ABSTRACT

AWAD, AMRO JIHAD. Towards Efficient Integration of Emerging NVM Technologies in Future Systems. (Under the direction of Dr. Yan Solihin.)

As emerging Non-Volatile Memory (NVM) technologies are expected to take place in future systems, many challenges raise up. Some of these challenges are due to inherent NVMs’ characteristics and limitations. Specifically, NVMs have relatively slower writes than DRAM. Furthermore, they have limited number of writes, which can significantly shorten the lifetime of NVM-based memory systems. Moreover, NVMs are non-volatile, thus data remanence attacks are much easier; stealing NVM-based DIMM chips and scanning them can reveal the memory data. Accordingly, using NVMs as main memory is expected to be paired with some kind of encryption.

In this thesis, we focus on studying several ways of integrating NVMs in future systems and propose several ways to overcome their challenges. We propose several techniques and mechanisms that have the potential to improve the performance and lifetime of NVM-based systems. First, we start our investigation from the most challenging use case of using emerging NVMs as main memory. We propose a novel secure memory controller, *Silent Shredder*. Silent Shredder enables zero-writes data shredding, which enables different use cases, starting from inter-process data leak prevention to efficient bulk zeroing. Our evaluation shows that Silent Shredder can efficiently eliminate large percentage of main memory writes and improve performance. Later, we investigate another use case that treats NVMs as a memory extension. Memory extension can be thought of as a second-level memory, where the OS ultimately migrates NVM pages to DRAM on-demand. We study and propose several write-aware management schemes that have the potential to reduce the write traffic to NVMs and hence mitigate the limited write endurance of emerging NVMs.
Furthermore, we investigate further on improving performance via OS-level prefetching. As emerging NVMs are extremely encouraging for replacing Flash-based SSDs as I/O devices, we also model the state-of-art NVM host controller interface, and study the expected impact of using it with very fast NVM technologies. We identified several bottlenecks that could hinder the expected returns of such technologies. Specifically, we found that the current I/O software layer overhead can be very limiting for emerging NVMs.

Furthermore, we discuss the challenges of current techniques for obfuscating the access pattern and how they become exacerbated when used with emerging NVMs. Accordingly, we proposed a lightweight design to obfuscate the access pattern. Our design *ObfusMem* exploits the logic on memory chips to encrypt the address bus, hence cryptographically obfuscating the access pattern. Finally, we discuss the TLB shootdown issue in the context of software-managed heterogeneous memory systems. To avoid the TLB shootdown issue and hence enabling better memory management, we propose Self-Invalidating TLB Entries (SITE), which allows avoiding large percentage of the TLB shootdowns completely.
Towards Efficient Integration of Emerging NVM Technologies in Future Systems

by
Amro Jihad Awad

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2016

APPROVED BY:

Dr. James Tuck
Dr. Huiyang Zhou

Dr. Xipeng Shen
Dr. Yan Solihin
Chair of Advisory Committee
BIOGRAPHY

I was born in a small town in Kansas, USA. Later on, my family moved to Germany where I spent good duration from my childhood. I did my undergraduate study at Jordan University of Science and Technology in Jordan, where I graduated with the top rank of my class. I joined NC State University in Spring 2012, working on interesting aspects of system-level optimizations and computer architecture. My research work appeared on top venues: HPCA, ISPASS, ASPLOS and ICS. I have more than five pending U.S patent applications.

After graduation, I plan to work on architecting future Exascale systems for the U.S Department of Energy (DoE) at Sandia National Laboratory.
ACKNOWLEDGEMENTS

I am thankful to all the great people helped me during my PhD. I would like to thank my advisor, Dr. Yan Solihin, for his help and guidance over the last couple of years. His hardworking inspired me the most. I am thankful for all your great help, brainstorming, insightful and inspiring discussions. I would also like to extend my thanks for my committee members, Dr. Huiyang Zhou, Dr. James Tuck and Dr. Xipeng Shen for their insightful feedback. I also would like to thank Dr. Gregory Byrd for his feedback about organizing the thesis and insightful discussions.

During my PhD work, I had the chance to intern under the mentorship of great people in Los Alamos Nationl Lab, HP Labs and AMD Research. I would like to thank all of my mentors and collaborators including Brett Kettering, Pratyusa Manadhata, Stuart Haber, William Horne, Sergey Blagodurov, Arkaprava Basu and Gabriel Loh.

I would like to thank my parents, brothers and sister for their support. I would like to also extend my thanks for my relatives in Detroit. I want to also thank Ahmad Samih and Rami El-Sheikh for their friendship and guidance. I also thank Devesh Tiwari for his insightful discussions and suggestions. I also extend my thanks to my friends James Ecker and Shi Ecker, my current group members and lab-mates, Hussein El-Nawawy, Seunghee Shin, Yipeng Wang, Joonmoo Huh, Abdullah El-Mughraby, Bagus Wibowo and Abhinav Agrawal. Thanks for my friends in NC State, Chao Li, Saurabh Gupta, Shahin Aghdam, Kostas Christidis, Xin Xu, Jose Manuel Valero Sarmiento, Rita Brugarolas, Mahdi Inaya, Elliot Forbes, Rangeen Roy Chowdhury, Piyali Roy Chowdhury, Sahil Sabhrawal and Jaewook Kawk. I appreciate the time we spent together.
# TABLE OF CONTENTS

## LIST OF TABLES
- viii

## LIST OF FIGURES
- ix

### Chapter 1 Introduction and Background 1
- 1.1 Introduction 2
  - 1.1.1 Utilizing Emerging NVMs as Main Memory 3
  - 1.1.2 Utilizing NVMs as Memory Extension 5
  - 1.1.3 Utilizing NVMs as I/O Devices 6
  - 1.1.4 Utilizing NVMs as Part of the Main Memory 7
  - 1.1.5 Low-Cost and NVM-Friendly Access Pattern Obfuscation 8
- 1.2 Organization of the thesis 8

### Chapter 2 Reducing Writes on Secure Non-Volatile Main Memories 10
- 2.1 Introduction 10
- 2.2 Background
  - 2.2.1 Non-Volatile Memories (NVMs) 15
  - 2.2.2 Memory Encryption 16
  - 2.2.3 Data Shredding 18
- 2.3 Motivation 20
- 2.4 Silent Shredder Design
  - 2.4.1 Attack Model 24
  - 2.4.2 Skipping the Zeroing Writes 25
  - 2.4.3 Silent Shredder's Design 29
- 2.5 Evaluation Methodology 33
- 2.6 Evaluation
  - 2.6.1 Write Reduction 36
  - 2.6.2 Read Traffic Reduction 37
  - 2.6.3 Overall Performance Improvement 38
  - 2.6.4 Counter Cache Size 39
- 2.7 Discussion
  - 2.7.1 Security Concerns 41
  - 2.7.2 Other Use Cases for Silent Shredder 42
- 2.8 Related Work 44
- 2.9 Conclusion 47

### Chapter 3 Utilizing Emerging Non-Volatile Memories as Main Memory Extensions 48
- 3.1 Introduction 49
- 3.2 Memory Extensions 49
3.3 Related Work ................................................................. 51
3.4 Methodology ................................................................. 53
  3.4.1 Emulation Infrastructure ........................................... 53
  3.4.2 System Configuration ............................................... 55
  3.4.3 NVM Device Model ............................................... 55
  3.4.4 Applications .......................................................... 56
3.5 Evaluation and Analysis .................................................. 56
  3.5.1 Write-aware Memory Management ............................... 57
  3.5.2 The Impact of DRAM to Memory Extension Ratio on Performance 66
  3.5.3 OS-level Page Prefetching ......................................... 68
  3.5.4 Page Replacement Policy ......................................... 72
3.6 Conclusion ................................................................. 74

Chapter 4 Utilizing Emerging Non-Volatile Memories as I/O Devices ........... 75
  4.1 Introduction ............................................................. 75
  4.2 Prior Work ............................................................... 76
  4.3 Non-Volatile Memory Express (NVMe) ............................. 77
    4.3.1 NVMe Memory Structures .................................... 77
    4.3.2 NVMe Controller ............................................... 78
    4.3.3 System Overview ............................................. 81
    4.3.4 NVMe Operation Scenario .................................... 83
  4.4 Evaluation Methodology ............................................... 84
    4.4.1 Simulator ........................................................ 84
    4.4.2 NVMe Controller Model ....................................... 85
  4.5 Study and Analysis .................................................... 86
    4.5.1 Scalability and Queuing Model ............................... 87
    4.5.2 Technology Impact on Overall Performance .................. 92
    4.5.3 Periodic Interrupts .......................................... 93
    4.5.4 Polling vs Interrupt .......................................... 95
    4.5.5 Protocol data traffic overhead ................................ 97
  4.6 Discussion ............................................................. 98
  4.7 Conclusion ............................................................. 99

Chapter 5 Low-Overhead Access Obfuscation for Trusted Memories .......... 101
  5.1 Introduction ............................................................. 101
  5.2 Background and Assumptions ....................................... 105
    5.2.1 Threat Model ................................................ 105
    5.2.2 Logic in Memory Modules .................................... 106
    5.2.3 Access Pattern Obfuscation .................................. 107
    5.2.4 Memory Encryption .......................................... 110
  5.3 ObfusMem Design ...................................................... 111
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1</td>
<td>Comparison between emerging NVMs, DRAM and Nand-Flash.</td>
<td>3</td>
</tr>
<tr>
<td>Table 2.1</td>
<td>Configurations of the baseline system.</td>
<td>35</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Comparison between Silent Shredder and other initialization techniques.</td>
<td>44</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Summary of the proxy applications we use.</td>
<td>54</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Configurations for the baseline system.</td>
<td>86</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Characteristics of the evaluated benchmarks.</td>
<td>124</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Configuration of the simulated system.</td>
<td>125</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Execution time overhead comparison of ORAM vs. ObfusMem.</td>
<td>127</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Comparing ORAM and ObfusMem.</td>
<td>132</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>Configuration of the simulated system.</td>
<td>158</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Applications</td>
<td>160</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>NVM as main memory.</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Example of duplicate data shredding.</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>State of the art counter mode encryption. AES is shown for encryption/decryption but other cryptographic algorithms are possible.</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>A code sample for initializing allocated memory.</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>The impact of kernel zeroing on performance.</td>
<td>22</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Impact of zeroing on the overall number of writes to the memory.</td>
<td>23</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Silent Shredder's shredding mechanism.</td>
<td>31</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Silent Shredder's mechanism for servicing an LLC miss.</td>
<td>32</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Savings on writes to the main memory.</td>
<td>36</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Percentage of the saved main memory read traffic.</td>
<td>38</td>
</tr>
<tr>
<td>Figure 2.10</td>
<td>The speed up of the main memory reads.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 2.11</td>
<td>Overall IPC improvement.</td>
<td>40</td>
</tr>
<tr>
<td>Figure 2.12</td>
<td>The impact of cache size on miss rate.</td>
<td>40</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Example of handling accesses to the memory extensions.</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>The expected lifetime of memory extensions (log scale).</td>
<td>50</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Handling page eviction in inclusive memory extensions.</td>
<td>58</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>The impact of inclusion property on number of writes.</td>
<td>59</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>The Most Frequent Word Reduction (MFWR) technique.</td>
<td>60</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>The impact of MFWR technique on number of writes.</td>
<td>61</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>The impact of DCW technique on number of writes.</td>
<td>63</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Summary of the impact of write reduction schemes on number of writes.</td>
<td>64</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>The impact of write reduction schemes on lifetime.</td>
<td>64</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>The Impact of DRAM:NVM Ratio on Performance of using PCM vs. Flash.</td>
<td>67</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Summary of DRAM:NVM Ratio Impact on Performance.</td>
<td>68</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Synchronous vs. Asynchronous Prefetching.</td>
<td>69</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>The Impact of OS-Level Prefetching.</td>
<td>70</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>The Impact of Replacement Policy.</td>
<td>72</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>NVMe Queue Memory Structure.</td>
<td>78</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>NVMe I/O command stages.</td>
<td>79</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>NVMe controller's Outstanding Requests Tracker (ORT).</td>
<td>79</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>NVMe Physical Region Descriptor Table (PRDT).</td>
<td>80</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Overview of a system that uses an NVMe controller.</td>
<td>82</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>I/O operation scenario using the NVMe protocol.</td>
<td>83</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.7</td>
<td>Different IO submission/completion queuing models.</td>
<td>87</td>
</tr>
<tr>
<td>4.8</td>
<td>Queuing model impact on overall execution time.</td>
<td>88</td>
</tr>
<tr>
<td>4.9</td>
<td>The number of row buffer misses for different queuing models.</td>
<td>90</td>
</tr>
<tr>
<td>4.10</td>
<td>Command submission time for various queuing models.</td>
<td>91</td>
</tr>
<tr>
<td>4.11</td>
<td>Command completion time for various queuing models.</td>
<td>91</td>
</tr>
<tr>
<td>4.12</td>
<td>NVM latency impact on performance.</td>
<td>93</td>
</tr>
<tr>
<td>4.13</td>
<td>Using periodic interrupt with different period $\tau$ lengths.</td>
<td>94</td>
</tr>
<tr>
<td>4.14</td>
<td>Periodic interrupt impact on performance.</td>
<td>95</td>
</tr>
<tr>
<td>4.15</td>
<td>Polling impact on performance.</td>
<td>95</td>
</tr>
<tr>
<td>4.16</td>
<td>Polling impact on traffic.</td>
<td>96</td>
</tr>
<tr>
<td>4.17</td>
<td>The traffic overhead of the NVMe protocol.</td>
<td>97</td>
</tr>
<tr>
<td>5.1</td>
<td>An example implementation of PCM-based DIMM [109].</td>
<td>106</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of counter mode memory encryption [8, 153].</td>
<td>110</td>
</tr>
<tr>
<td>5.3</td>
<td>Part (a): ObfusMem access pattern obfuscation for a write request (a write back of the LLC block). A read request due to read/write miss of the LLC is handled similarly in the reverse direction (not shown). Part (b): Illustration of dummy request generation.</td>
<td>116</td>
</tr>
<tr>
<td>5.4</td>
<td>The execution time overhead of ObfusMem, normalized to unprotected system.</td>
<td>128</td>
</tr>
<tr>
<td>5.5</td>
<td>The impact of the number of channels on ObfusMem performance, compared to unprotected system with equal number of channels.</td>
<td>129</td>
</tr>
<tr>
<td>6.1</td>
<td>A scenario demonstrating the process of TLB Shootdown.</td>
<td>141</td>
</tr>
<tr>
<td>6.2</td>
<td>The impact of the migration threshold on fast memory miss rate and number of migrations.</td>
<td>143</td>
</tr>
<tr>
<td>6.3</td>
<td>A scenario demonstrating the Self-Invalidating TLB system.</td>
<td>146</td>
</tr>
<tr>
<td>6.4</td>
<td>Number of TLB shootdown normalized to the baseline (no self invalidation) with varying lease length.</td>
<td>149</td>
</tr>
<tr>
<td>6.5</td>
<td>Impact of the lease length on execution time, normalized to non self-invalidating TLB</td>
<td>150</td>
</tr>
<tr>
<td>6.6</td>
<td>Desirable lease length for a given PTE.</td>
<td>151</td>
</tr>
<tr>
<td>6.7</td>
<td>Algorithms for dynamic lease length assignment.</td>
<td>153</td>
</tr>
<tr>
<td>6.8</td>
<td>Lookup of Expiration Time Table (ETT) as an extension to x86-64 page table. Entry for ETT is outlined in dotted lines. In x86-64, CR3 keeps the pointer to the root of the page table.</td>
<td>154</td>
</tr>
<tr>
<td>6.9</td>
<td>The impact of self-invalidating TLB entries (SITE) on performance of a system with heterogeneous memory (Page migration threshold = 10).</td>
<td>161</td>
</tr>
<tr>
<td>6.10</td>
<td>Normalized TLB shootdown count with varying lease assignment policy of SITE</td>
<td>162</td>
</tr>
<tr>
<td>6.11</td>
<td>Execution-time Breakdown for XSBench.</td>
<td>163</td>
</tr>
</tbody>
</table>
Figure 6.12  The impact of migration threshold on SITE’s performance. . . . . . . . 164
Figure 6.13  The impact of TLB shootdown latency on SITE’s performance. . . . . 165
Figure 6.14  Impact of page table walk latency on SITE’s performance. . . . . . . . 166
Figure 6.15  Impact of the latency ratio of slow to fast memory on SITE. . . . . . . . 167
In this chapter, we present a brief introduction about emerging Non-Volatile Memories (NVMs) and discuss their characteristics in Section 1.1. Later on Section 1.1.1, we briefly discuss the challenges when deploying emerging NVMs as main memories. In Section 1.1.2, we discuss the challenges that can raise up when using emerging NVMs as memory extension (second-level memories). Later, in Section 1.1.3 briefly discuss the limitations of integrating emerging NVM as pure I/O devices. In Section 1.1.4, we discuss the use of NVMs as part of the main memory that is directly accessible by the applications. Finally, in
Section 1.1.5, we discuss the challenges of current access pattern obfuscation techniques and how they are exacerbated when used with emerging NVMs.

1.1 Introduction

While the memory requirements for applications are continuously increasing, scaling down DRAM devices is becoming more difficult for several reasons. First, DRAM scalability is limited by the difficulty of fabricating small cylindrical cells for the capacitors. Second, future DRAM systems are expected to suffer from very high error rate, and hence an increase cost of error detection and correction logic. Finally, DRAM is volatile and needs refreshing cells frequently to overcome the capacitors charge leak. However, the refresh power has been shown to contribute to up to 40-50% of the overall power of the memory system [84, 99].

Due to DRAM scalability limitations, the research community has already started looking for other alternatives, such emerging Non-Volatile Memories (NVMs). Emerging NVMs are very promising candidates for replacing DRAM for several reasons: very fast, non-volatile and have promising scalability potentials. Furthermore, such emerging NVMs are orders of magnitudes faster than current technologies used in storage systems, e.g., NAND-Based Flash Solid-State Drives (SSDs).

Several NVM technologies are currently being explored and investigated. However, the most promising ones are Phase-Change Memory (PCM), Spin-Transfer Torque (STT) and Memristor. These technologies differ in read/write latencies, write endurance and density. Table 2.2 presents the expected characteristics for such technologies compared to DRAM and NAND-based Flash. The information were collected from [22, 35, 72, 78, 110, 152].
Table 1.1: Comparison between emerging NVMs, DRAM and Nand-Flash.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Read Latency</th>
<th>Write Latency</th>
<th>Write Endurance</th>
<th>Cell Size</th>
<th>Non-Volatility</th>
<th>Static Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>10-60 ns</td>
<td>10-60 ns</td>
<td>$&gt;10^{15}$</td>
<td>6-8 $F^2$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Nand-Flash</td>
<td>20-100 us</td>
<td>200 us</td>
<td>$10^4$</td>
<td>4-5 $F^2$</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PCM</td>
<td>50-80 ns</td>
<td>150 ns</td>
<td>$10^{-10^{-8}}$</td>
<td>6-16 $F^2$</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>STT</td>
<td>&lt;10 ns</td>
<td>12.5 ns</td>
<td>$&gt;10^{15}$</td>
<td>37 $F^2$</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Memristor</td>
<td>&lt;10 ns</td>
<td>200 ns</td>
<td>$10^{10}$</td>
<td>$&gt;5 F^2$</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

From the table, we can observe that PCM can be looked at as the most promising replacement for DRAM; denser with almost similar read latency. We can also observe that PCM is orders of magnitude faster than NAND-Flash and hence a good candidate for replacing NAND flash.

One important aspect to note about emerging NVMs is their relatively limited write endurance. As example, PCM can wear out tens of millions of times faster than DRAM. Furthermore, the writes to NVM devices are relatively slow and power consuming. Accordingly, the lifetime of NVMs is expected to affect directly the expected lifetime of the whole memory system.

1.1.1 Utilizing Emerging NVMs as Main Memory

Since emerging NVMs, such as PCM and Memristor, have promising densities, they are expected to replace DRAM as main memory [27, 32, 110, 139, 159], as shown in Figure 1.1. Many advantages come from using NVMs as main memory. First, the power savings result from not frequently refreshing the memory. Second, to exploit the low latency of such NVM
devices, it is better to have it directly accessible from the cache hierarchy rather than going through expensive software layers. One disadvantage of this scheme is the real need for wear-leveling and write reduction. Fortunately, several techniques and proposals have been proposed to tackle such problems [28, 110, 159]. However, the other disadvantage of this scheme is the security and data remenance. Unlike DRAM, NVMs keep data for long time after powering off the system, and hence any physical access to the machine can reveal processes’ data, e.g., a repairman can simply obtain a NVM chip, and later it can be scanned to reveal data. Therefore, using NVMs as main memory has been paired with encryption [27, 159].

Most of the work targeted the NVM memory writes problem can be divided into to main categories. First, **Wear-Leveling**, which is the optimization that tries to distribute the memory writes over the memory in a uniform way, so it avoids having parts of memory wear out much faster than expected [110]. Second, **Write Reduction**, which contains the optimizations that try to reduce the number of cells written per write operation. The former is based on the fact that writes are typically non-uniformly distributes and hence many memory locations can wear out way faster than expected. However, the later category aims
to find which cells didn’t have their values changed since the last write and hence avoid writing them again [28].

Unfortunately, as observed by Young et al. [159], the encryption can significantly reduce the effectiveness of write reduction schemes. Accordingly, reducing the number of writes in encrypted NVM systems is of paramount importance. Later on this thesis, we present a novel approach the takes advantage of encryption to reduce one of the most expensive system-level operations and hence reduce the number of writes.

1.1.2 Utilizing NVMs as Memory Extension

Using emerging NVMs as simple as direct replacement for DRAM in the main memory systems can be challenging for several reasons. First, the long write latency of NVM devices can degrade the whole system performance, even for the applications that don’t require large memory capacities. Second, the lifetime of the system can be intolerably shortened; NVMs can wear out millions of times faster than DRAM [81, 110, 159]. One possible way of integrating emerging NVMs would be as fast memory extensions. Memory extensions can be thought of as a second-level memory, where the OS ultimately chooses which pages are held in DRAM and which are moved to the memory extension. Unlike traditional swap systems, memory extensions bypass the filesystem and are dedicated for extending the capacity of the main memory system. Using emerging NVMs as memory extensions is attractive for several reasons. First, it is simple to integrate efficiently in current systems; no modifications are required at the hardware-level. Second, memory extensions are managed by the OS and hence their usage is transparent to applications. Finally, unlike swap systems, no interference with filesystem is required. Later in this thesis, we aim to answer the fol-
lowing questions. How can memory extensions affect the performance? How to maximize their performance? Most importantly, how fast can they wear out, and can we improve their lifetime?

### 1.1.3 Utilizing NVMs as I/O Devices

Emerging NVMs are very promising candidates for replacing NAND-Flash Solid-State Drives (SSDs) and Hard Disk Drives (HDDs) for many reasons. First, their read/write latencies are orders of magnitude faster. Second, some emerging NVMs, such as memristors, are expected to have very high densities, which allow deploying a much higher capacity without requiring increased physical space.

While the percentage of time taken for data movement over low-speed buses, such as PCI, is negligible for the overall read/write latency in HDDs, it could be dominant for emerging fast NVMs. Therefore, the trend has moved toward using very fast interconnect technologies, such as PCIe [105] which is hundreds of times faster than the traditional PCI [90]. Accordingly, new host controller interfaces are used to communicate with I/O devices, to exploit the parallelism and low-latency features of emerging NVMs through high-speed interconnects [2]. To understand the impact of using NVMs as I/O devices, we need to investigate the system bottlenecks and the communication protocol overhead.

Studying the performance of NVMs and the impact of host controller interface on the overall system performance is very important for two reasons. First, it gives the system architects a chance to foresee and estimate the performance gain of future technologies on system performance. Second, it highlights the system bottlenecks and protocol overheads, and allows the system architects and designers an opportunity to develop solutions that
overcome them.

Most of the recent studies focus on the implementation of the software layer of host controller interfaces [17]. They highlight a specific implementation changes and rarely discuss the bottlenecks in the architecture and hardware level.

Later in this thesis, we study the impact of using fast NVMs through high speed interconnects on the overall system performance. We discuss different possible implementations for the host controller interface and compare them from a system-level perspective. We also show how different implementations can impact performance when used with different latencies of NVM devices. Furthermore, we expose the host controller interface overhead in terms of the size of data transferred to and from the controller. The main purpose of this part of the thesis is to give insights for system architects and the NVMs' host controller interface developers about what to consider when designing and optimizing future systems and interfaces.

1.1.4 Utilizing NVMs as Part of the Main Memory

Another use-case for NVMs is to consider them as directly accessible part of the main memory by applications. This use case is very important as it increases the amount of memory directly accessible by the application without involving expensive page faults compared to memory extensions. However, the price of such an approach is a high overhead online profiling for pages behavior to dynamically migrate them across different memory technologies. One main challenge for such an approach is the costly migrations that are dominated by a process called TLB Shootdown. Later on this thesis, we propose a new design that can reduce the number of shootdowns significantly and hence enabling better
and more flexible management of heterogeneous memory systems.

\subsection{1.1.5 Low-Cost and NVM-Friendly Access Pattern Obfuscation}

While encrypting data is expected with the adoption of NVMs, hiding the access patterns is also required. Hiding the access pattern has recently received attention from the computer architecture community. For instance, several proposals have focused on obfuscating the access pattern by using random shuffling of memory locations, namely \textbf{Oblivious RAM (ORAM)} and its hardware realization Path ORAM[136]. Unfortunately, many limitations hinder the deployment of ORAM on real systems for several reasons. First, ORAM has non-zero failure probability. Second, the amount of wasted memory to keep failure probability reasonable can be up to 50\% of the overall memory. Third, reshuffling the data on each access, by reading and then writing a large number of blocks, e.g., 25 buckets (100 cache blocks), can affect both of the performance and the lifetime of emerging NVMs.

Later in this thesis, we discuss and propose new mechanism for hiding the access pattern at extremely cheap cost. Our proposed solution is inspired by the current trend of designing memory modules for future technologies.

\subsection{1.2 Organization of the thesis}

The rest of the thesis is organized as following. In Chapter 2, we discuss \textit{Silent Shredder}, our NVM main memory controller, which re-purposes the cryptographic initialization vectors (IVs) to eliminate expensive writes due to data shredding. Later, on Chapter 3, we investigate and propose several write-aware management schemes for NVMs as memory extensions.
We also propose several ways to improve performance. In Chapter 4, we model and analyze the impact of directly integrating emerging NVMs as I/O devices. We model state-of-art non-volatile memory host controller interface, NVM express, and study how using emerging NVMs can impact I/O performance. Later, in Chapter 5, we discuss our solutions towards practical and write-aware access pattern obfuscation. In Chapter 6, we discuss our solution towards avoiding TLB shootdowns for efficient and flexible management of heterogeneous memory systems, such as a mix of DRAM and emerging NVMs. Finally, we conclude the thesis with a summary of the proposed solutions and insights.
2.1 Introduction

As mentioned earlier in Chapter 1, there are still serious challenges in using NVMs as the main memory. Writing to NVM cells is often slow, requires large power consumption, and has limited write endurance, e.g., 10-100 million writes in Phase Change Memory [54,
81, 110, 156, 163]. Another serious challenge is the data remanence vulnerability, where NVMs retain data for a long time after a system is powered off whereas DRAM loses data quickly. Obtaining an NVM chip and scanning it can reveal data in memory. Memory encryption has been proposed to ward off data remanence attacks [27, 159]. While effective in dealing with the vulnerability, memory encryption worsens the write endurance problem. A good encryption scheme must have the diffusion property; the change in one bit in original data should change many bits in the encrypted data [135]. However, as pointed out by Young et al. [159], techniques to reduce the number of writes in NVMs, such as Data Comparison Write (DCW) and Flip-N-Write (FNW), lose effectiveness due to diffusion because these techniques are inspired by the observation that few bits will have their values changed after successive cache line writes.

Therefore, with memory encryption, reducing the number of writes is of paramount importance. The goal of this work is to find opportunities to reduce the number of writes to an encrypted NVM used as main memory. Specifically, we focus on the problem of data shredding, which is one of the most frequent operating system (OS) operations [24]. Data shredding is the strategy to initialize to zero each physical memory page before mapping it to a process [20, 119, 129]. The OS performs shredding to avoid inter-process data leak, where a process accidentally or intentionally reads the old data of another process. Also, hypervisors perform shredding to avoid inter-virtual machine (VM) data leak in virtualized systems. Our experiments show that data shredding can contribute to a large portion of the overall number of main memory writes, significantly costing processor cycles and memory bandwidth. Furthermore, up to 40% of the page fault time can be spent in page zeroing [144].
The frequency of shredding will further increase where there are many running processes or many virtual machines sharing a system, which is becoming increasingly more common in servers, due to virtualization and server consolidation. Data shredding can also occur multiple times for a single memory page. For example, as shown in Figure 2.1, hypervisors manage memory allocations across virtual machines in virtualized systems. As shown in Step 1, a VM requests host physical memory pages. However, the hypervisor needs to zero out the data to prevent inter-VM data leak [6, 142], as shown in Step 2. Furthermore, when a process requests memory pages, the Kernel/OS inside the VM zeroes out physical pages before mapping them to a new process to prevent inter-process data leak, as shown in Steps 3 and 4, respectively. The impact of data shredding necessitates that we rethink how data shredding should be implemented.
Virtual machines and local nodes' kernels may prefer to request memory allocations with large granularity, for several reasons. The hypervisor or resource manager does not need to go through a large number of extra page table walks beyond the page table walks that occur at the virtual machine or local node level, hence reducing the cost of address translation misses. Another reason is to reduce the intervention between virtual machines or local nodes and resource managers, hence a better scalability. Accordingly, for instance, assume a system with hundreds of terabytes of memory and a large number of nodes, memory allocations can be in order of gigabytes or terabytes. Zeroing out such large amount of memory would be very slow. Furthermore, there is a chance that the virtual machine or the local node never writes the whole space, but shredding the whole assigned physical space is still required.

Previous work targeted speeding up zeroing pages in the context of DRAM main memory [66, 124, 158]. However, since writes to DRAM are cheap, these studies proposed speeding up the zeroing operation, but not eliminating it. For example, Jiang et al. [66] suggest offloading the zeroing to a DMA engine, while Sehsadri et al. [124] shift the zeroing to within the DRAM. However, in both studies, the writes to the main memory still occur, and hence these methods are not suitable for use in NVM-based main memory.

In this chapter, we present a secure NVMM controller, Silent Shredder, that completely eliminates writes to NVM due to data shredding. Silent Shredder eliminates shredding-related writes to NVM-based main memory by exploiting initialization vectors (IVs) used in symmetric encryption. We prevent data leakage and eliminate shredding-related writes at the same time with minimal cost. One key insight for avoiding shredding-related writes is that to protect data privacy of a process, it is sufficient for the OS to make a data page unin-
telligible when the page is allocated to another process. By making the page unintelligible rather than zeroing it, the old data of a process is protected from being read by another process. The second insight is that when data in a newly allocated page is read, for software compatibility, the controller should skip the actual reading from NVM and instead should supply a zero-filled block to the cache. Silent Shredder achieves these goals by repurposing the IVs: it manipulates IVs to render data pages unintelligible and to encode the shredded state of a page. Note that Silent Shredder works with all modes of symmetric encryption that use IVs, including the counter mode used in recent secure NVM proposals. By avoiding shredding-related writes to NVM and the subsequent reads from NVM, Silent Shredder improves NVM’s write endurance, increases read performance, and reduces power consumption. All these are achieved using low-cost modifications to a standard counter-mode encrypted memory system.

To evaluate our design, we use gem5, a detailed full-system simulator, to run 3 graph analytics applications from the PowerGraph framework and 26 multi-programmed workloads from the SPEC 2006 benchmark suite. Silent Shredder eliminates about half (48.6%, on average) of the writes in the initialization and graph construction phases. Furthermore, it speeds up the memory reads by an average of 3.3 times and improves the instructions per cycle (IPC) by an average of 6.4%.

The rest of the chapter is organized as follows. We briefly describe NVM technologies, data shredding, and encrypted NVMs in Section 5.2. In Section 6.2.3, we present a motivational example. In Section 5.3, we introduce several design options for eliminating data shredding and discuss their advantages and disadvantages. Later, we motivate our choice of Silent Shredder as the preferred design and then show that our design can be implemented
at zero cost when using secure NVMM controllers. We introduce our evaluation methodology in Section 5.4 and present evaluation results and a parameter sensitivity study in Section 6.5. In Section 5.6, we discuss several use cases, security concerns, and remediation for Silent Shredder. We discuss related work in Section 5.7 and conclude with a summary in Section 5.8.

2.2 Background

In this section, we briefly describe emerging NVM technologies, memory encryption, and data shredding in OSes.

2.2.1 Non-Volatile Memories (NVMs)

Phase-Change Memory (PCM), Spin-Transfer Torque (STT-RAM) and Memristor are among emerging non-volatile memory technologies, some of which are considered as candidates for replacing DRAM for use as the main memory. Their read latencies and densities are either competitive or comparable with DRAM, and they may scale better than DRAM. However, NVMs suffer from slow and power consuming writes, and generally have limited write endurance, e.g., 10-100 million writes with Phase Change Memory [110]).

When used as main memory, NVMs may provide persistent memory, where regular store instructions can be used to make persistent changes to data structures in order to keep them safe from crashes or failures. Persistent memory enables persistent memory allocations [14, 85, 96, 148], and may allow future systems to fuse storage and main memory [32, 98, 139].
When an application or a VM requests and uses a persistent page, the OS should guarantee that its page mapping information is kept persistent, so the process or the VM can remap the page across machine reboots [96, 148]. There has been research on explicit software programming frameworks for exploiting persistent memory, e.g., ATLAS [26].

However, NVMM suffers from a serious security vulnerability: it retains data long after a system is powered off. Obtaining physical access to NVMM (through theft, repair, or improper disposal) allows attackers to read the plaintext of data [27, 159]. Accordingly, NVMM should be paired with at least some form of memory encryption.

### 2.2.2 Memory Encryption

There are two general approaches for encrypting NVMM. One approach assumes the processor is not modified, and the NVMM controller encrypts the main memory content transparently to the processor [27]. To minimize performance overheads, cold data, i.e., infrequently used data, stays encrypted, but hot data is proactively decrypted and stored in plaintext in memory. Another approach assumes that the processor can be modified and the processor chip is the secure base. Any data sent off chip in the main memory is encrypted. There are many example systems using the latter approach, including some recent studies [117, 153, 159].

There are several encryption modes that can be used to encrypt the main memory. One mode is *direct encryption* (also known as electronic code book or ECB mode), where an encryption algorithm such as AES is used to encrypt each cache block as it is written back to memory and decrypt the block when it enters the processor chip again. Direct encryption reduces system performance by adding decryption latency to the last level cache (LLC)
miss latency. Another mode is *counter mode* encryption, where the encryption algorithm is applied to an *initialization vector* (IV) to generate a one-time pad. This is illustrated in Figure 5.2. Data is then encrypted and decrypted via a simple bitwise XOR with the pad. With counter mode, decryption latency is overlapped with LLC miss latency, and only the XOR latency is added to the critical path of LLC miss. In state-of-the-art design, the IV of a counter mode encryption consists of a unique ID of a page (similar to a page address, but is unique across the main memory and swap space in the disk), page offset (to distinguish blocks in a page), a per-block *minor* counter (to distinguish different versions of the data value of a block over time), and a per-page *major* counter (to avoid overflow in counter values).

For the rest of the chapter, we assume counter mode processor-side encryption, similar to earlier papers [117, 153, 159], because counter mode is more secure: Encryption in ECB mode, without the use of counters or IVs is vulnerable to several attacks based on the fact that identical blocks of plaintext will encrypt to identical blocks of ciphertext,
wherever they occur in memory; vulnerabilities include dictionary and replay attacks. Memory-side encryption, e.g. with secure DIMMs, is vulnerable to bus snooping attacks. In addition, processor-side encryption makes it easier to interface with the OS, use per-user and per-application keys, and expose key management to users and applications (where appropriate).

Counters are kept in the main memory, but are also cached on chip with one major counter of a page co-located in a block together with all minor counters of a page. For example, in [153], a 4KB page has a 64-bit major counter that is co-located with 64 7-bit minor counters in a 64-byte block. Any parts of the IV are not secret and only the key is secret. The security of counter mode depends on the pad not being reused, hence at every write back, a block's minor counter is incremented prior to producing a new IV to generate a new pad. When a minor counter overflows, the major counter of the page is incremented and the page is re-encrypted [153]. Finally, counter mode integrity needs to be ensured to avoid attackers from breaking the encryption. While counters are not secret, they must be protected by Merkle Tree to detect any tampering to their values [117].

2.2.3 Data Shredding

Avoiding data leak between processes requires clearing, e.g., zeroing, a memory page of a process before allocating it to another process. The cost of data shredding is high. For example, a recent study showed that up to 40% of the page fault time is spent in page zeroing [144]. Different operating systems adopt different zeroing strategies. For example, in FreeBSD, the free pages are zeroed out early and kept in a memory pool ready for allocation. Once a process needs a new page to be allocated, a new virtual page is mapped to one
of the zeroed physical pages [43]. In Linux, the kernel guarantees that a newly allocated virtual page is initially mapped to a special physical page called the **Zero Page**. The Zero Page is shared by all new allocated pages. The actual allocation of a physical page happens when the process writes to the page for the first write, using a mechanism referred to as copy-on-write (COW). During COW, a physical page (from another process) is zeroed out and the virtual page is remapped to it [20]. Similar mechanisms are deployed by hypervisors to avoid data leak between virtual machines [142].

The zeroing step itself can be implemented with different mechanisms [30]. One mechanism relies on using temporal store instructions, e.g., `movq` in the x86 instruction set, which brings each block into the cache before writing zero to it. The use of temporal store can cause cache pollution, where useful blocks are evicted and cause subsequent cache misses. The impact of cache pollution increases when dealing with large pages, such as 2MB and 1GB size pages. In addition, the mechanism works well if all cache blocks of the page are used shortly after zeroing; however, often this is not the case. Also, without additional mechanisms, zeroing blocks of the page in the cache is not secure as it does not persist the modifications to the NVMM right away. Hence, upon system failures, the zeroing may be lost, and data leak may occur.

Another zeroing mechanism is using non-temporal store instructions, such as `movntq` in x86. Such store instructions bypass the entire cache hierarchy and write directly to the main memory; any blocks that are currently cached are invalidated. Non-temporal stores avoid cache pollution, but they also come with a risk of invalidating soon-to-be accessed or initialized blocks. Some overheads remain unchanged, e.g., the number of store instructions used, and latency and power from using memory bus bandwidth.
There have been proposals to improve zeroing performance. Some studies have proposed offloading the zeroing step to the DRAM controller or the DMA engine, so that the CPU time is not wasted in zeroing pages and cache pollution is avoided [30, 66, 124]. Such techniques are effective in DRAM main memory, but with NVMM, the zeroing still results in high performance overheads due to high power consumption and latency of writes in NVMs, and reduced write endurance. Since our work targets memory controllers for NVMM, we consider techniques that zero out pages in main memory and bypass the caches entirely, but with a significant reduction in the actual zeroing operations.

2.3 Motivation

We start with an example to study the impact of kernel shredding on system performance. Assume a simple code that allocates and initializes $SIZE$ bytes of memory, as shown in the code snippet in Figure 2.3. As explained earlier in Section 5.2, the Linux kernel allocates and

```c
// Allocating SIZE bytes
char ALLOC = (char) malloc(SIZE);
// Setting all allocated memory to 0
// Point 0
memset(ALLOC,0,SIZE);
// Point 1
memset(ALLOC,0,SIZE);
// Point 2
```  

Figure 2.3: A code sample for initializing allocated memory.
zeroes out a physical page right after the first write. Accordingly, the first store instruction in
the first memset causes a page fault, the page fault handler then allocates a new physical
page and zeroes it out, and finally maps it to the application process. As the application's
memset resumes after the page fault, if the region is the size of a page, page zeroing is
redundantly repeated. Overall, the first memset incurs the following delay: kernel page
allocation, kernel zeroing, and program zeroing. In contrast, the second memset only has
the program zeroing delay. Since program zeroing follows a similar mechanism as kernel
zeroing, the second memset's delay is a good proxy of kernel zeroing latency. memset is
more optimized than the kernel zeroing process- it uses non-temporal stores when the size
of the memory to be initialized is bigger than the LLC, and hence avoids cache pollution.
Hence it takes less time than kernel zeroing and in our experiments, memset's time is a
conservative proxy for kernel zeroing time. Figure 2.4 shows the time spent to execute the
first vs. the second memset.

The overall execution time is the time taken by the first memset (which includes page
zeroing, page faults, and program zeroing), and the height of the blue bars is the time
taken by the second memset (which only incurs program zeroing). We observe that on
average, roughly 32% of the first memset time is spent in kernel zeroing. Our observation
is consistent with previous studies that showed that up to 40% of the page fault time is
spent in kernel zeroing [144]. Note that writing latency for NVMs is multiple times slower
than that of DRAM, and hence the page zeroing is expected to become dominant and to
contribute for most of the page fault time.

Now let us examine real applications. Figure 2.5 shows the number of writes to the
main memory for several graph analytics applications from the PowerGraph suite [47]. The
Figure 2.4: The impact of kernel zeroing on performance.
number of writes was collected using performance counters. A system call was added to collect the number of times the kernel calls the `clear_page` function. For all real system experiments, we use Linux kernel 3.16.2 on a 12-core Intel Xeon E5-2620 CPU machine with 16GB main memory. The data sets we use are publicly released data sets from Netflix and Twitter [13, 160].

![The impact of allocation schemes on # Writes](image)

**Figure 2.5:** Impact of zeroing on the overall number of writes to the memory.

For each application, the first bar represents the number of writes when using temporal kernel zeroing (zeroing in caches). The second bar represents the number of writes when using non-temporal kernel zeroing (bypassing the cache hierarchy). Finally, the third column shows the number of writes to the main memory when zeroing is avoided in the main memory. We obtained the third bar by counting the number of writes caused by kernel zeroing in the non-temporal case and deducting it from the overall number of writes. We observe that in big data applications, such as graph analytics, a large percentage of the
overall number of writes to the main memory is caused by kernel zeroing. The main reason behind this is that many data analytics applications exhibit a write-once read-many times behavior, where the graph is rarely modified after the graph construction phase.

In summary, we found that kernel zeroing can contribute very significantly to the execution time and the number of main memory writes, regardless of whether temporal or non-temporal writes are used. While such overheads are costly with DRAM main memory, they are multiple times more costly with NVMM due to the slow and power consuming nature of writes, and the reduction in NVMs' lifetime until write endurance limit is reached.

2.4 Silent Shredder Design

In this section, we explore several design alternatives for data shredding and motivate the design choices in Silent Shredder.

2.4.1 Attack Model

Silent Shredder is designed to protect against attacks carried out by attackers who have physical access to the computers. The attackers can scan a computer's bus or main memory, tamper with main memory content, or play man in the middle between the computer's processor and the main memory. This attack model is the same as the models assumed in prior memory encryption studies [117, 153, 159]. Such attacks are possible due to several reasons such as lax physical security guarding server rooms in cloud computing facilities, improper disposal of computers at the end of their service, and malicious administrators.
or repairmen. Attackers may insert a bus snooper or a memory scanner to read out data communicated off the processor chip. They may also insert devices to overwrite the content of the main memory, or act as a man in the middle by returning false data requested by the processor chip. They may conduct known-plaintext attacks by observing the encrypted value of an expected data item, or employ dictionary-based attacks by guessing original values according to encrypted values’ frequency.

We leave out side-channel attacks because they are beyond the scope of this work and there are many side-channel protection mechanisms in the literature that can be added to Silent Shredder as needed.

Silent Shredder requires the OS to communicate page zeroing to hardware. This requires minimal modifications to the OS’s page fault handler; however, it requires the OS to be trusted, as an untrusted OS can maliciously avoid page zeroing in order to cause data leak between processes. If the OS is not trusted, then processes must run in secure enclaves. Although beyond the scope of this work, we believe that Silent Shredder can be adapted relatively easily to work with secure enclaves. For example, the hardware can notify Silent Shredder directly when a page from an enclave is going to be deallocated. Further discussion about several security concerns when using counter-mode encryption and their remediations can be found in Section 2.7.1.

### 2.4.2 Skipping the Zeroing Writes

The main goal of shredding data is to prevent data leakage between processes or VMs when a page is reused. One key observation we make is that page zeroing is only one way to
prevent data leakage. Writing any unintelligible or random content into the page before the page is reused achieves the same goal, as the process which gets the page will not be able to read any meaningful data from the page. In traditional (non-encrypted) memory, however, there is no advantage to writing a random pattern to a page compared to writing zeros to the page. In fact, writing zeros to a page is more efficient as various instruction sets may provide a special instruction to zero an entire cache block (e.g., `dcbz` in PowerPC).

However, in encrypted memory, the nature of cryptographic encryption can be exploited to initialize a reused page with a random pattern for free. For example, consider a page of data encrypted with the key $K_1$. Before the page is reused, if we change the key to a different one, $K_2$, then subsequent decryption of the page will result in unintelligible data. The nature of encryption ensures that the new decrypted data is not correlated with the initial data, nor is it any easier to break encryption when attackers read data decrypted with a different key. Most importantly, the entire initialization writes to the reused page are completely eliminated, resulting in significant performance improvement, power savings, and write endurance improvement.

The above technique can be achieved by giving every process a unique key, instead of the entire system using one key. However, giving each process a unique encryption key seriously complicates key management. First, the architecture and the OS need to ensure that no key is reused and all keys are securely stored. Second, it complicates data sharing among processes: shared pages need to be encrypted/decrypted using a common key that is known to all sharing processes. Third, from a practical perspective, the memory controller has no easy way to identify the process that owns the physical address of the memory read/write request; hence it is unable to decrypt the data without the correct key.
Addressing this requires the process ID or the process-specific key to be passed along with every memory request.

These challenges make the deployment of such a design costly. Thus, in Silent Shredder, we assume that processes share a single cryptographic key. In order to provide a way to initialize a reused page with a random pattern without using process-specific keys, we exploit the feature of counter-mode encryption where the initialization vector (IV) is what is encrypted to generate a pad. Essentially, if a block is encrypted with an IV, $I V_1$, but decrypted with a different IV, $I V_2$, then even when the two IVs differ by just one bit, the decrypted data is unintelligible. This is because $Data \neq D_k(I V_2) \oplus (E_k(I V_1) \oplus Data)$ where $E_k$ and $D_k$ are encryption and decryption using the key $K$. Therefore, to initialize a reused page with unintelligible random data, it is sufficient to modify the page's IVs. Note that this is true of any encryption mode that uses IVs, though we focus on counter mode in this work.

The next logical question is which parts of the IV should be modified. The page ID and the page offset components in an IV should not be modified as they ensure spatial uniqueness of the IV of a block. This leaves us with the per-page major counter and the per-block minor counter. We now discuss several possible approaches with different trade-offs. One option is to increment all the minor counters in a page. This changes the IVs of all blocks in a page and avoids actual page zeroing. However, this option has drawbacks. One drawback is that since minor counters are typically small, e.g., 7 bits are recommended [153], incrementing minor counters induces a high probability of minor counter overflow, which causes expensive page re-encryption. Essentially, a block in a page can only be written back from the LLC to the NVMM $2^7 = 128$ times before the page needs to be re-encrypted. Page re-encryption involves reading all blocks of a page from memory into the cache,
incrementing the page’s major counter, resetting all minor counters of the page, and writing back all blocks of the page from the cache to the NVMM. Page re-encryption is an expensive operation in DRAM main memory and is significantly more expensive in NVMM. Hence it should be avoided whenever possible.

To avoid increasing the frequency of page re-encryption, we can pursue a second option, where we increment the major counter and leave all minor counters unchanged. Since the major counter of a page is used to encrypt or decrypt all blocks in the page, incrementing the major counter is sufficient to initialize a reused page to a random pattern. Accordingly, we do not increase the frequency of page re-encryption since minor counters are unchanged.

A major drawback with both of these techniques is that they assume that the OS or applications are not implemented with the expectation that a newly-allocated page will have zero values. However, our observation is that this assumption is not valid in modern systems. For example, in Linux, we find that the libc system library’s runtime load (rtld) code contains an error-checking assertion that verifies that pointer variables in a newly-allocated page have the value of zero (NULL). Hence the two techniques give rise to software compatibility challenges.

However, we can pursue another approach when such assertion removal is not possible or not desirable. In this situation, we would still like to avoid zeroing writes but allow Silent Shredder to return a zero value when reading any cache block from a newly allocated page. This requires a third option as follows. When a reused page is initialized, the major counter of the page is incremented, and simultaneously all the minor counters are reset to zero. We reserve the value zero in a block’s minor counter to indicate that when the block is read, zeros are returned to the processor instead of a random pattern. The zero minor
counter is not used during regular page re-encryption; for example, when a minor counter overflows, the minor counter will be reset to 1 instead of zero. Upon an LLC miss, the minor counter value for the block is checked. If it is zero, then the cache block is not fetched from memory. Instead, a zero-filled block is allocated in the cache, and returned to the \texttt{ld} or \texttt{st} instruction that accesses the block. This approach has the side benefit of reducing page re-encryption frequency as it lengthens the time until page re-encryption. For the rest of the chapter, we use the third option to accomplish page shredding, where the major counter is incremented and minor counters are reset.

Note that without the Silent Shredder mechanism, shredding a memory page in conventional NVMM will cause every minor counter on a page to be incremented, \textit{in addition to} writing zeros to the main memory. Thus, Silent Shredder does not increase the number of writes to counters, but it completely avoids page zeroing writes.

### 2.4.3 Silent Shredder's Design

We now discuss Silent Shredder’s overall design. Figure 2.6 illustrates the high-level mechanism. First, when the OS wants to shred a page, $p$, the OS uses a mechanism to give a hint to the hardware, e.g., by writing $p$’s physical address to a memory-mapped register in the Memory Controller (MC). This is shown in the figure as Step 1. The MC then sends an invalidation request to the cache and coherence controller of remote caches and also the local cache (Step 2). The coherence controller sends out invalidation requests to any blocks in the page that may be shared in the caches of other cores. In addition, the block containing $p$’s counters is also invalidated from other cores’ counter caches. This step is
necessary for keeping the caches coherent, even though the blocks will not be actually written. When a remote core has a cache miss, it will be forced to reload updated counters for the page and the missing block in the page.

After the invalidation, the counters can be changed (Step 3) by incrementing the major counter of the page $p$ and resetting all the minor counters to zero. Finally, the counter cache controller acknowledges (Step 4) the completion of updating counters, and the MC signals the completion to the processor. At this time, the zeroing is completed without any blocks in the page ever written to the NVMM.

Note that steps 2, 3, 4 and 5 are correctness requirements for any kind of bulk zeroing that bypasses the cache hierarchy, e.g., non-temporal zeroing and DMA engine support. Most modern integrated memory controllers have the ability to invalidate and flush cache lines; DMA region writes should invalidate any stale data in the cache hierarchy [61, 124]. Furthermore, a simple serializing event, e.g., `sfence`, can be implemented to ensure that all invalidations have been posted and that the counter values have been updated. For example, in Linux, non-temporal zeroing stores are followed by `sfence` to guarantee that all writes have been propagated to main memory. A similar approach can be adopted in Silent Shredder, e.g., following shredding the command with `sfence` or `pcommit`, where the memory controller guarantees that serializing events such as `sfence` or `pcommit` will not be marked completed until all invalidations are posted and all counters are updated.

Figure 2.7 shows how a LLC miss is serviced. Suppose the LLC misses on a block, $x$. The block address of the miss is passed to the counter cache (Step 1), resulting in the minor counter being read out. The minor counter for the block $x$ is then compared to zero. If the minor counter value is not zero, then a request to fetch the block $x$ is passed on to
the memory controller (Step 3a), which then obtains $x$ from the NVMM, decrypts it, and returns it to the LLC (Step 4). On the other hand, if the value of the minor counter is zero, then a zero-filled block is returned to the LLC.

Overall, Silent Shredder's mechanism achieves the following savings compared to traditional page zeroing mechanisms:

1. During page shredding, none of the cache blocks of the page are written.

2. When any of the 64 blocks in a shredded page is read for the first time, the NVMM is not accessed, and instead a zero-filled block is installed in the cache.

Both of these savings improve performance, both during and after shredding. Furthermore, the first item reduces power consumption and improves write endurance of NVMM due to skipping writes.

**Silent Shredder's Extensibility:** Silent Shredder can be implemented with any encryption
Figure 2.7: Silent Shredder’s mechanism for servicing an LLC miss.

mode that uses IVs: CBC, PCBC, CFB, OFB, and CTR. Such encryption modes have the well-known property that encrypted data cannot be decrypted without both the encryption key and the IV that were used to encrypt it. In our scheme, Silent Shredder zeros out the minor counter and increments the major counter in the IV, thereby changing the IV originally used for encryption, making the encrypted data totally irrecoverable.

**Counter Cache Persistency:** Similar to any scheme that relies on IVs, Silent Shredder needs to maintain the counter cache persistency by using either a battery-backed write-back counter cache, or a write-through counter cache. For the rest of the chapter, we assume a battery-backed write-back counter cache. However, even in case of write-through counter cache, the overhead for counter cache is minimal per page zeroing (64B block per 4096B page write). One more approach could also be used instead of battery-backed counter cache, fast but not dense NVM technologies such as STT-RAM.
**Data Recovery:** In our case, recovering data in case of disasters is the same as with secure DIMMs. For security reasons, encryption keys need to be stored outside the DIMM or stored securely in the DIMM, e.g. encrypted with a master recovery key. Similar mechanisms to recover keys apply to both cases. In addition, the IVs must be backed up to NVMM. Once the keys and the IVs are obtained, the data can be recovered. In addition, redundancy and key sharing can be used for recovery.

### 2.5 Evaluation Methodology

We use a modified version of Linux kernel 3.4.91 in our experiments. Specifically, we replace the `clear_page` function used to clear new physical pages with our shredding mechanism. We use a memory-mapped I/O register for writing the physical address of the page to be shredded. We use the default page size of 4KB. Shredding any page larger than 4KB, such as 2MB and 1GB, can be done by sending a shred command for each 4KB of the large page. In Linux, the function for clearing large pages, `clear_huge_page`, already calls the `clear_page` function for each 4KB. Therefore, no more modifications are needed.

We run our modified Linux kernel on gem5, a full system cycle-accurate simulator [16]. We extended gem5's memory controller to include a counter cache and a memory-mapped I/O register. We run 26 SPEC CPU2006 benchmarks with the reference input [134]. We also run 3 benchmarks from PowerGraph benchmark suite: page rank, simple coloring and kcore [47]. All benchmarks were compiled with default settings and optimizations.

Our goal is to study the potential benefits for Silent Shredder under environments with high level of data shredding as a proof of concept. Data shredding is a performance bottleneck during graph construction and initialization phases. Hence we checkpoint the
PowerGraph benchmarks at the beginning of the graph construction phase, and the SPEC benchmarks at the beginning of the initialization phase. In the baseline configuration without Silent Shredder, we assume that when receiving a shred command, shredded cache blocks are invalidated and written back (if dirty) to the main memory. In other words, the baseline shredding uses non-temporal stores. Such an assumption guarantees that similar number of instructions are executed when comparing Silent Shredder with the baseline; similar kernel binary and checkpoints are used. For fair comparison, we assume that for both the baseline and Silent Shredder, invalidation of all cache blocks is sent right after receiving the shredding command.

We warm up the system caches for 1 billion cache accesses and then run the applications until at least one core has executed 500 million instructions (a total of approximately 4 billion instructions for an 8-core system). Since the shredding operation is microarchitecture-independent operation and given the impractical simulation time of detailed microarchitecture CPU model along with full-system simulation and modern kernel, we use the SimpleCPU model of Gem5 to run our modified kernel. For SPEC benchmarks, we run an instance of the benchmark on each core.

The system configuration used in our simulations is shown in Table 6.1. Similar to the expected trend of modern cache hierarchies, our system has a 4-level cache hierarchy [137]. L4 and L3 caches are shared among all cores, while L1 and L2 caches are private for each core. We also assume a battery-backed writeback counter cache.

Each valid cache block in the counter cache maps to a single physical page in the main memory. Each block contains 7-bit minor counters, one for each cache block in that page, and a 64-bit major counter for that page. In Section 2.6.4, we discuss the reason behind our
Table 2.1: Configurations of the baseline system.

<table>
<thead>
<tr>
<th>Processor</th>
<th>8 cores x86-64 processor, 2GHz clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>2 cycles, 64KB size, 8-way, LRU, 64B block size</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>8 cycles, 512KB size, 8-way, LRU, 64B block size</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>25 cycles, 8MB size, 8-way, LRU, 64B block size</td>
</tr>
<tr>
<td>L4 Cache</td>
<td>35 cycles, 64MB size, 8-way, LRU, 64B block size</td>
</tr>
<tr>
<td>Coherency Protocol</td>
<td>MESI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>16 GB</td>
</tr>
<tr>
<td># Channels</td>
<td>2 channels</td>
</tr>
<tr>
<td>Channel bandwidth</td>
<td>12.8 GB/s</td>
</tr>
<tr>
<td>Read Latency</td>
<td>75 ns</td>
</tr>
<tr>
<td>Write Latency</td>
<td>150 ns</td>
</tr>
<tr>
<td>Counter Cache</td>
<td>10 cycles, 4MB size, 8-way, 64B block size</td>
</tr>
</tbody>
</table>

| Operating System | |
| OS              | Gentoo |
| Kernel Version  | 3.4.91 |

choice of the capacity of the counter cache to be 4MB. The latency of the counter cache was obtained using the CACTI 6.0 tool [97].

### 2.6 Evaluation

In this section, we evaluate the potential benefits that come from our design. We start by showing the reduction of the number of main memory writes as a result of using Silent Shredder. Later, we show how our design also reduces read traffic and improves performance as additional benefits.
2.6.1 Write Reduction

The main goal of Silent Shredder is to reduce the number of writes occurring as a result of
the kernel zeroing process for shredding previous data. We find that an average of 48.6% of
the main memory writes in the initialization phase could be eliminated. Figure 2.8 shows
the percentage of writes to the main memory that we could save by using Silent Shredder.

![Write savings](image)

**Figure 2.8:** Savings on writes to the main memory.

From Figure 2.8, we observe that kernel zeroing contributes to large percentage of main
memory writes for some applications. A few applications, e.g., H264, DealII and Hmmer,
have a very small ratios of main memory writes to instructions. Accordingly, we find that
for such applications the majority of main memory writes result from the kernel zeroing
operation. Note that since we simulate the full system, the kernel zeroing can be a result of
the application initialization code itself, loading system libraries, writing metadata or any
other system activities. Since our work targets shredding from the kernel perspective, we
count all kernel shredding writes, regardless of the semantics of the program.

Eliminating a significant number of writes is very important; it reduces the memory traffic, increases performance by eliminating slow and power consuming NVMM writes, and increases the lifetime of the NVMM. Our evaluation focuses on the start up or graph initialization phase which is important because the graphs get rarely modified after initialization, hence significant percentage of the main memory writes occur at this phase. The startup phase is not the only scenario where Silent Shredder is effective. In a system that is highly loaded, data shredding will occur frequently because the high load from multiple workloads are placing a high pressure on the physical memory. For example, most data center servers today are not very good in terms of energy proportionality (idle power is approximately 50% of peak power), and therefore peak energy efficiency is achieved when the data centers are highly loaded resulting in very high processor utilization rates [38]. A highly loaded system will suffer from a high rate of page faults, and page fault latency is critical in this situation.

2.6.2 Read Traffic Reduction

Another important advantage of using Silent Shredder is the ability to recognize shredded cache blocks from their minor counter values. Once a read to a shredded block is detected, a zero-filled block will be returned to the cache hierarchy without the need to read the invalid data from the main memory. We find a surprisingly large percentage of the initialization read traffic can be eliminated.

As shown in Figure 2.9, an average of 50.3% of the read traffic during the initialization phase is due to reading shredded pages. Reducing the read traffic is crucial for systems that
run memory-intensive benchmarks with limited memory bus bandwidth. Identifying that a block is shredded happens quickly: any shredded block read request can be completed once its minor counter value is read. Figure 2.10 shows the speed up ratio of the average memory read latency for each benchmark. On average, Silent Shredder achieves an average memory read speed up ratio of $3.3 \times$ across the benchmarks.

### 2.6.3 Overall Performance Improvement

As mentioned earlier, in addition to eliminating shredding writes, Silent Shredder also speeds up reading shredded cache lines. These two improvements together improve the IPC. Figure 2.11 shows the relative IPC when using Silent Shredder (higher is better), normalized to the baseline.

We observe an average IPC improvement of 6.4% among all benchmarks, with a maximum of 32.1% for Bwaves. The overall IPC improvement depends on the fraction of
instructions that access either main memory or shredded cache blocks.

2.6.4 Counter Cache Size

The IV cache should be fast enough so that encryption, specifically OTP generation, starts as soon as possible. Accordingly, the IV cache needs to be as small as possible while achieving a high hit rate.

From Figure 2.12, we observe that increasing the cache size beyond 4MB has little impact on reducing the cache miss rate. However, increasing the cache size brings significant reduction in the cache miss rate until reaching a capacity of 4MB. Accordingly, we believe that a size of 4MB is the smallest cache size that achieves a reasonable cache miss rate.
2.7 Discussion

In this section, we start by discussing several security concerns when using Silent Shredder and counter-mode encryption in general, and describe possible remediations. We then discuss different use cases and applications that can benefit from Silent Shredder at no
2.7.1 Security Concerns

Several security concerns can arise when using counter-mode encryption in general. Other concerns arise when relying on hardware to achieve operating system privileged tasks. In this section, we discuss the concerns and their remediations.

- **Tampering with Memory and Encryption Counter Values**: Since data is already encrypted, tampering with the memory values can cause unpredictable behavior. However, tampering with or replaying the counter values can be a security risk. While out of the scope of this paper, to avoid such risk, techniques such as Bonsai Merkle Trees can be used for authenticating the counter values with low overhead (about 2%) \[117, 153\]. The only information that could be leaked to an adversary is whether a page is shredded or not; a minor counter of 0 will indicate shredding, however, such information is useless. If we want to hide such information, the only modification needed is to encrypt the IV counters when writing to the main memory.

- **Losing Counter Cache Values in Case of Power Loss**: In counter-mode encryption, the counter cache must be backed up to NVMM whenever there is a crash or power loss. Hence the counter cache must be battery-backed and guaranteed to be persistent. In addition, the kernel should have a mechanism to verify the integrity of the stored counter values. If the integrity has been violated, then the kernel should report an error and stop execution of the system. Further error handling mechanisms are needed to recover from this error, which are beyond the scope of this paper.
Another way for achieving persistence of the counter cache would be a write-through implementation, i.e., any update to the counter cache is also updated in NVMM. In the latter case, non-temporal stores or DMA engine zeroing write zeros to NVMM in addition to the counter cache block. However, in Silent Shredder only the counter cache block is written to NVMM.

• **Using the Shred Command from User Space**: As explained earlier, the Shred command hints the memory controller with a physical page address to shred. Allowing user-space applications to issue shred commands can be a security risk. Accordingly, we implement the shred command so that it can be executed only by the kernel (kernel-mode) and only when the kernel allocates a new physical page to a process. Any attempt to write the memory-mapped I/O register of the memory controller from a user-space process will cause an exception.

### 2.7.2 Other Use Cases for Silent Shredder

While our main goal in this paper is data shredding, Silent Shredder can be used in other areas.

• **Virtual Machine Isolation in Virtualized Environments**: In VMs, zeroing out pages is used heavily to shred any physical page before allocating to a VM [142]. This zeroing happens in addition to the kernel shredding occurring inside the VM to protect applications’ data. VMs typically request large pools of pages and prefer large pages due to the reduced translation overhead; large pages skip one or more levels
of translation and hence speed up the page table walk process [107]. Furthermore, large allocations can reduce the number of hypervisor interruptions. Silent Shredder can speed up the VM requests by completely avoiding zeroing out pages. In case of highly-shared systems, the hypervisor continuously reclaims physical memory pages from VMs in a process called memory ballooning, in order to be able to satisfy requests from VMs with higher memory demand [142]. Accordingly, data shredding will be very frequent and hence Silent Shredder can be of substantial use.

- **Initializing Large Blocks of Memory**: Some applications tend to zero out large amount of allocated memory, e.g., a sparse matrix, as part of the initialization phase. Such initialization can benefit from the hardware support provided by Silent Shredder. Applications would invoke a system call to provide the kernel with the starting virtual address and the number of pages to be zeroed out. The system call would be translated by the kernel shred command with the corresponding physical address.

- **Zero Initialization in Managed Programming Languages**: In managed languages, such as Java and C#, the language specifications require initializing new objects with zeros. Similarly, unmanaged native languages, such as C++, have started using zero initialization to improve memory safety [100]. Silent Shredding can be used to zero initialize allocated memory pages with low overhead. It only requires a system call to the kernel, providing it with the virtual address of the page(s) desired for zero initialization. The kernel will simply submit a shred command with the physical address of each page to be zero initialized.
Table 2.2: Comparison between Silent Shredder and other initialization techniques.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Cache Pollution</td>
</tr>
<tr>
<td>Non-temporal stores</td>
<td>✔</td>
</tr>
<tr>
<td>Temporal stores</td>
<td>x</td>
</tr>
<tr>
<td>DMA bulk zeroing</td>
<td>✔</td>
</tr>
<tr>
<td>engine</td>
<td></td>
</tr>
<tr>
<td>RowClone</td>
<td>✔</td>
</tr>
<tr>
<td>(DRAM-specific)</td>
<td></td>
</tr>
<tr>
<td>Silent Shredder</td>
<td>✔</td>
</tr>
</tbody>
</table>

2.8 Related Work

In this section, we discuss prior work related to our work. Since our work is interdisciplinary and has intersections with different research problems, we summarize previous work by the problems they addressed.

**Data Shredding:** Data shredding is the process of erasing data previously written by a process to prevent inter-process data leak. Chow et al. [30] discuss the importance of shredding data during page deallocation and investigate both temporal and non-temporal ways to zero out pages for shredding purposes. Currently, modern operating systems deploy shredding by zeroing out pages [20, 43, 119, 129]. Our work achieves the shredding process at zero-cost by eliminating the need to write to the main memory.

**Improving Initialization Performance:** Jiang et al. [66] suggest offloading the zeroing
process to a dedicated DMA engine close to the memory controller. Their design reduces both cache pollution and wasted processor time. Seshadri et al. [124] observe that significant memory bus bandwidth could be wasted by initialization. Accordingly, they suggest the zeroing operation to occur within DRAM by dedicating a zero row inside each DRAM subarray. Their design, RowClone, efficiently reduces the fraction of bus bandwidth used for initialization. However, in both the previously mentioned work, writes to the memory cells still occur. Lewis et al. [79] propose reducing the memory access required to fetch uninitialized blocks on a store miss by using a specialized cache to keep track of uninitialized regions of memory. While their work and Silent Shredder have similar advantages, our design’s main goal is to efficiently protect applications’ data. Speeding up applications and reducing memory traffic are additional advantages of the way we shred data. Yang et al. [158] study the performance implications when using temporal versus non-temporal stores for zeroing data. Sartor et al. [121] propose new ISA instructions to speed up zeroing in managed languages. Their solution, however, requires that the kernel zeros out processes’ new physical pages in main memory; hence Silent Shredder can be of great importance when using such a scheme. Table 2.2 presents a brief summary of the features of the most related initialization techniques. Note that for both temporal and non-temporal stores, the processor needs to execute a large for loop of movq/movntq instructions to zero out the whole page. In the case of temporal stores, it can also pollute caches and indirectly increase the main memory writes, as shown in Section 6.2.3. Furthermore, Silent Shredder enables fast read/write operations for the initialized blocks; shredded lines can be quickly recognized from their IV values. Silent Shredder also achieves persistent shredding, because the IV values must be backed up in case of power failure, otherwise, recovering the data
will be impossible.

**Secure Non-Volatile Memory Controllers:** Chhabra & Solihin [27] propose encrypting cache lines when writing to the main memory. Their design aims to efficiently avoid revealing data when NVMM chips get stolen. However, their implementation does not protect from bus-snoop, dictionary-based and replay attacks. Later, Young et al. [159] propose a design that uses counter-mode based encryption that eliminates bus-snoop attacks, replay attacks, dictionary-based attacks, and physical NVM access attacks. Their design, DEUCE, uses AES counter mode for encrypting cache lines before writing back to the main memory. Our secure NVMM controller is based on a design similar to the one in DEUCE. Accordingly, it prevents all the previously mentioned attacks.

**Write-aware Non-Volatile Memory Controllers:** Writes to NVM-based main memory are expensive in terms of latency, energy requirements, and lifetime reduction. Accordingly, many previous work have targeted reducing the actual number of writes to NVM-based main memory [110, 163]. Qureshi et al. [110] propose to dynamically change the bound of the main memory using start and gap registers to uniformly distribute writes across memory cells. Later, Zhou et al. [163] suggest using differential writes, i.e., first read the previous data, compare it with the new data, and then only write the cells whose values have changed. Cho & Lee [29] propose Flip-N-Write technique, their technique suggests that memory words can be written in a flipped manner to reduce the number of bits having their value changed and hence reducing the number of actual writes. In the context of secure NVMM controllers, Young et al. [159] observe that techniques such as Flip-N-Write and differential writes are inefficient in presence of encryption. They propose to reduce the number of writes by avoiding re-encrypting the unmodified partitions of a cache line. Our
work is orthogonal and can be easily integrated with their design, DEUCE. DEUCE targets reducing the need for changing cells’ values when a cache line write is certain to occur. However, Silent Shredder eliminates the cache line writes completely when shredding new pages.

### 2.9 Conclusion

NVM technologies are serious contenders for replacing DRAM as main memory in the near future. However, they face two key challenges for widespread adoption: limited write endurance and data remanence vulnerability. NVM encryption alleviates the data remanence vulnerability, but exacerbates the endurance challenge by increasing the number of main memory writes. In this chapter, we propose an approach to reduce the number of writes to encrypted NVM by completely eliminating the writes occurring due to OS data shredding. Our approach, Silent Shredder, manipulates counter mode encryption IV values to render memory pages unintelligible and hence obviates the writing of zeros to memory pages. For software compatibility, we encode IV values in a manner so that Silent Shredder quickly identifies shredded blocks from their IV values and then returns a zero block instead of returning unintelligible data stored in shredded pages. As a byproduct, this process also speeds up reading shredded cache lines. Hence Silent Shredder reduces the number of writes and speeds up reading in encrypted NVM, and hence reduces power consumption and improves overall system performance. We believe that encryption will be commonly used in NVMM for data protection; hence our Silent Shredder approach can contribute significantly to NVMM adoption by increasing NVMM lifetime and improving overall system performance.
CHAPTER 3

UTILIZING EMERGING NON-VOLATILE MEMORIES AS MAIN MEMORY EXTENSIONS
3.1 Introduction

The chapter is organized as follows. In Section 3.2, we discuss how memory extensions work and their expected impact on lifetime. We present the related work in Section 5.7. Later, we discuss our emulation methodology and assumptions in Section 3.4. In Section 6.6, we discuss and study our memory extension emulation. We also propose and study several performance optimizations and write reduction techniques. Finally, we conclude our work in Section 5.8.

3.2 Memory Extensions

Memory extensions are those memories dedicated for extending memory capacity. At any instance of time, a memory page can be either in the DRAM or the NVM. However, any access to an NVM page will be handled by the OS and this results in moving the page to DRAM, as shown in Figure 3.1.

Figure 3.1: Example of handling accesses to the memory extensions.
The kernel can keep track of the physical pages through a linked list of pages’ metadata structures (e.g., `struct page`). When a physical page is moved to NVM, all the virtual addresses that map to that physical page get unmapped, so that any later access to that page will raise a page fault. Note that there must be a mechanism to know the actual NVM physical address the unmapped virtual address used to map to, so the kernel can copy its content to a DRAM page and then map the virtual address to the new page. Since DRAM size is fixed, copying or moving a page from the NVM to the DRAM also requires evicting a page from the DRAM to the NVM. The actual page eviction to the NVM doesn’t need to occur in the critical path, but can be simply buffered in a pre-allocated fixed-size eviction buffer in DRAM. The eviction buffer is periodically flushed through a kernel thread without affecting the critical path.

![Expected Lifetime of Memory Extensions](image)

**Figure 3.2:** The expected lifetime of memory extensions (log scale).

NVM technologies have different endurance levels. As example, the NAND-based Flash
cells can wear out after few tens of thousands of cycles. In contrast, PCM cells can endure few millions to hundreds of millions of writes per cell. Furthermore, PCM is orders of magnitude faster than NAND-based flash. Such factors make emerging NVMs a much stronger candidate for extending the memory.

Figure 3.2 shows a comparison between the expected lifetime for a system that uses memory extension of PCM versus a one that uses Flash. For both memory extensions, we assume the memory extension has 4x the DRAM capacity (4GB DRAM and 16GB memory extension). We assume PCM with 150ns/1500ns page read/write latency and endurance of $10^7$ writes per cell. For NAND-Flash, we use the model calibrated in [146], a 25 $\mu$s read/write latency per page. We also assume $10^4$ writes per cell, which is typical for modern NAND-based Flash.

From Figure 3.2, we can observe that Flash-based memory extensions can wear out in less than a month for most of the HPC workloads we tried. In contrast, PCM-based memory extension can last for a few years for most of the applications, however, the lifetime can be shortened further for systems with other frequent write activities (e.g., checkpointing in HPC systems). For large-scale HPC systems, the write traffic can be multiple of times higher and hence making memory extensions less reliable. Accordingly, in this work, we focus on studying and improving both the lifetime and performance for memory extensions.

### 3.3 Related Work

In this section, we summarize the related work. We categorize the related work as following:

- **Memory Extension**: using a dedicated device for solely extending the virtual mem-
ory has been proposed by previous work [122]. FlashVM [122] extended the virtual memory using NAND-based Flash SSDs. However, emerging NVMs are orders of magnitude faster than Flash drives, which makes using them as memory extensions much more attractive. Furthermore, emerging NVMs can endure many more writes than Flash [32]. Accordingly, our work shows how extending the virtual memory with emerging NVMs can affect lifetime and endurance.

• **Emerging NVMs as I/O Devices**: Recent studies explored using emerging NVM devices, such as PCM, as building blocks for storage systems. In [149], the authors propose an optimized version from the state-of-the-art I/O host controller interface, NVM Express [2]. The authors propose DC Express, an optimized protocol that is more suitable for emerging NVM devices. Another work studied the performance bottlenecks and possible optimizations when using fast NVM devices over the standard NVM Express interface [7]. Our work is different in that we explore emerging NVM devices as extension for the main memory.

• **Emulating NVMs for Persistent Memory Allocations**: Previous work studied how an application can allocate a range of memory pages that are backed by a file in a NVM device [145]. Current Linux systems already have a system call called `mmap`, which can be used to back a specific range of virtual addresses by a file. Our work is different in that we don't require any modifications at the application-level, but all the system memory pages are managed by the OS (i.e., any page of the application heap can be located at NVM or DRAM, and that is solely managed by the OS). Furthermore, our design assumes no filesystem on the NVM device, hence no need to create files to hold application-specific pages. On the other side, our work focuses on enhancing
performance and leaves out persistency, which can be handled by specific APIs or kernel drivers, such as PMEM [106]. Another recent work uses a commercial platform, PMEP, to emulate how storage-sensitive workloads can have their performance affected when replacing DRAM with emerging NVMs [162]. Our work is different in that we evaluate using emerging NVMs as memory extensions, mainly for enhancing capacity rather than taking advantage of persistency. Furthermore, our work focuses on HPC workloads with large memory footprints.

3.4 Methodology

In this section, we explain the emulation infrastructure we use, our assumptions and the applications we run.

3.4.1 Emulation Infrastructure

Our emulation infrastructure is based on the PerMA NVRAM emulator [145]. PerMA is used to emulate the performance of persistent memory allocations for customizable NVM device latency. The PerMA NVRAM emulator is implemented as a Linux device driver that allows application execution at native speeds. If any application wants to allocate persistent memory, it can simply use the mmap system call to map specific range of virtual addresses to a file. Typically, this is done by passing the starting address and the size of allocation to the mmap system call. Any access to that virtual address range causes a page fault that will be handled by the PerMA driver.

As mentioned earlier, the typical use of PerMA [145] emulator is to mmap a space from
<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lulesh</td>
<td>It represents a typical hydrocode. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh [59].</td>
</tr>
<tr>
<td>XSBench</td>
<td>A mini-app representing a key computational kernel of the Monte Carlo neutronics application OpenMC [140].</td>
</tr>
<tr>
<td>RSBench</td>
<td>A mini-app to represent the multipole resonance representation lookup cross section algorithm [141].</td>
</tr>
<tr>
<td>SimpleMoC</td>
<td>A mini-app that demonstrates the performance characteristics and viability of the Method of Characteristics (MOC) for 3D neutron transport calculations in the context of full scale light water reactor simulation [50].</td>
</tr>
<tr>
<td>MiniFE</td>
<td>A proxy application for unstructured implicit finite element codes [52].</td>
</tr>
<tr>
<td>MiniAMR</td>
<td>A mini-app designed to support the study of adaptive mesh refinement (AMR) codes at scale [52].</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of the proxy applications we use.

the file of the PerMA device into the process address space by using the mmap system call. However, using such technique is limited and insufficient for our study for several reasons. First, we need to modify the applications code to modify every malloc and replace it with mmap. Second, every malloc will cause an mmap system call that is expensive. Third, managing the allocated space through the malloc calls is now shifted to the PerMA device, which deals with page granularity, unlike typical malloc/free implementations. Finally, we also need to have the shared libraries allocating memory from the PerMA device, not only the malloc calls visible in the source code. To overcome the previous issue, at the linking time, we inter-position standard malloc with a simple implementation of malloc/free and new/delete calls.

Initially, the first malloc/new mmap a very large space (e.g., 32GB) from the PerMA device into the process address space. Later, any subsequent malloc/free calls will use that
space as if it is the heap of the process (i.e., they initially see a free 32GB starting from the address returned from the mmap occurred in the first malloc). All subsequent malloc calls will be looking into free space within that 32GB space, while free calls will free the allocated space from that 32GB space. In other words, the mmap-allocated region is used as the heap space. Accordingly, malloc/free are now allocating regions from the space visible to the device and any page faults are handled through the PerMA device driver. We implement a simple first-fit malloc/free, with both coalescing at free and splitting allocations with large internal fragmentation.

3.4.2 System Configuration

We run all of our experiments on an AMD APU A10-7850K with 32GB main memory. We use Linux kernel version 3.19.6 with Ubuntu distribution.

3.4.3 NVM Device Model

Similar to current SSD drives, writes are expected to be hidden through several SSD optimizations, such as log-based writing and buffering. Indeed, previous papers calibrated read/write latencies of flash drives and found them to be similar [145]. For NAND-Flash, we use the model calibrated in [145], a 25 us read/write latency per page. For PCM, we use a page read and write latencies of 150ns and 1500ns, respectively. We assume a NAND-flash endurance of $10^4$ write cycles, and PCM endurance of $10^7$ write cycles. Our conservative estimations are based on projections from literature and recent announcements for emerging
NVM technologies [27, 32, 81, 111, 112, 159].

Current emerging NVM technologies, such as 3D Xpoint, are expected to be deployed with 4x the capacity of DRAM in future systems [33]. Accordingly, unless explicitly mentioned, we assume DRAM:NVM ratio to be 25%.

3.4.4 Applications

In this study, we focus on HPC workloads that are expected to run on large-scale systems with real demand for memory capacity. Accordingly, we select several open-source proxy applications from the U.S Department of Energy. The selected applications were configured to run with large memory footprints (approximately 16GB). Table 3.1 presents a summary of the used applications.

3.5 Evaluation and Analysis

In this section, we propose and discuss several write reduction schemes and compare their impact on number of writes and lifetime. Later, we study the impact of the DRAM to NVM ratio on performance and hence understanding how technologies with different densities can be used as memory extensions in future systems.
3.5.1 Write-aware Memory Management

One of the main limitations of emerging NVMs is their limited write endurance. For example, PCM cells can only endure few millions of writes. Furthermore, writing to PCM array consumes up to 43x energy compared to DRAM [76]. Accordingly, deploying such technologies as main memory extension requires careful consideration for how many writes they are exposed to. In this section, we study the impact of system software implementations on number of writes. Later, we investigate the impact of using state-of-art hardware techniques on write reductions and enhancing the lifetime.

3.5.1.1 System Software

One important aspect to consider is the inclusion property of memory extensions. The inclusion property determines if a page that is allocated in DRAM is required to be allocated in the NVM as well. For example, the system software can either copy the accessed page to the DRAM and keep a copy at the NVM or simply move the page to DRAM and free up the NVM copy.

We refer to keeping a copy from the page into the NVM as inclusive memory extension. Inclusive systems are very important in the context of NVM systems; many pages get written few times and for the rest of the application are only being read. As depicted by Figure 3.3, in inclusive systems, when a page gets evicted from DRAM, the system software checks if the page has been updated since the last time it was brought from the NVM. If the page was not updated, no write to the NVM is required, as shown in Case 1. However, if the page was updated, the page should be written back to the NVM device. Checking if the
Figure 3.3: Handling page eviction in inclusive memory extensions.
page was updated or not can be implemented by exploiting the page dirty bit which is set by the Memory Management Unit (MMU) hardware. Most of the modern processor systems support the dirty bit in MMU. Such simple checking can save lots of writes to the NVM device. To study the effectiveness of having the NVM inclusive, Figure 3.4 shows the number of cells written in inclusive memory extensions relative to non-inclusive system where old copies are simply discarded. We can see that up to 90% (for MiniFE) and an average of 49.5% of the writes could be eliminated by adopting an inclusive policy.

![The Impact of Inclusion Property on Number of Writes](image)

Figure 3.4: The impact of inclusion property on number of writes.

The cost of deploying the inclusion policy is the wasted space in the NVM capacity. Systems with low DRAM:NVM ratio will benefit from the inclusion property at negligible ratio, however, systems with high NVM ratio (e.g., 50%) might prefer to use the whole capacity and disable the inclusion property. The decision of making the memory extension inclusive or not depends on applications' memory footprint and the DRAM:NVM ratio.

To reduce the number of writes further, the Operating System can optimize writing
pages with large percentage of similar words within the same page. We call our technique Most Frequent Word Reduction (MFWR). As explained in Figure 3.5, a memory page can contain several words (e.g., 4 bytes words) that have similar content. Accordingly, at write time, the OS finds the most frequent word (MFW) in a page along with its bitmap, which indicates the locations where the value of MFW exists. Later, all other words are packed together to form a compact page. The bitmap along with the MFW can be stored either in the page table meta data (e.g., struct page) or packed together with other words to be written to NVM. Since DIMM-based NVMs can be written in cache line granularity (e.g., 64B) the packed words will be written using the least number of cache lines that can include all words. Finding the most frequent word or value was studied and shown to be promising in the context of compressing cache lines in caches [157].

The overhead can be a maximum of 3.2% per page, however, only those pages that benefit significantly from MFW reduction needs to store their MFW along with the bitmap.

MFWR requires a single pass over the page data to find the most occurring word and
its bitmap, then pack the words and the meta data, however, our evaluation showed that the overhead is negligible given that the writes occur in the background. Note that other reduction schemes, such as general compression, are possible, but they come at additional costly latency that can add to the read and write paths from the NVM device.

Figure 3.6 shows the impact of using MFWR technique on number of writes for the MFWR that only looks for the most frequent word (MFWR-1W) with other approaches that optimize the two most frequent words (MFWR-2W) and the three most frequent words (MFWR-3W).

![The Impact of Most Frequent Word Reduction (MFWR) Technique](image)

**Figure 3.6:** The impact of MFWR technique on number of writes.

As shown in Figure 3.6, adding MFWR-1W to inclusive memory extensions can eliminate an average of 61.5% of writes compared to non-inclusive memory extensions. However, adding the number of frequent words to optimize did not bring significant benefits than MFWR-1W. MFWR-2W and MFWR-3W reduce the number of writes by less than 1% relative to MFWR-1W. For the rest of the paper, we will use MFWR-1W as the default MFWR
technique.

3.5.1.2 Hardware-aware System Software Management

While the inclusion property is a system software optimization, other techniques for write reduction has been proposed in hardware. For example, Data-Comparison Writes (DCW) technique [154] has been proven to significantly reduce the number of writes. DCW compares the old values of the NVM cells with the new data to be written, and then eliminates programming those cells which don't have their values changed. DCW is efficient due to the fact that the probability of having the bit written to a cell being similar to the old value is 50%, hence a promising write reduction. Unfortunately, combining both techniques is counter-intuitively tricky; NVM devices are expected to deploy wear-leveling techniques that try to distribute the page writes uniformly across the device pages, however, DCW is most efficient when the pages get written to the exact physical pages. In other words, if the NVM controller deploys a wear-leveling mechanism, it will not have any clue if the current page write has any logical relationship with any free physical page (i.e., subsequent writes to the logical device address can be from different applications and runs and hence the data being written can be significantly different). A translation layer, similar to Flash Translation Layer (FTL), finds a free page depending on the wear-leveling algorithm and write the page there. Once the write is complete, the translation layer just logs the new NVM device physical address for that logical address. DIMM-attached NVM devices are also expected to deploy some wear-leveling mechanisms [111].

To tackle the problem, we propose a placement hint we denote by PIN. PIN hint is
provided to the NVM controller when writing a page. For example, we can use one of the reserved fields in the NVM Express protocol [2] command structure to hint the NVM controller to enforce placing the logical page to its previous mapping in the translation table. Otherwise, the NVM controller will simply deploy its default wear-leveling technique. In case of DIMM-attached NVM, the hint could be as simple as writing to a memory mapped register that is visible to the memory controller, and hence avoid applying wear-leveling techniques when start writing the physical page to NVM. To emulate the impact of wear-leveling on DCW effectiveness, we assume that the wear-leveling technique will pick a physical location that has no logical relationship with the actual value of the page to be written, hence we use a randomly filled data for the physical destination.

Figure 3.7 shows the impact of DCW technique with and without pinning hints.

![The Impact of Data Comparison Write (DCW) Technique](image)

Figure 3.7: The impact of DCW technique on number of writes.

From Figure 3.7, we can observe that DCW with pinning and without pinning can save an average of 95% and 83.8% of writes, respectively.
3.5.1.3 Comparison Between Write Reduction Schemes

![The Impact of HW/SW Approaches on # Writes](image)

**Figure 3.8:** Summary of the impact of write reduction schemes on number of writes.

![The Impact of HW/SW Approaches on NVMs Lifetime](image)

**Figure 3.9:** The impact of write reduction schemes on lifetime.

Figure 3.8 shows how different HW/SW approaches can affect the number of writes on the NVM device. The number of cell writes was calculated depending on the write reduction algorithm. As an example, the baseline writes 4096 bytes for every page write, while the
DCW technique only writes the cells having their values changed. We observe that for some applications, such as XSBench and MiniFE, the inclusion property could eliminate about 90% of the writes. The main reason behind this is that a large percentage of pages are only being read and rarely get written after initialization. However, for some other applications, such as MiniAMR and SimpleMOC, inclusion only saves less than 10% of writes. We can observe that by adding the simple MFWR scheme we propose can even reduce the number of writes further by 23.9% relative to the inclusive case. The main advantage for both of the previous techniques is that they are implemented completely from the system software side and hence no hardware support is needed. However, in the presence of hardware write reduction techniques, the system software should be aware of that and help guiding the NVM controller. NVM devices with DCW hardware support with system software hints can reduce the number of writes by an average of 32.3% compared to those without software hints.

In summary, we presented several practical schemes that can also benefit from hardware support that could eliminate up to an average of 95% of writes. Saving such amount of writes reduces overall energy and increases system lifetime. The additional lifetime can be used for other essential HPC-related purposes, like checkpointing and fault tolerance. Figure 3.9 shows a conservative estimation for the NVM lifetime depending on the application run on the system. Our calculation assumes that every single cell of the NVM device can endure ten million writes and we have a 16GB NVM device. We calculate the rate of writing cells for every application and calculate the estimated lifetime of the NVM device. We can observe that some applications, such as Lulesh, can wear out in about 5 years, but with using proper write reduction techniques it can survive up to 100 years.
3.5.2 The Impact of DRAM to Memory Extension Ratio on Performance

Different NVM technologies are expected to have very high densities. For example, PCM is expected to have 4x the DRAM density [112]. However, the way of organizing the cells can also affect the density. For example, a cross-point organization can highly increase the density. Another way of improving the density further is to use multi-level cells (MLCs) where different resistance levels for the same cell can encode more than a single bit. In this section, we study the impact of varying the DRAM:NVM ratio for the memory footprint of the applications. For example, 1:4 (25%) ratio indicates that a maximum of 25% of the application memory footprint can be present in DRAM and the rest is in the NVM device. The ratio itself can be determined by the OS and restricted with the maximum capacities of both of the DRAM and the NVM device. Figure 3.11 shows the impact of DRAM to NVM ratio on performance for the studied applications.

We can also observe that different applications can be affected differently with changing the DRAM:NVM ratio. For instance, some applications, such as MiniAMR and SimpleMoC, show less sensitivity for DRAM:NVM ratio than other applications. The main reason behind this is that the memory pages are accessed subsequently after the first access and rarely get reused after that, hence the number of page faults changes slightly with changing the DRAM:NVM ratio. In case of Lulesh, MiniFE and RSbench, the actual working set is large enough to fit into DRAM and hence changing the DRAM:NVM ratio can affect performance.

The exact DRAM:NVM ratio where the application performance gets hurt differ across applications. For example, MiniFE performance degrades significantly when DRAM:NVM ratio equals or smaller than 25%, however, for RSbench the DRAM:NVM ratio of 12.5% is the threshold point. Note that determining the DRAM:NVM ratio where the application
Figure 3.10: The Impact of DRAM:NVM Ratio on Performance of using PCM vs. Flash.
starts to get hurt significantly relies on the actual working set (not only memory footprint) of the application.

In summary, as shown in Figure 3.11, at DRAM:NVM ratio of 1:4 (25%), which is expected to be the actual DRAM:NVM ratio in future systems [33], PCM is expected to deliver an average performance with only 75.9% performance degradation compared to the ideal DRAM-only design. However, Flash is expected to deliver an average of 380% performance degradation to the ideal DRAM-only design.

### 3.5.3 OS-level Page Prefetching

Prefetching can be used to reduce the impact of OS page cache misses by speculating which pages will be used in the future, and bring them ahead of time. However, many parameters should be considered when using software prefetching (e.g., what drives the prefetching, where in the code to execute the prefetching). The most recent approaches for prefetching
suggest that software prefetching should be added to the page fault handler [103, 122]. In our implementation, we exploit the workqueue threads feature in modern Linux kernels. Workqueues enable asynchronous work submission, where a function can simply submit a work that can be shortly handled by a kernel thread. Figure 3.12 shows the difference between synchronous and asynchronous prefetching. We can observe that synchronous prefetching can add significant overhead to the page fault latency, and hence limiting the number of pages that can be prefetched without affecting the average page-fault handler latency; the page fault handler will be delayed until the prefetching is complete. For example, in [122], the authors found that prefetching more than two pages can increase the average page fault latency. Accordingly, we use asynchronous software prefetching. When a page fault occurs, the page fault handler decides whether to issue a prefetch or not, depending on if there is a currently running prefetch thread. If the decision is made to make prefetching, a separate kernel task/workqueue will be issued.

![Figure 3.12: Synchronous vs. Asynchronous Prefetching.](image-url)
As observed by Oskin & Loh [103], at page-level granularity, simple stride prefetcher can
be more efficient than a much more complicated prefetchers such as Markov prefetcher. Accordingly, to evaluate the effectiveness of prefetching, we use a simple stride prefetcher that records the most recent 8 strides. The prefetcher issues prefetching requests for strides that appeared at least for 5 times since it was recorded, hence avoid prefetching wrongly detected streams. The number of prefetches for each stride is determined by the depth of the prefetcher. Figure 3.13 shows the impact of prefetching on performance while varying the depth of the prefetcher. For each application, the primary axis shows the total number of page faults, while the secondary axis (to the right) shows the execution time. The page faults (i.e., DRAM misses) are categorized into: full and partial misses. Partial misses are those occurred while a prefetching for that page has already started, while full misses are for those not in DRAM and has no prefetching in progress. The top area on each figure represents the actual savings in number of misses compared to no prefetching.

We can observe that prefetching can reduce the number of page faults with different levels for different applications. For example, MiniFE and MiniAMR have significant percentage of their faults eliminated due to prefetching. However, other applications slightly benefit from prefetching. We also observe that for applications that don't get significant reduction in number of faults, the performance can be penalized. The main reason behind this is the processor time spent by the kernel thread for prefetching. Furthermore, submitting a work to the kernel worker is not free; our measures showed that about 3-4 microseconds could be added to the page fault handler just for submitting the work. For example, we can see that most of the applications get their performance degraded at depth of 1, because for almost every page fault, the page fault handler will find that there is no prefetching in progress and hence issuing a prefetching request. This adds the
work submission overhead to the page fault handler almost for each miss. Interestingly, some applications, such as MiniFE, could gain up to 37.2% performance improvement. While beyond the scope of this paper, the OS can adaptively enable/disable prefetching by learning how efficient is prefetching for different workloads.

3.5.4 Page Replacement Policy

As we have fixed number of pages that can fit in the DRAM, we need to have a policy to choose which pages to evict when bringing pages from NVM. For this purpose, we study the effectiveness the Clock algorithm [138] and compare it against the simple FIFO. Clock algorithm works as follows: a pointer points to the page right after the most recently inserted
one. Each time we want to evict a page, we start from the page pointed by the pointer to check the reference bit. The reference bit is set by the hardware at any access to the page. The reference bit can be checked by the operating system through the page table entry for the page. If the reference bit is set to 1, we clear it and move the pointer to the next page and check it. The process continues until finding a page with the reference bit cleared, that page will be chosen for eviction and the pointer will move to the next page. Finally, we also implement age-based algorithm that is different than the clock algorithm in that it has a counter per page. The counter is incremented if the reference bit is found to be set, and decremented otherwise. If a page has its counter with value less than or equal to zero, it will be selected for eviction.

Figure 3.14 shows the relative number of page faults for both Clock and Age-based replacement algorithms compared to FIFO. We can observe that most of the benchmarks don’t benefit and even some of them, such as Lulesh, get hurt from changing the replacement policy. Only XSBench benefits from the replacement policy change. We also vary the DRAM:NVM ratio and did not observe any change except for XSBench, where Clock and Age-based algorithms could save roughly 50% of the page faults. The reason why XSBench benefits from the Clock algorithm is that it uses some indirection table that holds pointers to small arrays. The accesses to the whole structure are random, but keeping the pointers structure, which is frequently accessed is beneficial. Clock and Age-based algorithms can detect such pages that are frequently accessed and try to keep them in DRAM as much as possible. For all other applications, our conclusion is consistent with previous researchers findings [143] in that at page-level granularity, a simple algorithm such as FIFO would be more efficient than other more complex schemes such as Clock algorithm.
3.6 Conclusion

Our work studied the effectiveness of using emerging NVMs as memory extensions. We showed how using emerging NVMs as memory extensions without write-aware management techniques can have serious impact on systems’ lifetime. We discussed and proposed several management schemes and evaluated their impact on reducing the number of writes. We also explored the impact of varying the DRAM:NVM ratio of memory extensions on performance, for both Flash and PCM. To enhance performance, we investigated and proposed OS-level prefetching, which showed promising results for some applications. Furthermore, we studied the impact of page replacement policy on number of page faults (NVM accesses). Our study showed that only one application out of six applications, benefited noticeably from changing replacement policy to temporal reuse aware policies, such as Clock and Age-based algorithms.

We believe that our work gives insights about the impact of using emerging NVMs as memory extension and helps in understanding how to manage such memory extensions to make them more reliable.
CHAPTER 4

UTILIZING EMERGING NON-VOLATILE MEMORIES AS I/O DEVICES

4.1 Introduction

This chapter is organized as follows. Section 5.7 discusses related work. Section 4.3 covers an overview of the NVMe host controller interface and the system architecture. Section 5.4 describes the testing methodology we use to analyze performance. Section 4.5 presents a
detailed study of major aspects that affect the system performance, such as the queuing model, NVM device technology, and interrupt impact on the system. Section 6.6 discusses the main observations and findings of the study and analysis. Finally, we conclude our work in Section 5.8.

4.2 Prior Work

Most of the prior work focused on protocol optimizations and the impact of different implementations on performance, without considering analyzing the performance bottlenecks of the system. In [149], the authors propose an optimization for the standard NVMe protocol, mainly the protocol communication latency. The paper focuses on the performance improvement over the original standard NVMe protocol. In contrast, our work is different in that it highlights general system bottlenecks and investigates the performance impact of different implementations. Our work is useful in providing researchers insights that will enable them to discover modern NVM host controller interface optimizations for fast NVMs.

Another work that discusses changes to the host controller interface implementation for I/O devices is found in [17]. The authors assume a null device that acknowledges I/O requests once they are passed to the device driver. Their study is limited to investigating the overhead of accessing shared I/O submission queues in the Linux block layer, and proposing the use of multi-queues instead. The work is specific to the Linux block-layer which is generic and contains requests to all I/O devices, and is not specifically optimized for fast NVMs. The paper did not investigate the system impact when using multi-queues; for example, how using multi-queues can affect the performance of the main memory. In our work, we investigate the impact of such changes, including, but not limited to, the use
of multi-queues instead of a single shared queue.

4.3 Non-Volatile Memory Express (NVMe)

Figure NVMe is the standard industry protocol for communication with non-volatile memories over the PCIe interconnect [2]. NVMe is featured for an optimized queuing interface, command set and feature set for communicating with fast NVMs. In the following sections, we will give an overview of the main structures of NVMe protocol, the corresponding model of the NVMe-capable controller, a system overview and a scenario of how NVMe protocol handles an I/O request.

4.3.1 NVMe Memory Structures

NVMe has two different types of commands: Admin commands and I/O commands. Admin commands are mainly used to inform the NVMe-capable controller for creating/deleting I/O queue and some related information, such as its base address in memory. I/O commands are the core of the NVMe protocol, its content can indicate read/write/flush request with some related information, such as the Starting Logical Block Address (SLBA) and the Number of Logical Blocks (NLB). Accordingly, NVMe defines queues to hold such commands. Each queue consists of two parts: Submission Queue (SQ) and Completion Queue (CQ) as depicted in Figure 4.1.

The SQs are used to submit commands, which should be read by the NVMe controller. The CQs are written by the NVMe controller, to report the status of the command and information about errors if occur. The queues are located in a direct memory access (DMA)
region in the host main memory. The tails and heads of the queues are synchronized through incrementing a memory-mapped I/O doorbell registers (for SQs) after submitting a command, or through an interrupt triggered by the NVMe controller after writing the status of a specific command by the NVMe controller. Both queues are circular, and each has 1024 entries by default.

4.3.2 NVMe Controller

NVMe controller is the hardware mainly responsible of fetching, decoding, executing and acknowledging the completion of I/O commands, as depicted in Figure 4.2. As shown in Figure 4.2, the NVMe controller executes the I/O command in several stages as follows. First, it fetches the command from the SQ in the main memory and writes it to its Outstanding Requests Tracker (ORT) as shown in Figure 4.3. The ORT table is mainly used to keep track
of the outstanding requests, and allow overlapped execution of I/O requests.

Second, the controller **decodes** the fetched command and accordingly determines the type of command (Read/Write/Flush), the Starting Logical Block Address (SLBA) and the Number of Logical Blocks (NLB). The third step depends on the number of pages that need to be read from or written to the host memory, where each page is a multiple of logical
blocks. If the number of pages is more than two, we need to read a table of pointers to these pages. This table is called Physical Region Descriptor Table (PRDT), similar to the table shown in Figure 4.4. The PRDTs allow the kernel to allocate physically non-contiguous page frames, instead of a large contiguous area starting from the address provided in the command. Note that different implementations of the direct memory access (DMA) engine can deal with PRDTs differently; for example, some DMA engines do not transfer the PRDT at all, but parse the PRDT using the DMA engine on the memory controller hub (MCH), which requires all the pages pointed to by the PRDTs to be transferred before moving to another request. Our choice of having the PRDTs read by the NVMe controller allows overlapping many I/O requests at the same time and giving more flexibility to the NVMe controller.

Figure 4.4: NVMe Physical Region Descriptor Table (PRDT).
Each entry in the PRDT is called a Physical Region Descriptor. It contains a pointer to the page to be read or written by the NVMe controller. In a read command scenario, reading PRDT can be done in parallel with reading from the device, because we do not need the PRDT until the data is ready to be written to the host memory. In contrast, in the case of a write command, the NVMe controller needs to know the physical addresses of where to read the blocks to be written to the device. Therefore, there is a field in the NVMe command called PRP1, which is used to pass the address of the first page, which allows starting the write operation without the need to read the whole PRDT. As discussed above, command execution may need an additional read of a PRDT, which is pointed to by a field called PRP2 in the command structure. Fourth, the NVMe controller has to notify the host of the completion of the executed command, the status of the completion and any errors occurred during execution. The notification is achieved by writing a new command completion entry to the Completion Queue (CQ) paired internally by the Submission Queue (SQ) of that command, then by interrupting the processor.

4.3.3 System Overview

As we described the details of NVMe protocol from the software perspective, in this section we focus on showing where the NVMe controller resides in the system. Figure 4.5 shows an example system with an NVMe controller and NVM devices. As shown in the figure, the NVMe controller communicates directly with the NVM devices, which may be various NVM technologies such as PCM, STT or SSD. The NVMe controller itself is attached to a PCI Express endpoint, the PCI Express endpoint is connected to the I/O bus through fast PCI Express lanes. The I/O bus is connected to the memory controller hub (MCH) via a
Figure 4.5: Overview of a system that uses an NVMe controller.
very fast bus, which has different industrial names such as QuickPath Interconnect (QPI) or HyperTransport (HT). The NVMe controller should be Direct Memory Access (DMA) capable, so it can initiate DMA writes to the buffers specified by the PRDT of the command.

### 4.3.4 NVMe Operation Scenario

In this section, we demonstrate a scenario of using the NVMe protocol for I/O operation. As shown in Figure 4.6, the operation starts by a read function call in a program as shown in Step 1. As a result of Step 1, the kernel will create one or more I/O request instances and pass it to a generic I/O software layer, where merging of I/O requests, scheduling and re-ordering occur before passing the requests to the appropriate device driver, which is represented...
by Step 2. Once the command has been passed to the specific device driver (NVMe), the NVMe device driver writes a new I/O command in NVMe format to a submission queue, as shown in Step 3. Afterward, the NVMe device driver notifies the NVMe controller about the new command by writing to a memory-mapped I/O register called tail doorbell register, as shown in Step 4. Later, the NVMe controller reads the command from the submission queue (which is allocated in DMA region), as shown in Step 5. Now, the NVMe controller executes the command through different stages as explained earlier in Section 4.3.2, ending up with writing a completion entry at the corresponding completion queue as shown in Step 6, and interrupting the processor in the next step, Step 7. Finally, the NVMe device driver processes the completion entry and returns to the generic layer and the application, as shown in Step 8 and 9, respectively.

4.4 Evaluation Methodology

4.4.1 Simulator

We use gem5 [16], a cycle-accurate full system simulator, to run IOR benchmark [125]. IOR benchmark is one of the most popular tools for I/O performance experiments, which can be used to generate synthetic I/O traffic with specific characteristics, depending on the provided parameters. We modify the gem5 simulator to model the NVMe Controller and NVM device latency. We use the NVMe device driver provided in Linux. We annotate the driver code with special no-operation instructions for timing purposes. We use the ruby interconnect model integrated with NVMain simulator [108]. NVMain simulator is a detailed cycle-accurate main memory simulator, which we use mainly to model the host
main memory. We use the IOR benchmark to read a 32 MB file from the NVM device. We choose a 32 MB file size and the number of processes to be up to 8 to keep the simulation time manageable. Simulating larger number of processes or larger file sizes can take very long simulation time, which can limit the scope of the study to very few experiments.

We focus on studying the performance of I/O read operations for many reasons. First, reading I/O data is typically needed before proceeding, while write operations do not need to be fully written to the I/O device to proceed. Second, written I/O data is often buffered in main memory (typically DRAM) or hardware caches, and not directly written to I/O device. Writing I/O data directly to the I/O device is required in very few cases, such as checkpointing to mitigate system failure.

The system baseline configuration is shown in Table 6.1. Later in this chapter, we will discuss the reasons behind using these parameters and the impact of changing them.

We assume L1, L2 caches are private per core, and the L3 cache is shared among each 8-cores. We assume that the NVM device is connected through a PCIe bus to the I/O bus (Southbridge or Platform Controller Hub). Finally, we assume the NVMe controller is integrated in the device (PCIe endpoint).

### 4.4.2 NVMe Controller Model

We model the NVMe controller in a way similar to what we described in Section 4.3.2. We assume that the ORT table has 32-entries, the fetching pointers stage is overlapped with accessing the NVMe device. Specifically, reading the first page occurs in parallel with reading the pointers of the other pages in write operation. Also, reading PRD entries is concurrent with reading the requested blocks from the device. We assume that the NVMe
Table 4.1: Configurations for the baseline system.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Assumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>x86-64 processor, timing mode, 2GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>16 cores</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>64KB size, 4-way, 64B blocks, 1 cycle latency, Pseudo-LRU</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>512KB size, 8-way, 64B blocks, 20 cycles latency, Pseudo-LRU</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>16MB size, 16-way, 64B blocks, 30 cycles latency, Pseudo-LRU</td>
</tr>
<tr>
<td>Coherency Protocol</td>
<td>MESI Protocol</td>
</tr>
<tr>
<td>Main memory</td>
<td>4 GB, 2 channels, 2 ranks per channel, 8 Banks per rank, 1KB Row Buffer(RB) 30 cycles RB hit, 102 cycles RB miss</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCIe 3.0, 4-lanes, 7.877 Gbps per lane</td>
</tr>
<tr>
<td>NVM Read Latency</td>
<td>20µs</td>
</tr>
<tr>
<td>NVM Write Latency</td>
<td>200µs</td>
</tr>
<tr>
<td>OS</td>
<td>Fedora release 20</td>
</tr>
<tr>
<td>Kernel</td>
<td>Linux 3.4.91</td>
</tr>
</tbody>
</table>

Controller is DMA-Capable with 64-bytes chunk size. Finally, we assume a strict ordering of the packets sent through the PCI Express bus.

4.5 Study and Analysis

In this section, we study the PRDT of different implementations of the host controller interface on the system performance. We also investigate the impact of using fast memory technologies on the overall system performance. In Section 4.5.1, we discuss how the organization of I/O submission/completion queues affects the system performance. In Section 4.5.2, we study the impact of changing the NVM latency in the overall execution time.
In Section 4.5.3, we study how using periodic interrupts instead of interrupt per command completion can affect performance. In Section 4.5.4, we discuss the use of polling instead of interrupt when using different NVM device latencies. Finally, in Section 4.5.5 we discuss the overhead of the traffic generated by the protocol related data, such as submission commands, completion entries and PRDTs.

4.5.1 Scalability and Queuing Model

Figure 4.7: Different IO submission/completion queuing models.

NVMe protocol uses the concept of submission and completion queues, as described in Section 4.3. In this section we discuss the impact of different organizations of the submission and completion queues, and how can this affect the overall system performance. Figure 4.7
depicts two queuing models, which we analyze in this section. In Figure 4.7a, the system has a single I/O submission and a single I/O completion queue, shared by all cores. Clearly, this implementation gives us the lowest memory overhead for allocating I/O queues. However, the single queue model has a synchronization bottleneck, because each time a core tries to submit an I/O command, it needs to acquire the lock for the queue. We refer to this model as SQ. In Figure 4.7b, the system has a submission and completion queue per core, which avoids the bottleneck of acquiring a single lock among all cores. The scalability advantage is at the expense of having a large number of allocated queues, which can also affect the system performance. We refer to this model as MQ.

![Graph: Queuing model impact on overall execution time.](image)

Figure 4.8: Queuing model impact on overall execution time.

In Figure 4.8, we show how the execution time changes when we increase the number of processes for both of the SQ and MQ queuing models. Each process reads a specific part of the file, which allows processes to proceed simultaneously. The MQ model scales better than the SQ model for up to 4 processes. However, in the case of 8 processes, it is just slightly better than the SQ model. The overall protocol scalability is affected mainly
by three major factors. First, the time spent submitting I/O command, which includes the
time to acquire the lock, establishing PRDTs, writing the command to the DMA-region
allocated submission queue and notifying the NVMe controller. Second, the time to identify
a completion entry and processing it, which includes accessing and modifying variables that
can be modified when writing a submission command, so the corresponding submission
queue should be locked. Third, reading/writing data, submission and completion entries
from and to host main memory (e.g. DRAM). The latter factor is highly affected by row
buffer locality in DRAM.

We found that the software layer is the largest contributor to the overall execution
time. It contributes to roughly 39% of the execution time. The software layer includes
but not limited to the I/O software stack. For example, in Linux, any I/O request needs to
go through a software layer called block layer, this layer is responsible of scheduling I/O
requests, merging requests and finally submitting it to the corresponding device driver (e.g.
NVMe). The dominance of the software layer on the overall execution time in the emerging
NVM technologies have been discussed in recent studies [131].

We also found that the main memory (DRAM) buffering can also contribute to the
overall execution time. The main memory buffering overhead includes buffering data to
the DMA buffers, remapping them and reading them by the application. Many researchers
have observed this in different situation (network I/O). As an example, in [56], the authors
propose to directly have the I/O data written to the last level cache. Note that our goal in
this work is not to optimize or tune the performance, but to show the bottlenecks. The
performance of the DRAM is significantly affected by the row buffer locality, which is a
very fast buffer, caches the most recently read row for each bank. Accessing the same row
subsequently will cause row buffer hits, which avoid the latency of closing the current row and precharging a new row. Our observation is that the SQ model is more friendly in terms of row-buffer locality than other models. Figure 4.9 shows how the number of row buffer misses with changing the number of processes for each queuing model.

As shown in Figure 4.9, the SQ model has fewer row buffer misses for any number of processes we tried. We can also see that the difference in the number of row buffer misses in general is increasing when increasing the number of processes. The main reason why the MQ model is less aware of row buffer locality is that each core has its own I/O submission and completion queues, which are allocated in direct memory access region (DMA), causing a lot of row buffer conflicts when writing/reading I/O commands.

The average submission and completion time per request are shown in Figure 4.10 and Figure 4.11, respectively. As mentioned earlier, the SQ model suffers from the bottleneck of accessing a single submission queue.

Figure 4.10 shows that the average time for submitting a command using the SQ model increases fast as the number of processes increases, but increases at a slower pace with the
MQ model. Changing the number of processes from 4 to 8 increases the average submission time by 30% and 5%, for SQ and MQ models, respectively. The time is measured beginning from trying to lock the submission queue, and end when the NVMe controller is notified of the submission.

The average time of command completion, starting from trying to acquire the lock of the completion queue and the corresponding submission queues, until the NVMe controller is notified of the completion, is shown in Figure 4.11. We can observe that the SQ model has
a higher average completion time when running 4 processes, which is mainly caused by the time to acquire the lock of the single submission queue in the system. Increasing the number of processes to 8 gives approximately the same average completion time for SQ and MQ, because the MQ model performance degrades as a result of the poor DRAM row buffer locality, as shown in Figure 4.9. Note that the conflicts in row buffers do not affect the average submission time as they do for the completion time, because in the submission most of the accesses to the submission queues are writes, which do not need to be fully written to DRAM to proceed, but just writing it to write buffers is sufficient. In contrast, in the case of the completion process, the completion entry must be read completely from the completion queue and having its status checked. We observe that the completion time trend is the same as the overall execution time, because it is affected by both of the locking overhead and the row buffer conflicts.

4.5.2 Technology Impact on Overall Performance

In this section, we study the impact of changing the technology of the IO device. We abstract our evaluation by changing the NVM latency for each 4KB block, starting from 1000 microseconds down to 1 microsecond. Figure 4.12 shows the impact of changing NVM Latency on the overall execution time of reading 32MB file using 1 process.

As shown in Figure 4.12, the gain from having faster NVM devices decreases significantly as we are getting close to the Phase-Change Memory latency (1 microsecond). The figure shows that the gain of having 10 times faster device is 7.23x times faster execution time, when moving from 1000 to a 100 microseconds latency device, which represents the move to SSD devices. Later, moving from 100 to 10 microseconds latency device gives a gain of
3.26x times faster execution time. Finally, moving from 10 to 1 microsecond latency device (the range of PCM device latency), gives only 1.3x faster execution time. This is consistent with our finding that the NVM latency and the transfer time are minor contributors to the overall execution time, which is mainly dominated by the software layer overhead. In conclusion, the overall I/O delay in the near future will be limited by the software layer overhead mainly and the DRAM buffering (writing to and reading from DMA buffers in DRAM) overhead.

### 4.5.3 Periodic Interrupts

Interrupting the processor each time an I/O command is ready can highly degrade performance. In [80], the authors show that at least 1.1 µs is spent in switching the context using modern x86 processor and Linux OS. In this section, we show how interrupting the processor for many completions (interrupt coalescing), instead of interrupt for each completion,
can affect performance. To the best of our knowledge, our study is the first to propose and study the impact of using one interrupt for variable number of I/O completions. We use periodic interrupts, where by the end of each period $\tau$ $\mu$s if at least one completion occurred, the NVMe controller triggers an interrupt, as depicted in Figure 4.13. From Figure 4.13, we can observe that using large interrupt detection period $\tau$, as shown in the scenario in Figure 4.13c, can reduce the number of times the processor is interrupted, but can highly increase the gap between the time the command gets ready and being processed by the processor. In contrast, using very small interrupt detection period $\tau$ causes issuing an interrupt for each completion as shown in scenario, as shown in Figure 4.13a. Using medium period $\tau$ length achieves a good trade off between the previous two scenarios, as it has moderate number of interrupts and gap between the readiness of the command and the time it gets processed, as shown in Figure 4.13b.

In Figure 4.14, we show how changing the period of interrupt $\tau$ affects the overall
execution time when using IOR benchmark to read 32 MB file using 1 process. As expected, having a medium length, which happens to be 100 $\mu s$ in our configuration, gives the shortest execution time.

4.5.4 Polling vs Interrupt

Figure 4.14: Periodic interrupt impact on performance.

Figure 4.15: Polling impact on performance.
As mentioned earlier in Section 4.5.3, the interrupt overhead can easily exceed the overhead of fast NVM technologies, such as PCM [149]. In this section we study polling as alternative for using interrupt. We modify the NVMe device driver and add a spin-wait instructions after submitting a command to the submission queue, polling on a memory-mapped I/O register to check when the data are ready and written to the DRAM buffers. We vary the NVM latency to from 100 µs to 1 µs and study how polling can affect execution time in comparison with interrupts, as depicted in Figure 4.15.

Polling can be efficient when used with very small latency devices [155]. As Figure 4.15 shows, at NVM latency of 1 µs, polling gives 3.1% speed up, but degrade performance significantly when used with high latency NVM devices. The performance degradation is a result of the high traffic generated by polling, which is proportional to the latency of the NVM device, as shown in Figure 4.16.

![Figure 4.16: Polling impact on traffic.](image)

From Figure 4.16, we can observe that using polling with high latency NVM devices can
cause an extremely high extra traffic, which can delay other packets to be sent through the I/O bus and the PCIe bus in our model.

### 4.5.5 Protocol data traffic overhead

NVMe protocol is featured for its low traffic overhead. We observed that only 0.18% of the overall traffic over the PCI Express are used for protocol commands, PRDTs and notifications used to synchronize the header and tail of the queues. Figure 4.17 shows the breakdown of the NVMe protocol traffic.

![Figure 4.17: The traffic overhead of the NVMe protocol.](image)

As Figure 4.17 shows, the largest overhead is coming from submission commands, then from completion entries written by the NVMe controller to the completion queues. We can see that the PRDTs along with reading/writing the doorbell registers (for synchronization and notification) are minor contributors to the protocol overhead. Reducing the overhead of the submission commands and completion notifications can be achieved by submitting
I/O requests with large sizes, but this depends on the time when the device driver receives the I/O requests, which affects the possibility of merging multiple requests by the device driver.

4.6 Discussion

In this section, we discuss the most important observations and findings from Section 4.5.

Our first counterintuitive observation is that using multiple I/O submission queues does not necessarily outperform a single I/O submission queue. We discussed how the queuing model can affect the DRAM row buffers efficiency and accordingly the overall performance. We validated our observation about the SQ and MQ performance experimentally using a real system with 24-cores (2 sockets of Intel E5-2695v2) and reading a 48 GB file from a 1.6TB NVMe device (Intel P3700 SSD), by comparing the performance when using SQ and MQ, the trend was similar to what our model shows. Our analysis also unveiled that the software layer is the largest contributor to the overall latency. Bypassing the unnecessary levels and optimizing the code of the software layer can help mitigating the software overhead, so it can be efficiently used with the emerging fast NVMs. DRAM buffering overhead is also a major contributor to the overall execution time, such an overhead can be mitigated by caching the I/O data in a special cache, for example in the memory controller hub (MCH) or directly writing it to the last level cache.

Using periodic interrupts with medium interrupt detection interval can enhance the performance; it reduces the number of times the processor is interrupted without significantly delaying the response time. Another implementation can be achieved by implementing a configurable-size interrupt buffer, which whenever it gets filled with the configured number
of interrupts, the controller interrupts the processor. This implementation no longer relies on a fixed period, but it still need to have a time out value to avoid having an interrupt waiting for very long time before being serviced. Finally, we observed that using polling can slightly enhance the performance when used with the fast emerging NVMs, such as PCM. However, using polling causes extraneous I/O bus traffic proportional to the NVM device latency, which can degrade performance significantly for high latency NVM devices.

4.7 Conclusion

In this paper, we have investigated different NVM host controller interfaces’ implementation aspects impact on system performance. We studied different queuing models, and explained different system bottlenecks and performance bottlenecks for each model. We also studied the impact of changing the NVM device latency, to foresee how replacing SSD or HDD with faster emerging NVMs affects the overall system performance. We observed that decreasing the NVM latency beyond 10 $\mu$s does not bring performance gain proportional to the speed of the NVM device, because the overall execution time is dominated by the I/O software layer and DRAM buffering. We studied the impact of using periodic interrupts instead of interrupt per completed command, which showed that using a medium interval for detecting interrupts can achieve a good trade off between using a large detection period and interrupt per completion. We also studied the impact of using polling vs interrupts while varying the NVM latency, which unveiled that using polling for high NVM latency devices (beyond 10 $\mu$s) can be highly inefficient and causes significantly higher I/O bus traffic. In contrast, using polling with very small NVM latency devices (1 $\mu$s) can enhance performance as it eliminates the interrupt overhead. We also investigated the traffic overhead of
the NVMe protocol and explained its breakdown. Finally, we suggested and discussed different performance tunes and optimizations which can mitigate the performance bottlenecks we studied.
5.1 Introduction

The big data era is pushing more and more computing and data storage into the cloud, raising increasing concerns about the privacy and security of user’s data. Cloud providers distribute their servers across geographically distributed areas with differing legal juris-
dictions and non-uniform security practices, e.g. [126]. Under this scenario, users’ data is vulnerable to being leaked, not just through software vulnerabilities, but also through physical attacks. Consequently, users and software makers increasingly demand stronger privacy and security guarantees from a secure trust base in hardware. In response, chip-makers including Intel [62] and AMD [67] are providing hardware support for basic security and privacy primitives such as enclaves and memory encryption.

Some of the most concerning physical attacks may focus on passive snooping or active tampering of the memory bus or scanning the DRAM or Non-Volatile Memory (NVM) chips. Attackers may observe the data stream as well as the access pattern revealed by the address stream on the memory bus as has been demonstrated [48, 55, 115, 165, 166]. This work focuses on the access pattern vulnerability.

Current solutions for obfuscating the access pattern are largely based on *Oblivious RAM* (ORAM) [45, 46]. ORAM relies on reshuffling the memory location of a datum (typically a cache block) after each access, such that observers cannot learn anything useful from the address stream of memory accesses. The state of the art hardware implementations of ORAM are Path ORAM [136] and its variants [34, 41, 115, 161]. While it promises better performance than other hardware implementations, Path ORAM still suffers from very significant practicality and overhead concerns. Path ORAM requires a block-level address translation table (called PosMap) and each block in memory associated with a path in a tree. Reading a memory block requires reading (and decrypting) all blocks in the path of the tree from the root to the leaf (100 blocks for 8GB memory), relocating the block to a new path, updating the PosMap, and writing back the block to the new path, in addition to any tree nodes that are evicted in the process. As a result, ORAM implementations incur
significant performance overheads: bandwidth increase by \(24\times\) and \(120\times\) in Ring and Path ORAM, respectively [115]. Such significant bandwidth overhead translates into significant execution slowdown. Furthermore, at least 50% of memory capacity is wasted through dummy blocks in order to achieve a reasonably acceptable failure rate \([34, 41, 115, 136, 161]\). Finally, even with 50% memory overhead, failure can still occur, where reshuffling cannot proceed because all buckets along a tree path are full, resulting in system deadlock.

In this chapter, we argue that the confluence of new memory technologies offers appealing opportunities for access pattern obfuscation in the memory bus. First, we observe that memory interfaces are becoming smarter, increasingly incorporating logic that receives request packets instead of specific memory commands. For example, DDR4 interfaces account for control logic inside the Dual-Inline Memory Modules (DIMMs) that deploy internal logic for buffering. Registered DIMMs \([1, 23]\) and Fully-Buffered DIMMs \([86]\) deploy internal logic to decode commands and addresses before sending them to memory chips. NVM memory controllers are expected to have logic to enable wear-leveling and internal buffering, and the logic is sometimes on chip \([4]\). The second technology change is 3D (and 2.5D) integration of memory. 3D and 2.5D stacked memory, such as Hybrid Memory Cube \([53]\), High Bandwidth Memory \([64]\), and Wide I/O 2 \([150]\), will have at least one logic layer \([18]\) containing memory controller logic, buffering, and interconnection. 3/2.5 D memory allows a sophisticated logic to be implemented on the same chip as memory banks. ORAM was designed with an assumption of conventional memories with very limited computational power. It is time to revisit this assumption.

3/2.5D integration of memory and its increasing logic capability present an opportunity to implement a simpler address stream obfuscation approach. In this chapter, we propose a
low-overhead access pattern obfuscation design, ObfusMem, which adds memory modules to the trusted computing base. Instead of reshuffling the memory locations continuously to obfuscate the memory access pattern as in ORAM, ObfusMem encrypts commands and addresses on the memory bus, hence the access pattern is cryptographically obfuscated from external observers.

Designing ObfusMem presents several challenges. First, in ORAM, read and write are indistinguishable. However, an additional mechanism to obfuscate the type of memory requests is needed in ObfusMem. Second, a realistic memory system has multiple memory channels, so we need to obfuscate the access pattern over multiple channels. We discuss these challenges and solutions for them. We also analyze the security protection of ObfusMem and compare it to that of ORAM, and point out their similarities and differences.

To evaluate our design, we implement ObfusMem on gem5, a full-system cycle-accurate simulator, and run several memory intensive workloads from SPEC 2006. Our evaluation shows that the ObfusMem performance overhead is only 10.9% on average, inclusive of the overheads from regular memory encryption and communication authentication. ObfusMem also does not increase the wearout of the memory (if NVM). We believe that ObfusMem's simplicity, low performance overheads, and zero memory overheads, make it an attractive alternative to ORAM.

The rest of the chapter is organized as follows. Section 5.2 discusses the threat model, logic in memory, and overview of ORAM. Section 5.3 discusses ObfusMem trust architecture and design. Section 5.4 describes our evaluation methodology, while Section 5.5 presents quantitative evaluation of ObfusMem. Section 5.6 discusses security analysis and comparison of ORAM and ObfusMem. Section 5.7 discusses the most relevant work. Finally,
Section 5.8 concludes this work.

5.2 Background and Assumptions

5.2.1 Threat Model

Our threat model is similar to that of much previous work in secure processor studies [8, 27, 34, 115, 153, 159, 161]. We assume an attacker can passively observe and actively tamper with the data stream and access pattern (read vs. write, and address stream) on the bus connecting the processor chip and memory chips. Our threat model also includes any change of data value in memory not caused by legitimate store instructions, including the row hammer attack [21, 123]. In addition, the attacker can physically obtain any of the memory chips and scan their contents. Most of these attacks require physical access to the computer systems, which is a realistic possibility in geographically-distributed data centers, with very significant consequences if the described attacks occur and succeed.

On the memory side, we assume that the logic layer of the 3/2.5D memory system is not susceptible to physical attacks. This is the same assumption applied to the processor chip in most secure processor studies [8, 27, 34, 115, 153, 159, 161]. We also assume that all internal wires of processor and memory modules are well-buried inside the silicon wafer and difficult to probe. This is achievable due to the advances in packaging technologies, 3D stacking and Through Silicon Vias (TSVs) technologies.

With such threat models, the minimum requirements for security include encryption and integrity verification of data stored in memory and communicated on the processor-memory bus, as well as obfuscation of the access pattern that appears on the processor-
memory bus. The former requires traditional memory encryption and Merkle Tree integrity verification [117], and the latter is provided by our ObfusMem.

Finally, memory accesses and communication with the processor leak side channel information such as power, timing, and electromagnetic signals. We consider them out of the scope of this work. We point out that there is a wealth of studies in literature that discuss these attacks and/or solutions to them [15, 51, 57, 58, 71, 91, 92, 151].

5.2.2 Logic in Memory Modules

Due to signal integrity, complexity of memory controller tasks and scalability, memory modules deploy some control logic inside the memory modules. For instance, many large capacity DDR4 modules have logic inside each DIMM with registers that can be configured by the memory controller [1, 23]. For performance scalability, many servers use a new industry standard memory module, Load-Reduced DIMMs (LR-DIMMs). In LR-DIMM, an isolation buffer is used to buffer the data and commands inside the DIMM [87].

![Diagram of memory module implementation]

Figure 5.1: An example implementation of PCM-based DIMM [109].
This trend is expected to become more common for future memory technologies. For example, Non-Volatile Memory (NVM) products, such as Intel 3D XPoint, have TBs of capacity and write endurance problem, requiring logic for wear leveling, scheduling, and failure remapping logic that need to be incorporated into the NVM, rather than in the processor-side memory controller. A patent application from Intel contains a figure where a PCM memory controller includes logic with such capabilities [109], as shown in Figure 5.1. Similar logic can also be observed in recent PCM prototypes [4].

Another important trend is 3/2.5D stacked memory, which not only has substantial logic budget on a separate die, but also bury this logic die layer under memory die layers [18]. The communication between logic and memory in the stacked memory system all occur within the chip, unexposed to attackers. In this context, only the communication between the processor chip and the memory chip is transmitted on exposed wires and is vulnerable to attacks. The logic layer is expected to have moderate power budgets: up to 42 W for commodity servers and up to 55 W for high-end servers [37]. This presents an opportunity for simplifying access pattern obfuscation.

### 5.2.3 Access Pattern Obfuscation

While secure processors encrypt memory contents [8, 27, 153, 159], memory devices require memory addresses to be transmitted plainly over the memory bus. An attacker with physical access to the computer can snoop the memory bus and observe the access pattern (address stream and type of requests) [55], which was shown to leak control flow and data leading to leaking cryptographic keys [166]. Obfuscating the access pattern is costlier than obfuscating data. The state of the art method for this is Oblivious RAM (ORAM).
ORAM relies on reshuffling the memory location of a datum (typically a cache block) after each access, such that observers cannot learn anything useful from the address stream of memory accesses. The state of the art hardware implementations of ORAM are Path ORAM [136] and its variants [34, 41, 115, 161]. Path ORAM requires a address translation table (called PosMap) that translates a physical address to an actual address in memory, similar to a page table but operating at the block level. When memory is accessed, a PosMap lookup reveals the mapping of the physical address to the actual memory address. After the memory access completes, the block is randomly assigned a new memory address. A tree is used to assign the new memory address. Each memory block is assigned to a leaf of the tree, ensuring a unique path from the root of the tree to the block at the leaf. When a block’s memory address is reshuffled, it is re-assigned from one tree path to a new randomly chosen tree path. The PosMap is then updated to reflect the new memory address. Each node contains a bucket of blocks, some real and some dummy. Path ORAM ensures that the following invariance is maintained [136]:

If a block is mapped to leaf \( l \), then it must be either in some bucket on path \( l \) or in the stash. Blocks are stored in the stash or ORAM tree along with their current leaf and block address.

When there is a memory access, all blocks along the path are read and decrypted into the *stash* (a secure temporary storage in the processor chip). The path length determines the number of blocks read for each memory access, and is dependent on the size of the memory, e.g. 100 for 8GB memory. All the real blocks are added to the stash. After the block is assigned a new leaf, as many blocks in the stash from the old path are evicted and encrypted. Any remaining space is filled with encrypted dummy blocks.
In short, ORAM obfuscates the access pattern in two ways. First, the memory location of blocks get reshuffled after each access. Second, dummy blocks may be read along with the intended block to provide additional obfuscation. Clearly, the deployment of ORAM comes at the cost of many additional memory accesses, due to fetching blocks from a tree path, and the additional address translation. Much efforts have been made to reduce the overhead of ORAM. However, even the state-of-the-art designs, in addition to the multiple times execution slowdown, Ring ORAM and Path ORAM incur a bandwidth overhead of $24 \times$ and $120 \times$, respectively [115]. Furthermore, at least 50% of memory capacity is wasted in order to achieve a reasonably acceptable failure rate [34, 41, 136, 161]. Finally, even with 50% memory overhead, failure can still occur, where reshuffling cannot proceed because all buckets along a randomly chosen new tree path are full, resulting in deadlocking the entire system.

When the memory is non-volatile, there are additional problems. When we consider Non-Volatile Memory (NVM), the cost and practicality concern increase. In NVMs, writes are expensive because NVM cells have limited write endurance (a few hundred million writes for PCM cells), a write takes much more time to perform than a read, and a write takes much higher energy to perform than a read. Path ORAM and other ORAM implementations cause write amplification: every access to memory whether it is a read or a write results in reading a whole path of blocks (about 100 cache blocks for 8GB memory for $L = 24$ and $Z = 4$) and then writing them back after reshuffling [136]. This write amplification may reduce the memory lifetime by orders of magnitude, significantly increase power consumption, and slow down execution time.
5.2.4 Memory Encryption

Depending on the threat model assumed, there are two approaches for encrypting data in memory. In the i-NVMM approach [27], only data remanence attacks were assumed. Hence, NVM is designed to self encrypt to match the non-volatile retention time of DRAM. In another approach [8, 153, 159], passive bus snooping, physical bus traffic tampering, and memory content readout and modifications were assumed. In this case, data in memory is encrypted at all time and only the processor chip is trusted. Any data sent off chip to the main memory is encrypted. Any data brought into the processor is decrypted and its integrity is verified.

![Counter Cache Diagram]

Figure 5.2: Example of counter mode memory encryption [8, 153].

The state of the art memory encryption uses counter mode encryption [153], where the
encryption algorithm is applied to an *initialization vector* (IV) to generate a one-time pad. This is illustrated in Figure 5.2. Data is then encrypted and decrypted via a simple bitwise XOR with the pad. The decryption latency is overlapped with the LLC miss latency, and only the XOR latency is added to the critical path of an LLC miss. In state-of-the-art design, the IV of a counter mode encryption consists of a unique ID of a page (similar to a page address, but is unique across the main memory and swap space in the disk), page offset (to distinguish blocks in a page), a per-block *minor* counter (to distinguish different versions of the data value of a block over time), and a per-page *major* counter (to avoid overflow of counter values).

In relation to the ObfusMem, we note that access pattern may leak through data. For example, when data contain pointers, and the program flow chases pointers, then the access pattern leaks through data. It is possible to guess with relatively good accuracy whether a datum contains an address or not [31]. Therefore ObfusMem, like ORAM [34, 41, 115, 136, 161], uses counter-mode processor-side memory encryption of data items. For the rest of the chapter, we assume counter mode (processor-side) memory encryption, similar to earlier papers [117, 153, 159].

### 5.3 ObfusMem Design

#### 5.3.1 Trust Architecture

An ORAM system’s processor is trusted, meaning that we rely on the correct operation of the processor to ensure ORAM’s security and privacy characteristics. ObfusMem relies not only on the correct operation of the processor, but also on the logic of the memory-side
memory controllers to perform essential cryptographic operations. Thus, ObfusMem's 
trusted computing base (TCB) includes both the processor and the memory controller. The 
manufacturers of both components are trusted. The rest of this section discusses how the 
trust architecture of an ObfusMem system is bootstrapped from its TCB.

First, we assume that the processor and the memory are both trustworthy components, 
in the sense that they are manufactured to the specifications and do not contain design 
errors and faults that may introduce security vulnerabilities. Next, the processor and mem-
ory must have unique identities that can be used to secure communication between them. 
To achieve this, the manufacturers of the processor and memory bind the processor and 
memory each to its own public key. In other words, the manufacturers act as the certifica-
tion authorities for these components. The processor and memory manufacturers do not 
need to coordinate key generation or binding with one another; they do not even need to 
know one another. Each manufacturer burns a public key and corresponding private key 
into its chip(s). However, only the public keys can be read out from the chip pins. If there is 
more than one memory chip, each chip is given its own public/private key pair.

In order to establish authenticated and private communication between processor 
and memory, each must learn the public key of the other. There are two approaches to 
achieve this. In the first approach, trust between the two components is bootstrapped 
during execution of a BIOS protocol in which the two components exchange public keys 
in the clear. However, this assumes the bootstrapping process to be isolated from any 
physical attacks. Since bootstrapping may occur at the user site (possibly at the same site 
as the operational site), the validity of this assumption is unclear, since it may be difficult 
to ensure such isolation. In the second approach, the system integrator programs the
processor's public key into the memory chips and the public keys of the memory chips into the processor chip. These keys can be burned into write-once non-volatile registers inside the processor and memory chips, possibly protected in Trusted Platform Modules (TPMs) of their respective chips. The system integrator is trusted to correctly program the public keys into the corresponding chips. In this trusted integrator approach, when the system is shipped to users, the processor and memory chips already know who they should communicate with to establish secure communication. A drawback of the approach is the difficulty in upgrading components after system integration, which requires the system integrator to be involved. There are clearly trade-offs between the two approaches, but since ObfusMem is designed for environments in which hardware attacks are anticipated, we focus on the trusted integrator model.

The final step in establishing the ObfusMem trust architecture is for the processor and memory to establish an authenticated and private communication channel. During BIOS execution, the hardware controller in the processor will initiate a Diffie-Hellman key exchange [89] with each of the memory controllers in the memories to establish a different shared session secret key with each controller. Each shared secret key enables a private communication channel between processor and memory. If there is more than one memory chip, then the processor must keep track of multiple secret keys, one corresponding to each memory chip. The shared secret key enables a private, authenticated communication channel that persists until the system shuts down. When the system is re-booted, the BIOS execution results in a new shared secret key for the communication channel.

The ORAM TCB includes only the processor and not the memory, so the step of establishing authenticated, private communication between these components is not needed.
While ObfusMem requires a larger TCB with trust bootstrapping during BIOS execution, it offers the benefits of simplicity in obfuscating the access pattern, zero storage overheads, suitability for emerging memory technologies, and very low performance overheads.

### 5.3.2 Access Pattern Obfuscation

So far, we have described how a processor and memory establish a secure communication channel. Now, we discuss how the memory access pattern is encrypted in this channel. Several approaches can be used to encrypt the addresses and commands on the memory bus. The first approach uses a simple *Electronic Code Book* (ECB) encryption mode where encrypting address $X$ will result in address $Y$, where $Y = E_{Key}(X)$. ECB is sufficient for obfuscating spatial locality across blocks. Specifically, since $E_{Key}(X + 1) \neq Y + 1$, observers cannot learn the spatial access pattern of the program. However, ECB has several security vulnerabilities. First, the temporal access pattern can be inferred over time, because if attackers find two requests showing the same ciphertext address, they can infer that the two requests share the same plaintext address. Second, ECB reveals the memory footprint of the application as the number of unique addresses appearing on the bus reveals the actual number of used cache blocks. Finally, dictionary attacks can be deployed to guess the plaintexts of addresses by matching the frequency distribution of accesses to different addresses in plaintext and ciphertext forms.

A second approach uses non-ECB memory encryption modes. While many modes are possible, *counter mode* is especially appealing because we can pre-generate encryption pads for future use, since future counter values are known ahead of time. In an AES-CTR mode, an *Initialization Vector*(IV) is encrypted to generate a pad. The IV is implemented as
a single use counter called a *nonce*. Later, the generated pad is XORed with the address and command. The resulting ciphertext $Y = E_{Key}(IV) \oplus X$. The IV (counter) changes after every memory access, as shown in Figure 5.3(a). Note that a 64-bit counter is large enough that it will not overflow for millennia.

The figure shows how a request resulting from the last level cache (LLC) write back, consisting of command and address, and data, is obfuscated. Data is passed to the processor encryption (Step 1a) to obtain its ciphertext. In the case of LLC read/write miss, there will not be data involved. In parallel, the address of the request is used to index the Session Key Table to extract the session key for the memory chip which will be handling this request (Step 1b). The command and address themselves are also passed to the next step (Step 1c), while a dummy request generator generates a matching request consisting of dummy command and address, and dummy data (Step 1d – to be explained later). The session key is fetched from the table and is used as an encryption key for the encryption engine (Step 2). The counter value $Ctr$ and subsequent $Ctr + 1, Ctr + 2, \ldots, Ctr + 5$ were generated and input to the encryption engine to generate six 128-bit pads (Step 3). The reason for six pads is that one pad will be used to encrypt the original request address by XORing the pad and command and address (Step 4a), one pad will be used to encrypt a dummy request (Step 4b), and four pads (16-byte each) are used to encrypt the 64-byte block encrypted data (Step 4c). Finally, the counter is increased by six to prepare it for the next encryption (Step 4d).

Note that in ObfusMem, the processor-encrypted data is encrypted a second time before it is sent out to the memory. If we were to skip this second encryption, when a block is fetched multiple times without being modified, the block’s ciphertext would appear repeatedly.
Figure 5.3: Part (a): ObfusMem access pattern obfuscation for a write request (a write back of the LLC block). A read request due to read/write miss of the LLC is handled similarly in the reverse direction (not shown). Part (b): Illustration of dummy request generation.
on the memory bus. Observers could infer temporal reuse of data: if the same ciphertext occurs on the bus on two different requests, even though the obfuscated addresses are different, there is a higher probability that their original addresses are the same, especially considering that much data in memory is read only. By encrypting data for the second time, even unmodified data will appear different every time it appears on the memory bus.

At the memory side, encrypted data sent from the processor is decrypted. The session key and counter values are used to generate pads. One pad is used to decrypt the command and address of the original request (Step 5a), the dummy request is discarded (Step 5b – to be explained later), and the data is decrypted using four pads (Step 5c). Finally the counter is incremented by six to keep it synchronized with the processor-side counter and to prepare it for the next decryption (Step 5d).

5.3.3 Obfuscating the Type of Memory Requests

Revealing the memory operation type, read or write, is also a potential security vulnerability. In typical Path ORAM designs, the operation types are indistinguishable because every access is treated equally by reading a path and then filling the path from the stash. While we are not aware of security attacks solely based on the type of memory requests, we cannot rule out observers improving their odds of attack success using knowledge of the request type. To address this, ObfusMem piggybacks every read operation with a *dummy write* operation. Similarly, every write operation is preceded by a *dummy read* operation. This is illustrated in Step 1d in Figure 5.3(a). Accordingly, every access now appears as a *read-then-write* request to an external observer. We could have chosen *write-then-read* instead of read-then-write, however, the latter has a distinct performance advantage: write operations
are typically due to block evictions from Last-Level Cache (LLC) and hence are not in the critical path delay of instructions executing in the processor pipeline. In contrast, reads are usually in the critical path, hence read-then-write for a read operation can proceed much faster and return the data to the LLC once fetched from memory.

Another design question with the dummy requests is what address the dummy request should access. In one possible design, we can select a random address to attach to the dummy request. However, each additional address being accessed consumes energy and reduces performance due to the loss of temporal locality. In a second alternative design, we can exploit the fact that ObfusMem uses counter mode encryption, hence even the same address will appear different at different encryption instances. Thus, we can use the same address for the dummy request as was used in the original request. This way, temporal locality is not reduced and at the same time, observers still cannot distinguish between actual and dummy requests. Unfortunately, even the latter design is expensive if the main memory is NVM: every read now incurs an actual write to the NVM, reducing the lifetime of the NVM and increasing energy usage. To overcome this problem, we choose a third design, where we set aside a location on the memory to hold the dummy (illustrated in Figure 5.3(b)). Specifically, every memory module will reserve one 64-byte block as dummy. Note that the observer cannot infer that the request is a dummy because, once encrypted with counter mode, the dummy address appears different each time due to the change of the encryption pad. However, when the memory module decrypts the request and finds that the address matches the reserved dummy address, it discards the actual writing of the data to the dummy location (illustrated in Step 5b of Figure 5.3(a)), hence saving write energy and avoiding premature wear-out. If instead the request is a dummy read, random
data can be returned and discarded at the processor.

5.3.4 Obfuscating Inter-Channel Access Pattern

So far, our discussion has assumed a single memory channel. However, a realistic 3/2.5D stacked memory may have multiple channels, where channels operate independently and use different pins. For example, High Bandwidth Memory allows 4-8 independent channels using separate pins [64]. While our discussion in this section is not specific to a particular 3/2.5D stacked memory, we will address the security challenge that comes from having multiple memory channels.

While it may be counter-intuitive, the access pattern may leak across memory channels, because obfuscation in ObfusMem is performed independently on a channel. This is due to addresses being interleaved across channels. Address interleaving may be designed differently in different systems depending on requirements for memory level parallelism, target bandwidth, and spatial locality. For instance, on the fine grain side, physical cache blocks $X$, $X+1$ and $X+2$ may be interleaved across channels 0, 1 and 2, respectively. In this case, an attacker can easily infer spatial locality by observing accesses being sent out to channel 0, 1, and 2, in succession, without knowing anything about addresses or data. Access obfuscation per channel only obfuscates accesses to that channel. Accesses to different channels appear on different physical pins so they cannot be obfuscated easily. Attackers can directly snoop these pins and observe which channels receive requests, and in which order the requests are received.

One approach to eliminating the cross-channel security vulnerability is to generate dummy requests on other channels. To illustrate, suppose that there are four channels
and that there is a request on channel 2. Dummy requests can be created on channel 0, 1, and 3, so that observers cannot tell which channel is receiving an actual memory access request. However, note that this full channel dummy replication scheme is expensive, and the cost increases linearly with the number of channels. Due to the cost, we explore another approach. We note that fundamentally, observers cannot infer spatial locality if all the channels are accessed simultaneously and with equal number of times. Viewed from this criteria, the full channel dummy replication is unnecessary. If other channels already have requests, there is no need to generate a dummy request for them. In other words, dummy requests are only needed for idle channels. With this idle channel dummy replication scheme, at any time there is a memory request to be serviced on one channel, the processor-side ObfusMem controller checks the status of all other channels and injects dummy requests on the idle ones.

Note that with the idle channel dummy replication scheme, dummy requests are generated when channels are idle, and few requests are generated when channels are busy. Thus, no extra penalty is paid when bandwidth is very limited. As with other types of dummy requests, unique address values enable dummy writes to be discarded at the memory side.

### 5.3.5 Integrity Verification of Processor-Memory Communication

So far, we have discussed access pattern obfuscation when the attackers’ only capability is to passively observe anything communicated between the processor and memory. However, an attacker may have an additional active capability for communication tampering via changing the request type, address, or data of the request. The attacker may drop a request completely, inject a bogus request, or replace a request with a bogus one or replay a valid
request from the past. Address stream obfuscation makes meaningful attacks more difficult because the attacker does not know the type, address, or data of a request that can lead to accessing a specific location. Also an attacker cannot easily select a request for injection whose content will be meaningful when decrypted. If a request or data reply is removed, counters in the processor and memory may not match, preventing the processor and memory from any further meaningful communication. Since ObfusMem uses counter mode encryption and counters are never reused, valid requests from the past cannot be replayed meaningfully either. Thus, an attacker’s impact is limited to a denial of service (DoS) by sabotaging ObfusMem. Protecting ObfusMem against DoS attacks is a challenge left for future work.

While a successful active attack against an ObfusMem-protected system is difficult, detecting the incidence of attempted active attacks is valuable to protect user data from a determined attacker that will seek alternative avenues of attack. We now consider attack detection. Note that first, with counter-based memory encryption, it is common to protect data (or counters) using a Merkle Tree [117]. A Merkle Tree is sufficient to detect tampering of data, but not tampering of communication between the processor and memory. The standard mechanism for protecting a message from tampering is to use a message authentication code (MAC), which is a one-way hash function, such as MD5 [116] and SHA-1 [36]. In our application, the message sent from/to the processor to/from the memory is the encrypted memory request. Mathematically, the message is $M = E_K \oplus (r | a | D)$, where $E$ is encryption, $K$ is the encryption key, $r$ is the request type, $a$ is the request address, and $D$ is the processor-encrypted data block of the request. If the cryptographic MAC function is $H$ and $\alpha = H(M)$, then what the processor sends to the memory is $M$ and $\alpha$. Upon receiving
this, the memory computes its own $H(M)$ and compares it to $\alpha$. Any tampering to $M$ or $\alpha$ would be discovered. Note, however, this standard mechanism adds a significant latency to the critical path of the request to memory, and also the reply from memory, as the encryption must be completed before the MAC can be computed. Thus, we design an alternative approach.

In the alternative approach, the MAC function is not applied over the message, but applied to the plaintexts of components that make up the message. In this case, the MAC is $\beta = H(r|a|c)$. This MAC can be computed early since its components are available early. For example, a dirty block near the least recently used (LRU) position in the LLC can be anticipated to generate a memory write request soon. A stream or stride predictor can be used to anticipate future LLC read/write miss, which will generate memory read request. So, we can compute $\beta$ before the memory request is generated, allowing overlap of request encryption and MAC generation. At the memory side, when $M$ and $\beta$ are received, it can decrypt $M$ to obtain $r$ and $a$, and use its own counter $c$ to recompute $H(r|a|c)$. If it does not match the received $\beta$, then tampering has been detected.

Now let us examine several tampering scenarios. In the first scenario, the attacker changes the message to $M'$. This will result in the memory decrypting $M$ and obtaining a wrong request $r'$ or address $a'$. When the memory recomputes the MAC, it will calculate $H(r'|a|c)$ or $H(r'|a'|c)$, which does not match $\beta$, so tampering is detected. If the attacker deletes a request or a reply, the counter in the processor and memory will not match. Suppose that the processor counter is $c$ while the memory counter is $c'$. In this case, the memory computes $H(r|a|c')$ which does not match $\beta$, so the tampering is also detected. Finally, if the attacker replays an old valid message, the computed MAC will again fail
to match \( \beta \) because the memory will use its fresh counter value to compute the MAC, while \( \beta \) reflects a stale counter value. Overall, any types of tampering will be detected, unless the system crashes prior to detection. Because the attacker must know the original components of the encrypted command to successfully tamper with communication between the processor and memory, a lightweight MAC function is sufficient to detect tampering. We assume MD5 in our implementation, however other MAC functions can also be used.

5.4 Methodology

To evaluate ObfusMem, we constructed a Gem5-based full-system cycle-accurate simulator [16]. To enable performance comparison with ORAM, we chose representative benchmarks from the SPEC CPU2006 suite [134]. The characteristics of the benchmarks, such as Instructions Per Cycle (IPC), Last-Level Cache Misses Per Kilo Instructions (LLC MPKI) and average latency gap between consecutive memory requests are shown in Table 5.1. Eight of the benchmarks have average memory request gaps of less than 100ns, indicating very high memory request rates, while seven of the benchmarks have average memory request gaps higher than 100ns. We warmed-up the simulator and fast-forwarded the workloads to skip the initialization phase, then simulated 200 million instructions.

To ensure a fair comparison of ObfusMem to ORAM, we model ORAM performance optimistically by choosing aggressive assumptions for ORAM implementations. We assume a base Path ORAM of 25 levels, 4 blocks per bucket and a capacity waste of 50%, similar to previous studies [34, 161]. For ORAM, we assume a fixed memory access latency of 2500 ns, obtained by extrapolating the ORAM access latency from [41]. Our latency assumption is
Table 5.1: Characteristics of the evaluated benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>IPC</th>
<th>LLC MPKI</th>
<th>Avg Gap (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bwaves</td>
<td>0.59</td>
<td>18.23</td>
<td>44.32</td>
</tr>
<tr>
<td>mcf</td>
<td>0.17</td>
<td>24.82</td>
<td>74.95</td>
</tr>
<tr>
<td>lbm</td>
<td>0.35</td>
<td>6.94</td>
<td>67.97</td>
</tr>
<tr>
<td>zeus</td>
<td>0.53</td>
<td>4.81</td>
<td>63.56</td>
</tr>
<tr>
<td>milc</td>
<td>0.42</td>
<td>15.56</td>
<td>51.54</td>
</tr>
<tr>
<td>xalan</td>
<td>0.52</td>
<td>0.97</td>
<td>945.62</td>
</tr>
<tr>
<td>omnetpp</td>
<td>4.30</td>
<td>0.10</td>
<td>1104.74</td>
</tr>
<tr>
<td>soplex</td>
<td>0.25</td>
<td>23.11</td>
<td>69.06</td>
</tr>
<tr>
<td>libquantum</td>
<td>0.33</td>
<td>5.56</td>
<td>146.82</td>
</tr>
<tr>
<td>sjeng</td>
<td>0.95</td>
<td>0.36</td>
<td>1382.13</td>
</tr>
<tr>
<td>leslie3d</td>
<td>0.49</td>
<td>9.85</td>
<td>58.91</td>
</tr>
<tr>
<td>astar</td>
<td>0.70</td>
<td>0.13</td>
<td>5660.18</td>
</tr>
<tr>
<td>hmmmer</td>
<td>1.39</td>
<td>0.02</td>
<td>2687.60</td>
</tr>
<tr>
<td>cactus</td>
<td>1.05</td>
<td>1.91</td>
<td>128.09</td>
</tr>
<tr>
<td>gems</td>
<td>0.40</td>
<td>11.66</td>
<td>66.25</td>
</tr>
</tbody>
</table>

aggressive, as we include the process of reading and evicting a full tree path to the memory assuming unlimited bandwidth and unconstrained phase-change memory (PCM) write power budget. Note that the ORAM access latency depends on the memory layout, the probability of finding a block in the leaf, applying bandwidth reduction techniques (e.g., XOR) and many other parameters (bucket metadata size, row buffers locality, etc.), hence our optimistic estimate.

The machine configuration is shown in Table 6.1. We model a 4-core processor with a three-level cache hierarchy and a DDR-interfaced PCM memory with PCM parameters from [76]. Similar to the design in [76], writes to PCM cells are incurred only when evicting data from dirty row buffers.

Note that memory encryption is orthogonal to access pattern obfuscation and is re-
quired in all protected systems, including ORAM. In both cases, we overlap fetching data with encryption pad generation.

Table 5.2: Configuration of the simulated system.

<table>
<thead>
<tr>
<th>Processor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4 core, each 2GHz, out-of-order x86-64</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>private, 2 cycles, 32KB, 8-way, 64B block</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>private, 8 cycles, 512KB, 8-way, 64B block</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>shared, 17 cycles, 8MB, 8-way, 64B block</td>
</tr>
<tr>
<td>Coherence</td>
<td>MESI protocol</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DDR-based PCM Main Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>8 GB</td>
</tr>
<tr>
<td># Channels</td>
<td>1 (base), 2, 4 and 8;</td>
</tr>
<tr>
<td>Channel bw</td>
<td>12.8 GB/s</td>
</tr>
<tr>
<td>PCM Latencies</td>
<td>60ns read, 150ns write, based on [76]</td>
</tr>
<tr>
<td>Organization</td>
<td>2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address mapping</td>
</tr>
<tr>
<td>DDR Timing</td>
<td>tRCD 60ns, tRP 150ns, tBURST 5ns</td>
</tr>
<tr>
<td></td>
<td>tCL 13.75ns, 64-bit bus width, 800 MHz Clock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operating System</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Gentoo, Linux, kernel v2.6.22.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encryption Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter Cache</td>
<td>5 cycles, 256KB size, 8-way, 64B block</td>
</tr>
</tbody>
</table>

For the memory encryption, we model a counter cache with 5-cycle access latency and a size of 256KB. For ObfusMem, we model ObfusMem’s AES-CTR latency based on our synthesize results using 45nm libraries and a publicly available pipelined AES-128 implementation [102]. The overall AES encryption latency is 24 cycles (cycle time of 4ns) and the AES unit can produce a 128-bit on each cycle. The AES engine consumes 15.1 mW. The required area is 0.204 mm². Each channel requires an AES engine on both sides
(processor and memory). To enable detection of communication tampering on the channel, we use a publicly available 64-stage pipelined implementation of MD5 [101]. Our synthesize results show that the MD5 unit consumes about 12.5 $mW$ and has an area overhead of 0.214 $mm^2$.

## 5.5 Evaluation

### 5.5.1 Performance Overhead

ObfusMem and ORAM provide similar protection to the memory bus, hence our first evaluation results compare their overhead for such protection. Table 5.3 shows the execution time overheads of ORAM and ObfusMem with authenticated communication (ObfusMem+Auth) over unprotected execution, and the speedup ratio of ObfusMem+Auth over ORAM. We can see that the overheads of ORAM vary across benchmarks depending on their LLC MPKI: benchmarks with high LLC MPKI suffer from large ORAM execution time overheads. Across all of the benchmarks, ORAM adds 946.1% to the execution time of programs, or a slowdown ratio of slightly over one order of magnitude. This is in line with results reported for the common benchmarks in the literature [41]. In contrast, ObfusMem+Auth adds 10.9% to the execution time, on average, and 32.1% in the worst case. Note that these overheads already include counter-mode memory encryption, ObfusMem's access pattern obfuscation, and authenticated memory bus communication. ObfusMem+Auth achieves speedups ranging from $1.3 \times$ to $17.1 \times$, with an average of $9.1 \times$. So overall, ObfusMem+Auth is almost one order of magnitude faster than ORAM. This very substantial improvement makes a significant difference in the feasibility of incorporating obfuscation of access patterns in general.
purpose computer systems.

Table 5.3: Execution time overhead comparison of ORAM vs. ObfusMem

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ORAM</th>
<th>ObfusMem+Auth</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>bwaves</td>
<td>1561.0%</td>
<td>18.9%</td>
<td>14.0×</td>
</tr>
<tr>
<td>mcf</td>
<td>1133.3%</td>
<td>32.1%</td>
<td>9.3×</td>
</tr>
<tr>
<td>lbm</td>
<td>1298.6%</td>
<td>12.5%</td>
<td>12.4×</td>
</tr>
<tr>
<td>zeus</td>
<td>1644.3%</td>
<td>14.9%</td>
<td>15.2×</td>
</tr>
<tr>
<td>milc</td>
<td>1846.6%</td>
<td>28.4%</td>
<td>15.2×</td>
</tr>
<tr>
<td>xalan</td>
<td>137.7%</td>
<td>0.8%</td>
<td>2.4×</td>
</tr>
<tr>
<td>omnetpp</td>
<td>64.96%</td>
<td>1.2%</td>
<td>1.6×</td>
</tr>
<tr>
<td>soplex</td>
<td>1878.6%</td>
<td>15.7%</td>
<td>17.1×</td>
</tr>
<tr>
<td>libquantum</td>
<td>604.8%</td>
<td>2.9%</td>
<td>6.8×</td>
</tr>
<tr>
<td>sjeng</td>
<td>152.5%</td>
<td>1.1%</td>
<td>2.5×</td>
</tr>
<tr>
<td>leslie3d</td>
<td>1626.6%</td>
<td>15.1%</td>
<td>15.0×</td>
</tr>
<tr>
<td>astart</td>
<td>30.7%</td>
<td>0.1%</td>
<td>1.3×</td>
</tr>
<tr>
<td>hmmmer</td>
<td>86.6%</td>
<td>0.0%</td>
<td>1.9×</td>
</tr>
<tr>
<td>cactus</td>
<td>784.8%</td>
<td>5.2%</td>
<td>8.4×</td>
</tr>
<tr>
<td>gems</td>
<td>1340.9%</td>
<td>14.3%</td>
<td>12.6×</td>
</tr>
<tr>
<td><strong>Avg</strong></td>
<td><strong>946.1%</strong></td>
<td><strong>10.9%</strong></td>
<td><strong>9.1×</strong></td>
</tr>
</tbody>
</table>

Next, Figure 5.4 breaks down the execution time overheads of ObfusMem into various levels of security, starting from memory encryption only without access pattern obfuscation, plain ObfusMem, and ObfusMem with authenticated communication. The figure shows that roughly a quarter of ObfusMem overheads come from the memory encryption (2.2% vs. 8.3%). Furthermore, communication authentication increases the overhead of ObfusMem only slightly, from 8.3% to only 10.9%, as authentication can be partially overlapped with encryption.

As discussed earlier, one of the key design aspects of ObfusMem is to obfuscate the
Figure 5.4: The execution time overhead of ObfuscMem, normalized to unprotected system.

inter-channel access pattern, through dummy request injection for every memory access (ObfuscMem-UNOPT) and only on idle channels (ObfuscMem-OPT). The execution time overheads of these two injection strategies are evaluated with various numbers of channels, with and without communication authentication. Figure 5.5 shows these results. The overhead of ObfuscMem-UNOPT reaches up to 18.8% and 16.3% (with eight memory channels), with and without authentication, respectively. In contrast, the overhead of ObfuscMem-OPT reaches up to 13.2% and 10.1% with and without authentication, respectively. The gap between the non-optimized and optimized versions, as expected, declines as the number of memory channels declines.
Figure 5.5: The impact of the number of channels on ObfusMem performance, compared to unprotected system with equal number of channels.
5.5.2 Impact on Memory Energy and Lifetime

One of the key challenges of main memory NVMs is the limited and energy consuming writes. However, a typical Path ORAM implementation is based on reading a whole tree path that consists of about 100 blocks (assuming $L = 24$ and $Z = 4$) and then evicting it later. Assuming PCM cells’ write energy is $6.8 \times$ the read energy [76], a basic ORAM implementation may incur $(1 + 6.8) \times 100 = 780 \times$ the read energy, while ObfusMem incurs only $\frac{1 + 6.8}{2} = 3.9 \times$ the read energy, assuming 50:50 read:write distribution (a 200× PCM energy reduction). More importantly, ObfusMem does not incur any extra writes, which can improve the lifetime of the system by roughly 100× compared to the base ORAM, which requires 100 blocks to be evicted after each access, regardless of the access type (read or write).

However, the savings are not only limited to the memory read/write energy, but should also include the encryption energy. The basic ORAM requires 100 blocks to be read and decrypted, then later encrypted and written back. This costs $200 \times 4 = 800 \times$ the energy for 128-bit encryption. In contrast, ObfusMem requires two 128-bit address encryption pads (dummy and real), and four 128-bit encryption pads for data point-to-point encryption, in addition to the four 128-bit pads used for data encryption. This adds up to ten 128-bit pads for the processor side of the channel and six 128-pads for the memory side, i.e., 16 pads per channel. Even though having a high density NVM DIMM will reduce the number of required channels, having a 4-channel system will require 64 128-bit pads in the worst case, which is 12.5× fewer than that required for a typical ORAM (800 128-bit pads). Note that in the best case, ObfusMem only requires 16 128-bit pads (a 50× reduction in the number of pads); other channels are already busy with real requests.
5.6 Security Analysis and Discussion

5.6.1 Security Analysis

We will now analyze the security protections provided by ObfusMem and compare them with those provided by ORAM. The leakage of multiple types of access pattern information may aid attackers in their attacks, including spatial patterns, temporal patterns, read vs. write, and memory footprint. We will discuss each of these in greater detail, with a summary of the comparison shown in the top half of Table 5.4. Spatial pattern refers to the pattern of accesses to adjacent or non-adjacent memory blocks of a program. ORAM obfuscates the spatial pattern exhibited by a program by reshuffling the locations of a data block each time it is accessed. ObfusMem obfuscates the spatial pattern by encrypting addresses with counter-mode encryption. In the table, we indicate that both ObfusMem and ORAM provide full spatial pattern protection.

Temporal pattern refers to the pattern of block reuse over time. ORAM partially leaks the temporal pattern because as data is read, its entire tree path (e.g. $P_1$) is read and then reshuffled to a new path (e.g. $P_2$). If the same block is accessed again by the processor, then we will see an access to the previous path $P_2$ and reshuffling to a new path (e.g. $P_3$), and so on. Observers can detect that $P_2$ appeared twice, once through eviction and again through fetching, indicating temporal reuse. Thus, the incidence of block reuse can be detected by ORAM. However, if the stash does not evict the old path right away, but, instead, keeps it on chip along with other paths for awhile, the temporal pattern may be less evident to an observer. In this case, when a path is evicted, that path may correspond to any of several recently read paths. However, this non-determinism in block reuse is implementation
dependent, rather than a guarantee provided by ORAM. In contrast, with ObfusMem, addresses are encrypted. Hence even if the same address is accessed over and over again, the accesses appear randomized in the memory bus. Thus, ObfusMem completely hides the temporal pattern, but ORAM partially leaks the temporal pattern.

Read accesses are indistinguishable from write accesses in both ORAM and ObfusMem: ORAM treats both access types the same way, by reading/writing from/to old/new paths, while ObfusMem inserts a dummy read/write for a write/read request.

Table 5.4: Comparing ORAM and ObfusMem.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>ORAM</th>
<th>ObfusMem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial pattern</td>
<td>Full</td>
<td>Full</td>
</tr>
<tr>
<td>Temporal pattern</td>
<td>Partial</td>
<td>Full</td>
</tr>
<tr>
<td>Read vs. write</td>
<td>Full</td>
<td>Full</td>
</tr>
<tr>
<td>Memory footprint</td>
<td>Partial</td>
<td>Full</td>
</tr>
<tr>
<td>Communication authentication</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TCB</td>
<td>Proc only</td>
<td>Proc+Mem</td>
</tr>
<tr>
<td>Exe time overheads</td>
<td>±10×</td>
<td>11%</td>
</tr>
<tr>
<td>Storage overheads</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Write amplification</td>
<td>100×</td>
<td>None</td>
</tr>
<tr>
<td>Deadlock possibility</td>
<td>Low</td>
<td>Zero</td>
</tr>
<tr>
<td>Component upgrade</td>
<td>Easy</td>
<td>Harder</td>
</tr>
</tbody>
</table>

The memory footprint may be used by attackers to infer information about the program. For example, the attacker may guess the number of data records or the graph size. ObfusMem hides the memory footprint by encrypting the addresses on the memory bus (i.e., attackers cannot distinguish whether an access is to a previously accessed block or to a new block. If the application makes \( n \) accesses, the possible memory footprint \( M \) is
bounded by $1 \leq M \leq n$. As the program’s run time increases, the accuracy of the attacker’s estimate of $M$ decreases. However, in Path ORAM and its variants, observing the number of unique paths read by the processor can reveal the memory footprint. Unique paths can be obtained by deducting the reads to previously evicted/written paths from the overall number of read paths. Let us denote the number of unique paths $U$ and the path length (in cache blocks) by $L$ ($L$ is typically 100). For a long running application, the memory footprint of the application is bounded by $L \leq M \leq U L$. Note that the bound is not dependent on the number of accesses that have been made by the application (unlike in ObfusMem), hence the memory footprint may be inferred even for long running applications. In theory, ORAM can generate a large number of dummy path accesses to eliminate this issue. However, this comes at the cost of extra memory bandwidth and increased probability of stash overflow.

As discussed in Section 5.3.5, ObfusMem can be equipped with communication authentication with roughly 2% additional overheads. None of the ORAM implementations we are aware of include communication authentication. Thus, ObfusMem can detect the incidence of tampering of memory requests, whereas ORAM cannot.

Finally, side channels may allow attackers to correlate system timing, power, thermal, or electromagnetic characteristics with the access pattern or data. Here, we briefly compare ORAM with ObfusMem on thermal side channels, but leave deeper analysis and other side channels for future work. That ObfusMem does not reshuffle data locations in the main memory is its key advantage (resulting in low overheads) but also makes it more vulnerable than ORAM to side channel attacks. Attackers may be able to thermally analyze the memory chips to infer which rank, bank, row, etc. are activated, leading to a leakage in spatial and temporal access patterns. This is more difficult to do in ORAM due to the
reshuffling. System designers should consider such tradeoffs.

5.6.2 Discussion

Summary comparison of ObfusMem to ORAM. Our comparison of ObfusMem to ORAM is summarized in the bottom half of Table 5.4. The Trusted Computing Base (TCB) of ORAM is the processor chip only vs. processor and memory chips in ObfusMem. The execution time of programs on ORAM is roughly one order of magnitude higher than unsecured processors, but is only 11% slower on ObfusMem. The storage overhead of ORAM is at least 100% (i.e. at least 50% of the memory capacity is unusable) vs. zero in ObfusMem (excluding the storage of counters needed for memory encryption which is not a part of ObfusMem). ORAM amplifies writes by about 100× which is harmful for NVM life time and energy consumption, versus zero write amplification for ObfusMem. Whole system deadlocks are possible in Path ORAM due to tree bucket overflows vs. zero possibility in ObfusMem. Memory component upgrade is easy in ORAM since the TCB only includes the processor. Memory component upgrade is harder with ObfusMem since the TCB includes memory chips, and with the trusted integrator model, the system integrator needs to be involved. However, this concern does not affect all systems, as most computer systems do not get component upgrades during their service life.

Multi-Socket System. For systems with multiple sockets, the operating system (OS) treats the memory attached to each socket as a separate pool arranged in Non-Uniform Memory Access (NUMA) manner. Inter-socket memory modules may cause the inter-socket access pattern to leak. This affects not just ObfusMem, but also ORAM. However, since per-socket memory is typically very large, one way to address this is to allocate important
data structures on a single node.

5.7 Related Work

Exploiting on-die logic for security. Gundu et al. [49] proposed using near-data computing capability in future memory technologies to implement data authentication. Our work is different in that we obfuscate the access pattern. Our work is orthogonal to [49] and can potentially be integrated with it.

Secure processor designs. Much prior work considered designing efficient secure processor systems where the TCB only includes the processor, and memory content is encrypted and protected with integrity verification [8, 117, 153, 159]. ObfusMem uses memory encryption to protect data, and additionally provides access pattern obfuscation.

Obfuscating access pattern. Obfuscating the memory access pattern has been a long time research challenge [45, 46]. The concept of Oblivious RAM was proposed by Goldreich et al. [45, 46]. Later, a practical realization of the concept of ORAM was proposed in Path ORAM [136]. Path ORAM has been a target of optimization for several state-of-the-art implementations of ORAM [34, 41, 115, 161]. We contrasted ObfusMem with ORAM and compare their security properties in Section 5.6.1 and characteristics in Section 6.6. Prior to Path ORAM, there were studies looking into hardware support for permuting the address space at chunk-level granularity [44, 164, 165]. Such approaches obfuscate the blocks within a specified small chunk (typically 64KB) using hardware support. Because of their small granularity, their performance overheads are much smaller than those of ORAM. In contrast, ObfusMem obfuscates the access pattern of the whole memory but at a low performance
overhead.

Other studies focused on preventing information leakage due to access timing [39, 42, 49, 57, 151]. These proposals study efficient ways to hide the information leak due to the timing of the memory accesses through scheduling mechanisms and memory controller modifications. ObfusMem is orthogonal to them and can potentially be integrated efficiently with them.

5.8 Conclusion

Obfuscation of memory access patterns offers important hardware-level protection for both system security and user privacy. In this chapter, we proposed ObfusMem, a memory access pattern obfuscation system, which offers overhead advantages in comparison with ORAM. ObfusMem relies on cryptographic logic in both processor and memory to encrypt and authenticate commands, addresses, and data that are communicated via the exposed wires connecting the processor and memory. Our evaluation showed that ObfusMem incurs only 11% execution time overhead on average above the cost of unprotected memory accesses, making memory access pattern obfuscation feasible for commodity computer systems. ObfusMem is nearly an order of magnitude faster than ORAM. Critically, for non-volatile memory, ObfusMem does not cause early wear out.
CHAPTER

6

AVOIDING TLB SHOOTDOWNS THROUGH SELF-INVALIDATING TLB ENTRIES
6.1 Introduction

The virtual memory system in modern computers enable a wide range of features, from standard virtualization of the machine’s physical address space and memory protection, to a wide variety of optimizations such as copy-on-write, zero-page and other memory compression/deduplication, garbage collection, late binding/allocation of physical memory, memory relocation and NUMA performance tuning, and many aspects of processor virtualization. Processors support high performance virtual memory translations through low-latency caches of recently accessed page table entries (PTEs) in the processor’s translation lookaside buffers (TLBs).

In the process of supporting the various virtual memory maintenance or optimization operations, however, modifications of the page table (e.g., changing a mapping or modification of page permissions) typically require a TLB shootdown operation. The TLB shootdown is an expensive operation involving the operating system (OS) and hardware that removes all possible stale copies of the modified PTE from all TLBs to ensure any future accesses make use of the correct, most up-to-date mapping. Depending on the system architecture, number of threads, and other issues, the latency for a TLB shootdown can take up to 13.2 $\mu$s [104].

In this work, we propose a new hardware-software cooperative approach to manage TLBs, which greatly reduces the number of costly TLB shootdown operations. We make the observation that many PTEs are used for a relatively limited duration, and then the corresponding pages are not re-referenced for a very long time. Rather than let these PTEs linger on indefinitely in the processor TLBs, we assign each PTE a expiration time. This acts
like a “self-destruct” mechanism, where the processor automatically treats expired TLB entries as invalid. At the same time, the OS keeps track of the expiration times assigned to PTEs. If any PTE is updated/modified and its previous expiration time has already passed, then the OS is guaranteed that no valid copies of the PTE are cached in any TLBs. In such scenarios, the OS may freely update these PTEs without issuing any corresponding TLB shootdown operations.

This chapter details the architectural and OS support required to build an effective virtual memory management system based on self-invalidating TLB entries (SITE). We demonstrate the potential of this approach with a use case based on emerging heterogeneous memory systems, e.g., a mix of DRAM and non-volatile memory (NVM). In such systems, there is a tension frequent page migrations from the slower NVM to the faster DRAM (to increase the fraction of memory requests serviced by the high-performance DRAM) and less frequent migrations (to decrease the performance impact of TLB shootdowns). With our SITE approach, we present a solution that relaxes this tension to provide both high DRAM service rates and low TLB shootdown overheads, resulting in a substantial improvement in performance over traditional approaches based solely on TLB shootdowns.

In summary, we make the following contributions.

• We introduce the idea of Self-invalidating TLB entries (SITE) to eliminate many TLB shootdowns.

• We detail the hardware and OS modifications needed for implementing a SITE-based system.

• We demonstrate the efficacy of SITE in improving the performance of a heterogeneous
memory system by reducing the number of TLB shootdowns needed for migrating pages.

6.2 Background and Motivation

In this section, we define the TLB shootdown process, how it is implemented and why it does not scale. We also introduce emerging heterogeneous memory systems, as they serve as our primary working example in this chapter, and we motivate our work by demonstrating the tension between page migrations and TLB shootdowns in such systems.

6.2.1 TLB Shootdown

Modifications to a page table entry (PTE) typically must be accompanied by a TLB shootdown operation. If a processor core is allowed to continue executing with a stale TLB entry, then undesirable behaviors may manifest from the application level (e.g., incorrect program results, application crash) to the entire system (e.g., security violations, system crashes). The TLB shootdown process ensures that any cached copies of the modified PTE are properly discarded before continuing the execution of the affected application.

Conventional processor caches (e.g., data caches) have direct hardware support to maintain cache coherence, so that modifications to a cache line result in the automatic invalidation or update of any other copies of the same line in other caches. Modern processors do not implement hardware coherence support for TLBs, and so the TLB shootdown mechanism is employed instead. The TLB shootdown process maintains the coherency of page table entries by interrupting all participating cores and guaranteeing they execute a
TLB entry invalidation instruction (e.g., `invlpg` in x86 systems). As shown in Figure 6.1, the same page table entry (PTE) can be cached in different private TLBs 1\(\rightarrow\)2. Later, when the OS updates the PTE due to page migration or a change in access permissions 3, the OS issues or initiates an inter-processor interrupt (IPI) across all participating cores 4\(^1\).

Each receiving core executes an interrupt handler routine that invalidates the entry for any cached copies of the PTE in the core’s private TLBs, and then sends an acknowledgment back to the initiating core. The OS waits for acknowledgment from each receiving core.

\(^1\)In a multi-process scenario, when the OS knows that a process does not have any threads running on particular cores, the IPI does not need to be issued to those “non-participating” cores. There are other optimizations that allow filtering of some IPIs, but in general the IPI will still need to be sent to all cores involved in executing threads for the affected process.
before concluding the overall PTE update process 5.

The overhead of interrupting cores and waiting for all of the acknowledgments is very high, and the overhead increases with the number of participating cores. There are multiple sources for the shootdown overheads, including multiple user/kernel mode transitions and the usage of legacy APIC hardware (on x86-based systems) [104]. Rather than reduce the shootdown latency, this work presents a new mechanism to greatly reduce the number of shootdown operations needed. We discuss other related mechanisms to address shootdown latency in Section 6.7, although they are generally orthogonal/combinable with our approach.

6.2.2 Heterogeneous Memory Systems

Conventional memory systems are based on DRAM technology, which is rapidly running up against difficult fundamental physical limitations as the DRAM cell sizes continue to shrink [99]. At the same time, a variety of emerging non-volatile memory (NVMs) technologies are gaining traction, including phase-change memory [76, 81], STT-MRAM [54], memristors [156], and 3D-XPoint memory [33]. While the exact characteristics of the different technologies vary, a common challenge is that the performance of these NVMs is typically worse than DRAM. As a result, using NVM as a wholesale replacement for DRAM is unlikely. Instead, many researchers are advocating heterogeneous memory systems consisting of some DRAM for performance, coupled with NVM for capacity [9, 113, 122].

There are many possible organizations of heterogeneous memory systems. In this work, we consider a system with a mix of fast DRAM and slower NVM. Other possibilities include a combination of high-bandwidth in-package (3D-stacked) DRAM [53, 64] with either
conventional DRAM or NVM outside of the package. While several different approaches have been proposed to manage such heterogeneous memory systems, in this work we consider a system where the OS is responsible for migrating pages between the different memories in a manner that is transparent to the application programmer. More details of the exact mechanisms are described later, and other alternative approaches to manage heterogeneous memory systems are discussed in Section 6.7.

### 6.2.3 Shootdowns vs. Migration

Using the heterogeneous memory system described above as a working example, we now examine the tension between aggressive memory management and the corresponding TLB shootdown overheads. We consider a system with a fast, first level of memory (e.g., DRAM), and a slower-but-larger second level of memory (e.g., NVM). The full details of our system and experimental methodology can be found in Section 6.5.1. Our baseline heterogeneous memory management policy simply keeps track of each page currently mapped to the slow...
memory. If the page is accessed more than a threshold number of times, then the page is migrated to the fast memory. If necessary, a page from the fast memory may at the same time also need to be moved back to the slow memory to make room (selected with a clock replacement policy[138]).

For a low migration threshold, a page in the slow memory only needs to be referenced a few times before being migrated to the fast memory. This allows the system to be more responsive, aggressively moving pages to (hopefully) allow more future references to be serviced from the faster memory. In a complementary fashion, a large migration threshold requires that a page exhibit high levels of reuse before promoting it from the slow memory to the fast memory. This reduces the likelihood of migrating a page with few future uses, but it also increases the number of requests that must be serviced out of the slower memory.

Figure 6.2 shows results for two representative applications as we vary the migration threshold. Let us first consider the Lulesh application on the left. The left y-axis shows the fast memory’s miss rate (the fraction of memory requests not serviced by the fast memory; lower is better). Lulesh is an application that demonstrates reasonably good memory locality; when a page is accessed, it is likely to be accessed many times again in the near future. As a result, when we set the migration threshold to 1 (i.e., move a page from slow to fast memory on the very first access), we achieve a very low miss rate from the fast memory. As the threshold is increased, the miss rate steadily climbs because each memory access on our way to reaching the migration threshold results in a miss in the fast memory. However, simply setting a low migration threshold can still be disastrous for performance, as each and every migration is accompanied by a costly TLB shootdown. The right y-axis shows the number of TLB shootdowns incurred, and as the migration threshold
is increased, the number of shootdowns drops.

The right side of Figure 6.2 shows another application, RS Bench, that shows the same macro-scale trends, but the slopes and concavities of the curves differ. RS Bench has poorer spatial locality than Lulesh, with many pages that are accessed only a few times. Even with a migration threshold of 1, almost half of the memory accesses miss in the fast memory. When increasing the migration threshold to 10, we observe a massive drop in the TLB shootdown rate, indicating that there are a huge number of pages that are accessed fewer than ten times before being evicted out of the fast memory. Further increases in the migration threshold do not substantially decrease the TLB shootdown rates for RS Bench.

**Take-aways:**

These results illustrate the tradeoff between making the best use of a heterogeneous memory system’s fast memory against the memory management overheads of trying to maintain those low miss rates. If the overall cost of TLB shootdowns could be reduced, then the heterogeneous memory management system could afford to be more aggressive in its migration decisions, leading to more memory traffic being serviced from the faster memory. In this work, we take the approach of reducing the frequency of shootdown events through self-invalidating TLB entries (described in the next section).

Beyond illustrating the need to reduce TLB shootdowns to effectively manage a heterogeneous memory system, the results also illustrate that different applications react differently in terms of fast-memory miss rates and the corresponding TLB shootdown rates. This suggests that our proposed solution should have some dynamic component to it to adapt to variations between and even within workloads.
6.3 Self-invalidating TLB Entries (SITE)

Traditionally, address translations/page table entries (PTEs) are loaded into the TLB by a hardware page table walker (e.g., in ARM and x86-based systems) on a TLB miss and remain valid until they are evicted by TLB’s replacement policy or explicitly invalidated by a TLB shootdown. Instead, we propose Self-invalidating TLB Entries (SITE) to cache PTEs in TLBs with an expiration time. SITE enforces an invariant that a translation entry in the TLB is valid only if its expiration time is in future. Thus, an entry in the TLB with an expired lease does not need to be explicitly invalidated. A shootdown can then be avoided by taking advantage of this invariant if the latest expiration time of a given PTE is known to be in the past.

For a concrete exposition of SITE, consider the scenario shown in Figure 6.3. The left-
hand side of the table shows an example sequence of events and their timestamps in a SITE-based system. The right-hand side depicts these events in a system with two CPU cores. Here, ① core \( C_0 \) accesses virtual address \( A \). The corresponding TLB in that core is searched for a translation for \( A \). ② Assume that the lookup misses and a page table walk is initiated to find the in-memory PTE that holds the desired translation at time \( T = 5 \). ③ At time \( T = 205 \), the hardware page table walker locates the PTE. However, different from a traditional system, the walker in a SITE-based system retrieves the translation and assigns an expiration time for \( A \). The expiration time is calculated to be 1205 by adding a chosen lease length (1,000 in this example) and the current time \( (T = 205) \). Furthermore, the walker records \( A \)'s expiration time in a new in-memory data structure called the Expiration Time Table (ETT). ④ At time \( T = 210 \), Core \( C_0 \)'s TLB receives \( A \)'s translation and installs it in its TLB along with the expiration time. \( C_0 \) then initiates its data cache access using the provided physical address. Design details such as lease-length selection are described in the next subsection.

⑤ Later at \( T = 1300 \), another core \( C_1 \) initiates the process of invalidating the address translation for \( A \) (e.g., due to a PTE modification). This involves calling a specific TLB shootdown routine in the OS. ⑥ At \( T = 1400 \), The modified OS routine for SITE modifies the PTE for address \( A \) in the page table and then consults the corresponding entry in ETT. The OS routine finds the expiration time for address \( A \) in the ETT is 1205, which is in the past. The OS then immediately returns without performing the shootdown and thus avoiding its cost. ⑦ At \( T = 1500 \), core \( C_0 \) again attempts to access data with virtual address \( A \). As usual, the TLB is looked up to find \( A \)'s translation, but this time the translation is found to have already expired (expiration time 1205). The SITE-based TLB cannot use this TLB entry
and thus a TLB miss is triggered. This ultimately results in a new page table walk, similar to what was explained above ②.

In summary, we observe that a SITE-based system can avoid performing TLB shootdowns by utilizing the invariant that a PTE whose latest-known expiration time (according to the ETT) ensures that no valid copies of the PTE can exist in any TLBs. At the same time, using SITE can also introduce extra TLB misses due to expired TLB entries that would have remained valid in a conventional non-SITE system.

6.3.1 Design Considerations for SITE

While the general idea of self-invalidating TLB entries (SITE) is intuitive and simple, there are several key design considerations to assemble an effective TLB-management solution. Below, we discuss a few important design dimensions and our rationale for the choices we make for each one.

6.3.1.1 Unit of Lease and Expiration Time

A fundamental need for supporting SITE is a way to mark the passage of timestamps. An obvious option is to use the physical clock tick as the unit for lease and expiration times, as has been previously proposed in the context of hardware cache coherence (e.g., globally synchronized timestamps) [127, 130]; commercial vendors also suggest the possibility of realizing such global timestamp at least within a single chip [60]. While an embodiment of SITE could use such global physical time, we find it is unnecessary. Unlike cache accesses and coherence traffic, TLB shootdowns occur much less frequently, and so a fine-grained timestamp based on clock ticks is not needed for SITE. On the other hand, SITE is more
valuable in large systems, possibly with multiple chips/sockets, as TLB shootdown costs increase with system size [118, 147]. Keeping a physical clock synchronized across large systems can be challenging.

We instead use the memory access count as the logical time unit for lease and expiration times in our proposed SITE implementation. There are at least two reasons behind this choice. First, the main memory (DRAM and NVM) access count changes at a much lower rate than clock tick as memory is accessed only after a miss in the last-level cache and is therefore easier to keep synchronized in larger systems. Second, there are only handful of memory controllers in a system and therefore the required hardware modifications are limited.
Figure 6.5: Impact of the lease length on execution time, normalized to non self-invalidating TLB

6.3.1.2 Choosing the Lease Length

Choosing an effective lease length is crucial. If the lease is too short, then TLB entries will expire quickly, incurring additional TLB misses. On the other hand if the lease length is too long, then the TLB entries will not self-invalidate quickly enough to avoid shootdowns. Figure 6.4 shows how the fraction of incurred TLB shootdowns (lower is better) changes with lease length for two representative benchmarks (the rest are in Section 6.5). As expected, shorter leases reduce the number of shootdowns. On the other hand, however, the address translation cost grows with shorter lease lengths as additional TLB misses are incurred (Shown in Figure 6.11 in Section 6.5).

These two conflicting trends mean that the overall application execution time can be minimized if the lease length is neither too short nor too long as depicted in Figure 6.5.
Dynamic Lease-length Assignment: However, the best-performing lease length is not the same across all workloads. For example, in Figure 6.5, we observe that RSBench prefers a much shorter lease length compared to LULESH. Furthermore, even within a single application, different memory regions (e.g., stack vs. heap) experience different access patterns and are thus likely to prefer different lease lengths. These suggest that the desired lease length should be dynamically discovered at runtime.

To appreciate what lease length (range) may be desirable for a PTE, Figure 6.6 shows the timeline of an imaginary sequence of shootdowns and page table walks for a virtual address $A$. In the figure, the page walks $PageWalk_{e0}$ and $PageWalk_{e1}$ occur due to expired entries in the TLB. These page walks could have been avoided if the lease length was long enough.
to cover the time between the last page walk due a true TLB miss (here, PageWalk<sub>m1</sub>) and the most recent walk (PageWalk<sub>e1</sub>). We call this the minimum desirable lease length. To avoid the second shootdown (Shootdown<sub>1</sub>), the lease length should be shorter than the time between the latest page walk (PageWalk<sub>m2</sub>) and the shootdown to ensure that the TLB entry has already expired. We call this the maximum desirable lease length. Ideally, one would pick a lease length between these two minimum and maximum values. Note that the minimum and maximum desirable values may not always overlap, and therefore it may not be possible to simultaneously minimize both additional page walks and shootdowns. Because shootdowns are orders of magnitude slower than a page table walk, it is reasonable to avoid shootdowns even at the cost of a few additional TLB misses.

Based on the above observation that lease length should be maintained within the minimum and maximum desirable range, the algorithms described in Figure 6.7 dynamically adjust lease lengths based on a program's observed behaviors. On a PTE invalidation, if the OS finds that the copies of the PTE in TLBs may not have expired, it lowers the lease length following the insight from the Figure 6.6 (typically, constant \( C = 2 \)). When the hardware page table walker observes several consecutive page walks due to expired entries, it correspondingly increases the lease length. We empirically found that a typical value of the threshold \( T h = 16 \) is effective to ensure that the algorithm favors the reduction of shootdowns even at the cost of a few extra page table walks.

### 6.3.1.3 Storing Expiration Times and Lease Information

Our SITE-based system needs to store the latest expiration time of TLB entries corresponding to a given PTE. Furthermore, for dynamic lease length assignment, additional informa-
On invalidating a PTE for address A:
IF (ExpirationTime(PTE_A) > CurrentTime) Then
{ //Shorten the lease, C is constant
    Lease_A = (CurrentTime – LastPTWTS)/C
}

On a Page Table Walk for address A:
IF (Page walk due to expiration) Then
{
    numConsecutiveTLBMiss++;
    IF (numConsecutiveTLBMiss > Th) Then
    { //Increase the lease length
        Lease_A = Max ( (CurrentTime – LastPTWTSMiss + C’, Lease_A*C’)
        numConsecutiveTLBMiss = 0
    }
    } else {
    LastPTWTSMiss = CurrentTime //Updating timing information
}
LastPTWTS = CurrentTime

Figure 6.7: Algorithms for dynamic lease length assignment.

In the SITE information (as shown in Figure 6.7) needs to be kept at a per-page granularity. This information is comprised of the latest expiration time \(2\), current lease length, and timestamps for the latest \textit{true} page table walks and that for any page walk \((\text{LastPTWTS}_{\text{Miss}} \text{ and LastPTWTS in Figure 6.7, respectively})\). At 48 bits per timestamp, this amounts to 24 bytes of storage per page, which is less than 0.58\% overhead for 4KB pages (and lower still for large pages).

The PTE format in modern page tables barely has any free bits available \cite{5}, and so we propose to store the SITE information in an auxiliary in-memory data structure called the Expiration Time Table (ETT). To avoid constructing a whole new page-table-like structure,

\footnote{Section 6.6 discusses how overflow can be handled}
Figure 6.8: Lookup of Expiration Time Table (ETT) as an extension to x86-64 page table. Entry for ETT is outlined in dotted lines. In x86-64, CR3 keeps the pointer to the root of the page table.

The ETT reuses the non-leaf page-table nodes as shown in Figure 6.8 (based on the x86-64 page-table structure). The sharing of top-level/non-leaf nodes reduces the number of memory accesses to look up the ETT. Each 8-byte PTE entry is paired with 24 bytes of SITE information, and so for each page of PTE information in the original page table, we interleave three additional pages to store the ETT entries. With this layout, we can simply

---

3At first glance, the overhead of the ETT may appear unattractive because it amounts to an effective page-table size increase by \( \sim 3 \times \) (less due to the sharing of the non-leaf nodes). This can be easily reduced by re-encoding the various ETT timestamps because their upper bits will almost always be identical, and so there are well-known encoding optimizations that can be applied. However, even with an unoptimized additional 24 bytes per ETT entry, page tables are very small compared to the corresponding application's
reuse the calculation of the final PTE entry with a modified offset to retrieve the desired
ETT entry.

6.4 Implementation: Putting it All Together

We now describe the hardware and software modifications necessary to implement one
possible embodiment of a SITE-based system.

6.4.1 Hardware Modifications

**TLB Structure and Lookup:** SITE extends TLB entries to hold their expiration times (48
bits). This adds state overhead roughly equal to TLB tag overhead. However, a TLB's tag
array does not contribute much to a processor's area budget and thus, this extension barely
impacts a processor's overall area budget. The TLB lookup process in SITE needs to compare
the expiration time of a given TLB entry with the current logical time (here, number of
DRAM accesses). On a tag match in a TLB, a hit is signaled *only if* its expiration time is
greater than the current logical time. The tag and expiration time lookups occur in parallel
and do not significantly impact the latency of a TLB hit or miss.

Also note that the overheads of SITE are minor compared to previously-proposed tem-
poral cache coherence works that use the idea of self-invalidation for caches, but they
need to extend each cache block with a timestamp and alter the hardware cache coherence
protocol [127, 130].

**Hardware Page Table Walker:** The hardware page table walker (PTW) for SITE has three
memory footprint to begin with, and so we do not bother optimizing the ETT size here as the total overhead
is only 0.58% as mentioned earlier.
additional responsibilities. The PTW assigns an expiration time to each address translation it returns to TLBs. The PTW does so by adding the current logical time and the lease value for the PTE (found in the ETT) as described in Section 6.3.1. Next, the PTW needs to store the expiration time in the corresponding entry in the ETT. This becomes the latest expiration time for that given PTE. At the same time, the PTW updates the statistics needed for the dynamic lease assignment as described in Figure 6.7, which may cause the PTW to increase the lease length for later accesses to that PTE.

All of the above steps need the PTW to access the ETT. As depicted in the Figure 6.8, the page table and the ETT share the top levels or non-leaf nodes and thus nothing extra needs to be done there (effectively re-using the existing PTW operations). While accessing the leaf level of the page table, the corresponding ETT entry is concurrently accessed. Thus, we do not expect page table walk latencies to increase significantly due to this extension, but we have accounted for the possibility of longer page walks for SITE by assuming a very conservative 33% of extra latency in our evaluations.

**Per-core Logical Time:** We use the main memory access count as the logical clock. Memory controllers can easily keep count of memory accesses. However, on every load or store, a core cannot query the memory controller for the current count. Therefore in our implementation, we add a register to each core to hold the current logical time (i.e., memory access count). On each DRAM access, the memory controller broadcasts a special message over the cache coherence network (similar to a coherence control message) to update the per-core registers. Note that these messages are broadcasted to cores (not caches) at a negligible rate compared to cache coherence messages that keep the private cache hierarchies coherent in multi-core systems. Thus, they introduce negligible contention or overhead in the coherence network.
6.4.2 OS Modifications

SITE is truly an OS-hardware co-designed system. Below, we describe the OS enhancements to support SITE.

PTE-invalidation Routine: The OS’s TLB shootdown routine is invoked to invalidate a PTE (Section 6.2). This routine is enhanced to avoid actually performing costly shootdowns, when possible. Specifically, it looks up the entry in the ETT corresponding to the PTE to be invalidated. If the expiration time in that entry is less than the current logical time (memory access count), then the shootdown is avoided and the routine returns immediately. If not, the default path of TLB shootdown is followed as shown in Figure 6.1.

This routine may also need to decrease the lease length as per the dynamic lease assignment policy in Figure 6.7 and update the corresponding entry in the ETT. This does not add any observable overhead as PTE invalidation routines are already several thousands of cycles long.

Allocating ETTs and Page Fault Handler: At the time of allocating a PTE to map a newly allocated memory page, the corresponding entry in the ETT is also created and initialized. The page fault handler is easily extended for the purpose. This adds an insignificant overhead as the latency of page faults is already in the order of several micro-seconds [122].

6.5 Evaluation and Analysis

In this section, we evaluate the ability of SITE-based memory management to reduce the cost of TLB shootdowns in a heterogeneous memory system. However, SITE is more broadly applicable to other use cases like copy-on-write and garbage collection that critically de-
Table 6.1: Configuration of the simulated system.

<table>
<thead>
<tr>
<th></th>
<th>Processor and Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8 in-order cores, single hardware thread per-core</td>
</tr>
<tr>
<td>L1 TLB</td>
<td>32-entry fully associative per core (1 cycle hit latency)</td>
</tr>
<tr>
<td>L2 TLB</td>
<td>256-entry 8-way associative per core (10 cycles hit latency)</td>
</tr>
<tr>
<td>Page Walk Latency</td>
<td>150 cycles page-walk (L2 TLB miss), 50 extra cycles for SITE</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>private, 1 cycles, 32KB, 4-way, 64B block</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>private, 10 cycles, 256KB, 8-way, 64B block</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>shared, 25 cycles, 8MB, 16-way, 64B block</td>
</tr>
<tr>
<td>Coherence</td>
<td>MOESI protocol</td>
</tr>
</tbody>
</table>

| Main Memory                    |                                                            |
| Fast to Slow Memory Ratio      | 1:8                                                       |
| Fast Memory Latency            | 150 cycles                                                |
| Fast Memory Replacement        | Clock replacement policy [138]                           |
| Slow Memory Latency            | 600 cycles                                                |

| Page Migration Overhead        |                                                            |
| TLB Shootdown Latency          | Issuing (Receiving) core 20,000 (5,000) cycles            |
| Page Copy Latency              | 1000 cycles                                               |

pends upon TLB shootdowns, which is discussed in Section 6.2.

### 6.5.1 Simulation Model

To evaluate SITE, we used trace-based simulation using the PIN toolset [114]. The simulator models detailed three levels of caches, two levels of TLBs, and keeps the caches coherent using a MOESI coherence protocol. We model heterogeneous memory comprising of fast (DRAM) and slow memory (PCM). The ratio of capacity between these two types of memory is 1:8. We then modeled our threshold-based page migration policy where a page is migrated to the fast memory only after a given number of accesses (threshold) to a page residing in slow memory. Unless mentioned otherwise, we use a page-migration threshold of 10 for rest of the evaluation. We also model detailed page migration and TLB shootdown costs.
Table 6.1 details the configuration of the system we modeled.

Using such a PIN-/trace-based simulation allowed us to simulate workloads with large memory footprints (> 1 GB) and run the applications for long run times. Both of these are important to reliably characterize a system with heterogeneous memory. We then use the statistics from this simulation infrastructure to feed a detailed performance model. At a high level, it models three types of latency.  

1. Non-load/store instructions are assumed to execute in one cycle.  
2. Data access costs for load/store instructions are calculated using hit/miss counts at the three levels of caches and their corresponding hit/miss latencies as listed in Table 6.1. Accesses that misses in the last-level cache incur a latency to either fast or slow memory depending upon the current memory mapping.  
3. Address translation costs for each load/store instruction are calculated by combining L1 and L2 TLB hit/miss rates with their corresponding latencies and the latency of page table walks (Table 6.1). For the baseline without self-invalidating TLB entries, we assumed a page table walk latency of 150 cycles. However, in case of SITE, we conservatively increase this by 33% to 200 cycles to model any extra latencies associated with ETT accesses. This is very conservative because it is possible for the page table walker to access the page table leaf node and ETT entry in parallel.  
4. Finally, we calculate the cost of migrating a page from slow memory to fast memory by taking into account the latency of copying the page between memories and the cost of the TLB shootdown. For TLB shootdown cost, we model it accurately assuming a larger latency incurred at the initiator (issuer) core of the shootdown and \( \frac{1}{4} \) that latency at the receiving cores. The issuer core needs to wait for all other cores to acknowledge the shootdown while receivers do not (Section 6.2 and Figure 6.1), thus this asymmetry. The latencies and ratio of asymmetry are similar to those reported in previous works [118, 147].
### 6.5.2 Workloads

We evaluate SITE using eight multi-threaded applications as listed in Table 6.2. We choose these workloads from the NAS parallel benchmark suite [12] and from publicly released HPC proxy applications [52, 59, 140, 141]. The applications are chosen such that they are memory intensive, a requirement to exercise systems with heterogeneous memory in any meaningful way. We use input set size of C for NAS workloads, and the memory footprint of each proxy application is around 1.25 GB.

### 6.5.3 Performance Evaluation

In the evaluation, we seek to answer the following questions: ① What performance improvement can SITE achieve by avoiding TLB shootdowns? ② What are the sources of improvements (degradations)? ③ What is the sensitivity of the evaluation against varying parameters such as the migration threshold, shootdown latency, page table walk latency, and slow memory access latency?

Figure 6.9 shows how execution time of different applications under page migration...
Figure 6.9: The impact of self-invalidating TLB entries (SITE) on performance of a system with heterogeneous memory (Page migration threshold = 10).

(migration threshold = 10) improves with SITE. The y-axis shows the execution time normalized to a baseline with no self-invalidating TLB (lower is better). There is a cluster of bars for each application listed along the x-axis. The right-most bar in each cluster represents the baseline (no self-invalidation). The left-most bar shows the normalized execution time with SITE employing the dynamic lease assignment algorithm (Dynamic lease) described in Figure 6.7. The bars in the middle of each cluster represent SITE with static lease length assignment. For example, Static-Lease-1K represents SITE employing a constant lease length of 1K units of logical time. We make two observations from Figure 6.9. ① SITE greatly helps performance, with dynamic lease assignment (Dynamic) reducing the execution time by 49.6% on average across all workloads. This is expected though, as it is well known that the majority of overhead from page migration is due to TLB shootdowns [9, 82, 104], and the benefit of migration can be negated by such overheads. Because SITE avoids many shootdowns, it helps performance significantly. ② The dynamic lease algorithm helps across all workloads. Except RSBench, the dynamic algorithm beats the best of all static
Next, we analyze the above-mentioned performance numbers. Figure 6.10 shows the number of TLB shootdowns normalized to the baseline for each configurations presented in the previous figure. The y-axis of Figure 6.10 shows the number of shootdowns normalized to that in the baseline (lower is better). The x-axis is same as the previous figure for execution times. We see that a large fraction of TLB shootdowns is avoided by SITE with both dynamic and static lease assignment policies. For example, on average across all workloads, 65.2% of the shootdowns are avoided by SITE with dynamic lease assignment. While SITE can avoid TLB shootdowns, it can increase address translation costs by incurring additional

---

4RSBench prefers a very short lease length, but our dynamic algorithm was less aggressive and did not reduce the fraction of shootdowns by as much compared to the best static lease values.
TLB misses due to expired leases. To understand how these two opposing trends impact the overall execution time, we dig into one of the representative workload XSBench in Figure 6.11. The y-axis shows the execution time normalized to the baseline. There are stacked bars for different lease assignment policies of SITE and for the baseline. Each bar is broken into three parts (stacks) showing the fraction of execution time spent on TLB shootdown (*Shootdown*), the fraction of time spent on address translation (*Translation cost*), and the rest that includes data access and page copy costs (*Others*). From the figure, we observe that a large fraction of the execution time in the baseline is attributed to the overhead of TLB shootdowns due to page migrations. This is in line with the findings in previous works [9, 82, 104]. This fraction, however, decreases significantly with SITE. Reduction is more significant with shorter leases and best with dynamic lease assignment. The fraction for address translation is much smaller compared to that for TLB shootdowns in the baseline because shootdowns are orders of magnitude larger than the cost of a page table walk. However, this fraction grows as the lease length decreases, but not enough to
negate the advantage of the lower shootdown costs until a very small static lease length of 1K. On the other hand, the dynamic lease assignment policy for SITE does a good job of balancing the lease length to achieve significant overall performance improvement. We omit such breakdown for other workloads as they show the same overall trends.

### 6.5.4 Sensitivity Studies

Next, we evaluate the resiliency of our SITE-based solution against several key design parameters.

**Impact of Migration Threshold:** In a system with heterogeneous memory, lowering the threshold for the number of accesses to a page in slow memory before it is migrated to fast memory (migration threshold) can increase the hit rate in fast memory, but it also increases the number of shootdowns. Figure 6.12 shows how SITE (with dynamic lease assignment) fares with migration thresholds of 1 (first touch), 10, and 100 (Threshold=1, 10,
100, respectively, in the figure). Lowering the threshold increases the number of shootdowns in the baseline due to aggressive page migration and thus offers more opportunity for SITE to avoid shootdowns. Consequently, the efficacy of SITE increases significantly with lowering migration threshold as shown in the figure.

**Impact of TLB Shootdown Latency:** SITE avoids TLB shootdowns and thus changes in shootdown latency affect its efficacy. Previously, we assumed a shootdown cost of 20K cycles to the initiator (issuer) core and $1/4$th (i.e., 5K cycles) at receiving cores (Table 6.1). In Figure 6.13, we varied the shootdown latency to 10K cycles and to 40K cycles for the initiator core while maintaining the same ratio to the latency for the receiving cores. As expected, the efficacy of SITE increases or decreases with an increase or decrease in the shootdown latency. However, even with a relatively small shootdown latency of 10K, SITE still reduces the average execution time of applications by 41.4%.

**Impact of Page Table Walk Latency:** SITE can increase the number of page walks, and thus we vary page walk latencies to see how that impacts SITE’s performance. Previously, we
assumed a page walk latency of 150 cycles and 200 cycles (33% more) for SITE. We vary the baseline page walk latency to 100 and 200 cycles, while also proportionally increasing the page walk latency for SITE. In Figure 6.14, we present execution times of SITE (with dynamic lease length), normalized to the baseline (lower is better) with these variations. As expected, higher page walk latency decreases efficacy of SITE, but it still remains potent even at a page walk latency of 200 cycles with more than 45% reduction in execution time on average.

**Impact of Slow Memory Latency:** Previously, we assumed the slow memory in our heterogeneous memory system is 4× slower than the fast memory (Table 6.1). Figure 6.15 shows the impact of varying the latency ratio between fast and slow memory (2× and 8×). We observe that the efficacy of SITE does not alter with varying the latency ratio of fast and slow memory.


6.6 Discussion

In this section, we discuss a few topics related to the idea of SITE and its proposed implementation.

6.6.1 Energy Efficiency of SITE

SITE extends each TLB entry with a 48-bit expiration time. On every TLB lookup, SITE also needs to compare this expiration time with the current logical time to confirm that the entry has not yet expired. Because expiration time can be looked up in parallel with the TLB tag, this does not significantly impact latency, but it adds to the dynamic energy consumption. However, industry projections show that the TLB contributes to around 6% of a chip's power [132]. Thus, SITE is unlikely to add much to the overall energy consumption of the chip.
More importantly however, SITE is likely to significantly reduce static energy. Static energy (leakage) consumption is proportional to the execution time, and SITE can reduce execution times significantly (e.g., by up to 65%). As the technology feature size for processors shrinks, the transistors tend to leak more current, and therefore the static energy can become a larger problem [70]; static energy already contributes more than 30-40% of chip's energy budget even at 22nm technology node [10]. Thus, SITE is likely to help reduce the net energy consumption of a system.

6.6.2 Scalability of SITE

SITE needs globally visible logical timestamps to ascertain when TLB entries have expired, and so SITE’s overall scalability may be limited by the scalability of maintaining globally synchronized timestamps. If the \textit{time} progresses too fast, then it becomes challenging to keep it globally synchronous. Thus, as discussed in Section 6.3.1.1, we avoided using physical clocks, and we instead used main memory access count as logical time. Main memory access count is incremented relatively infrequently. We avoid using physical clock since the TLB shootdowns happen relatively infrequently and thus do not require high-resolution timestamps.

Furthermore, our evaluation in Section 6.5 shows that most applications prefer lease lengths on the order of thousands of main memory accesses. Thus, SITE is unlikely to lose its efficacy if we lower the resolution of our logical time by $10 \times$ or even $100 \times$ by rounding timestamps to tens or hundreds of memory access counts. This allows incrementing the logical timestamp only once in every ten (or hundred) memory accesses, further aiding the scalability of SITE. Also note that the proposed implementation of SITE uses infrequent
coherence (control) message to keep the copies of timestamp updated, and so the already small coherence traffic related to broadcasting memory count increments would be further reduced. Researchers have shown that cache coherence can scale well to large numbers of cores [88], and therefore SITE’s proposed implementation for maintaining synchronized timestamps should scale along with it.

### 6.6.3 Timestamp Overflow

Our proposed implementation of SITE uses 48-bit logical timestamps, which would overflow at a very slow rate. We estimate that even with a sustained, high memory bandwidth demand of 20 GB/sec, the timestamp counter would overflow in 9.71 days, assuming the timestamp is incremented on every memory access. In case of the rare overflow, we propose to simply flush all TLBs and invalidate all ETTs.

### 6.7 Related Work

In this section, we summarize the most related work.

**Self-Invalidation:** The concept of self-invalidation has been explored in other contexts, such as in cache coherence protocols [73–75, 95, 127, 130, 133], in cache replacement [69, 83], and in cache power management [3, 40, 68]. Min et al. proposed a timestamp-based software-assisted cache coherence [95]. Lebeck et al. proposed to proactively invalidate shared cache blocks through dynamic self-invalidation (DSI) to eliminate invalidation messages [75]. Shim et al. improved DSI by delaying writes until all shared copies are expired [127]. Lai et al. proposed a Last-Touch Predictor (LTP) to improve the accuracy of
predicting the shared blocks to be proactively invalidated [74]. Somogyi et al. proposed predicting the last store to cache blocks via PC addresses [133]. Employing self-invalidation to reduce cache coherence overhead has also been extended for GPUs and accelerators [73, 130]. In the context of cache replacement, various studies have looked into identifying dead blocks as potential replacement victims, using counters events [69] or cache bursts [83]. Other work proposed using self-invalidation to reduce leakage power by turning off dead cache blocks [3, 40, 68].

To the best of our knowledge, this is the first work to propose using self-invalidation for TLB entries. Not only is the context new, but the challenges are unique because TLBs, page table walkers, and TLB shootdowns are not entirely managed by the hardware, and the OS plays an important role in these. For example, there is no hardware-enforced TLB coherence in commercial processors and the OS needs to get involved. Similarly, the OS allocates entries in the page table, which is then looked up by the hardware page table walker. This results in a significantly different design space compared to past work on conventional caches.

**Reducing TLB Shootdown Cost:** This category of work is the most related to our work, and includes the following studies. Romanescu et al. [118] proposed hardware coherence support for TLBs [118]. While feasible, hardware coherence for the TLBs may be an overkill because PTEs are updated much less frequently compared to data caches, and hardware coherence may add complexity to the core TLB structures and circuitry. Cache coherence is already tedious to validate due to its complex state machine [11]; TLB coherence is likely to add to that. Villavieja et al. [147] proposed a shared hardware TLB directory (DiDi) that is inclusive of all TLB entries in the system. DiDi helps filter extraneous shootdowns to
cores without cached copies of the PTE. The centralized nature of this hardware-based solution, and the fact that the proposed shared TLB needs to be inclusive to all other TLBs, makes it difficult to scale to a larger number of cores and to large per-core TLBs that are becoming more common. It is not clear how it can be adapted to handle multi-socket (SMP) systems, either. Oskin and Loh [104] proposed hardware-assisted TLB shootdowns to reduce inter-processor interrupt latency, but the overheads are still large, i.e., in the microsecond range. Our proposed self-invalidating TLB scheme is distributed, and scalable to implement, without relying on any centralized hardware structures.

**Heterogeneous Memory Systems Management:** There are many prior studies exploring the management of page placement in heterogeneous memory systems. Some studies relied on a hardware-only management approach [19, 63, 65, 77, 120, 128]. While transparent to system software, hardware caching is not without its costs. First, the hardware cache typically requires non-trivial resources for bookkeeping (i.e., the tags) that either require the construction of large on-chip structures (SRAM) that increases chip costs, and/or cannibalizes part of the fast memory capacity to store the bookkeeping state, thereby reducing the effective size of the faster memory resource. Furthermore, by making the fast memory software-transparent, the total system memory capacity is effectively reduced. An alternative to hardware-only management is a software management approach. Software management can be categorized into OS-based and application-explicit management. OS-based management approaches focus on different migration policies and improving performance via prefetching [9, 94, 104]. In application-level management work, the application has the flexibility to choose where to allocate memory objects and how to migrate pages across different memory technologies. One examples is Memkind [25], which is a
user-level heap manager that provides programmers with different memory services. Similarly, Meswani et al. proposed explicit management of two-level memory (TLM) memory through explicit calls from applications [93].

Our work investigates a complementary and crucial question of reducing the cost of TLB shootdowns. Lowering the number and cost of TLB shootdowns is a crucial enabling technology for the heterogeneous memory management discussed above.

6.8 Conclusion

We introduced the idea of Self-invalidating TLB Entries to avoid TLB shootdowns when possible. The key idea is that copies of PTEs are cached in TLBs with associated expiration times, and SITE maintains the condition that PTEs are valid only if their expiration times are still in the future. Utilizing this, SITE can avoid a shootdown if expiration times of all copies of a PTE in any TLB are known to be in the past.

We then showed how a SITE-based system can be implemented and detailed both the hardware and OS enhancements necessary for such an implementation. We evaluated the efficacy of a SITE-based system in the context of emerging heterogeneous memories where efficient page migrations, and thus associated TLB shootdowns, play a crucial part in overall performance. We find that SITE can reduce execution time by almost half and can easily eliminate more than 65% of TLB shootdowns.
In this thesis, we discussed the impact of integrating emerging NVMs in several places in future systems. We also proposed several techniques and mechanisms that have the potential to improve the performance and lifetime of NVM-based systems. We started our investigation from the very optimistic use case of having NVM-based main memory, where we proposed a novel secure memory controller, *Silent Shredder*. Silent Shredder enables zero-writes data shredding, which can have different use cases, starting from inter-process data leak prevention to efficient bulk zeroing. Our evaluation showed that
Silent Shredder can efficiently eliminate a large percentage of main memory writes and improve performance. Later, we investigated another use case that treats NVMs as a memory extension. Memory extension can be thought of as a second-level memory, where the OS ultimately migrates NVM pages to DRAM on-demand. We studied and proposed several write-aware management schemes that have the potential to reduce the write traffic to NVMs and hence mitigate the limited write endurance of emerging NVMs. Furthermore, we investigated further on OS-level prefetching and the impact of page replacement policies. As emerging NVMs are extremely encouraging for replacing Flash-based SSDs as I/O devices, we modeled the state-of-the-art NVM host controller interface, and studied the impact of using it with very fast NVM devices. We identified several bottlenecks that could limit the performance gains from such technologies. Specifically, we found that the current I/O software layer overhead can be very limiting for emerging NVMs. In Chapter 5, we discussed the challenges of current techniques for obfuscating the access pattern and how they become exacerbated when used with emerging NVMs. Accordingly, we proposed a lightweight design to obfuscate the access pattern. Our design ObfusMem exploits the logic on memory chips to encrypt the address bus, hence cryptographically obfuscating the access pattern. Finally, in Chapter 6, we discussed the TLB shootdown issue in the context of software-managed heterogeneous memory systems. To avoid the TLB shootdown issue and hence enabling better memory management, we proposed Self-Invalidating TLB Entries (SITE), which allows avoiding large percentage of the TLB shootdowns completely.

In summary, in this thesis, we investigated and proposed several design recommendations to enable integrating emerging NVM technologies efficiently in future system.
7.1 Research and Personal Reflections

In this section, I briefly share my own personal and research reflections of the thesis.

7.1.1 Research Reflections

In this section, I summarize my personal opinion and conclusions.

Emerging NVMs are still in their infancy stage, however, many directions and issues are still unveiled. My work in integrating emerging NVMs as I/O devices showed the significance of the software overhead of the I/O layer. While delving deep into the internals of block layer, I/O system, and device drivers was a lengthy process, it was very educational and helped me understand the big picture of the system and how subsystems interact with each other. Many opportunities are there for optimization, as the whole I/O layer requires reworking to deal with byte-addressable NVMs, rather than dealing with block devices as done previously for many years. In my work in Silent Shredder, my work and design forced me to dig deep into the kernel to understand the memory management unit and its interaction with hardware. This part was the most challenging, as it required going through thousands of lines of code just to understand how things work and where we can begin modifications. Again, what I learned during that process was priceless, from learning concepts, to designing a solution and implementing it. The generous feedback from my PhD advisor and the other mentors at HP Labs was essential to keep an eye on the big picture and to help steer me in the right direction. I believe that there are lots of interesting interactions between hardware, security and OS that are becoming attractive points of optimizations, given the issues of NVMs and the deployment of encryption engines in
memory controllers.

Utilizing emerging NVMs as memory extensions is a very promising use case, and I would personally think it will be the first to happen after using NVMs as pure I/O devices.

The process of evaluating the work involved lots of engineering efforts that were informative and fun; starting from porting an old kernel driver to a new kernel, learning from code, and through trial and error. Our proposed write-reduction techniques and kernel-level prefetching have good potentials as they require zero to minimal hardware changes. There are lots of topics that are not well-studied and are worth investigating further, including NVM-aware memory allocators, garbage collectors, and application-level hints to heap managers.

Using emerging NVMs as part of the main memory is another interesting use case that has its pros and cons. Nonetheless, our TLB shootdown avoidance mechanism facilitates such use case and enables better management for a heterogeneous memory system. Understanding the details of the virtual memory and its interaction with hardware and OS would have been an arduous task by only relying on textbooks or white-papers. I was fortunate enough to work closely with my PhD advisor and other people who are experts in that area and those who are involved in the actual design process. I believe that the virtual memory in general will change a lot to adapt to the abundance of memory provided by emerging NVMs. An expected change in the virtual memory, which we alluded to in Silent Shredder, is the use of large pages to reduce the translation costs. However, many hidden issues can arise, including migration overhead and reducing the chances of deduplication.

In summary, I believe that there are lots of unanswered questions that are yet to be open research problems to enable emerging NVMs in future systems. While we looked at many of
these issues and provided promising solutions, I believe there is still much more to tackle.

7.1.2 Personal Reflections

In this section, I summarize my research experience during PhD and reflect it with my own words.

I found it extremely important to stay up to date with recent research directions, reading papers from different areas, and understanding current research problems. Such efforts enrich your knowledge and help you relate research problems to each other, and ultimately give you the vision of the big picture. My personal opinion about PhD is that it is a well-designed process that teaches you how to find a problem, estimate the importance of a problem, propose a solution to the problem, evaluating your solution to the problem, and finally communicating the solution to the community in a systematic manner.

Finding a problem is the most challenging part; you need a problem that aligns well with your research interests and the expertise and interests of the people working with you. I personally always liked to share and discuss what I thought of as potential research problems with my PhD advisor, who was always willing to discuss them and give his personal opinion. Moreover, industrial internships and technical experience helped introduce me to interesting research problems. Estimating how important the problem is typically requires considering many things, including which systems are affected by the problem and how the significance of the problem is likely to change in the future. Frankly, answering these questions requires a visionary and well-experienced person, who was in many cases my PhD advisor and other mentors.

Proposing a solution is an interesting part of the PhD processes, as it requires lots of
brainstorming and challenges. My process of finding a solution typically starts from a simple naive solution, then adding other solutions while clearly identifying the trade-offs between them. Certainly, coming up with a perfect solution is difficult, however, identifying the trade-offs, and looking at the pros and cons are typically the guidance for my final solution. Evaluating a solution typically consists of first identifying the main results and insights you want to learn from your experiments, then identifying the right tool to use and its limitations, and finally conducting the experiments with said tool.

While all the previously aforementioned steps are important parts of the research process, communicating the ideas and putting them in a real life context are very important. The writing skills and ability to communicate why you think the problem is important and the clarity of your solution and insights are critical aspects for presentations whether in technical papers, conference presentations, or seminars. I believe that the writing skills are acquired with time and require serious desire to learn.
BIBLIOGRAPHY


185


[123] Seaborn, M. & Dullien, T. “Exploiting the DRAM rowhammer bug to gain kernel privileges”.


[131] SNIA non volatile memory (NVM) programming technical work group (TWG).


[142] Understanding Memory Resource Management in VMware vSphereÂ© 5.0.


