Hardware Error Injection, Analysis and Tolerance at Operating System Level

A thesis submitted in partial fulfillment of the requirements for the degree of Master in Innovation and Research in Informatics with specialization in High Performance Computing by

Sergiy Bagdasar

Advisor: Ramon Canal Corretger (DAC)
Co-advisor: Juan José Costa Prats (DAC)

Facultat D’Informática de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Date of defense: June 25th, 2018
Abstract

As the high performance computing systems demand more and more memory to handle the increasing size of data sets, the error tolerance in DRAM becomes a major concern. Even though a considerable amount of works has been dedicated to hardware errors in memory both for injection and tolerance, none was aiming to a granularity of a process.

In this work, we present a software approach to hardware error injection in a running process on Linux which allows users to test a specific program for memory error tolerance. Towards this direction, we are going to introduce three tools for users ranging from advanced to unexperienced: `va_pid_to_pa`, `inject_mce_to_process` and `tui`. We are going to show that the overhead of the injection on the target process when running on a single thread is linear to the number of injected errors with an overhead of around 2% for 1.5 injected errors per second and up to 85% for 35 injected errors per second.

Afterwards, we are going to analyze the existing hardware error reporting tools on Linux and the error tolerance techniques they provide. We are going to propose a new approach to tolerate memory errors at process level by deploying selective process check-pointing after analyzing hardware corrected errors on physical memory pages.

Finally, we are going to present our evaluation and experimental results where we show that the overhead of this approach on the execution time of a running process ranges from less than 1% to about 5% and it solely depends on the amount of memory used by the process and the amount of memory changes made during the execution time.
We can only see a short distance ahead, but we can see plenty there that needs to be done. - Alan Turing
Acknowledgements

For this work I would like to deeply thank my advisor Ramon Canal Corretger and my co-advisor Juan Jose Costa Prats for our weekly meetings. My advisors assisted and guided me throughout this work with their knowledge and experience, with patience and understanding.

I would also like to thank my family and especially my fiancee, Christina, for their never ending support and help, but most of all, for believing in my skills and abilities.

Finally, I would like to thank the open source community who works tirelessly to produce free and open source software without which this work would have been impossible.
# Contents

Abstract ............................... i

Acknowledgements .................. iii

List of Tables ....................... vi

List of Figures ...................... vii

1 Introduction ......................... 1
   1.1 Motivation .......................... 1
   1.2 Contribution ....................... 3
   1.3 Document organization ............. 3

2 Related Work ....................... 5
   2.1 Memory errors impact on the system .... 5
   2.2 Hardware error injection ............. 5
   2.3 Hardware error tolerance ............. 6

3 Hardware Error Injection in a Running Process ............................ 7
   3.1 Machine-check architecture .......... 7
   3.2 MCE Injection tools .................. 9
      3.2.1 Translating a virtual address of a process to a physical address . 9
      3.2.2 Available interfaces on Linux .... 10
      3.2.3 Translation algorithm and validation .......................... 11
      3.2.4 Injecting MCE in a process ................... 14
      3.2.5 Adding abstraction and functionality .......... 16
   3.3 Tui overhead ....................... 18
   3.4 Conclusions .......................... 20

4 Error Reporting Mechanisms ........ 21
   4.1 mcelog ............................. 21
   4.2 edac ............................... 24

5 Improving Error Tolerance .......... 25
   5.1 Using non-volatile dual in-line memory modules (NVDIMMs) ............ 25
## Contents

5.2 Mapping a part of persistent memory to the user memory space ............................ 26  
5.3 Process checkpoint ................................................................................................. 27  
5.4 Process restore ...................................................................................................... 31  
5.5 Conclusions .......................................................................................................... 32  

6 Evaluation & Results ................................................................................................ 33 
  6.1 Experimental Setup ............................................................................................. 33  
    6.1.1 Hardware ......................................................................................................... 33  
    6.1.2 Software .......................................................................................................... 35  
    6.1.3 Methodology .................................................................................................... 37  
  6.2 Results .................................................................................................................. 39  
    6.2.1 Dummy process analysis .................................................................................. 39  
    6.2.2 CPU SPEC2006 benchmarks analysis ............................................................. 44  
  6.3 Conclusions .......................................................................................................... 48  

7 Conclusions & Future Work ...................................................................................... 49  

Appendices .................................................................................................................. 53  
  A Dummy process data .............................................................................................. 54  
  B CPU SPEC2006 processes data ............................................................................. 55
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Pagemap 64 bit value.</td>
<td>11</td>
</tr>
<tr>
<td>6.1</td>
<td>CPU information</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>Memory device information</td>
<td>35</td>
</tr>
<tr>
<td>A.1</td>
<td>Dummy process times(s)</td>
<td>54</td>
</tr>
<tr>
<td>B.1</td>
<td>CPU SPEC2006 benchmarks times</td>
<td>55</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 The relative failure rate for servers with different DIMM capacities [1]. . . . 1
1.2 The relative failure rate for servers with different chip densities [1]. . . . 2

3.1 Software injection vs hardware failure. . . . . . . . . . . . . . . . . . . . . 8
3.2 Maps example of a dummy process. . . . . . . . . . . . . . . . . . . . . . 10
3.3 MCE injection to a process. . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.4 Tui. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.5 Execution time with injections on one available thread. . . . . . . . . . . . 19
3.6 Injection overhead over error injection rate on one available thread. . . . 20

4.1 mcelog basic error flow. . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.2 mcelog log file CE example.. . . . . . . . . . . . . . . . . . . . . . . . . . . 23

5.1 Error Tolerance Scheme. . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

6.1 401.bzip2 memory usage. . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
6.2 Application execution with two error thresholds. . . . . . . . . . . . . . . 37
6.3 Application execution with one error threshold. . . . . . . . . . . . . . . 38
6.4 Time breakdown for 6 dummy processes after 4 CE’s. . . . . . . . . . . . 39
6.5 Time breakdown for 6 dummy processes after 8 CE’s. . . . . . . . . . . . 41
6.6 Time breakdown for 6 dummy processes after 12 CE’s. . . . . . . . . . . . 42
6.7 Checkpoints overhead for 6 dummy processes after 12 CE’s. . . . . . . . . 43
6.8 Restore time for 6 dummy processes. . . . . . . . . . . . . . . . . . . . . . 44
6.9 Time breakdown for 3 spec processes after 4 CE’s. . . . . . . . . . . . . . 45
6.10 Time breakdown for 3 spec processes after 8 CEs. . . . . . . . . . . . . . 46
6.11 Time breakdown for 3 spec processes after 12 CEs. . . . . . . . . . . . . . 47
6.12 Checkpoints overhead for 3 spec processes after 12 CEs. . . . . . . . . 47
6.13 Restore time for 3 spec processes. . . . . . . . . . . . . . . . . . . . . . . 48
Chapter 1

Introduction

1.1 Motivation

Tolerating hardware errors in DRAM is an increasingly important topic. High performance computing systems are using more and more memory to handle the increasing size of the data sets. Inevitably, the increasing number of transistors in DRAMs provides more surface for hardware errors.

As we see in Figure 1.1, from the research that has been done on the entire Facebook server fleet [1], the server failures are not related to the different DRAM-DIMM capacity. Figure 1.2, however, shows that the server failures are related to the density of the DRAM. This observation supports the argument above, because the DRAM density is increasing over time in order to provide DRAM devices with more capacity which is necessary to handle increasing data sets.

![Graph showing relative failure rate for servers with different DIMM capacities.](image)

**Figure 1.1:** The relative failure rate for servers with different DIMM capacities [1].
Figure 1.2: The relative failure rate for servers with different chip densities [1].

There are a few other reasons which could lead to DRAM errors like the manufacturing process, the wearing out and the cosmic rays. High performance computing systems deploy techniques based on error correction codes (ECC) in order to detect and correct hardware errors, if possible. As we are going to present later, software techniques like bad page off-lining are deployed as well.

For this work, it is useful to define two types of hardware errors: corrected errors (CEs) and uncorrected error (UCEs). A corrected error is a hardware error condition that has been corrected by the hardware or the firmware by the time that the operating system is notified about the presence of the error condition. An uncorrected error is a hardware error condition that cannot be corrected by the hardware or the firmware.

Uncorrected errors are further classified as either fatal or nonfatal. A fatal hardware error is an uncorrected or uncontained error condition that is determined to be unrecoverable by the hardware. When a fatal uncorrected error occurs, the operating system generates a bug check to contain the error. A nonfatal hardware error is an uncorrected error condition from which the operating system can attempt recovery by trying to correct the error. If the operating system cannot correct the error, it generates a bug check to contain the error.
1.2 Contribution

After some research in this field, we realized that although there are software tools to inject memory hardware errors, we could not find any tool that could inject errors to a specific process of the system and we also found a limited number of approaches to tolerate hardware errors at a software level. Thus, we decided to set four goals:

1. Create a tool for hardware error injection to the memory of a running process on Linux.
2. Add abstraction and functionality, so that even unexperienced users would be able to test their software by injecting hardware errors.
3. Increase the tolerance to hardware errors in memory at an OS level.
4. Exploit emerging technologies like NVDIMMs, but also make our tools usable on conventional systems.

1.3 Document organization

Our work is presented in this document as follows. We start with the related work in Chapter 2. In Chapter 3, we offer a set of tools that allow users to add hardware errors to a running process on Linux; while in Chapter 4, we present two available tools on Linux that allow hardware error reporting. Chapter 5 focuses on our hardware error tolerance approach by using process checkpoints after the analysis of corrected errors on a physical memory page. In Chapter 6, we perform a number of experiments with our tools to evaluate and analyze their performance and finally; in Chapter 7, we reach our conclusions and propose future work.

From now on in this work, we will be also referring to hardware errors as "machine-check", as we will use an Intel-based CPU to test our developments and to be consistent with Intel’s documentation.
Chapter 2

Related Work

In this chapter we are going to present prior studies that show the impact of memory errors on the system performance and the efficiency of software error tolerance approaches. Both of these focused on large-scale systems. Then, we are going to present prior works related to hardware error injection and tolerance.

2.1 Memory errors impact on the system

The first work was performed on Windows Servers with Intel’s Haswells in 2012 [2]. Researchers observed that for SPEC CPU2006 benchmarks, corrected memory errors can slow down the average execution time by up to 2.5x. For an interactive web-search workload, average query latency degrades by up to 2.3x for a light traffic load and up to an extreme 3746x under peak load.

The second work [1] was performed in 2015 on the entire Facebook fleet. The difference is that this work focused on much larger systems with Linux operating system. Among other observations, researchers showed that the offlining at scale can reduce memory error rate by 67%. We are going to explain page offlining in details in Chapter 4.

2.2 Hardware error injection

There are many works related to hardware error injections [3] [4] [5] [6] [7]. However, none of them developed techniques for error injection at process granularity. The *FAUmachine* [3] is a virtual machine with three very distinctive features: it runs as a user process on top of Linux on i386 and x86_64 hardware, it allows fault injection capability for experimentation and finally, it provides a VHDL interpreter for automating experiments.
and tests based upon another project called fauhdle. In practice, this virtual machine allows the injection of errors into virtual hardware components.

Another older work [4] which was published in 1997 summarized the available tools for hardware and software error injections. At a software level, the injection can be done at compile time by modifying a program’s instructions or at runtime where an interrupt handler can be linked to an event or malicious code can be injected to a running program. This work mentions tools like Ferrari or Xception [4] towards this direction. At hardware error the injection can be done with or without contact. With contact the injection has direct impact on the system, for example by changing the voltage, and without contact an external source produces some physical phenomenon, such as heavy ion radiation and electromagnetic interference, inside the target chip. Tools like Messaline, FIST or MARS [4] have been developed since then towards this direction.

An approach which involves faulty virtual parts is presented in [5]. This approach shares some similarities with ours, but it adds an abstraction level. In order to inject errors to memory, the user needs to translate a virtual address of a process running on a virtual machine to a physical address of the guest. Then, the user manually provides it to a tool running on the guest which writes the error on the model specific registers of the virtual CPU. In our approach, it is possible to inject errors to memory directly to the OS and the process can be automated.

The most important related works to ours are the mce-inject [6] and mce-test [7]. The first tool allows error injection to model specific registers and the latter provides many different error templates to test the RAS features of a system. We are going to provide more details on these in the next chapter.

### 2.3 Hardware error tolerance

The tools available for hardware error tolerance in memory are HWPOISON [8] and mcelog [9]. HWPOISON uses the machine check architecture - more on it in the next chapter - to connect a poisoned memory data handler, which then manages the memory at a physical page level and tries to isolate the physical pages affected by the errors. Mcelog uses a similar method and it seems to be the natural evolution of HWPOISON since both share the same author. The method is called page offlining. Mcelog also provides bad cache offlining and error reporting mechanisms. Error reporting mechanism is provided by the edac tool [10] as well. Mcelog is a very important tool for this work and it is analyzed in Chapters 4 and 5.
Chapter 3

Hardware Error Injection in a Running Process

In this chapter, we will describe the tools we developed, the problems we encountered and the solutions we provided to achieve two goals. Firstly, we wanted to allow the users to "attack" the physical address space of a running process by using all the power of the mce-inject [6]. Secondly, we wanted to add an abstraction layer to allow even the most unexperienced users in the area to examine the memory space of a process and perform some basic error injections. However, in order to explain the implementation of our tools, first we are going to describe the machine-check architecture (MCA) as well as introduce the mce-inject tool.

3.1 Machine-check architecture

As mentioned in the introduction, the first big part of this work is about the machine-check error (MCE) injection using software. This is particularly useful for testing and debugging purposes, as it can be used to test MCE tolerance at OS level without physically causing any hardware malfunction. After some research in this field, we were able to find existing tools that work towards this direction. The most popular is the mce-inject [6]. This tool allows to inject machine-check errors on software level into a running Linux kernel. The user has to provide an error description file as input, which specifies the error fields that depend on the Model-Check Architecture (MCA) of the underlying machine.

MCA provides a mechanism for detecting and reporting hardware (machine) errors, such as: system bus errors, ECC errors, parity errors, cache errors and TLB errors. It consists
of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs are used for recording any detected errors. Details on MCA and the available MSRs can be found in Intel’s Manual [11]. The model-specific registers are unique to any processor model and are found in the respective data sheet volume 2 in Intel’s Resources [12]. In fact, there are tools to test the available MSRs on Linux distributions by loading the `msr` module to the Linux kernel. The tools are the `rdmsr` and the `wrmsr`. By defining the register number, a user can read the content of this register or write a new value to it using the above tools respectively.

In Figure 3.1, we notice the difference between an MCE which was caused by a hardware failure or cosmic rays (red box) and the software injection using mce-inject (blue box). For the operating systems the outcome of both is exactly the same. The difference is that, in the first case, a fault is detected as error upon the access of a memory location and written to MSRs by the hardware and in the second case, the same error details (e.g. address, socket, status, error type) were written to MSRs by the software. The difference between a fault and an error is
that a fault becomes an error when the faulty memory location is accessed. Faults that do not become errors are not used in any case throughout this work.

Although the mce-inject can be used to inject almost any machine-check error (MCE), we were not able to find tools that would associate an error with a specific process executing on the system. This case was particularly interesting for us, since it would allow testing the hardware error tolerance of sensitive processes. We were interested in memory errors in the first place because memory provides the biggest surface for data corruption. Thus, we decided to work towards this direction.

### 3.2 MCE Injection tools

Based on our goals, the existing tools and the available Linux resources we developed three following tools:

1. *va_pid_to_pa*, developed in C programming language, compiled with *gcc* 7.2. This tool translates the process ID and a virtual address to a physical address.

2. *inject_mce_to_process*, developed in Python 3 programming language executed with Python 3.6.3 interpreter. This tool receives a process ID, a virtual address and an error description file. Then, it translates the PID and the VA to a physical address, substitutes any appearance of *0x&* in the error description file with the physical address and starts the injection.

3. *tui*, developed in Python 3 programming language and executed with Python 3.6.3 interpreter. This tool adds abstraction and usability by guiding the user through the MCE injection process.

All three will be presented in detail in the next sub-sections.

### 3.2.1 Translating a virtual address of a process to a physical address

According to the mce-inject documentation [6], the tool accepts an address as input which in Intel’s developer manual [11] is considered to be a physical memory address on modern memory systems. We also know that in modern systems any process sees the available storage as a unified address space where all the memory is available at any moment. In other words, a process can only access its virtual memory space and it is not able to see the memory of other processes. Moreover, modern operating systems use
structures, like page tables (PT), to store the translations between virtual and physical pages and a Translation Look-aside Buffer (TLB), to cache translations and speed up future accesses.

This implies that to inject MCEs to the main memory of a running process, we need to translate virtual addresses (VAs) that are used by this process to physical addresses (PAs) and then use them with mce-inject. To solve this problem, we developed our first tool: `va_pid_to_pa`.

### 3.2.2 Available interfaces on Linux

Linux operating systems offer two major interfaces that are used in order to perform the translation of a process VA to a PA. The first can be accessed through the `/proc/PID/maps` file where the PID is the ID of a process under execution. In Figure 3.2, there is an example of the information we can obtain from the maps interface.

In order to get this information, we used a dummy process which is a simple process that loops forever for three minutes and sleeps for one second at the end of each iteration. This process is used for testing and evaluation repeatedly in this work and it is explained in more detail later on.

<table>
<thead>
<tr>
<th>#</th>
<th>Address region</th>
<th>Perm</th>
<th>F offset</th>
<th>Dev</th>
<th>Inode</th>
<th>Pathname</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>55da8b985000-55da8b988000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>6430243</td>
<td>/home/wespal01/error_injection_tnf/tmf-ugo/dummy_proc.out</td>
</tr>
<tr>
<td>1</td>
<td>55da8b986000-55da8b987000</td>
<td>r-p</td>
<td>00010000</td>
<td>08:08</td>
<td>6430243</td>
<td>/home/wespal01/error_injection_tnf/tmf-ugo/dummy_proc.out</td>
</tr>
<tr>
<td>2</td>
<td>55da8b987000-55da8b988000</td>
<td>r-xp</td>
<td>00020000</td>
<td>08:08</td>
<td>6430243</td>
<td>/home/wespal01/error_injection_tnf/tmf-ugo/dummy_proc.out</td>
</tr>
<tr>
<td>3</td>
<td>55da8b98d000-55da8b98e2000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>5</td>
<td>7f4d000000-7f4d00070000</td>
<td>---</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>6</td>
<td>7f4d000700-7f4d000e8000</td>
<td>r-xp</td>
<td>00010000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>7</td>
<td>7f4d000e8000-7f4d000e9000</td>
<td>r-xp</td>
<td>00010000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>8</td>
<td>7f4d000e9000-7f4d000f0000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>01230774</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>9</td>
<td>7f4d000f0000-7f4d000f4000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>10</td>
<td>7f4d000f4000-7f4d000f5000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>11</td>
<td>7f4d000f5000-7f4d000f6000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>12</td>
<td>7f4d000f6000-7f4d000f7000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>13</td>
<td>7f4d000f7000-7f4d000f8000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>14</td>
<td>7f4d000f8000-7f4d000f9000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>15</td>
<td>7f4d000f9000-7f4d000f9000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>16</td>
<td>7f4d000f9000-7f4d000f9000</td>
<td>r-xp</td>
<td>00000000</td>
<td>08:08</td>
<td>12320851</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
<tr>
<td>17</td>
<td>fffffff000000-fffffff0001000</td>
<td>r-xp</td>
<td>00000000</td>
<td>06:00</td>
<td>0</td>
<td>/lib/x86_64-linux-gnu/libc-2.26.so</td>
</tr>
</tbody>
</table>

**Figure 3.2:** Maps example of a dummy process.

Each row in the maps interface describes a region of contiguous virtual memory in a process or a thread. In the example above, we have enumerated 18 continuous virtual memory regions under the R# column. The Address region column provides the starting and the ending virtual address of a given region. The Perm column describes the permission of the process to access a given region. There are four different permissions: read(r), write(w), execute(x) and share(s). If none of the three first permissions is granted, a "-" appears and if the region is private, then a "p" will appear instead of an "s". The F_offset is the offset in file if the file was previously mapped with mmap.
otherwise a zero appears as an offset. The \textit{Dev} is the major and minor device number (in hex) of the device where the previously mapped file was located, otherwise this field remains zero as well. The \textit{Inode} field is the file number if the region was mapped from a file. Finally, the \textit{Pathname} column represents the path to a file if any file was mapped to this region, it remains blank for anonymous regions and it is tagged for special process sections like a stack, a heap or a vvar.

The \textit{maps} interface is not necessary for the translation procedure itself, but it is extremely important to provide the user with a picture of the virtual space of any process that is actually mapped and to build an abstraction layer with multiple options on top of that.

Another file provided by Linux, which is necessary to perform the translation, is the \texttt{/proc/PID/pagemap} \cite{13}. This file is available since Linux 2.6.25 and shows the mapping of each of the processes virtual pages into physical page frames or swap areas. It contains one 64-bit value for each virtual page, with the bits set as explained in Table 3.1

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>If set, the page is present in RAM.</td>
</tr>
<tr>
<td>62</td>
<td>If set, the page is in swap space.</td>
</tr>
<tr>
<td>61</td>
<td>(since Linux 3.5) The page is a file-mapped page or a shared anonymous page.</td>
</tr>
<tr>
<td>60:56</td>
<td>(since Linux 3.11) Zero.</td>
</tr>
<tr>
<td>55</td>
<td>(since Linux 3.11) PTE is soft-dirty.</td>
</tr>
<tr>
<td>54:0</td>
<td>If the page is present in RAM (bit 63), then these bits provide the page frame number (PNF). If the page is present in swap (bit 62), then bits 4:0 give the swap type, and bits 54:5 encode the swap offset.</td>
</tr>
</tbody>
</table>

\textbf{Table 3.1:} Pagemap 64 bit value.

Before Linux 3.11, bits 6 – 55 were used to encode the base-2 log of the page size. The 55 least significant bits, which provide the page frame number, in essence provide the translation of a virtual page address to a physical page address.

\subsection{Translation algorithm and validation}

The previous leads us to the complete translation algorithm. Provided the \textit{process ID (PID)} and a \textit{virtual address (VA)} that is mapped in this process, here is the algorithm:

1. Read the \textit{pagesize} from the system.
2. Compute the page offset \( PO \) as \( VA \) modulo pagesize.

3. Open /proc/PID/pagemaps file as pagemaps.

4. Seek to \( VA \) in pagemaps.

5. Read a 64 bits value.

6. Compute the PNF as a bitwise AND between the 64 bits value and \( 0x7FFFFFFFFFFFFF \). This masks out the 9 most significant bits.

7. Compute the physical address \( PA \) as \( (PNF \times \text{pagesize}) + PO \)

8. Return \( PA \).

A crucial side of this algorithm is to use the appropriate data types for the implementation process. The 64 bits value from the pagemaps, the page frame number and the physical address itself should all take 64 bits data types, which in our case was unsigned long long. It is also very important to use the unsigned types to avoid any unexpected behavior that might appear because of the bitwise AND operation that is used.

The algorithm above provided us with an number which should represent a physical page number. We needed to develop an approach to validate it. The first approach towards this direction was to modify the memory of the translation tool itself. The steps for this test are as follows:

1. Create a buffer in the heap and initialize it to the "Hello, World!" string.
2. Translate the starting virtual address of the buffer to a physical address.
3. Check physical address stability.
4. Open the memory device /dev/mem.
5. Write the string "Hello, Linux!" to the physical address on the memory device.
6. Check if the buffer’s value contains "Hello, Linux!" by using its virtual address.
7. Return Success if true and Failure otherwise.

The first issue we wanted to observe with this test is the stability of the physical address in step 3. The problem here is that the mapping of a virtual address to a physical address does not remain the same during the life of a process. The physical address might be changed by the operating system because another process requests a memory size that is not currently available and thus, the OS might swap out physical pages that were
assigned to the first process [14]. If the part of the virtual memory that was swapped out is accessed again in the future, it is very likely that the physical pages assigned to it are different. For our purpose, this means that in the time period from the address translation to the error injection, the physical address might have changed and no longer belong to the process that we actually want to ”attack”.

The only way we were able to find in order to solve the problem is to use the \texttt{mlockall} [15] system call, which locks all pages mapped into the address space of the calling process into the memory. The problem when using mlockall() is that the developer of the target program must perform the call itself. Another way would be to add a wrapper process (fork and execv) of the target process and call mlockall(), but according to documentations, the locks are not inheritable to the child process and are automatically removed when the execv is called. Therefore, there is no workaround to this: the developer of the target process has to call mlockall() explicitly.

Of course, this does not mean that the injection cannot be successful, even without the locking. It is still possible to detect that the injection was performed in the correct address with very high probability if the translation returns the same physical address after the injection, as it did before.

The stability check was a simple loop which performed the same translation many times with some time difference in between to count the number of address changes. Meanwhile, we tried to stress the memory of our system to the limits in order to force the OS to swap out the pages of our process. In our experiments, a change only occurred once when we stressed the memory to 100% use and the swap space to 90% of use. However, even with that load we could not see any further changes. We also did not encounter any injection problems with our experiments in Chapter 7.

The fourth step in this test opens the memory device (\texttt{/dev/mem}) [16], which is a Linux file that is an image of the main memory of the computer. Byte addresses in \texttt{/dev/mem} are interpreted as physical memory addresses. The problem here is that, since Linux 2.6.26, the access to this device is forbidden and the only way to use it is to recompile the kernel with the \texttt{CONFIG\_STRICT\_DEVMEM}=$n$ configuration option. In order to perform the validation steps, we did recompile the kernel and any user who wants to invoke the tests needs to do it as well. Solely for the translation, the tool works on any Linux version $\geq$ 3.11 without any modification.

After the test execution, we confirmed the validity of the physical address, but we wanted to modify a memory location of another process as well, in order to ensure that there are not any other security restrictions and to provide extra validation.
Chapter 3. Hardware Error Injection to a Running Process

The new test consisted of the translation tool invoked with special switches and a dummy process. The dummy process is the same as the one mentioned earlier. It was a simple loop that would sleep for a second at every iteration for three minutes. Before the loop, a dummy process would initialize an integer variable $i$ and then print $i$'s virtual address alongside with its own PID. At the beginning of every iteration, the process would check if the value of $i$ has changes, but it would never change it by itself. Then we would invoke the translation process and provide it with the PID and the virtual address. The translation process would then translate this address to a physical one and access /dev/mem to replace the value of $i$ with another randomly generated integer. When this happened, the dummy process would realize the change of the $i$ variable and return a message.

The second test was also successful without any extra steps. Therefore, having had a tool that can accurately translate a PID and a virtual address to physical address, we moved on to accomplish our second goal, which was adding abstraction and usability.

### 3.2.4 Injecting MCE in a process

To inject an MCE in a process, the user must now manually invoke the `va_pid_to_pa` tool, provide the virtual address and the PID, get the physical address back and then prepare an error description file using the physical address. Finally, the user has to invoke the `mce-inject` and provide the error description file. To make this process faster and easier, we decided to develop another tool which is called `inject_mce_to_process`. 
Chapter 3. Hardware Error Injection to a Running Process

Figure 3.3: MCE injection to a process.

Figure 3.3 shows the localization of this tool in our stack. Both the tools that we have introduced are in the blue color and are running in user space with root permissions. Mce-inject, however, runs in the kernel space because it uses a kernel module. When `inject_mce_to_process` is used, the following steps are followed:

0. The user invokes the tool and provides the PID, the VA and a Template. The PID is the process ID of the target process, the VA is the virtual address that the user wants to ”attack”. The Template is an error description file which will be used by the mce-inject, but instead of using any particular address, the user will use the `0x&` symbol. In fact, some Templates can be pre-prepared for the user. Many error description files can be found in the mce-test [7] repository. The mce-test tool is a collection of tools and test scripts for testing the Linux RAS related features, including CPU/Memory error containment and recovery. We included two error templates for corrected and uncorrected page errors. In Chapter 5, we will discuss this in more detail.

1. `inject_mce_to_process` invokes `va_pid_to_pa.c` by providing the PID and the VA to get the PA back.
2. `inject_mce_to_process` reads the error description template and replaces any appearance of `0x&` with the obtained PA.

3. A temporal error description file with the correct PA is created.

4. The mce-inject is invoked with the temporal file as input.

5. The mce-inject performs the injection by writing the error details into MSRs.

After this, the temporal file is cleared and step 1 is repeated. The new PA is compared with the old PA to detect if the injection was performed on a correct address. If the address has changes, it is likely that the injection was performed to a wrong PA, otherwise the probability of success is very high, as explained earlier. Finally, the user gets the appropriate notification. Although this tool performs a relatively simple task, it is very useful when a user wants to exploit the full power of mce-inject with a specific process.

### 3.2.5 Adding abstraction and functionality

Up until this point, we assumed that the user knows exactly the virtual addresses that they used for the error injection. This may be true for advanced users, but it still includes the overhead of manually examining the virtual address space of a process and picking a virtual address. The task becomes more difficult when a user wants to repeat an injection or to attack a specific section of the process. To solve these problems, we developed a text user interface (tui).

Figure 3.4 shows the flowchart that represents the user’s interaction with the tool. Initially, the user is asked for the target process’s ID. Then, the user can choose 1 out of the 4 available options: List Maps, Random VA, Already know the VA and Random VA in a region. The first option will simply enumerate the available virtual memory regions for inspection. The Random VA option will generate a random virtual address in one of the available regions. The Already known option will provide three more choices: the user can provide one VA, a list of VAs or a continuous VA range.

When the VA generation is finished the tool will start preparing the MCEs. By listing the VAs one by one, the user needs to choose the error type for each VA. The error types depend on the templates that the tool reads from a pre-configured directory. As was already mentioned, we provided a template for corrected and uncorrected page errors. Then, the user will provide the number of repetitions of this MCE and finally, confirm the injection.
After the confirmation, the tool will start the injection of MCEs one by one. Each MCE will be repeated according to the user’s choice. One MCE is injected by using the `inject_mce_to_process` which takes the PID, the corresponding VA and the error template, as it was chosen at the error type selection by the user.

It is important to mention that the random generation of virtual addresses can be time consuming in some cases. The random generation of one virtual address consists of two parts. Firstly, a virtual region is chosen randomly after parsing the `/proc/PID/maps` file. Then, the virtual page address is chosen randomly from the addresses that belong to the previously chosen region. This address, however, is not guaranteed to be mapped to a physical address. If it is not, another address is randomly generated and the procedure is repeated. If the number of unsuccessful attempts exceeds a particular threshold, then another virtual region is chosen. If the number of requested virtual addresses is too high, this process could fail or take a considerable amount of time. The second part of the virtual address is the offset, which can by any random number in $[0, \text{pagesize} - 1]$.

Initially, the random VA generation would try to pick the whole address at once, but this is much more difficult. Since there can be a big amount of virtual addresses, the ones that belong to the same virtual page will always fail if that page is not mapped to a physical address. To avoid the unnecessary repetition, we changed this logic to the one above.

This tool was the last of the MCE injection stack that we developed for this work. All three tools allow users of all experience levels to inject MCEs to a running process on Linux operating systems. After this part, we wanted to analyze the error reporting mechanisms that are present in a modern OS like Linux and their software approaches to tolerate hardware errors.
In this section, we are going to briefly analyze the overhead that can be caused by \textit{tui} to the target process. In order to understand this, we performed the worst case scenario, where the \textit{tui} execution will directly affect the performance of the target process. We

\section*{3.3 Tui overhead}

In this section, we are going to briefly analyze the overhead that can be caused by \textit{tui} to the target process. In order to understand this, we performed the worst case scenario, where the \textit{tui} execution will directly affect the performance of the target process. We
disabled all of the available threads on our processor, except for one, using the command

```
echo 0 | sudo tee /sys/devices/system/cpu/CPU-NUM/online
```

Then, we executed three different target processes from CPU SPEC 2006 benchmarks one by one: 458.sjeng, 470.lbm and 401.bzip2. Briefly, we choose these benchmarks because they are CPU intensive with increasing memory usage and diverse execution times. We are going to reuse and analyze those programs in more detail later. Also, the details of our CPU can be found in Chapter 6.

We repeated the execution of each benchmark with 0, 10, 100, 1000, 10000 corrected errors. The result is presented in Figure 3.5, where we see the execution time of the target process for the different number of corrected errors. This information reveals that the time is linear to the number of corrected errors for each process and thus, tui can be considered scalable. Of course, every process causes an overhead to the tui as well, since they run on the same physical single-thread core. Thus there is a possibility that, if the number of errors is too high, there will not be enough time to finish the injection before the target process is finished.

![Figure 3.5: Execution time with injections on one available thread.](image)

In Figure 3.6, we see the execution time overhead of the injection for each process over the error injection rate. As was expected, the overhead is proportional to the error injection rate, but also inversely proportional to the execution time of each benchmark. Thus, for any given error injection rate bzip2 has the highest overhead, the lbm is in the middle and sjeng has the lowest overhead.
The overhead is quite high for the bzip2 benchmark (up to 85% for 35 errors per second) because this benchmark is the fastest of the three, as shown above. Generally, the error injection rate is considerably high and, even on our small core, the tui can finish the injection with a very reasonable overhead for the two of the three benchmarks and an acceptable overhead for the last one.

![Figure 3.6: Injection overhead over error injection rate on one available thread.](image)

### 3.4 Conclusions

In this chapter, we presented three tools that can help the user to inject MCEs to a running process on Linux. The user can translate a virtual address of a process with `va_pid_to_pa` to a physical address and then use it with mce-inject. Instead, the user can run `inject_mce_to_process` and avoid manually updating the error description file of mce-inject, while still being able to use all of its features. Finally, the user can invoke `tui` to inspect the virtual address space of a process, select random virtual addresses or provide their own and start the injection. We have also shown that the execution time of the target process when injecting CEs using `tui` on a single thread is linear to the number of corrected errors. The overhead on the target process execution time is also linear to the error injection rate.
Chapter 4

Error Reporting Mechanisms

In this chapter, we are going to present existing MCE reporting tools on Linux, their features and even their own mechanisms to tolerate MCEs at software level. This is needed because our own tools, which we are going to describe in the next chapter, make heavy use of one of them, the `mcelog`. There are two well known MCE reporting tools on Linux: `mcelog` [9] and `edac` [10].

4.1 mcelog

Mcelog is a daemon for handling and reporting hardware errors. It is able to use trends in corrected error reports to implement specific error prevention algorithms. In modern x86 systems the errors are reported to the system through the machine check architecture (MCA), basically using the model specific registers (MSRs). The uncorrected errors are directly handled by the kernel which can kill the process currently pointed by the instruction pointer, if it is associated by the error or shut down the entire system. From our experience, it is most likely to be the latter and that is why we chose an old Linux machine for our work.

The corrected errors which we are interested in in this work are handled by an interrupt and reported to the mcelog. The mcelog daemon decodes the errors, performs accounting and book keeping. The corrected errors can be used to recognize patterns and predict future failures and this is exactly how we are going to use mcelog for our own error tolerance tool.
In Figure 4.1, we notice a high level representation of the mcelog memory error flow. When the memory controller detects an error using techniques based on ECCs, it is going to write the error details to the model specific registers. An interrupt will be generated and the kernel will send the information to the mcelog daemon. Then, mcelog will update its memory accounting database and also add the error to the log file. Using the memory accounting database, the mcelog can send information through its socket client or initiate a trigger, if a certain error threshold is surpassed. In the next chapter, we are going to explain how do we interact with mcelog and why.
Chapter 4. Error Reporting Mechanism

In Figure 4.2, we see an example of the mcelog log file after we have injected a few corrected errors in the physical page with address 0x132088a30. As we can see, it provides the exact number of errors on the page, the physical address of the page, the MCE that was injected, the CPU information and more that can be found in Intel developer’s manual [11].

There are currently two error tolerance features offered by mcelog: bad page offlining and cache error handling. Bad page offlining aims to solve the stuck bit problem which could develop in the main memory. The problem is that the bit is stuck in a specific state which constantly generates corrected errors on reads. This is easy to correct even with the simplest ECC based solutions, but it can eventually lead into uncorrected error if another bit on a memory word gets stuck. To avoid constant corrected errors which can hit the system performance and prevent uncorrected errors at software level, the OS which has control over physical memory pages will copy the content of the page to another page and silently remap it. This is also called soft offlining, since the process which uses the affected physical page does not have knowledge of this change. The downside is that offlining does not work for all pages, but only for pages in the Linux page cache (that is containing application memory or file data) and free pages. This includes the majority of the memory of common work loads, but some page types are excluded. Our approach is more generic and does not have any such restrictions.

The cache error handling works on recent Intel processors which mark the health status of caches. According to [11], the processor tracks the corrected errors on the cache lines and marks the cache with a green or a yellow status. The cache will be marked with a green status if the error corrections are below a certain point and a yellow otherwise. When a cache is yellow, Intel suggests to maintain the system in the next few weeks. When the mcelog realizes the yellow status of a cache, it will disable the core that uses
that cache and allow the operation of the system with reduced resources. We only worked on memory errors, so we did not develop any error tolerance techniques for caches.

### 4.2 edac

Since we did not use edac [10] in this work, but it is the only known alternative to mcelog, this is going to be a brief reference to this tool. EDAC stands for "Error detection and correction" and it is a set of separate drivers for different hardware platforms that allows hardware error accounting, but does not offer advanced error tolerance methods, like bad page offlining or cache error handling of mcelog. The software does not seem to be maintained actively.
Chapter 5

Improving Error Tolerance

By looking into the existing error tolerance mechanisms provided by a modern OS, like Linux, we realized that more software approaches can be developed. In the same way as in the Chapter 3, we were interested in developing a new approach at process granularity. Moreover, we were interested in exploiting new technologies that could provide advantages to our error tolerance mechanism over the existing ones, in the near future, but still be able to work on today’s systems. In this chapter, first, we present a technology called non-volatile dual in-line memory modules (NVDIMM’s). Then we present an initial attempt to use it as a part of our error tolerance approach which at the end was not successful. After that, we present our final approach which involves creating process checkpoints.

5.1 Using non-volatile dual in-line memory modules (NVDIMMs)

We were aware of an emerging technology called non-volatile dual in-line memory modules (NVDIMM’s), so we decided to build our solution around it. The main two characteristics of an NVDIMM is that it can provide data persistence at speeds that are much higher than conventional hard drives and SSD’s and, at some cases, are even comparable to DRAM speed [17].

The technology itself was not available to us, but fortunately there is the Persistent Memory Development Kit (PMDK) [18] which was developed by the community with strong contribution by Intel. PMDK is a collection of libraries built on the Direct Access (DAX) feature available in both Linux and Windows, which allows applications
direct load/store access to persistent memory by memory-mapping files on a persistent memory aware file-system. In other words, DAX bypasses the page cache.

According to T.Krenn [19], under Linux, the page cache accelerates many accesses to files on non volatile storage. This happens because, when it first reads from or writes to data media like hard drives, Linux also stores data in unused areas of memory, which acts as a cache. If this data is read again later, it can be quickly read from this cache in memory.

For block devices that are memory-like (for example an NVDIMM), the page cache memory pages would be unnecessary copies of the original storage. The DAX code removes the extra copy by performing reads and writes directly to the storage device. For file mappings, the storage device is mapped directly into the userspace.

The official PMDK website [18] suggests that the users of PMDK follow the SNIA [20] programming model, seeing as the available libraries are built around it. Towards this direction, the most convenient way to use the advantages of persistent memory, like NVDIMM, is to port the data structures of an application to the data structures provided by PMDK libraries which will be stored in the persistent memory automatically during the program execution.

One example of this approach was developed by Intel’s developer Eduardo B. [21]. He presents the famous map-reduce algorithm for persistent memory (PMEM), using the C++ bindings of libpmemobj, which is a core library of PMDK. This implementation demonstrates the fault tolerance ability of this algorithm when using persistence memory. By replacing the conventional data types with data types provided by libpmemobj library, all the intermediate results are stored in persistency and the application can be resumed even after a total crash. This can potentially happen with an acceptable performance hit.

In our case, it was impossible to follow this approach because we wanted a generic technique that could be applied to any process in the system without any code modification. Based on that, we analyzed two main ideas.

5.2 Mapping a part of persistent memory to the user memory space

The first was to map a part of the NVDIMM to the memory. Since the mcelog already provides page off-lining as was explained in the previous chapter, the OS could move
pages with high number of errors to the NVDIMM. This would have the advantage of saving space on the DRAM, but there were two problems.

We could not find any way to use the persistence provided by the NVDIMM, thus we would be using the NVDIMM as volatile memory, which brings us to the second problem. By design, NVDIMMs are slower than DRAM because of their persistence, so what we wanted to do is actually slow down a given process without providing any significant advantages.

Still, since this technique could help reduce the pressure on the DRAM during intensive errors, we thought that it might be useful to measure the performance of a program when some of its pages have been moved to the NVDIMM and this brought us to another problem: we did not possess any real NVDIMM device. The idea was to emulate the NVDIMM interface by setting a part of the DRAM to act as an NVDIMM device. The emulation process, though, was only useful to try out the PMEM interface and was not emulating any real delays on the read/write times. In order to do that, we had to catch any read/write system calls manually using a tool like Intel’s ”Pin tool” and then add an artificial delay. We also needed to track any pages that would have been moved by the OS to the NVDIMM in order to measure the delay on those accesses only.

This might sound like a good solution, but it was not. The main reason is that the OS may decide to move a page to any other part of the memory for any reason, so it was possible that a page which was moved to the NVDIMM by mcelog would soon be moved to a different place and there was no way to know where and when. Moreover, we did not have any knowledge on the actual delay that we wanted to introduce, since there are few different types of NVDIMMs with different characteristics and we could not find any accurate numbers on any of those because they are not adopted yet. Finally, we decided to abandon this strategy and move on to the second, which is the strategy we converted into a tool.

5.3 Process checkpoint

The new strategy was to take advantage of the NVDIMM speed against conventional storage devices, as well as of the DAX capability that they might have. This solution should also work with any storage device. The idea was, as previously, to track the hardware errors per page using mcelog and use that information to checkpoint a process which was suffering from multiple errors on its memory pages. The first obstacle was that the mcelog only reports the physical addresses of the pages that suffer from hardware errors. We needed to actually find out which was the affected process.
What makes this task difficult is that the page table in any system maps the virtual addresses to physical addresses, but not backwards and this is how the `/proc/pid/pagemap` file on Linux was designed. In order to actually decide if a physical address was used by a process, we needed to scan the whole `/proc/PID/pagemap` file and make sure that the address actually appears there. While it might sound easy, this file is quite big (since it contains all the mapping to the physical memory) and there might be hundreds of running processes. Particularly on our system with 48 bits virtual address space and 4096 bytes long pages, this file contains $2^{36}$ entries of 8 bytes which makes 512GB and it is mostly filled with zeros. This would take years to scan for each process. For this reason, in reality the `/proc/PID/pagemap` is not a text file, it is an interface.

Instead, we used a list of PIDs in which the user is interested in, we parsed the `/proc/PID/maps` and checked every virtual address that is mapped by that process to see if it actually translates to the physical address which was ”attacked”. The whole scheme with more details is provided in Figure 5.1. The blue boxes mark the tools that we developed, the `error_tolerance` is the new tool introduced in this chapter. This tool works as follows:

![Figure 5.1: Error Tolerance Scheme.](image-url)
Chapter 5. *Improving Error Tolerance*

0. The user edits the configuration file to add the PIDs that they want to protect from possible hardware errors.

1. A fault occurs on a physical memory page. This could be a fault caused by a hardware failure, cosmic rays or the steps 1 and 2 could be software emulated, as explained in Chapter 3.

2. Upon a read action on the same page, the error detection mechanism would detect and correct the error but also write any related details to MSRs. This mechanism could be any known memory error check and correction technology in combination with Intel’s MCA architecture.

3. The mce-log module reads the error details from MSRs.

4. There are several ways to use the error reporting mechanism of mcelog. First, we could read the log file where the errors are filed by the mcelog. However, this would take too much effort to actually parse the log file. Another option could be to use mcelog server and develop a client which would communicate with mcelog server through a socket following a particular communications protocol. Our purpose was to detect hardware errors on memory pages. Using the client we could get the number of errors for every physical page, but this would incur a huge overhead, since a system can have a big amount of pages (1 KB pages on our system with 4GB of RAM would result in more than 1 million pages). For these reasons, we decided to use triggers. When a certain number of errors occur on a physical memory page (this number can be set in the mcelog configuration), the mcelog triggers a pre-configured script.

For example, if 10 hardware errors are encountered on page 0x123, then a pre-configured script will be invoked. Moreover, mcelog will pass some parameters to the script using environment variables. By examining the environmental variables, we decided to use two of them. The first is the MESSAGE variable. This was the only one that actually contained the physical page address of the page with multiple errors. The other was the THRESHOLD variable, which provides the number of errors (both corrected and uncorrected) that happened to a page in a period of time (for example 24 hours).

5. The tool reads the PIDs provided by the user.

6. The tool passes every PID to the `parse_maps`. The `parse_maps` reads the `/proc/PID/maps` file and creates an object for every region with the starting and the ending virtual address of that region. The regions are passed back to the `error_tolerance`. 
7. To speed up the translation of each virtual addresses in the virtual memory region (vmem) we developed a C extension called `is_my_pa`. This receives a PID, a vmem region and a physical page address. For every one of those addresses it then calls the address translation tool. If any of the virtual addresses translates to the physical addresses that we are looking for, this means that the process with the given PID is affected. Finally, the `error_tolerance` receives an answer. Initially, this process was considerably slow, even by using the C extension, so we decided to add parallelism using OpenMP. This was quiet straightforward, since all the addresses are independent and the speedup is linear. Another improvement was to translate only the addresses of virtual pages and not all of them, for the same reason mentioned in Chapter 3. This made the process page-size times faster.

Another improvement is the PA cache. The PIDs in the `config/watch.pids` are scanned sequentially, so that if the affected PID is the last in the list it will be delayed until the address spaces of the previous processes are scanned. To improve this we are caching every PA alongside its PID, so that from the second time onwards the affected PID will be scanned first because it is quite likely that the PA still belongs to the same PID. We will examine the overhead of this step in detail in the next chapter.

8. If the physical page address belongs to any of the loaded PIDs, the tool proceeds to the error analysis. The error analysis basically has to make one decision which is either ignore or fully checkpoint the process that is being "attacked" or checkpoint it incrementally.

At first, we were always using a full checkpoint. Whenever the number of corrected errors exceeds a certain threshold the tool calls `criu dump`. Checkpoint/Restore In Userspace, or CRIU [22] is a software for the Linux operating system. Using this, you can freeze a running application (or part of it) and checkpoint it as a collection of files on disk. You can then use the files to restore the application and run it exactly as it was during the time of the freeze. To fully checkpoint a process, `CRIU` receives a `dump` command, a directory to store the images, a PID and some minor switches. This will create a directory with all the images that are needed to restore the process and then kill it. CRIU has some limitations of its own that will also be mentioned in the next chapter.

The problem that appears when we only do the full checkpoint is that, by the time the number of corrected errors reaches the threshold, an uncorrected error might happen and the user will never be able to restore the process. On the other hand, by lowering the threshold we might end up killing the process too soon.
To solve this, we also introduced a low threshold. If the number of corrected errors on a page is between the low and the high threshold, then the tool calls CRIU with different commands in order to perform an incremental checkpoint. The incremental checkpoint creates a directory that only contains the differences with the previous incremental checkpoint. The directory management at this point is crucial, because CRIU requires a new directory for every new checkpoint and it also requires the directory of the previous checkpoint as an input. When the number of corrected errors finally exceeds the high threshold, the tool will create a full checkpoint by providing the directory of the last incremental checkpoint and then kill the process. A full checkpoint is recommended, but in case the process was killed by an uncorrected error before the full checkpoint was performed, a restore can be attempted with the files from the incremental checkpoint. This approach introduces some overhead over the first one which is the time to perform the intermediate incremental checkpoints. We will analyze this overhead and compare both approaches in the next chapter.

### 5.4 Process restore

In the previous sub-chapter, we described the way our tool prepares process checkpoints. All of the checkpoints and incremental checkpoints are stored in the `/etc/criu-mce-backup/` directory. Then, the full checkpoint directories start with the prefix `p-` where the incremental ones start with the prefix `pr` followed by the ID of a given process. For example, the full checkpoint of the process 123 will be stored in `/etc/criu-mce-backup/p-123` and any incremental checkpoints will be stored in `/etc/criu-mce-backup/pr123`. The latter will also contain one directory for every incremental backup starting from 0.

When the user realizes that a process of their interest was killed, they can attempt a restore. To do this, they can invoke `error_tolerance` with the `-r` switch. The tool will then scan the `/etc/criu-mce-backup/` directory for available checkpoints. For every available process the tool will annotate the PID with one of Incremental or Full tags and provide a list to the user. If a particular process has a full checkpoint, then only the Full will be available to the user for restore. After the list is presented to the user, they can input the PID number and the tool will attempt to restore. For this purpose, the CRIU is called with `restore` instruction as input, as well as the directory that contains the images.

Any user can decide to manage the files and directories in the `/etc/criu-mce-backup/` manually, but the naming conventions of the directories must be respected in
order to use `error_tolerance -r` as explained above.

5.5 Conclusions

In this chapter, we presented a tool called `error_tolerance`. It is used to checkpoint a running process on Linux after the analysis of the corrected errors on its physical pages. The error accounting is done by mcelog which triggers `error_tolerance` after a certain number of corrected errors on a physical page is exceeded. Then, our tool scans the virtual memory spaces of the processes that the user is interested in and decides to create either an incremental or a full checkpoint of that process. If the process is killed either by an uncorrected error or the CRIU, the user can attempt to restore it using the same tool with the `-r` switch.
Chapter 6

Evaluation & Results

In this chapter, we are going to present the results from our evaluation process. The purpose of the evaluation was firstly, to prove the correctness of the tools presented in Chapter 3 and Chapter 5 and then, to analyze the performance of the `error_tolerance` tool. Particularly, we wanted to provide a time breakdown which includes the overhead of the incremental checkpoint, the time of the process filtering which is the time to decide if a physical address belongs to a certain process or not, the time to perform a full checkpoint and the time to restore a certain process. We will also compare the two different approaches mentioned earlier in Chapter 5.2, the approach with one error threshold per page and the approach with a high and a low error threshold.

6.1 Experimental Setup

In this sub-chapter, we will provide details on the hardware and the software, as well as the methodology that we followed to produce the results presented in the following sub-chapter.

6.1.1 Hardware

Firstly, in Table 6.1 we see the characteristics of the CPU of the machine that we used to run our experiments. The CPU was a fourth generation Intel Core i3 with two cores and two logical threads per core with x86_64 architecture. The frequency of this CPU ranges from 800 MHz to 1.9 GHz depending on the system load. A three level data cache was also available with the 32K private, 8-way set-associative first level cache, 256K private, 12-way set-associative second level cache and a shared 3MB third level cache.
12-way set-associative cache.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU op-mode(s)</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order</td>
<td>Little Endian</td>
</tr>
<tr>
<td>CPU(s)</td>
<td>4</td>
</tr>
<tr>
<td>Thread(s) per core</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket</td>
<td>2</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>1</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>1</td>
</tr>
<tr>
<td>Vendor</td>
<td>Intel</td>
</tr>
<tr>
<td>CPU family</td>
<td>6</td>
</tr>
<tr>
<td>Model</td>
<td>69</td>
</tr>
<tr>
<td>Model name</td>
<td>Intel(R) Core(TM) i3-4030U CPU @ 1.90GHz</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>1598.422</td>
</tr>
<tr>
<td>CPU max MHz</td>
<td>1900.0000</td>
</tr>
<tr>
<td>CPU min MHz</td>
<td>800.0000</td>
</tr>
<tr>
<td>Virtualization</td>
<td>VT-x</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32K</td>
</tr>
<tr>
<td>L1i cache</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache</td>
<td>3072K</td>
</tr>
<tr>
<td>NUMA node0 CPU(s)</td>
<td>0-3</td>
</tr>
</tbody>
</table>

Table 6.1: CPU information

In Table 6.2, we also see the memory information. The setup had a simple DDR3 4GB memory which could support up to 1600 MT/s.
## Evaluation & Results

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Width</td>
<td>64 bits</td>
</tr>
<tr>
<td>Data Width</td>
<td>64 bits</td>
</tr>
<tr>
<td>Size</td>
<td>4096 MB</td>
</tr>
<tr>
<td>Set</td>
<td>None</td>
</tr>
<tr>
<td>Locator</td>
<td>ChannelB-DIMM0</td>
</tr>
<tr>
<td>Bank Locator</td>
<td>BANK 2</td>
</tr>
<tr>
<td>Type</td>
<td>DDR3</td>
</tr>
<tr>
<td>Type Detail</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Speed</td>
<td>1600 MT/s</td>
</tr>
</tbody>
</table>

**Table 6.2:** Memory device information

The persistent storage of the setup was a 512GB SSHD with an 8 GB SSD cache and a 504GB HDD drive.

### 6.1.2 Software

All the development and evaluation was executed on *Ubuntu 17.10* operating system which was running on the *Linux kernel version 4.14.23*. All tools written in C programming language were compiled with *gcc 7.2.0* and all tools including those used to gather and process data were written with Python 3 and were executed with the *Python 3.6.3 interpreter*.

For the performance analysis we used six different processes. Three of them were started from the same program, but with different parameters. We called this program *the dummy process*. The dummy process is basically the same described in earlier chapters with one addition. Before entering the loop of 180 iteration we would dynamically allocate memory to store an array of integers. The number of integers would depend on the input given to the program as a number of MB. For example, if the program was executed with 200 as an input, it would allocate 200 MB for an integer array. By the end of the execution and after the main loop has finished, we would compute the sum of the integer numbers and print it to the user. This last step is in fact unnecessary, but it prevents the compiler from deleting the first part as dead code during optimizations. The reason for adding the memory allocation part was that we wanted to associate the possible overheads of *error_tolerance* with the memory usage of the running process. Thus, we executed three processes with 200 MB, 400 MB and 800 MB dynamically allocated memory.
The other three processes were executed from three CPU SPEC 2006 benchmarks. To select the benchmarks we had set very specific criteria. Firstly, the CRIU software has some limitations with applications that use resources like sockets or file reading/writing. This happens because if a process is checkpointed and killed for example during a network interaction, it is not guaranteed to be restored because the resource might no longer be available. To increase the chances for success we use CPU intensive, scientific applications that are provided by the CPU SPEC 2006. More than that, we wanted applications with increasing memory usage, to associate it with the results as previously, but we also wanted diverse execution times in contrast with the dummy process where the execution times were very close. Finally, we needed processes with stable memory usage to be sure about the memory consumption of the target for every error injection.

This brought us to the following benchmarks: \textit{458.sjeng}, \textit{470.lbm} and \textit{401.bzip2}. According to [23] \textit{458.sjeng} uses 180 MB of memory, \textit{470.lbm} 417 MB and the \textit{401.bzip2} uses 856 MB of memory. The exact memory usage during our experiments was at 178 MB, 418 MB and 850 MB respectively. The same sources also mark \textit{458.sjeng} and \textit{470.lbm} as stable, which means they use the same amount of memory through their execution. On the other hand, \textit{401.bzip2} is not as stable, but it provided almost double the memory of the previous benchmark which was useful for our analysis. You can see the memory usage of \textit{401.bzip2} in Figure 6.1.

![Figure 6.1: 401.bzip2 memory usage.](image-url)
Although it is not stable throughout all the execution, it remains close to 850 MB for a long time, during which the error injection was performed.

### 6.1.3 Methodology

In Figure 6.2, we represented the work flow in order to get the incremental checkpoint, full checkpoint, process filter and restore times for each of the aforementioned processes. This version includes two error thresholds.

**Figure 6.2**: Application execution with two error thresholds.

The first step is to start the process and then update the `config/watch.pids` file by adding the process ID. Then, using the `tui` tool, we would select a virtual address in the stack section of the process and inject 4 corrected errors. The same address will be used for the next two injections. At this point the mcelog, which is configured to raise a trigger every 4 corrected errors, will invoke `error_tolerance`. This tool will receive the affected physical page address and the error count and will start the process filtering. When the tool reaches the virtual address of the affected process that we used for injection, it will match it with the affected physical address and proceed to the error analysis. Since there are 4 corrected errors which are more than 2 (low threshold), but less than 10 (high threshold) the tool will decide to start the incremental checkpoint. From this point and until the incremental checkpoint is finished, the process is frozen and this time will be measured as overhead. The same procedure happens at 8 corrected errors, but instead of creating a complete new incremental checkpoint, the tool will only create...
an incremental checkpoint that includes the changes from the last one. When the last injection happens, the number of corrected errors in the affected page has grown to 12 and the tool will perform a full checkpoint by using the incremental checkpoint as a basis and kill the process. This time is also considered as an overhead.

The first reason we chose 4 and 8 CEs is that both are below the high threshold (10 CEs, which was suggested by mcelog [9]), but both are higher than the low threshold (2 CEs). We also chose 12 CEs because it is higher than the high threshold. All three are also multiples of 4 which is the number of errors we configured mcelog to invoke the trigger. Finally, we wanted to avoid previous CEs of the pages and any intermediate triggers, so we picked the first three.

After the process is killed, there is the "Dead time", which is the time passed from the process death until the user invokes the `error_tolerance -r` to restore it. When this happens, it will take some time for the process to resume from the state it was just before it was killed.

In Figure 6.3, we represent a similar work flow but without incremental checkpoints. This happens because we only use one threshold (10 corrected errors), which implies that the `error_tolerance` does not add overhead for the first two injections.
The two setups will help us to compare the overhead of using incremental checkpoints or not, later in this chapter.

6.2 Results

6.2.1 Dummy process analysis

In this section, we are going to present the results of our experiments where each of the metrics is an average of a ten times repetition of the experiment. In Figure 6.4, we see the time breakdown of the error_tolerance execution after the injection of the first 4 CEs to the physical pages of the six processes. The first three processes (d200, d400, d800) were initiated from the dummy program with 200, 400 and 800 MB of memory. For these processes the error_tolerance was configured with two error thresholds. The other three processes (df200, df400, df800) were exactly the same, but the error_tolerance was configured with only one threshold. All the data used to create the next plots is available in Appendix A. Also, all the experiments can be recreated with the tools provided on our publicly available repository [24] where we created a special branch called experiments.

![Figure 6.4: Time breakdown for 6 dummy processes after 4 CE’s.](image-url)
The execution time of the first three processes is divided among the process filter and the incremental checkpoint. Unsurprisingly, the first takes 5% of the time for the d200 process, 32% for the d400 process and 24% for the d800 process. For the d200 the process the filtering part takes a lot less time compared to the other two. This is due to the way we placed the PIDs in our config/watch.pids file and the virtual address we selected for the injection.

The PIDs were placed in the same order the processes appear in the plot. This means that, in order to check if a PA belongs to d400, firstly we checked whether it belonged to d200 and, in order to confirm if it belongs to d800, the other two are checked first. As mentioned earlier, the virtual address of the “attack” was always a part of the stack section of any process. The stack section is one of the first virtual regions that we scan, so if the virtual address belongs to the first PID in the configuration, the process filtering will be very fast and this is what happens in the case of d200.

The process filtering particularly for the d200 takes only 0.0082s. The process filtering for the d400 takes 0.421s which is the time to scan the whole address space of d200 and up to the stack of d400. This time almost doubles for the d800 as expected, since the address space of d400 is two times bigger than d200. Generally, we expect this behavior since the process filtering is totally linear to the number of addresses in the virtual space of a process. For the other three processes the filtering time is the only one to measure, since we do not use incremental checkpoints.

The incremental checkpoints take 0.1791s, 0.909s and 3.825s for the first three respectively. Seeing as the only difference between the processes is their usage of the heap, we can clearly notice the relation between the incremental checkpoints time and the memory footprint. The relation is not linear, as the time from d400 to d800 is a lot more than double (4 times) and this is due to the way the CRIU software works, which was not a part of this work to analyze.

Another useful observation is that for this amount of errors the approach with two thresholds adds an overhead of 0.1791s, 0.909s and 3.825s on the running process, where the second adds none.

In Figure 6.5, is presented a similar plot when adding 4 more CEs to the selected physical pages. The scale of this plot is two orders of magnitude smaller than the previous and this happens for two reasons. Firstly, the process filtering time is very similar for all of the processes (0.007 - 0.009) and it is practically the same as the time it took for d200 in the previous plot (0.0082). In Chapter 5, we explained that in order to improve the performance of the process filtering for repeated physical page addresses, we have added a cache. If a coming PA exists in the cache, then the associated PID is checked.
before any other. So what happens is that all of the physical pages were cached from the previous injection and their PIDs were used first. Since the respective virtual address was located in the stack of any process, the filtering happened very quickly. In fact, in almost the exact same time as in d200 of the previous injection.

Secondly, the incremental checkpoint time is much lower. From 0.1791s we now have 0.024s, from 0.909s we now have 0.028s and from 3.825s we now have 0.064s. This is expected because the dummy process is designed in a way that it does not change any memory during its execution after the first allocation phase. This means that there are almost no changes from the last incremental checkpoint. However, in comparison with the last three processes there is still some overhead.

In Figure 6.6, we see the time breakdown after the last injection. As previously explained, the process filtering time was very similar for all six processes and very low. Because the physical pages for each process has now 12 CEs there is no incremental checkpoint; instead, all of the processes are fully checkpointed and that is the time we see in green color. The full checkpoint times for each processes are: 0.0358s, 0.0528s, 0.0741s, 0.1953s, 0.937s, 4.245s which is again increasing as the memory usage of a process is doubled. The noticeable behavior here is that the first three process are fully
checkpointed very quickly and this happens because they use the existing incremental checkpoint and also, because hardly any changes happened afterwards. The other three take much more time, since this is the first time the processes are checkpointed. These times are very similar to the times of the incremental checkpoints of the first three processes after the first injection.

Figure 6.6: Time breakdown for 6 dummy processes after 12 CE’s.

In Figure 6.7, we see the overhead of the checkpoints on top of the execution time of the six processes. This plot is useful to compare the overhead of the two thresholds versus the one threshold. We consider the overhead to be the sum of the incremental checkpoint times and the full checkpoint time. For processes d200 and df200 the overhead is very small, lower than 1%. For d400 and df400 the overhead takes 1% of the overall time and the d800 and df800 the overhead takes 2%.

The overhead of d200 is slightly larger than the overhead of df200 and the same holds for d400 and df400. However, the d800 overhead is smaller that df800. These observations indicate that the overhead of using two thresholds can be larger than the overhead of using one threshold, but it is not consistent and also, it is very slight. Moreover, it provides the possibility of restoring our process even if it was killed by an uncorrected error before the full checkpoint. Nevertheless, in case the first three processes were making changes in the memory during the execution, the incremental checkpoint time
could have be larger. We will see this behavior later when we analyze three of the CPU SPEC2006 benchmarks.

![Checkpoints overhead for 6 dummy processes after 12 CE’s.](image)

**Figure 6.7:** Checkpoints overhead for 6 dummy processes after 12 CE’s.

Figure 6.8 is the last plot that analyzes the dummy processes and here we see the time it takes to restore each of them. The restore time seems to be linear to the memory used by each process, as it doubles every time we double the memory usage. We can also see that the restore times of the d and df categories are very similar, although the checkpoints files in the first three have more directories to travel. But it seems to have a very small impact. Generally, the times are quiet low for processes with this amount of memory being only half a second maximum.
6.2.2 CPU SPEC2006 benchmarks analysis

In this section, we are going to present similar plots, but for three different processes. All the data used to create these plots can be found in Appendix B. As analyzed earlier, we will be using 458.sjeng, 470.lbm and 401.bzip2. Figure 6.9 presents the process filtering and the incremental checkpoint time for the processes after the injection of 4 CEs. The process filtering time is very low and quite similar, because we were only adding one PID in the config/watch.pids. This means that the time for the process filtering represents the time that it takes to scan all the virtual addresses of one process up to the virtual address that translates to the PA used for the ”attack”. The first incremental checkpoint time is increasing as the memory usage is increased, but again it does not seem to be linear.
In Figure 6.9, we see the same time breakdown after the second injection. Here, we notice a difference compared to the previous analysis. The incremental checkpoint time is not related to the memory usage of the processes. This happens because the incremental checkpoint only backs up the changes in memory since the last incremental checkpoint. From the plot it is very clear that the process \texttt{bzip2-850MB} changes the least amount of memory, while \texttt{458.sjeng} changes slightly more, and the \texttt{470.lbm} changes two times more memory than the other two. Still, all of the three processes create an incremental checkpoint in less than one second, because the kernel can track the memory changes and provide the changed addresses to the CRIU software. Thus, the CRIU software does not have to scan the whole memory to find out the memory changes, something which makes it very fast.

**Figure 6.9:** Time breakdown for 3 spec processes after 4 CE’s.

In Figure 6.10, we see the same time breakdown after the second injection. Here, we notice a difference compared to the previous analysis. The incremental checkpoint time is not related to the memory usage of the processes. This happens because the incremental checkpoint only backs up the changes in memory since the last incremental checkpoint. From the plot it is very clear that the process \texttt{bzip2-850MB} changes the least amount of memory, while \texttt{458.sjeng} changes slightly more, and the \texttt{470.lbm} changes two times more memory than the other two. Still, all of the three processes create an incremental checkpoint in less than one second, because the kernel can track the memory changes and provide the changed addresses to the CRIU software. Thus, the CRIU software does not have to scan the whole memory to find out the memory changes, something which makes it very fast.
The same behavior repeats after the last injection which is seen in Figure 6.11. The full checkpoint takes slightly more time that the incremental checkpoint did in the previous plot.

In Figure 6.12, we can see the overhead of this approach. This is very similar to the overhead of the dummy process. The overhead takes less than 1% for the first process, about 2% for the second process and about 5% for the third process. An interesting insight is that the overhead is not related to the execution time of the process at all, but it solely depends on the memory usages and the memory modification rate. This is why the fastest bzip2 has the biggest overhead with 7.49s and the slowest sjeng has only 1.3s overhead.
Finally, Figure 6.13 shows that the restore time is totally related to the size of the memory used by a process and again it seems to be linear, since it roughly doubles every
time the memory footprint is doubled.

Figure 6.13: Restore time for 3 spec processes.

6.3 Conclusions

This concludes the experimental setup and result analysis, where we analyzed the time performance of the error tolerance for our dummy processes and three of the benchmarks of CPU SPEC2006. We confirmed the correctness of the four tools and showed that the overhead of error tolerance is from 1% to 5%, which is quite low. We have also seen that the overhead is not related to the execution time of a process; instead, it is related to the memory footprint and the memory change rate. Finally, we demonstrated that by using two error thresholds we can provide the restore option, even if the target process was killed before the full checkpoint, but with a slight overhead increase in comparison to one error threshold.
Chapter 7

Conclusions & Future Work

During this work we tried to accomplish four different goals.

1. Create a tool for MCE injection to a running process on Linux. This was accomplished with the development of the `va_pid_to_pa` and `inject_mce_to_process` tools.

2. Add abstraction and functionality, so that even unexperienced users would be able to test their software by injecting MCEs. We accomplished this by developing the `tui`.

3. Increase the MCE tolerance at OS level. To do this, we developed the `error_tolerance` tool which works with mcelog and the CRIU software in order to read physical page errors, analyze them and checkpoint the affected process.

4. Exploit emerging technologies like NVDIMMs, but also make our tools usable on conventional systems. This is accomplished by the way `error_tolerance` manages the persistent storage. In the future and with the deployment of NVDIMMs, our tool could still be used with a simple switch which would provide a simple directory on an NVDIMM. This kind of directory can be created using PMEM libraries.

On top of these, we also executed several experiments to evaluate, understand and measure the performance of our tools. From these experiments, we noticed that all of the tools of our stack work correctly and also perform well, even on a system with a conventional HDD setup. The MCE injection overhead on a target process is linear to the error injection rate of `tui` when executing them on a single thread. The overhead of `error_tolerance` was from 1% to 5% and was independent of the process execution time,
but dependent on the process memory footprint and the memory changes performed by the target process. The process filtering time was linear to the memory footprint of the target process, but never exceeded 1.2s, even when three processes with 200 MB, 400MB and 800MB had to be scanned to reach the ”attacked” virtual address. The restore time was between 0.1s and 0.5s for processes which were using from 200 MB to 850 MB of memory.

All of the tools are available on the main branch of our public repository [24] under the GPL license. The data from our experiments, as well as the extra tools we developed in order to perform them and process the data, can be found on the experiments branch on the same repository [24]. Guidance to compile, install and use the tools can be found in the README files which are included.
Bibliography


References.


Appendices
Appendix A

Dummy process data

In this appendix you can find the data that was generated by error_tolerance for the dummy process and was used to create the plots in the previous chapters. Here are some points that help understand the data:

- Process name e.g ”d200”.
- Metric e.g ”Process filter”. There are five different metrics.
- Number of corrected errors e.g 4, 8, 12.

The same data in JSON format can be found on the experiments branch on our public repository under the /logs/backup directory. This data can be used as input to stats.py in order to generate the plots in Chapter 5. All of the data except for ”Execution time” is an average of ten iterations of the same experiment.

<table>
<thead>
<tr>
<th>Process Filter</th>
<th>Incr. Checkp. 4, 8 CEs</th>
<th>Full Checkp. 12 CEs</th>
<th>Restore</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>d200</td>
<td>0.0082, 0.0083, 0.0088</td>
<td>0.1791, 0.0238</td>
<td>0.0358</td>
<td>0.1195</td>
</tr>
<tr>
<td>d400</td>
<td>0.4207, 0.0078, 0.0076</td>
<td>0.9093, 0.0278</td>
<td>0.0538</td>
<td>0.2306</td>
</tr>
<tr>
<td>d800</td>
<td>1.2028, 0.0088, 0.0082</td>
<td>3.8248, 0.0642</td>
<td>0.0741</td>
<td>0.4503</td>
</tr>
<tr>
<td>df200</td>
<td>0.0076, 0.0085, 0.008</td>
<td>-</td>
<td>0.1953</td>
<td>0.1168</td>
</tr>
<tr>
<td>df400</td>
<td>0.4204, 0.0075, 0.0083</td>
<td>-</td>
<td>0.9371</td>
<td>0.2282</td>
</tr>
<tr>
<td>df800</td>
<td>1.2024, 0.0074, 0.0082</td>
<td>-</td>
<td>4.2451</td>
<td>0.4459</td>
</tr>
</tbody>
</table>

Table A.1: Dummy process times(s)
Appendix B

CPU SPEC2006 processes data

In this appendix, you can find the data that was generated by error_tolerance for the CPU SPEC2006 benchmarks and was used to create the plots in the previous chapters.

<table>
<thead>
<tr>
<th></th>
<th>Incremental Checkp. 4, 8, 12 CEs</th>
<th>Full Checkp. 12 CEs</th>
<th>Restore</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>sjeng</em></td>
<td>0.0351, 0.0386, 0.0424</td>
<td>0.4111, 0.4287</td>
<td>0.46</td>
<td>0.168</td>
</tr>
<tr>
<td><em>lbm</em></td>
<td>0.035, 0.0395, 0.0359</td>
<td>1.06, 0.95</td>
<td>1.48</td>
<td>0.2639</td>
</tr>
<tr>
<td><em>bzip2</em></td>
<td>0.027, 0.029, 0.038</td>
<td>6.776, 0.334</td>
<td>0.3791</td>
<td>0.5101</td>
</tr>
</tbody>
</table>

*Table B.1: CPU SPEC2006 benchmarks times*

The same data in JSON format can be found on the *experiments* branch on our public repository under the */logs/backup* directory. This data can be used as input to *stats.py* in order to generate the plots in Chapter 5.