ABSTRACT

ELNAWAWY, HUSSEIN MOHAMED. Analyzing and Mitigating the Cost of Persistence in High-Performance Computing Systems. (Under the direction of Gregory T. Byrd.)

Non-Volatile Main Memory (NVMM) technologies provide many opportunities for applications in modern and future systems. One of these opportunities is the ability to achieve data persistence in the main memory rather than having to store data in the much-slower storage. This major opportunity makes it possible to use the main memory for checkpointing mechanisms to provide failure safety to applications. However, the non-volatility characteristic of NVMMs raises a potential data inconsistency problem. Therefore, applications need to ensure data consistency when making updates to critical data structures.

In light of these opportunities and challenges of NVMMs, we believe legacy checkpointing and logging schemes must be revisited as they incur high execution time overheads as well as a large number of additional writes to NVMM, which may significantly impact its write endurance. We analyze different checkpointing techniques when used with NVMMs. Then, we propose a novel recompute-based failure safety approach and demonstrate its applicability to loop-based code. Rather than keeping a fully consistent logging state, we only log enough state to enable recomputation. Upon a failure, our approach recovers to a consistent state by determining which parts of the computation were not completed and recomputing them. Effectively, our approach removes the need to keep checkpoints or logs, thus reducing execution time overheads and improving NVMM write endurance, at the expense of more complex recovery.

We then go deeper into the different implementations of General Matrix Multiply (GEMM) algorithms to analyze their performance and impact on NVMM with our Recompute technique for checkpointing. Our aim is to analyze the persistence/performance trade-offs that the programmer should be aware of. To approach this problem, we perform experiments on the different implementations of GEMM from one of the most widely used
basic linear algebra algorithms, the GotoBLAS library.

After that, we address the overhead of persistence in NVMMs, in general. To address the problem of potential data inconsistency, applications need to make sure that updates to persistent data comply with the crash consistency model. To achieve that, transaction-based approaches, such as write-ahead logging, have been proposed to ensure data consistency in the NVMM. These approaches use cache line flushing followed by a memory or store fence to ensure data durability.

While cache line flush and write back instructions can complete in the background, fence instructions expose the latency of flushing to the critical path of the program’s execution incurring significant overheads. We observe that if flush operations are started earlier, the penalty of fences can be significantly reduced.

We propose PreFlush, a lightweight and transparent hardware mechanism that predicts when a cache line flush or write back is needed and speculatively performs the operation early. Since we speculatively perform the flush, we add hardware to determine when it is more profitable to flush, and we also handle cases where the preflush misspeculates and still ensures correct execution without the need for any complex recovery mechanisms. Also, PreFlush requires no modification on existing NVMM-enabled code.
Analyzing and Mitigating the Cost of Persistence in High-Performance Computing Systems

by

Hussein Mohamed Elnawawy

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

2020

APPROVED BY:

James M. Tuck

Huiyang Zhou

Guoliang Jin

Gregory T. Byrd
Chair of Advisory Committee
DEDICATION

To my parents who went to great length to make sure I have a secure future. My father, thank you for always supporting me and pushing me to realize my potential. My mother, thank you for teaching me how to love and give unconditionally.

To my amazing wife, thank you for being my best friend. Thank you for reminding me that there is more to life than just career accomplishments. You helped me reconcile with my sentimental and spiritual self. With you by my side, I feel loved and valued, and more importantly, I do not fear losing sight of who I am and what I want.
BIOGRAPHY

I was born in a relatively small city in rural Egypt called Al-Mahalla Al-Kobra. I lived in Al-Mahalla until I was fifteen years old then I moved with my parents to Cairo.

I was the top of my class most of the time, scoring above 98% in the nation-wide high school exams. There was not much to do in Al-Mahalla, though. Therefore, I was able to stay focused on my studies. Nevertheless, the small and slow pace of Al-Mahalla made the move to the massive Cairo hard.

I went to the Engineering school in one of the greatest and oldest universities in Africa and the Middle East, Cairo University. At the time, my journey of great success took another turn. I staggered along the road time after time. It was very hard to recover from that hurdle. At that time, my father encouraged me to start anew by pushing me to continue my education in the U.S. He wanted to provide me with another chance to find my passion and success.

During my education journey in the U.S., the 2011 Egyptian revolution took place. For me, the revolution was a dream. It was change; the real one; the elusive dream to live in my country with dignity and human rights. It was very hard for me to see the revolution happening and not being able to be part of it. This time had its toll on me mentally and psychologically. I worked really hard to get out of the hole that I dug for myself. I worked very hard everyday whether it was thanksgiving, Christmas, new year, or any other holiday. Eventually, I emerged victorious over my hardships.

I am grateful for everything I lived in this amazing Ph.D. journey. It taught me to work very hard and be resilient. I am grateful for my advisors Professors Gregory Byrd and James Tuck. I am grateful for this country for allowing me to have this opportunity and supporting me financially throughout my doctoral degree. My father always told me to be like a beacon of light; to blossom wherever I am. I hope I will be able to use what I learnt, to contribute to society wherever I am.
ACKNOWLEDGEMENTS

Everything I learnt, accomplished, and worked towards in my Ph.D. journey was the result of complex system of support and mentorship. I would not be here if it was not for my academic advisor, Professor Gregory Byrd, who believed in me, supported me in hard times, and helped me develop a solid foundation of parallel computer architecture knowledge. More importantly, thank you for treating me with kindness and for caring about my well-being and my progress in my journey of learning and evolving.

I would also like to extend my gratitude to Professor James Tuck for co-advising me. Thank you for your kindness and respect. Thank you for long meetings of brainstorming, exploring ideas, and helping me with technical and conceptual hurdles.

Thank you Professor Byrd and Professor Tuck for showing me that I can live with ethics and integrity and still reach big accomplishments. For that, you have my utmost respect, gratitude, and unwavering loyalty.

I would also like to extend my thanks to my committee members, Professor Huiyang Zhou and Professor Guoliang Jin for their constructive input and guidance.

I want to thank Professor Amro Awad. In his last year of his Ph.D. at NC State, Amro was my mentor through all the difficult technical knowledge. Furthermore, Professor Awad gave me strong foundation to begin research in Non-Volatile Memory. Through our long discussions, I became a much better architect. I want to thank Professor Yan Solihin for advising me during my Master’s and part of my Ph.D. Professor Solihin made me love research and taught me how to think as a researcher.

I want to thank Trey Cain for his tremendous support and mentorship during my internship in Marvell Semiconductors. Trey was always very friendly and humble despite being a very well-accomplished and well-known computer architect. Trey is a living example of a great manager. I would also like to thank Rangeen Basu being my friend and my mentor during my internship in Intel, summer 2017.
For all my friends, I want to thank you for making this journey fun learning experience. I could not have been able to push through the stressful times if it was not for the get-togethers with my very dear friends Amro Awad, Abdulla Mughrabi, Islam Hashem, Karim Ali, and Moustafa Radwan.

For all my lab mates, I want to thank you for being my companions throughout my years of the Ph.D. Special thanks goes to Bagus Wibowo, my senior lab-mate, for his technical and conceptual help in the beginning of my Ph.D. I want to thank all my colleagues: Mohammed Alshboul, Abdulla Mughrabi, Mahita Nagabhiru, Mohannad Ibrahim, Seunghee Shin, Joonmoo Huh, Abhinav Agrawal, Zhen Lin, Ahmed Samarah, Abdulla Al-Kafi, and Zachary Johnston for offering your valuable experiences and input.
# TABLE OF CONTENTS

## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Chapter 1  INTRODUCTION

1.1 Introduction .................................................. 1
   1.1.1 Challenges Facing DRAM .................................. 2
   1.1.2 Non-Volatile Memory Technologies ....................... 2
   1.1.3 Organization of This Chapter ........................... 5

1.2 Contributions .................................................. 5
   1.2.1 Utilizing NVM in Application-Level Checkpointing ....... 5
   1.2.2 Analysis of General Matrix Multiply Algorithms in NVMM .. 7
   1.2.3 Data Flushing and Maintenance .......................... 8

1.3 Summary ......................................................... 9

## Chapter 2  BACKGROUND

2.1 Non-Volatile Memory ........................................... 10
2.2 Crash Consistency in Persistent Memory ....................... 13
2.3 Intel Persistency Programming Support ....................... 14
   2.3.1 New Instructions ........................................ 14
   2.3.2 Clflushopt and Clwb Operation .......................... 16
   2.3.3 Flush Instructions with Memory Barriers ................. 16
   2.3.4 PMEM Programming Model ................................ 17
   2.3.5 Durable Transactions with Write-Ahead Logging ......... 18
   2.3.6 Tiled Matrix Multiply (TMM) ............................ 19
   2.3.7 TMM Crash Consistency with Traditional Checkpointing .. 21
   2.3.8 TMM Crash Consistency with Application-Level Checkpointing ... 22

## Chapter 3  UTILIZING NON-VOLATILE MEMORY FOR APPLICATION LEVEL CHECK-POINTING

3.1 Introduction ................................................... 24
3.2 Our Recompute Approach ....................................... 26
   3.2.1 Hybrid Recompute/Checkpointing ........................ 32
3.3 Methodology ..................................................... 33
   3.3.1 Simulation configuration ................................ 33
   3.3.2 Benchmarks ............................................... 34
3.4 Evaluation ....................................................... 36
   3.4.1 Execution Time and Number of Writes .................... 36
   3.4.2 Sensitivity to Checkpointing Frequency ................ 37
   3.4.3 Sensitivity to Thread Count ............................. 39
   3.4.4 Other comparisons ....................................... 41
   3.4.5 Other Benchmarks ........................................ 42
   3.4.6 Programming Complexity ................................ 42
3.5 Related Work .................................................. 44
3.6 Conclusion .................................................... 46

Chapter 4 ANALYSIS OF GENERAL MATRIX MULTIPLY ALGORITHMS IN NVMM 48
4.1 Introduction .................................................... 48
4.2 Goto Implementation of GEMM ............................ 51
4.3 GEMM Impact on NVMM Write Endurance .............. 52
4.4 GEMM with Recompute Checkpointing .................. 53
  4.4.1 Impact of Checkpointing Granularity ................. 54
4.5 Methodology .................................................. 55
  4.5.1 Simulation configuration ............................... 55
  4.5.2 Workloads ............................................. 57
4.6 Evaluation .................................................... 57
  4.6.1 Row Writes Without Checkpointing .................. 57
  4.6.2 Row Writes With Checkpointing ...................... 59
  4.6.3 Impact of Packing ..................................... 60
4.7 Related Work ................................................ 60
4.8 Conclusion .................................................... 60

Chapter 5 CACHE FLUSHING AND MAINTENANCE .............. 62
5.1 Introduction .................................................. 62
5.2 Motivation .................................................... 63
5.3 PreFlush Design ............................................. 64
  5.3.1 Speculating on a flush ................................. 65
  5.3.2 Predicting the Last Store .............................. 68
  5.3.3 Baseline Design ....................................... 69
  5.3.4 Last Store Optimization .............................. 73
  5.3.5 Sample of Sets Optimization ......................... 73
  5.3.6 Optimized Design Operation ......................... 73
  5.3.7 Recovering from Ordering Violation ................. 75
  5.3.8 Handling Flushes in Loop-Based Algorithms ........ 78
5.4 Methodology .................................................. 80
  5.4.1 Simulation configuration ............................... 80
  5.4.2 Workloads ............................................. 82
5.5 Evaluation of PreFlush ...................................... 83
  5.5.1 Evaluated Designs ..................................... 83
  5.5.2 Structures Sizing ..................................... 84
  5.5.3 Prediction Accuracy ................................... 84
  5.5.4 PreFlush Performance for Undo-Logging Workloads 87
  5.5.5 PreFlush Execution Time for No-Logging Workloads 88
  5.5.6 Area overhead ....................................... 89
5.6 Related Work ................................................ 90
5.7 Conclusion .................................................... 92

Chapter 6 CONCLUSION ............................................. 93
LIST OF TABLES

Table 2.1  Comparison between NVMs, DRAM, and Nand-Flash. .................. 13
Table 3.1  The baseline system configuration. ................................. 33
Table 3.2  Multiple approaches applied to the tiled matrix multiplication to
achieve write durability of the resultant matrix. The matrix is 1024 ×
1024 in size, and the block size is 16 in order to align them to the
cache block size. Thus, to persist one stride, only one clflushopt is
required. ................................................................. 35
Table 3.3  Other benchmarks we evaluated. ........................................ 35
Table 3.4  Execution time and number of write comparison between Check-
pointing (tmm+CP) and different granularity of our Logging (tmm+L),
Recompute (tmm+R) and Hybrid Recompute (tmm+HR) schemes,
normalized to the base tiled matrix multiplication (tmm). ............ 36
Table 3.5  Normalized execution time as the thread count varies from 1 to 12
threads, for various schemes. ........................................ 39
Table 3.6  Normalized number of NVMM writes as the thread count varies from
1 to 12 threads, for various schemes. .................................. 40
Table 3.7  Various performance statistics: instruction count (IC), store count
SC), L2 dirty eviction ratio (ER), L2 miss ratio (MR), number of L2
evictions (EC), and number of clflushopt's (CFOC). All numbers are
normalized to tmm (except for number of clflushopts, which is nor-
malized to tmm+R). ...................................................... 41
Table 4.1  The baseline system configuration. ................................. 56
Table 5.1  The baseline system configuration. ................................. 81
Table 5.2  Evaluated workloads. The top four are from Whisper library. The
bottom three are loop-based scientific microbenchmarks. ............ 82
Table 5.3  The sizes of the different structures we used in our PreFlush designs. 84
LIST OF FIGURES

Figure 2.1  (a) Regular NMOS with a gate controlling the flow of charge from the source to the drain through the silicone dioxide, (b) Floating-gate NMOS. The floating gate holds the charge. The silicone dioxide ensures persistence by insulating the floating gate from the channel.  

Figure 2.2  (a) Amorphous state; (b) Chrystalline state  

Figure 2.3  Clflushopt & CLWB journey, starting from the Store Queue (SQ), for different cache line states: (a) line is clean in the entire cache hierarchy, (b) line is dirty (i.e. has to be written back to the Memory Controller "MC) and the modifier is the flush issuer, (c) line is dirty but the modifier is not the flush issuer. 

Figure 2.4 Tiled Matrix Multiplication Code 

Figure 2.5 Tiled Matrix Multiplication.  

Figure 2.6 Tiled Matrix Multiplication With Logging  

Figure 3.1 Tiled Matrix Multiplication with Recompute technique. Granularity of checkpoint is ii.  

Figure 3.2 Tiled Matrix Multiplication with Recompute technique. Granularity of checkpoint is jj.  

Figure 3.3 Recovery pseudo-code for Tiled Matrix Multiplication with jj granularity. 

Figure 3.4 Recovery pseudo-code for Tiled Matrix Multiplication with ii granularity.  

Figure 3.5 Extrapolation of execution time overheads normalized to Checkpointing (tmm+CP) and our ii-granularity Hybrid Recompute with matrix saving frequency of 64×. 

Figure 3.6 Execution time (top) and number of writes (bottom) overheads for different benchmarks. For each benchmark we compare Checkpointing and our Recompute schemes normalized to base (base is the model without persistence or checkpointing). The figure also shows the geometric mean (gmean) for each scheme.  

Figure 4.1 The different implementations of GEMM. The labels have the form gexy where the letters chosen for x and y indicate the shapes of matrices A and B, respectively  

Figure 4.2 GEMM1: anatomy of GEMM using GEPP1 kernel  

Figure 4.3 Number of row writes to NVM relative to naive matrix multiplication (top) and speedup (bottom).  

Figure 5.1  (a) Coherence overhead for store followed by flush. (b) Coherence overhead for eagerly flushing after the store.  

Figure 5.2 Flush instructions with fences and thread synchronization  

Figure 5.3 Path of execution for store instructions.  

Figure 5.4 Path of execution for flush instruction.
Figure 5.5  PreFlush algorithm. ......................................................... 74
Figure 5.6  Sequence of coherence messages that ends with PreFlush invalidation due to ordering violation. ........................................ 76
Figure 5.7  Loop-based flushing following loop of stores. ...................... 79
Figure 5.8  Percentage of clwb flushes that our PreFlush technique predicted, did not predict, and mispredicted. The bars from the left are: the trace-based design without counter-mode prediction ($PF_{trace\_noCounter}$), the last-store design without counter-mode prediction ($PF_{LS\_noCounter}$), the sampled-set design ($PF_{ss}$), and the last-store design with counter-mode prediction ($PF_{LS}$). ......................................................... 85
Figure 5.9  The percentage of clwb latency savings for our two PreFlush designs normalized to the base undo-logging model ................. 86
Figure 5.10 The percentage of performance improvement of our two PreFlush designs normalized to the undo-log model ...................... 86
Figure 5.11  The percentage of performance improvement of our two PreFlush designs normalized to the optimistic no-log model ............ 89
CHAPTER

1

INTRODUCTION

1.1 Introduction

For decades, Dynamic Random Access Memory (DRAM) has been the de-facto technology deployed as the main memory of computer systems. This market dominance is attributed to DRAM’s speed, low cost, and simple design compared to other memory technologies, such as Static Random Access Memory (SRAM). For instance, each DRAM cell is composed of one transistor and one capacitor while each SRAM cell is composed of at least six transistors. The simplicity of design in DRAM is responsible for its lower price and higher density. Nevertheless, the emergence of big data and memory-intensive applications with their large dataset put significant pressure on the DRAM main memory of computer systems. This trend of memory-intensive applications is expected to continue well into the future.
with the advent of machine learning and neural networks applications.

1.1.1 Challenges Facing DRAM

Although DRAM is cheaper and scales better than other memory technologies, it won’t be able to keep up with the scalability demands of memory-intensive workloads in the future. The reason is that DRAM’s density will soon hit a scaling wall beyond forty nanometer [54, 58, 68]. In addition, DRAM’s simple design comes at the cost of data volatility, meaning that DRAM loses its data upon power failure or shut down. The reason DRAM is volatile is that it stores each data bit in a capacitor. Since capacitors leak their charge over time, DRAM has to periodically refresh its cells to be able to retain its data. The power consumption of this refresh operation is a limiting factor to scaling DRAM for use in future systems; it contributes to up to 40-50% of total power of the memory system [51, 60]. Due to DRAM’s scalability limitations, the industry and academic community have been researching alternatives for decades.

1.1.2 Non-Volatile Memory Technologies

Non-Volatile Memory is a memory technology that retains its data even after power crash or shutdown. Therefore, they can be used to store data and retrieve them later on without the need for constant power supply, and hence the non-volatility. In contrast, volatile memory technologies need constant power supply to continue to retain the data. Of the different non-volatile memory technologies, NAND and NOR flash memories are the most popular nowadays, especially as the dominant technology for storage [2, 16, 20, 55, 73].

NAND flash has the advantage of high density and faster write and erase latency compared to NOR flash. On the other hand, NOR flash has the advantage of faster reads at the cost of the density. These characteristics are attributed to the structure of these two technologies. NOR flash achieves the high read latency by providing direct access to every memory cell to be able to read data in byte granularity. This structure is also the reason
it is not as dense as NAND flash since more area is needed for the interconnects. On the other hand, NAND flash groups each sixteen to 64 cells together to share the same interconnect, which in turn reduces the number of interconnects needed compared to NOR and increases its density [55]. The cost of sharing interconnect is that NAND flash has to access data in page granularity (ranging from four to sixteen kilobytes) which results in higher read latency compared to NOR flash [62]. Based on these attributes, NOR flash is better suited for code execution because of its lower read latency. On the other hand, NAND flash is better suited for storage due to its better density and faster erase and write operations [15, 62, 73]. Nevertheless, both NOR and NAND flash NVMs have read and write latency that are orders of magnitude slower than DRAM [15, 55, 62]. Therefore, they cannot replace DRAM as main memory but they are excellent solutions as storage-class memory [16]. However, since flash memory is MOS-based, it faces the same scalability limitations that DRAM faces [15, 16, 55, 60, 68, 73].

Aside from flash memory technology, new NVM technologies have been advancing rapidly over the years. They have already been deployed as a storage class module that is much faster than NAND flash [7, 34]. They have also been deployed as memory modules to augment the DRAM and extend main memory’s capacity. NVMs have proven they are strong contenders to replace DRAM as a future main memory [3, 9, 11, 12, 34, 39, 43, 47, 65]. They are byte-addressable which means they can be accessed with the granularity of a cache block just like DRAM. They are also non-volatile, which means they can retain the data with almost zero refresh power. Furthermore, they are much denser and cheaper than DRAM. In the rest of this dissertation, we will refer to emerging NVM technologies that are used in main memory either to replace or augment the DRAM as Non-Volatile Main Memory (NVMM). We will discuss some of these technologies in more detail in chapter 2.
1.1.2.1 Challenges Facing NVM

Although promising, the new class of NVM technologies has some limitations, namely limited write endurance and relatively high write latency, albeit still comparable to DRAM [12, 34, 47]. Therefore it is imperative to keep the number of writes low. In addition to these limitations, the non-volatility of NVMM is both an advantage and a challenge. As we mentioned, being non-volatile means less power consumption. The challenge for using NVM as main memory is that the data remains in the NVM after shut down or power failure. Therefore, there is a concern of leaving the program memory in inconsistent state upon a crash. As a result, many software and hardware techniques have been proposed to address this consistency challenge. Most of these proposals perform persistence, which is the operation of making sure the data is made durable in the NVM. By durable, we mean in the fail-safe domain, which is any memory domain that can retain its data upon crash or power cut. Therefore, battery-backed buffers are considered part of the fail-safe domain. We will talk in more detail about persistence and durability in the next chapter.

1.1.2.2 Crash Consistency in NVM

Sometimes when applications modify critical data, they need these modifications to be done in an atomic way (i.e. either all the modifications commit or none of them). The reason is that we would not know where in the code to continue execution when restarting the system following a crash or power failure. Even if we knew where exactly in the code the crash happened, we do not know if some of the modifications we made were written to the NVMM as a result of normal cache eviction operation. Therefore, whenever the persistence operation is required to execute in an atomic way, there is a need for some form of a data log (i.e. backup) that we can roll back to if a failure happens in the middle of the persistence operation. Implementing these data log guarantees can have up to $2.3 \times$ execution time overhead and $2 \times$ more writes to the NVMM [5, 26, 35, 69].
1.1.3 Organization of This Chapter

In the rest of this chapter, we introduce the challenges facing NVMM that we address in this dissertation and our contributions to solving these problems. In the next chapter, we give more detailed background about each of these problems.

1.2 Contributions

1.2.1 Utilizing NVM in Application-Level Checkpointing

High-Performance Computing (HPC) systems rely on an increasing number of compute nodes and components for executing long-running scientific applications and algorithms. As the number of these computing nodes increases, the Mean Time To Interrupt (MTTI) decreases. Therefore, HPC systems often stop the application execution to take a checkpoint, i.e. save the entire address space and processor state to the durable filesystem (i.e. disk or SSD). Following a system interruption, the memory and processor state of the system are rolled back to the most recent checkpoint, which means the application would continue execution from the most recent checkpoint rather than restarting execution from the beginning.

Checkpoints can be taken for the entire system memory and address space (system-level checkpointing or SLC) or only for the application address space (Application-Level Checkpointing or ALC [18]). With ALC, the programmer or compiler determines when checkpoints are created and possibly which data is included in the checkpoint. Hence, ALC can reduce the amount of data that is checkpointed. The constant increase in number of compute nodes in HPC systems and decrease in MTTI means that some long-running applications are guaranteed not to finish before the next system interruption. Therefore, frequent checkpointing has become a must, not just an optimization, to ensure forward progress. Currently, the MTTI for the Titan supercomputer, one of the fastest supercomputers today, is around
160 minutes [1]. Each checkpoint takes around 55 seconds to commit (i.e. finish writing to the filesystem) [28]. However, this essential checkpointing process will soon become even more critical in HPC systems: For future exascale systems to achieve 90% progress rate, a checkpoint would need to finish in nine seconds [1]. Therefore, checkpointing in disk will become very challenging, if not prohibitive.

The non-volatility and speed of NVMMs can revolutionize checkpointing for failure-safety in HPC systems, which is a critical concern, especially with exascale HPC systems on the horizon [6, 26]. With NVMM, the main memory can hold the persistent state of a program, thereby serving the same role as a checkpoint. To achieve that, a consistent state can be constructed *in-place* in the NVMM utilizing the working data structures used by the applications. Consequently, only very minimal additional state beyond what the program already saves to memory needs to be recorded.

Nevertheless, depending on the application, a system failure while checkpointing data could leave the memory in inconsistent state. To deal with this problem, transaction-style hardware and software optimizations have been proposed. There are several ways to implement these transactions. One of these ways is Write-Ahead Logging (WAL) with undo log. The idea is to take a log of the data we are about to modify and make sure this log is made durable (i.e. persisted in NVM). Then we perform the transaction and make it durable by persisting it in the NVM. At that point, the log can be discarded. If a failure happens during the transaction execution or persistence, we do not know what was successfully persisted and what was not. Hence, the need for the log. In that case, recovery from failure is basically restoring the data from the log then continuing program execution from where the crash happened.

This process of logging data and making it persistent involves a significant number of writes and the performance overhead of logging can be significant [5, 26]. Therefore, we propose a technique called *Recompute* to get rid of logging completely. This technique works on loop-based algorithms. Since most applications running on supercomputers are
scientific algorithms, accelerating checkpointing in loop-based algorithms using our Re-compute technique can be very vital. Our Re-compute technique is based on the observation that if we track certain information about loops, such as the loop indices, we can determine where in the program the crash happened. We can then recompute the part of the data that we were computing or persisting when the crash happened. That way, we allow data to be inconsistent and get rid of the logs. If a crash happens, we only recompute the part of the data that was left inconsistent. We will give an analysis of the different styles of Re-compute and the design trade-offs between them in chapter 3.

1.2.2 Analysis of General Matrix Multiply Algorithms in NVMM

We discussed how our Re-compute technique can be used to get rid of logging while checkpointing loop-based algorithms in NVMM. Scientific applications rely heavily on linear algebra operations in the computation. Therefore, developing highly-optimized linear algebra libraries was and still is of utmost importance. There is a very rich literature of different libraries in academia and industry aimed at optimizing linear algebra algorithms. All of these libraries rely on a set of highly-optimized Basic Linear Algebra Subprograms (BLAS) [24] to implement more complex linear algebra algorithms. One of the most popular BLAS implementations is GotoBLAS [29]. Since General Matrix Multiplication (GEMM) is the most fundamental linear algebra operation, it is widely used in scientific applications. Therefore, GEMM is arguably the most important candidate for checkpointing in NVMM.

In this work, we analyze in-NVMM checkpointing of the widely-popular GEMM implementations from the GotoBLAS linear algebra library [29]. We study in-NVMM checkpointing overhead and the impact on NVMM write endurance. We also discuss the design choices to be considered when checkpointing GEMM in NVMM. Our analysis shows that the best-performing GEMM implementation, which is recommended in the GotoBLAS paper, incurs \(32 \times\) the number of writes compared to other implementations with lower performance. This significant number of writes, which has a major negative impact on the
write endurance of NVMM, could make other GEMM implementations that incur fewer writes more practical, even if they are a bit slower. We also show that if checkpointing were to be used for these GEMM implementations, which is very likely from our discussion in the previous section, the number of writes each implementation incurs would change. We will discuss GEMM and its different implementations in more detail in chapter 4.

1.2.3 Data Flushing and Maintenance

We discussed how NVMM can accelerate checkpointing significantly. We have also discussed how applications that modify important data structures must ensure their data is made durable in an atomic way to avoid leaving the memory in an inconsistent state upon crash. To achieve that, transaction-based techniques can be used to ensure the data are not lost upon failure. In these techniques, combinations of flush instructions followed by store fence are used to ensure durability beyond a certain point in the program. Flush instructions force the modified data out of the cache while the store fence blocks any flushes or stores that come after it in the program order from execution until the flush is complete (i.e. the data is written to the memory controller).

Many software and hardware optimizations have been proposed to address the added overhead of flushing and persisting logs [4, 5, 21, 36, 61, 64, 74]. Other techniques have been proposed to do away without logging altogether; one of these techniques is our Recompute proposal that we introduced in section 1.2.1. Nevertheless, whether it is logging-based or log-free, both techniques primarily target reducing the number of data to flush for persistence.

Although some of the software and hardware techniques accelerate the time it takes to persist a cache line, they are targeted at transaction-style workloads. Therefore, the proposed logging software and hardware techniques do not provide much benefit for this kind of log-free applications. Even for log-free applications or techniques, some flushes and fences are still needed.
To that end, we propose the PreFlusher, a lightweight hardware mechanism to predict and issue flush requests early. Our work is different than previous NVMM transactions hardware and software optimizations in that it accelerates persist operations regardless of how many they are. Therefore, both log-based or log-free applications can transparently benefit from our technique without having to change any line of code. We will discuss our programming model and our proposed design in chapter 5.

1.3 Summary

Taken together, the techniques described in this dissertation provide novel ways to incorporate NVMs in future systems as main memory. Our work is mainly focused on HPC systems. We study legacy scientific routines in context of NVM as main memory. We aim to incorporate NVMMs in the checkpointing process of these algorithms. We also provide detailed analysis of the highly-optimized BLAS-based implementation of GEMM and the considerations and modification that need to be made to be better suited for use with NVMM. To reduce the cost of persistence, we propose a lightweight hardware technique to accelerate cache line flush or write back operations no matter how many they are using our predictor-based hardware mechanism. This data flushing or write back is a core operation in using NVMM for checkpointing. Our technique is unique in that it is transparent to the programmer. Therefore, it would accelerate most algorithms and applications that use NVMM. On the whole system level, we propose a hardware-software co-design technique to accelerate page migration to and from the NVMM in heterogeneous memory systems.
In this chapter, we will cover the background needed for the work we present in this dissertation. We will start by giving a background on NVMs and the different technologies with the potential to either become main memory extensions or to completely replace the DRAM. Then we will talk about persistency models. After that, we will cover the PMEM instructions and programming model. We will explain the semantics of Intel’s x86 instructions: \textit{clflushopt} and \textit{clwb}.

### 2.1 Non-Volatile Memory

The market of Non-Volatile Memory (NVM) has been dominated by NOR and NAND flash memory for decades (also called Solid State Drive "SSD") [2, 16, 20, 55, 73]. Similar to DRAM,
Figure 2.1 (a) Regular NMOS with a gate controlling the flow of charge from the source to the drain through the silicone dioxide, (b) Floating-gate NMOS. The floating gate holds the charge. The silicone dioxide ensures persistence by insulating the floating gate from the channel.

these NVM technologies are composed of transistors based on Metal-Oxide Semiconductor (MOS). What gives flash memories the non-volatility attribute is that their cells are based on a special type of MOS called the Floating-Gate MOS (FGMOS). Figure 2.1 shows the structure of FGMOS compared to a normal MOS. The gate in both transistors is connected to a conductive terminal. Its function is basically controlling the transistor (i.e. switching it on or off). However, the FGMOS has an extra gate that floats inside the insulating silicone dioxide layer. This surrounding insulation allows the floating gate to trap any charge it contains and prevent it from leaking which gives flash memory its non-volatility attribute [55].

Emerging Non-Volatile Memory is a class of new memory technologies that are not based on MOS transistors. Examples of emerging NVM technologies include Phase-Change Memory (PCM) [11, 49], Memristor [80] and Spin-Transfer Torque RAM (STT-RAM) [43]. The main idea behind these technologies is to use material that can take two distinct states providing varying resistance. For example, PCM, which is the most mature among these technologies, is made of a material called chalcogenide glass. This material can take the form of either a chryssalline or amorphous phases as seen in figure 2.2. There are orders
of magnitude difference between the high resistivity of the amorphous phase and the low resistivity of the crystalline phase. The amorphous phase is achieved by applying a short and high voltage pulse to a terminal connected to the material. The voltage should be high enough to heat the material above its melting point before quenching it rapidly to solidify in the amorphous phase. To change its phase to crystalline, a lower and longer pulse is applied to cool down the material slowly.

Some of these technologies are very promising candidates to replace DRAM as main memory. They have read latencies that are not much higher than DRAM and while exhibiting sufficient write endurance. Table 2.1 shows the different latencies and attributes of the emerging NVMs. The latencies reported in the table reflect the performance of future versions of these memory technologies, not necessarily current performance. Although these technologies are still under development, we are beginning to see examples of emerging NVM-based products. One example is 3D-Xpoint, the result of collaboration between Intel and Micron. Their first product was a PCIe-based storage class NVM. It was released under the market names Intel Optane and Micron QuantX in 2017 [34]. Intel reported speeds up to 5.8× faster than NAND Flash SSD [34]. Their collaboration continued and resulted in the first Non-Volatile Main Memory (NVMM) module with a Dual-In Memory Module (DIMM) interface similar to DRAM. These main memory modules are used to extend the DRAM’s
Table 2.1 Comparison between NVMs, DRAM, and Nand-Flash.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Latency</th>
<th>Write Endurance</th>
<th>Cell size</th>
<th>Static power</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read</td>
<td>Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>10-50 ns</td>
<td>10-50 ns</td>
<td>$&gt;10^{15}$</td>
<td>6-8 $F^2$</td>
</tr>
<tr>
<td>Nand-Flash</td>
<td>20-100 $\mu$s</td>
<td>200 $\mu$s</td>
<td>$10^4$</td>
<td>4-5 $F^2$</td>
</tr>
<tr>
<td>PCM</td>
<td>50-80 ns</td>
<td>150 ns</td>
<td>$10^{-1}$-$10^4$</td>
<td>6-16 $F^2$</td>
</tr>
<tr>
<td>STT-MRAM</td>
<td>&lt;10 ns</td>
<td>12.5 ns</td>
<td>$&gt;10^{15}$</td>
<td>$37 F^2$</td>
</tr>
<tr>
<td>Memristor</td>
<td>&lt;10 ns</td>
<td>200 ns</td>
<td>$10^{16}$</td>
<td>$&gt;5 F^2$</td>
</tr>
</tbody>
</table>

Out of these memory technologies, PCM is considered the most mature at this moment. We note that Intel and Micron did not reveal the technology used in 3D-Xpoint; however, the speculations, based on the performance measurements, point to PCM as the likely technology used in the 3D-Xpoint. Therefore, in the remainder of the documentation, we assume PCM-level performance for NVMM in our analysis.

### 2.2 Crash Consistency in Persistent Memory

While memory consistency defines the order with which loads and stores become globally visible (i.e. visible by all processors or cores), memory persistency defines the ordering with which stores are made durable in NVM [63]. Over the years, various persistency models and libraries have been proposed including strict persistency [63], epoch persistency [22], buffered epoch persistency [22, 37], strand persistency [63], and transactional persistency [42, 74].

Applications that modify important data such as databases or filesystems must ensure the memory is left in a consistent state upon a crash. To achieve that, they must make their memory updates durable in an atomic way (i.e. either all updates are made durable or none
Transactions have long been used to achieve crash consistency in non-volatile storage. Implementing atomic and durable software transactions for NVMM can be a very complex and error-prone. Therefore, several libraries, such as Mnemosyne [74], NVML [64] and NV-Heaps [21] have been proposed to make data durable while ensuring their consistency in case of crash. These libraries can free the programmer from the burden of reasoning about and using native machine instructions at the cost of added overhead associated with the meta-data needed to maintain the persistent data in the NVMM. However, they can also inhibit low-level software optimizations.

Another way to program for persistent memory is to directly use the native machine instructions to ensure durability. This low-level programming allows higher levels of control over the use of persistent memory at the cost of higher complexity.

In both programming models, combinations of flush instructions followed by store or memory fences are used to achieve durability. In this work, we use the flush instructions offered by the x86 ISA to support programming for NVM.

## 2.3 Intel Persistency Programming Support

### 2.3.1 New Instructions

With the advent of NVMs, Intel announced a new PMEM persistency programming model, and published it at pmem.io [64]. PMEM added two new instructions to the x86 Instruction-Set Architecture (ISA): Cache Line Write Back (clwb) and Cache Line Flush Optimized (clflushopt), which is a more relaxed version of the legacy clflush instruction [64]. These two instructions can be used to write fail-safe software in conjunction with existing X86 instructions, such as cache line flush (clflush) and store fence (sfence).\(^1\) clflushopt is similar

---

\(^1\)Another PMEM instruction that was a part of PMEM is pcommit. The instruction was designed to flush a cache block from the memory controller buffer to the NVMM. However, Intel has deprecated pcommit because it requires the memory controller to have a "failure-safe" queue, where when a cache block is accepted by the
in functionality to \textit{clflush}. However, \textit{clflushopt} is optimized for performance by relaxing the ordering requirements of \textit{clflush}. In contrast to \textit{clflush}, \textit{clflushopt} does not impose strict ordering with all other stores. Instead, it is only strictly ordered with respect to stores to the same block address, giving some room to overlap the execution of stores to different blocks. Due to that, we rely on \textit{clflushopt} in our design. \textit{clwb} differs from both \textit{clflush} and \textit{clflushopt} in that it cleans a cache line by writing it back but retains the line in the cache. Another difference is that \textit{clwb} only orders stores to the same address, instead of ordering stores to the same block as in \textit{clflushopt} [33].

The last instruction that we use is \textit{sfence}. \textit{sfence} was originally designed for memory consistency to specify a point where all older stores have been made visible to other threads, but has now been extended to persistency. If placed right after PMEM instructions such as \textit{clwb} and \textit{clflushopt}, \textit{sfence} waits until the PMEM instructions are durable before it completes, while stalling younger stores or PMEM instructions from being visible to the cache until it completes.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.3}
\caption{Clflushopt \& CLWB journey, starting from the Store Queue (SQ), for different cache line states: (a) line is clean in the entire cache hierarchy, (b) line is dirty (i.e. has to be written back to the Memory Controller "MC) and the modifier is the flush issuer, (c) line is dirty but the modifier is not the flush issuer.}
\end{figure}

\footnote{queue, it can be considered durable. In light of this deprecation, we omit the use of \textit{pcommit} in our schemes.}
2.3.2 Clflushopt and Clwb Operation

Both clflushopt and clwb instructions execute similar to stores. They issue to both the Store Queue (SQ) and the Reorder Buffer (ROB). Then they remain in the ROB until they are done execution (i.e. the address is generated), then they retire. They remain in the SQ until it is time for them to access the L1 cache, off the critical path of the program’s execution. Figure 2.3 shows the operation of both instructions starting from the SQ.

When the L1 cache receives the flush request, it propagates it all the way down to the directory. If the cache line is dirty, the L1 sends the dirty data along with the request. In case of clflushopt, it also invalidates the cache line, even if it was clean.

The directory writes the cache line back to the memory controller, if it finds it is dirty. If the cache line is in modified state in another private cache, the directory forces the owner to write back the dirty line (and invalidate its copy, if the request is clflushopt), then it sends the cache line to the memory controller. Therefore, if the L1 cache that received the flush request finds that the line is invalid or shared, it still has to send the flush request to the directory to make sure there is no dirty version somewhere else in the cache hierarchy. After the directory sends the dirty cache line to the memory controller, it waited until it receives an acknowledgement back. When it does, it propagates this acknowledgement all the way up to the L1 flush requestor, which further forwards it to the SQ. Only then the flush instruction is considered complete and can safely be removed from the SQ.

2.3.3 Flush Instructions with Memory Barriers

To ensure that the outstanding stores and flushes in the SQ are completed (i.e. globally visible in the whole cache fabric) before newer stores or flushes can access the cache hierarchy, clwb and clflushopt instructions have to be followed by store fence (sfence) or memory fence (mfence). The problem with using fences is that they expose the latency of clwb and clflushopt instructions to the critical path of execution. Considering the semantics
of operation for these two flush instructions that we described above, the execution time overhead can be relatively significant.

### 2.3.4 PMEM Programming Model

Various persistency programming models have been proposed [22, 37, 53, 63, 64], including strict persistency [63], epoch persistency [22], buffered epoch persistency [22, 37], strand persistency [63], and transactional persistency [42, 74]. In contrast to those, Intel’s PMEM extension recognizes that not all applications need persistency support, hence it was designed to be flexible enough that it can be mixed with traditional programming models.

To illustrate the contrast, consider strict persistency. Strict persistency requires that when a store is *globally performed* (i.e. visible to other threads), it is also *durable* (its value is accepted by the NVMM). With PMEM, programmers specify which stores need to persist and in which order they persist. To illustrate this, suppose that we wish to persist a store to address X before a store to address Y, but do not care about persisting or ordering stores to addresses U and V. Then, we would write:

```
i1: st U, 5;
i2: st V, 1001;
i3: st X, 1;
i4: clwb X;
i5: sfence;
i6: st Y, -300;
```

Instructions i1 (store to U) and i2 (store to V) are not augmented with any ordering or persistency constraints. Instruction i3 (store to X), however, is followed by i5 (clwb), which forces the store X to be written back from the cache. The sfence (i5) stalls the next store instruction (i6) until all stores preceding it have been accepted by the NVMM (either received by a failure-safe buffer in NVMM or fully written into NVMM cells), guaranteeing durability. This example illustrates the instructions that need to be inserted by programmers.
in order to specify which stores must persist and the ordering of the persist relative to other stores. We also note that programmers can construct other persistency models such as strict and epoch persistency using PMEM instructions.

2.3.5 Durable Transactions with Write-Ahead Logging

Using the PMEM programming extensions we can force stores to become durable in an order of our choosing. However, this alone does not guarantee that memory is left in a failure-safe state since a failure could occur while making changes to data. What is also needed are atomic updates such that either all updates become durable or none of them do.

Transactions have long been used to achieve failure safe updates to non-volatile storage [42, 74]. We use write-ahead logging or WAL [56] for our implementation of durable transactions. WAL makes a durable undo log of all the data that will be updated in the transaction before making any modifications. If a failure occurs during the transaction, we can use the undo log to recover the correct data. A durable transaction can be constructed in software using the following steps:

- Perform the undo-logging by making a copy of the data we want to update, and make the undo-log durable.

- Set a transactionRunning bit indicating the beginning of a transaction and make the bit durable.

- Update the data that needs to be modified and make the updates durable.

- Unset the transactionRunning bit to mark the transaction complete and make the bit durable. At this point, the log is no longer needed.

The recovery mechanism is quite simple. If a failure occurs, we check the transactionRunning bit. If it was set to true, that means the failure occurred in the middle of a transaction.
In that case, we restore all data using the entries in the undo-log. Then, we resume execution from the point of the failed transaction. On the other hand, if the transactionRunning was unset, it means there was no transaction running when the failure happened, so no further recovery actions are needed on the data protected by the transaction.

Now that we discussed PMEM instructions and undo logging, we will apply them to tiled matrix multiplication to make it failure-safe. We chose tiled matrix multiplication as a case study for checkpointing due to its popularity and importance.

![Tiled Matrix Multiplication Code](image)

**Figure 2.4 Tiled Matrix Multiplication Code**

### 2.3.6 Tiled Matrix Multiply (TMM)

Before we show how to apply PMEM-style undo logging to tiled matrix multiplication, we will give some background about tiled matrix multiplication algorithms. Tiling is a well-known cache optimization technique to improve temporal locality and reduce the cache miss ratio. We use a standard 6-loop tiling for matrix multiplication. We refer to the tile size as bsize (as in block size). In this version, tiling splits a, b, and c into tiles or blocks of bsize × bsize. The operation of the tiled matrix multiplication (tmm) is shown in Figure 2.4, and the corresponding code is shown in Figure 2.5. We will briefly walk through the matrix multiplication example.

\[ tmm \]

\[ tmm \text{ consists of 6 loops: } kk, ii, jj, i, j, \text{ and } k. \text{ The } kk \text{ loop splits matrices } a \text{ and } c \text{ into vertical groups; each vertical group consists of a number of } bsize \text{ columns. The } kk \text{ loop also} \]
for (kk=0; kk<n; kk+=bsize) {
    for ( ii =0; ii <n; ii +=bsize) {
        for ( jj =0; jj <n; jj +=bsize) {
            for (i=ii; i<(ii+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum = c[i][j];
                    for (k=kk; k<(kk+bsize); k++)
                        sum += a[i][k]*b[k][j];
                    c[i][j] = sum;
                } //end of for j
            } //end of for i
        } //end of for jj
    } //end of for ii
} //end of for kk

Figure 2.5 Tiled Matrix Multiplication.

splits the b matrix into horizontal groups; each horizontal group consists of bsize rows. The ii loop splits each of matrix a's vertical kk groups into square bsize × bsize tiles as illustrated in Figure 2.4. Similarly, The jj loop splits each of matrix b's horizontal kk groups into square bsize × bsize tiles. Moving to the innermost loops i, j, and k. As the iterations names indicate, each of these loops will move through within a tile that was made by the corresponding upper loops, ii, jj, and kk, respectively.

The tiled matrix multiplication operation starts with a kk group from matrix a and its corresponding kk group from matrix b. Then the ii loop selects the first tile from a's kk group, and the jj loop selects the first tile from b's kk group. After that, the i loop goes over all of the strides from a's ii group and multiplies them by the selected all the vertical strides from matrix b's jj tile. At the end of the i loop, the result is a (bsize × bsize) square in from matrix c (corresponding to the first ii, jj tiles) with intermediate values (not the final values). Then, we move on to the next jj tile from b's kk group. When all jj tiles of b's kk group are done, the ii loop moves to the next ii tile from the a's kk group. When all ii tiles of a's kk group are done, the kk loop moves on to the next vertical kk group of a and the next horizontal kk group of b. The operation keeps going until all kk groups have been
computed. We note that each $kk$ loop iteration touches every element of the result matrix $c$ once. Therefore, at the end of the computation, we would have written to every element of matrix $c$ a number of times equal to $\frac{n}{\text{size}}$.

### 2.3.7 TMM Crash Consistency with Traditional Checkpointing

A traditional solution for making an application failure-safe is by creating checkpoints of the application. A checkpoint typically includes a snapshot of memory and the full processor context (all registers, PC, etc.). The snapshot of memory should be taken at a particular instruction, such that any older instructions have their results reflected in the state but none of the younger instructions have taken effect. The first step toward a consistent snapshot is interrupting all processor cores using a precise interrupt mechanism. Then, the computation state in all cores, including the program counter, register files, and memory, must be saved to secondary storage. Note that any dirty data in the caches must be written back or flushed to the main memory before this snapshot is taken.

Since we are targeting a system with NVMM, instead of copying the checkpoint to secondary storage we can copy it to a region in the NVMM instead. In our example of tiled matrix multiply in this section, a checkpoint is created by copying matrices $a$, $b$, and $c$ elsewhere in non-volatile memory. To make evaluation simpler, we ignore saving the processor state. The performance of this model is more efficient than system-level checkpointing because the checkpoint excludes non-application address space. In addition, the cost of saving the processor state is not included since we ignore it in our model. However, this model is still less efficient than application-level checkpointing in that the checkpoint could be made even smaller if we consider deeper application information. For example, if we know matrices $a$ and $b$ are not needed after the matrix multiplication, only the result matrix $c$ needs to be included in the checkpoint.
2.3.8 TMM Crash Consistency with Application-Level Checkpointing

Another way to checkpoint data is to save the key data structures of the application. Upon a crash, we enter the recovery mode during which we use the checkpointed data to reconstruct the application to the point of failure.

2.3.8.1 Tiled with Write Ahead Logging

One way to implement application-level checkpointing for TMM is to use write-ahead logging. We use Intel PMEM instructions to achieve that. For the tmn code, it is clear that updating the result matrix needs to be wrapped into a durable transaction since the matrix needs to be consistent. Upon recovery, we use the undo logs to recover the inconsistent data and redo the transaction. However, in order to continue execution from the point of failure, we have to know how much of the work was already completed. For loop-based codes, a straightforward solution is to log all of the current loop indices by explicitly writing them to memory. Upon failure, we recover a consistent state by reading these indices and determining where in the loop-nest to resume execution.

The transactionalized tmn code that illustrates this approach is shown in Figure 2.6. Each transaction is the body of the $ii$ loop iteration – a larger or smaller transaction on a different loop is also possible. We must log the elements of matrix $c$ and the indices before updating them. The logging part is shown in lines 4 to 7. In this part, we log all the $c$ elements we are about to modify inside the $ii$ loop, which is basically a full horizontal $kk$ group of a number of $b$size rows. We log the lastPersistedII variable, as well, because we will update it in the transaction (line 9). Then, we make the logs durable by flushing them from the caches using clflushopt instructions (lines 10-14). Then we use an sfence as shown in line 15 to ensure that flushing the log is complete (i.e. the log is made durable). After that, in line 18, we set the insideTx bit indicating we are about to begin updating the data. Next, we flush it using clflushopt followed by an sfence to make sure the update to insideTx is...
for (kk=0; kk<n; kk+=bsize) {
    for(ii=0;i<n; i+=bsize){
        //Step1: Create the undo log
        for(r=ii; r<(ii+bsize); r++) {
            for (l=0; l<n; l++) {}
            cLog[r][l] = c[r][l];
        }
    }
    //Step2: Make the log durable
    //lastPersistedII = lastPersistedII;
    for (r=ii; r<(ii+bsize); r++) {
        for(l=0; l<n; l+=bsize)
        CLFLUSHOPT(&logC[r][l]);
    }
    CLFLUSHOPT(&lastPersistedII);
    SFENCE;
    //Step3: Set the flag marking beginning
    //of the transaction and make it durable
    insideTx = true;
    CLFLUSHOPT(&insideTx);
    SFENCE;
    //Step4: Enter the transaction
    for (jj=0; jj<n; jj+=bsize) {
        for (i=ii; i<(ii+bsize); i++) {
            sum = c[i][j];
            for(k=kk; k<(kk+bsize); k++) {
                sum += a[i][k]*b[k][j];
            }
            c[i][j] = sum;
        }
    }
    //end of for jj
    //end of for ii
}

Figure 2.6 Tiled Matrix Multiplication With Logging

durable before moving to execute the transaction itself (lines 19 & 20). At this point, we would have a durable log of the data we are about to modify. Then, we begin the transaction execution (lines 22-40). After finishing the transaction, we reset the insideTx bit and make it durable, indicating that the transaction has completed (lines 43-45). Once we have reset the insideTx flag, we no longer need the log. We follow a similar approach for updating the indices (lines 48-61).
3.1 Introduction

As we discussed in section 1.2.1, checkpointing commit time will pose a major challenge in future exascale systems. NVMM provides a great opportunity for improving application-level checkpointing by taking an in-place checkpoint instead of having to checkpoint in the file system. A common approach to in-place checkpointing is logging, where modifications to key data structures are made durable at a transaction granularity. If the programmer
wraps changes that need to be durable together in a transaction, data structures can always be in a consistent state. Durable transactional logging has been applied to the file system (e.g. BPFS [22]), heap (e.g. NV-HEAPS [21], NVML [64], and Mnemosyne[74]), and linked data structures (e.g. [37, 42, 63]). We implemented logging using Intel PMEM instruction extension [64] on a machine model built on gem5, and measured the performance of tiled matrix multiplication. Our implementation of logging reduces the execution time and write overheads compared to system-level checkpointing: 8% vs. 207% execution time overheads, and 111% vs. 330% additional NVMM writes. While this represents much improvement in execution time overheads, the write overheads are still unacceptable. Hence, we explore another avenue.

In this chapter, we present a new Recompute approach, where we do away with the log entirely and allow the memory state to become inconsistent. Some of the result matrix could become or appear inconsistent at the point of failure. However, we track enough additional information so that we can discard the possibly inconsistent state and re-run computation as needed. Recompute has the interesting property of being restore-free, in that no checkpoint needs to be copied back to memory or consistent state restored using a log. We demonstrate our Recompute approach on loop-based codes that are essential in HPC workloads, and present it in detail for matrix multiplication. Loop-based code is interesting because we must consider optimizations such as tiling in the study.

We show that Recompute achieves failure safety with almost no penalty in execution time and write endurance. For tiled matrix multiplication, we find that Recompute has an execution time overhead of only 5% compared to an unmodified version of the kernel, while adding only 7% write overhead. Across various workloads, the geometric mean execution time overhead ratio is 1.03× for Recompute vs. 1.91× for traditional checkpointing, and write overhead ratio of 1.08× vs. 1.38× for traditional checkpointing.
3.2 Our Recompute Approach

As we mentioned in chapter 2, we use Tiled Matrix Multiplication (TMM) as a case study due to its popularity and importance. Figure 2.6 in chapter 2 shows the code to checkpoint tmm using write-ahead undo logging. While tmm with logging allows us to avoid creating copies of the matrices, relying on logs still incurs significant performance overheads. In addition, the log must be made durable, thereby adding execution time and write overhead and reducing the write endurance of NVMM.

We propose a new approach that does not rely on durable transactions for updating matrix \( c \). Instead, it persists the result matrix as it goes, in a non-atomic manner. Hence, while updating the \( c \) matrix, there is no guarantee of precisely consistent state. However, we do make all updates durable and periodically wait for them to finish. If failure occurs before we have made the updates to \( c \) durable, it is possible that some elements of the \( c \) matrix are in an inconsistent state. However, because we continue to atomically update induction variables in the loop nest, we know the exact region of the \( c \) matrix that may be inconsistent. If a failure occurs, we discard just this state and recompute their values to restore the state of the matrix back to where it was at the point of the failure. Thus, we make computation faster and simpler, but at a cost of more complex and longer recovery.

3.2.0.1 Checkpointing Granularity

The traditional system-level checkpointing does not have restrictions on when the checkpoint can be taken. The reason is that at any point in the program execution, the system stops and takes a snapshot of the whole address space and processor state. Since application-level checkpointing does not save the whole address space, the programmer has to reason about where in the code to stop and save the critical data structures.

For our 6-loop tiled matrix multiply algorithm, shown in figure 2.5 in chapter 2, we can stop the algorithm at any point to save the data. For example, we can stop after each \( kk \)
loop iteration to save the cells of the result matrix that were updated during the iteration. In that case, if a crash happens while we are in the middle of a \( kk \) iteration, we fall back to that last value of \( kk \) that we saved and we lose any progress since then. We can also choose to stop the program execution and save the modified cells of the result matrix after each \( ii \) loop iteration. Since the \( ii \) loop is inside the \( kk \) loop, stopping after each \( ii \) iteration means higher frequency of checkpointing. But it also means that if a crash happened, we only lose the progress of the most recent \( ii \) iteration. As a result, the recovery of the more aggressive \( ii \) granularity is better than the \( kk \) granularity, but at the cost of more frequent checkpoints and higher performance overhead.

Although recovery is the rare case, there is a limit to how long it takes. This limit is needed to ensure forward progress of the program execution. With applications’ data sets increasing rapidly and Mean Time To Failure (MTTF) getting more critical with the advent of exascale computers, the time to recover from one \( kk \) loop could easily surpass the MTTF which would render checkpointing pointless, altogether. Therefore, the recovery time is an important design choice to ensure forward progress of the program execution.

For our case study tiled matrix multiply, we call this choice the checkpointing granularity. We can stop to checkpoint at the end of every \( jj, ii, jj, ii, or kk \) loop iteration. In this work, we evaluate two checkpointing granularities: \( jj \) and \( ii \).

### 3.2.0.2 Recompute Checkpointing Algorithm

Figure 3.1 shows the algorithm of our Recompute approach with \( ii \) granularity. In this code, we compute a full \( ii \) iteration before making all updated values durable (similar to the durable transaction example). In lines 13 to 17, we make all the \( c \) elements we modified in the \( ii \) loop durable by flushing them to NVMM. Then, we update the \( lastPersitedII \) variable to indicate which loop of \( ii \) we completed, so that we continue from this loop after recovery (lines 18-20). Upon the completion of the \( sfence \) instruction in line 20, the modified elements of matrix \( c \) are guaranteed durable. Thus, with Recompute, we eliminated the need for
for (kk=0; kk<n; kk+=bsize) {
    for (ii=0; ii<n; ii+=bsize) {
        for (jj=0; jj<n; jj+=bsize) {
            for (i=ii; i<(ii+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum = c[i][j];
                    for (k=kk;k<(kk+bsize);k++)
                        sum += a[i][k]*b[k][j];
                    c[i][j] = sum;
                }
            }
            for (i=ii; i<(ii+bsize); i++){
                for (j=0; j<n;j+=16) {
                    CLFLUSHOPT(&c[i][j]);
                }
            }
            SFENCE;
        }
    }
}

Figure 3.1 Tiled Matrix Multiplication with Recompute technique. Granularity of checkpoint is ii.

logging while updating the c elements and the lastPersistedII variable. However, we still use logging for updating the indices (lines 22-35). From this technique we can see that we remove much overhead from the normal execution path but add more burden on the recovery code to recompute all the previous parts of the elements of matrix c, from the beginning up until the point of the failure.

We also show the algorithm for the jj granularity in figure 3.2. As we can see the code becomes more complex as we step inside the ii loop iteration. This complexity also means potential negative performance impact in favor of faster recovery.

3.2.0.3 Recovery Mechanism

Figures 3.4 and 3.3 shows the recovery steps for the ii and jj granularities, respectively. The recovery starts by setting the indices for the range of c matrix cells we will have to recompute. This range is determined from the lastPersistedII variable that we saved during
```c
for (kk=0; kk<n; kk+=bsize) {
    for (ii=0; ii<n; ii+=bsize) {
        for (jj=0; jj<n; jj+=bsize) {
            for (i=ii; i<(ii+bsize); i++) {
                for(j=jj; j<(jj+bsize); j++) {
                    sum = c[i][j];
                    for(k=kk;k<(kk+bsize);k++)
                        sum += a[i][k]*b[k][j];
                    c[i][j] = sum;
                }
            }
            SFENCE;
            lastPersistedII += bsize;
            CLFLUSHOPT (&lastPersistedII);
            SFENCE;
        }
    }
}

Figure 3.2 Tiled Matrix Multiplication with Recompute technique. Granularity of checkpoint is jj.
```
//Step 1: Set the index of the tile we will recompute
if(insideTxII) {
    lastPersistedJJ = firstLoop-tile;
    lastPersistedII = lastPersistedIILog + bsize;
    return; //recovery complete
} else if (insideTxKK) {
    lastPersistedJJ = firstLoop-tile;
    lastPersistedII = -tile;
    lastPersistedKK = lastPersistedKKLog + bsize;
    return; //recovery complete
}
firstJJ = lastPersistedJJ+bsize;
if(firstJJ >= lastLoop){
    return; //recovery complete
}
firstII = lastPersistedII+bsize;
lastPersistedKKLoop = lastPersistedKK+(2*tile);

//Step 2: Reset (zero out) the cells we will recompute
for(i=firstII; i<(firstII+bsize); i++) {
    for(j=firstJJ; j<(firstJJ+bsize); j++)
        c[i][j] = 0;
}

//Step 3: Recompute the affected cells up until the kk in which we crashed
for(kk=0; kk<lastPersistedKKLoop; kk+=tile) {
    for(i=firstII; i<(firstII+bsize); i++) {
        for(j=firstJJ; j<(firstJJ+bsize); j++) {
            sum = c[i][j]; the value of the current element
            for(k=kk; k<(kk+bsize); k++)
                sum += a[i][k]*b[k][j];
            c[i][j] = sum;
        }
    }
}

//Step 4) Make the recomputed cells durable
for (i=firstII; i<(firstII+bsize); i++) {
    for (j=firstJJ; j<(firstJJ+bsize); j+=tile)
        CLFLUSHOPT (&c[i][j]);
}
for(i=firstII; i<(firstII+bsize); i++) {
    for(j=firstJJ; j<(firstJJ+bsize); j++)
        c[i][j] = 0;
}
}

Figure 3.3 Recovery pseudo-code for Tiled Matrix Multiplication with jj granularity.
//Step 1: Set the index of the tile we will recompute
  if (insideTxKK) {
    lastPersistedII = -tile;
    lastPersistedKK = lastPersistedKKLog + bsize;
    return; //recovery complete
  }
  firstII = lastPersistedII + bsize;
  if (firstII >= n) {
    return; //recovery complete
  }
  lastPersistedKKLoop = lastPersistedKK + (2*tile);

//Step 2: Reset (zero out) the cells we will recompute
  for (i=firstII; i<(firstII+bsize); i++) {
    for (j=firstLoop; j<lastLoop; j++)
      c[i][j] = 0;
  }

//Step 3: Recompute the affected cells up until the kk in which we crashed
  for (kk=0; kk<lastPersistedKK; kk+=bsize) {
    for (jj=0; jj<n; jj+=bsize) {
      for (i=firstII; i<(firstII+bsize); i++) {
        for (j=jj; j<(jj+bsize); j++) {
          for (k=kk; k<(kk+bsize); k++)
            sum += a[i][k]*b[k][j];

          c[i][j] = sum;
        }
      }
    }
  }

//Step 4) Make the recomputed cells durable
  for (i=firstII; i<(firstII+bsize); i++) {
    for (j=0; j<n; j+=tile)
      CLFLUSHOPT (&c[i][j]);
  }
  SFENCE;
  lastPersistedII += bsize;
  CLFLUSHOPT (&lastPersistedII);
  SFENCE;

Figure 3.4 Recovery pseudo-code for Tiled Matrix Multiplication with ii granularity.
normal program execution. After that, in step 2, we reset (zero out) all the elements of matrix $c$ that need to be recomputed. This removes any intermediate values in these elements, and prevents any potential consistency problem. After that, in step 3, we recompute the elements we zeroed out in step 2 from the beginning up until the loop iteration in which the failure occurs. We determine this loop iteration from the lastPersisted copy we saved during normal program execution (i.e. lastPersistedII in figure 3.1 and lastPersistedJJ in figure 3.2). Upon the completion of the step 3, we would have returned to the state right before the crash happened. Finally, in step 4, we use a loopnest of clflushopt instructions followed by an sfence to make the recomputed values durable.

### 3.2.1 Hybrid Recompute/Checkpointing

One limitation of the Recompute approach is the potential large amount of work that must be repeated for the elements of matrix $c$ that were left in an inconsistent state, in the event that a crash occurs after running for a long time. This may result in a long recovery time. In the worst case, for a huge matrix, the recovery time may approach or even exceed the MTTF.

To avoid an overly lengthy recovery time, we can periodically save matrix $c$ so that if a failure occurs, recomputing matrix elements can commence from the saved copy, rather than starting over from the beginning. The recovery code only needs to re-execute from the iteration in which the copy was made until the point of failure. Furthermore, we can devise approaches that infrequently save a copy of each element by spreading it over many iterations of the outermost loop nest. For example, if we save $\frac{1}{64}$ of the matrix at every loop, the entire matrix $c$ is fully copied every 64 iterations of the algorithm. We can determine which part of the matrix is scheduled for copying by using a modulo of the iteration index. Similarly, the recovery code can be changed accordingly to determine how much recomputation is needed based on the iteration in which the last copy was taken.
Table 3.1 The baseline system configuration.

<table>
<thead>
<tr>
<th>Processor</th>
<th>8-core, 2GHz, out-of-order x86-64, 8-wide issue/retire ROB: 192 Fetch/Issue/Store Queues: 32/32/32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coherence Protocol</td>
<td>Ruby System, MESI Two Level Protocol</td>
</tr>
<tr>
<td>L1I &amp; L1D Cache</td>
<td>private, 2-cycle access, 64KB, 16-way, 64B block</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared, 20-cycle access, 1MB tile/core, 16-way, 64B block</td>
</tr>
<tr>
<td>MSHRs</td>
<td>32</td>
</tr>
<tr>
<td>DDR-based PCM Main Memory</td>
<td></td>
</tr>
<tr>
<td>Capacity</td>
<td>8GB</td>
</tr>
<tr>
<td>Organization</td>
<td>1 channel, 2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address mapping, FRFCFS scheduling policy</td>
</tr>
<tr>
<td>PCM latencies</td>
<td>50ns read, 150ns write</td>
</tr>
<tr>
<td>DDR Timing</td>
<td>tRCD 55ns, tWR 150ns, tCL 12.5ns, 64-bit bus width, 800 MHz Clock</td>
</tr>
<tr>
<td>Read &amp; write queues</td>
<td>32-entry RQ &amp; 64-entry WQ, 85% WQ high threshold, 50% low threshold</td>
</tr>
</tbody>
</table>

3.3 Methodology

3.3.1 Simulation configuration

We evaluated our Recompute and Hybrid techniques using Gem5 [59], an open source cycle-accurate full system simulator. Our simulator uses x86-64 instruction set architecture (ISA). It models a detailed 4-way out-of-order processor pipeline for each core, with parameters shown in Table 3.1. The table also shows parameters for the 2-level caches that we model, with private per-core L1 caches and an L2 cache shared by all cores. The memory hierarchy model is built on top of Ruby [66], with MESI protocol keeping the L1 caches coherent with respect to one another and with respect to the shared L2 cache. We modeled a Phase Change Memory (PCM) type of NVMM based on the DDR timing parameters shown in
the table. This configuration for PCM is in based on the widely used parameters from Lee et al. [46] and is consistent with PCM values shown in chapter 2. It is also consistent with previous work in the area of NVMM [5, 10, 13, 14, 36, 46, 52]. These parameters would yield read and write latency of 60ns and 150ns, respectively. We note that if the latencies are higher, the relative benefits of our Recompute and Hybrid Recompute schemes will increase, because they incur the fewest writes to the NVMM.

We implemented Intel PMEM instructions, namely clflushopt (clflush is implemented but unused). The ordering constraints of clflushopt are implemented as described in the Intel manual [33]. Specifically, clflushopt is ordered only with respect to memory fences (including sfence and mfence), and with respect to older loads/stores to the same cache line address. Similar to stores, clflushopt accesses the cache after it is retired from CPU pipeline. Moreover, clflushopt becomes durable and the instruction completes when the dirty cache block has been written back to the buffer in memory module.

### 3.3.2 Benchmarks

Table 3.2 shows multiple approaches applied to the tiled matrix multiplication that we evaluated. tmm is the baseline tiled matrix multiply with no persistence or checkpointing at all. tmm_CP is the Checkpointing approach. tmm+R, tmm+L and tmm+HR are our Recompute, Logging, and Hybrid Recompute schemes, respectively. We evaluated two persistence granularities, namely ii loop granularity and jj loop granularity, with the ii granularity being the default (e.g. in Figure 3.1). For most experiments, we run the matrix multiplication with eight worker threads plus one master thread. They run on nine cores.

Other benchmarks that we evaluated include LU decomposition (LU), Fast Fourier Transform (FFT), Gaussian elimination (Gauss), and 2-dimensional convolution (2D-conv). These benchmarks constitute kernels that are frequently used in the high performance computing domain and beyond. They are taken from [71, 77, 78] and the Splash-2 benchmark suite [79]. Table 3.3 shows the input that we use.
Table 3.2 Multiple approaches applied to the tiled matrix multiplication to achieve write durability of the resultant matrix. The matrix is $1024 \times 1024$ in size, and the block size is 16 in order to align them to the cache block size. Thus, to persist one stride, only one `clflushopt` is required.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmm</td>
<td>Tiled matrix multiplication</td>
</tr>
<tr>
<td>tmm+CP</td>
<td>Checkpointing (1 checkpoint)</td>
</tr>
<tr>
<td>tmm+R_ii</td>
<td>Recompute (ii granularity)</td>
</tr>
<tr>
<td>tmm+R_jj</td>
<td>Recompute (jj granularity)</td>
</tr>
<tr>
<td>tmm+L_ii</td>
<td>Transaction with Logging (ii granularity)</td>
</tr>
<tr>
<td>tmm+L_jj</td>
<td>Transaction with Logging (jj granularity)</td>
</tr>
<tr>
<td>tmm+HR_ii_x32</td>
<td>Hybrid Recompute (ii granularity), checkpoint interval $32 \times$ of tmm+CP</td>
</tr>
<tr>
<td>tmm+HR_jj_x32</td>
<td>Hybrid Recompute (jj granularity), checkpoint interval $32 \times$ of tmm+CP</td>
</tr>
<tr>
<td>tmm+HR_ii_x64</td>
<td>Hybrid Recompute (ii granularity), checkpoint interval $64 \times$ of tmm+CP</td>
</tr>
<tr>
<td>tmm+HR_jj_x64</td>
<td>Hybrid Recompute (jj granularity), checkpoint interval $64 \times$ of tmm+CP</td>
</tr>
</tbody>
</table>

In all of our simulations, we execute approximately 250 million instructions, on average, to warm up the caches and other structures, and we simulate and report the timing for an additional 300 million instructions, on average.

Table 3.3 Other benchmarks we evaluated.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>1-million nodes vector FFT</td>
</tr>
<tr>
<td>LU</td>
<td>1k-square input matrix LU decomposition</td>
</tr>
<tr>
<td>Gauss</td>
<td>4k-square input matrix gauss elimination</td>
</tr>
<tr>
<td>2D-conv</td>
<td>1k-square input matrix 2D convolution</td>
</tr>
</tbody>
</table>
3.4 Evaluation

We evaluate the execution time overhead of Checkpointing and compare it against Logging, and our Recompute and Hybrid Recompute schemes. We modeled Checkpointing optimistically by only counting the time to store all three matrices (a, b, and c) to NVMM, while ignoring the time to write to the file system, context switching, register file saving, and the precise interrupt. In addition, we further reduced the time it takes to write the checkpoint to the NVMM in our Checkpointing model by using the x86-64 SSE quadword store instruction, which is a SIMD vector store.

3.4.1 Execution Time and Number of Writes

Table 3.4 Execution time and number of write comparison between Checkpointing (tmm+CP) and different granularity of our Logging (tmm+L), Recompute (tmm+R) and Hybrid Recompute (tmm+HR) schemes, normalized to the base tiled matrix multiplication (tmm).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Exe Time</th>
<th>Num Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>base (tmm)</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>tmm+CP</td>
<td>3.07</td>
<td>4.30</td>
</tr>
<tr>
<td>tmm+L_ii</td>
<td>1.08</td>
<td>2.11</td>
</tr>
<tr>
<td>tmm+L_jj</td>
<td>1.09</td>
<td>2.23</td>
</tr>
<tr>
<td>tmm+R_ii</td>
<td>1.05</td>
<td>1.07</td>
</tr>
<tr>
<td>tmm+R_jj</td>
<td>1.06</td>
<td>1.15</td>
</tr>
<tr>
<td>tmm+HR_ii_x32</td>
<td>1.08</td>
<td>1.13</td>
</tr>
<tr>
<td>tmm+HR_ii_x64</td>
<td>1.07</td>
<td>1.08</td>
</tr>
<tr>
<td>tmm+HR_jj_x32</td>
<td>1.09</td>
<td>1.20</td>
</tr>
<tr>
<td>tmm+HR_jj_x64</td>
<td>1.08</td>
<td>1.16</td>
</tr>
</tbody>
</table>

Table 3.4 shows the execution time and number of writes for all of the schemes we study, normalized to the base tiled matrix multiplication which is not failure safe. The number of
writes represents the number of L2 writebacks plus cache line flushes.

As can be seen in the table, Checkpointing (tmm+CP) more than triples the execution time (3.07×) and quadruples NVMM writes (4.3×). This is because a snapshot of the matrices are copied to another memory location in the NVMM. The number of writes Checkpointing incurs is troublesome since NVMM often has limited write endurance. Logging (tmm+L) incurs acceptable small execution time overheads: 8% and 9% for the $ii$ and $jj$ granularities, respectively. However, Logging causes a significant increase in the number of writes: 2.11× & 2.23× for the $ii$ and $jj$ granularities, respectively. Although lower than those in Checkpointing, the number of writes overheads are still problematic for NVMM write endurance. Note that the $ii$ granularity yields lower execution time and write overheads compared to the $jj$ granularity. This is expected as the $ii$ loop envelopes the inner $jj$ loop. Recompute simultaneously achieves low execution time overheads (5% and 6% for the $ii$ and $jj$ granularities, respectively) and low write overheads (7% and 15% for $ii$ and $jj$ granularities, respectively). This shows that for loop-based code, it is possible to achieve failure safety without incurring much execution time or write overheads.

Finally, we also evaluated the Hybrid Recompute scheme with 32× and 64× the checkpoint interval of tmm+CP (both denoted by tmm+HR prefix). As expected, less frequent checkpointing reduces both execution and write overheads: 7% with tmm+HR$_{ii}$ x64 vs. 8% for tmm+HR$_{ii}$ x32. The same observation applies for the $jj$ granularity as well. Compared to Recompute, Hybrid Recompute with 64× checkpointing interval incurs 1-2% slightly higher execution time and write overheads. However, the recompute effort during failure recovery is much lower now, and is bounded by 64 iterations. Compared to Checkpointing, the overhead is much lower since only a small part of matrix $c$ is copied at each iteration.

### 3.4.2 Sensitivity to Checkpointing Frequency

The high execution time and number of writes overheads for Checkpointing are closely related to the frequency of taking checkpoints. On a system that takes frequent checkpoints,
Figure 3.5 Extrapolation of execution time overheads normalized to base for Checkpointing (tmm+CP) and our ii-granularity Hybrid Recompute with matrix saving frequency of 64×.

execution time and write overheads will be higher than on a system that takes checkpoints less frequently. In contrast to Checkpointing, the overheads of Logging, Recompute, and Hybrid Recompute are constant (i.e. we incur this overhead every loop iteration). Thus, a question arises: can the frequency of taking checkpoints be reduced such that the execution time overhead for Checkpointing is equal to or lower than Hybrid Recompute?

To answer this question, we plot the execution time of Checkpointing normalized to Hybrid Recompute at ii granularity at 64× the checkpointing frequency, as we vary the number of kk loop iterations. This is shown in Figure 3.5. We note that the execution time speedup ratio of Checkpointing decreases inversely proportionally as the number of kk loop iteration increases, as the checkpoint creation overhead is amortized across more iterations, whereas it remains constant for tmm+HR. While we have not collected results for a large number of kk iterations, we extrapolated the tmm+CP curve using a least square method. The intersection of the two curves is at 42. Note, however, that this number is not exact as we relied on extrapolation. The key point, though, is that there is a loop iteration
count $N$ such that checkpointing may become cheaper than *Recompute*, and $N$ is likely quite high.

Repeating the same methodology for the number of writes, the parity point between the write overhead of *Checkpointing* and *Hybrid Recompute* occurs at 155 $k$ loop iterations, which is significantly higher than the ratio we observed for the execution time overhead.

Overall, system designers need to take these intersection points into account when choosing an approach to achieve failure safety. If they are high (especially when they approach or exceed MTTF), our *Recompute* and *Hybrid Recompute* are more attractive. However, when the intersection points occur at a low iteration count, *Checkpointing* is acceptable.

Table 3.5 Normalized execution time as the thread count varies from 1 to 12 threads, for various schemes.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>1T</th>
<th>4T</th>
<th>8T</th>
<th>12T</th>
</tr>
</thead>
<tbody>
<tr>
<td>base (tmm)</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>tmm+CP</td>
<td>1.88</td>
<td>2.85</td>
<td>3.07</td>
<td>3.27</td>
</tr>
<tr>
<td>tmm+L_{ii}</td>
<td>1.30</td>
<td>1.25</td>
<td>1.08</td>
<td>1.08</td>
</tr>
<tr>
<td>tmm+L_{jj}</td>
<td>1.36</td>
<td>1.26</td>
<td>1.09</td>
<td>1.10</td>
</tr>
<tr>
<td>tmm+R_{ii}</td>
<td>1.15</td>
<td>1.12</td>
<td>1.05</td>
<td>1.04</td>
</tr>
<tr>
<td>tmm+R_{jj}</td>
<td>1.18</td>
<td>1.14</td>
<td>1.06</td>
<td>1.05</td>
</tr>
<tr>
<td>tmm+HR_{ii} x32</td>
<td>1.21</td>
<td>1.17</td>
<td>1.08</td>
<td>1.07</td>
</tr>
<tr>
<td>tmm+HR_{ii} x64</td>
<td>1.21</td>
<td>1.16</td>
<td>1.07</td>
<td>1.06</td>
</tr>
<tr>
<td>tmm+HR_{jj} x32</td>
<td>1.26</td>
<td>1.17</td>
<td>1.09</td>
<td>1.07</td>
</tr>
<tr>
<td>tmm+HR_{jj} x64</td>
<td>1.26</td>
<td>1.16</td>
<td>1.08</td>
<td>1.06</td>
</tr>
</tbody>
</table>

### 3.4.3 Sensitivity to Thread Count

Table 3.5 and Table 3.6 show the execution time and number of writes overhead, respectively, for running the different schemes with different number of threads (thread count). These
Table 3.6 Normalized number of NVMM writes as the thread count varies from 1 to 12 threads, for various schemes.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>1T</th>
<th>4T</th>
<th>8T</th>
<th>12T</th>
</tr>
</thead>
<tbody>
<tr>
<td>base (tmm)</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>tmm+CP</td>
<td>4.10</td>
<td>4.09</td>
<td>4.03</td>
<td>3.65</td>
</tr>
<tr>
<td>tmm+L_ii</td>
<td>2.12</td>
<td>2.05</td>
<td>2.04</td>
<td>1.71</td>
</tr>
<tr>
<td>tmm+L_jj</td>
<td>2.04</td>
<td>2.12</td>
<td>2.23</td>
<td>1.43</td>
</tr>
<tr>
<td>tmm+R_ii</td>
<td>1.03</td>
<td>1.03</td>
<td>1.07</td>
<td>1.02</td>
</tr>
<tr>
<td>tmm+R_jj</td>
<td>1.03</td>
<td>1.10</td>
<td>1.15</td>
<td>1.09</td>
</tr>
<tr>
<td>tmm+HR_ii_x32</td>
<td>1.09</td>
<td>1.09</td>
<td>1.13</td>
<td>1.06</td>
</tr>
<tr>
<td>tmm+HR_ii_x64</td>
<td>1.04</td>
<td>1.05</td>
<td>1.08</td>
<td>1.04</td>
</tr>
<tr>
<td>tmm+HR_jj_x32</td>
<td>1.09</td>
<td>1.17</td>
<td>1.21</td>
<td>1.19</td>
</tr>
<tr>
<td>tmm+HR_jj_x64</td>
<td>1.05</td>
<td>1.12</td>
<td>1.16</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Overheads are normalized to the base tiled matrix multiplication (tmm) running with the respective thread count. We evaluated each scheme's sensitivity to thread count for 1, 4, 8, and 12 threads, while adjusting the number of cores to scale with the thread count. All runs in Table 3.5 and Table 3.6 are for the same amount of work (i.e. running the same number of iterations of the tiled matrix multiplication).

Table 3.5 shows that Checkpointing (tmm+CP) incurs higher execution time as thread count increases. This is mainly because it requires all threads to synchronize before it can take the checkpoint. While the base execution time decreases as thread count increases due to increasing parallelism, the checkpoint creation time remains the same, and the synchronization time increases slightly. Thus, relative to the base execution time, the overhead from checkpoint creation increases.

The execution time overheads for Logging (tmm+L), Recompute (tmm+R), and Hybrid Recompute (tmm+HR) slightly decrease as the number of threads increases. This is true for both $ii$ & $jj$ granularities. Comparing across schemes, Recompute holds on to its execution time overhead advantage compared to other schemes. For 12 threads, its execution time overhead is only 4%, vs. 8% for Logging and 6% for the Hybrid Recompute. All other previous
observations remain, i.e. the $ii$ granularity produces less overheads vs. the $jj$ granularity, and
64× Hybrid Recompute incurs less overheads compared to its $32×$ counterpart. Regarding
the write overheads, Table 3.6 does not show any strong trend affected by the thread count.

Table 3.7 Various performance statistics: instruction count (IC), store count (SC), L2
dirty eviction ratio (ER), L2 miss ratio (MR), number of L2 evictions (EC), and number
of clflushopt’s (CFOC). All numbers are normalized to tmm (except for number of
clflushopt’s, which is normalized to tmm$+$R.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>IC</th>
<th>SC</th>
<th>ER</th>
<th>MR</th>
<th>EC</th>
<th>CFOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>base (tmm)</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>N/A</td>
</tr>
<tr>
<td>tmm+CP</td>
<td>8.7</td>
<td>11.2</td>
<td>1.19</td>
<td>3.18</td>
<td>3.4</td>
<td>2.97</td>
</tr>
<tr>
<td>tmm+L_ii</td>
<td>1.11</td>
<td>1.02</td>
<td>0.56</td>
<td>2.02</td>
<td>1.81</td>
<td>2.01</td>
</tr>
<tr>
<td>tmm+R_ii</td>
<td>1.13</td>
<td>1.08</td>
<td>0.02</td>
<td>1.41</td>
<td>1.07</td>
<td>1.00</td>
</tr>
</tbody>
</table>

3.4.4 Other comparisons

To get further insights into the comparison, Table 3.7 compares the instruction count (IC),
store count (SC), L2 dirty eviction ratio (ER), L2 miss ratio (MR), number of L2 evictions
(EC), and number of clflushopt’s (CFOC), for select variations of the 4 schemes. In terms
of instruction count and store count (number of committed store instructions), tmm+CP
increases them the most, due to the checkpoint creation. Both tmm+L_ii and tmm+R_ii
increase them by a modest percentage. The L2 dirty eviction ratio (ER) is measured as
the number of evictions of dirty blocks divided by total number of L2 evictions. Here,
tmm+CP increases the ratio because, as it creates a checkpoint, useful data that is dirty in
the cache gets evicted prematurely to make room for the checkpoint. In contrast, tmm+L_ii
and tmm+R_ii perform L2 cache line flushes after the computation for particular matrix
elements. Hence, they are flushed to the NVMM before they get evicted by the natural cache
replacement policy, resulting in the ratio dipping below 1. However, tmm+L_ii creates
more dirty blocks overall due to the log creation, thus it suffers from a lot more dirty block evictions than *Recompute*. In terms of total L2 eviction counts, both tmm+CP and tmm+L_\text{-}ii expand the memory footprint due to checkpoints or logs, hence their total eviction counts increase by $3.4 \times$ and $1.81 \times$, respectively. In contrast, tmm+R_\text{ii} does not expand the footprint much, hence it only increases the eviction count by $1.07 \times$. Finally, the number of cache line flushes is shown in the CFOC column, normalized to the tmm+R_\text{ii} case. The number of clflushopts is significantly higher tmm+CP and tmm+L_\text{ii}, again corroborating the increase in footprint size observation.

### 3.4.5 Other Benchmarks

We demonstrate similar results on the other workloads we studied. Figure 3.6 shows the normalized execution time and number of writes for four other benchmarks and their geometric means, with each benchmark running with eight threads. The top part of figure 3.6 shows *Recompute* incurring execution time overheads between 1-7% (averaging 3%), compared to a range of 31-507% (averaging 91%) for *Checkpointing*. The bottom part of the figure shows *Recompute* incurring NVMM write overheads between 2-30% (averaging 8%), compared to a range of 8-71% (averaging 38%) for *Checkpointing*. The benefit from *Recompute* differs from one benchmark to another, depending on the nature of the benchmark. For *Gauss*, the saving in execution time from *Recompute* is significantly higher than the saving in the number of writes. The reason for this is that *Gauss* has decreasing computation effort per iteration as the program runs. This makes the execution time overhead of *Checkpointing* higher compared to *Recompute*.

### 3.4.6 Programming Complexity

While incurring less execution time and write overheads, *Recompute* requires a more complex code transformation and recovery compared to *Checkpointing*. Matrix multiplication and other applications are commonly used in HPC (and beyond) through a library. Thus, a
Figure 3.6 Execution time (top) and number of writes (bottom) overheads for different benchmarks. For each benchmark we compare Checkpointing and our Recompute schemes normalized to base (base is the model without persistence or checkpointing). The figure also shows the geometric mean (gmean) for each scheme.
library can implement the Recompute scheme efficiently, allowing end users to benefit from its performance without programming it on their own. Another approach for mitigating the implementation difficulty could be through the addition of language or compiler support to help automate the process. While we have not designed a compiler algorithm to automate it, we will sketch what such automation may require.

In the five loop-based benchmarks we evaluated, we observed the same transformation steps for achieving failure safety using Recompute. First, the programmer needs to provide the compiler with three hints: 1) the outermost loop of the persistence region (i.e. the $kk$ loop in tmm); 2) the variables to be persisted (i.e. matrix $c$ in tmm); and 3) the granularity of persistence (i.e. $ii$ or $jj$ in tmm). By specifying the outermost loop, the compiler infers which loop induction variables need to be logged. By specifying the name of all variables that must be persisted, the compiler can identify which stores must be made durable. By specifying the granularity of persistence, the compiler knows where to insert the cache line flushing instructions to make data durable. It can also validate that all updates to the specified persistent variables are nested within the specified loop and that it can correctly generate a loop that flushes all modified elements to make them durable. These steps must be performed in a compiler pass after other loop transformations (e.g. tiling and unrolling) are considered. An example of how the programmer can provide such hints is through source code-level directives a la OpenMP directives.

### 3.5 Related Work

The traditional approach to achieve fault tolerance is the Checkpoint/Restart (C/R) approach. In C/R, the computational state of the machine that constitutes a full checkpoint, such as the PC, register file and the address space, are saved periodically to stable storage. When a machine crashes and subsequently restarts, it restores the checkpoint by copying the saved structures back into memory before it can resume execution. Since the failure
usually occurs between checkpoints, some work must be re-run to recover the state of
the system back to the point of the failure. A key disadvantage of the C/R approach is its
significant performance overhead [67]. C/R overhead can be reduced in many ways. For
example, checkpoints can be compressed [31] to reduce the time to write them to storage.
Faster secondary storage options, like NVM [23, 38], can reduce the overhead of copying
checkpoints. Multi-level checkpointing [57] can reduce the frequency and overhead of
copying checkpoints to slower disk-based storage. However, C/R still poses significant
overhead.

Application-Level Checkpointing (ALC) provides an improvement over C/R. It exploits
the observation that most iterative scientific applications have certain key data structures or
variables from which the computational state of the program can be recovered and resumed.
For example, in order to restore an n-body application, we only need to save the positions
and velocities of all the particles; therefore, ALC would checkpoint only these key data
structures in each checkpoint [18]. Programmer instrumentation is needed to determine
good points in the program to save a checkpoint of the key data structures or variables. This
approach has two main advantages over C/R. First, it is not machine or OS specific. Second,
it can significantly reduce the amount of checkpointed data by saving only the necessary
ones. Reducing the data copied to secondary storage significantly improves the execution
time overhead and reduces the memory needed to take the checkpoint compared to C/R
approach [18].

There are even techniques that can provide recovery without the use of C/R. Algorithm-
Based Fault Tolerance (ABFT) can be used to detect and correct silent errors [30] or even
node failures [17, 27] in matrix multiplication. ABFT depends on saving encoded global
variables that represent the state of the program. For example, encoding and saving a
checksum to detect silent errors in matrix multiplication. The main advantages of ABFT
approach over C/R and ALC is a much better scalability and the ability to not only detect
but also correct silent errors. ABFT incurs overheads from encoding the checksum and
applying the necessary transformation to the redundant data. The main problem with ABFT algorithms is that they need significant algorithm and support environment changes [75]. These changes are hindering ABFT algorithms from being deployed. Although ABFT algorithms have been heavily researched for a while, MPI forum has not agreed on any of them to be standardized [70]. In addition, ABFT algorithms are mainly aimed at silent errors. System-wide failures still need to roll back to a previous checkpoint [27]. Therefore, ABFT algorithms can be thought of as orthogonal approach to checkpointing approaches. It does not completely eliminates the need for checkpointing, rather it is aimed at reducing the number of needed checkpoints. One good approach to complement ABFT is Checkpoint-on-Failure (CoF) [17]. It basically checkpoints the surviving nodes only when other nodes fails, which achieves the most optimal checkpointing interval by definition. Similar to ABFT, CoF requires heavy MPI environment changes.

3.6 Conclusion

In this work, we studied several approaches for checkpointing on loop-based codes on NVMM. We observed that Logging applied to a tiled loop increases the number of writes to NVMM significantly, hence reducing write endurance of NVMM.

Our new approach is based on the novel observation that inconsistent state can be tolerated to gain both performance and reduce the number of writes to NVMM for loop-based code. Rather than logging all of the state modified during a transaction which incurs large overheads, we only log enough state to enable recomputation. We also optimize our Recompute-based approach to avoid recomputing from the beginning and we show this incurs little overhead.

We compare our new approach against Logging and Checkpointing on five scientific workloads, including tiled matrix multiplication, by running on the gem5 simulator with
support for Intel PMEM instruction extension. For tiled matrix multiplication, we find that our Recompute approach has an execution time overhead of only 5% compared to a 8% overhead with Logging and 207% overhead with Checkpointing. Furthermore, Recompute incurs only 7% additional NVMM writes, compared to 111% and 330% more with Logging and Checkpointing, respectively. Other workloads show similar trends. Hence, Recompute simultaneously achieves good execution time performance and does not adversely affect NVMM write endurance.
4.1 Introduction

The core computations of many scientific applications are dense linear algebra operations. As a result, the industry and academic community have studied linear algebra routines for decades to reach highly optimized implementations of linear algebra routines. One of the most popular and widely used linear algebra implementations is the BLAS (Basic Linear Algebra Subprograms) library. BLAS is a set of hard-coded highly-optimized basic routines, such as vector addition and matrix-matrix multiplication, written using machine-specific assembly code to tailor them for utilizing the hardware features of the machine they run
on. Some of the most popular linear algebra libraries, such as Linear Algebra Package (LAPACK) [8], Automatically Tuned Linear Algebra Software (ATLAS) [76], and Intel’s Math Kernel Library (Intel MKL) [32], rely on BLAS to implement complex algorithms from the basic kernels BLAS offers.

In the domain of dense linear algebra, it is well-established that GEMM (stands for GEneral Matrix Multiplication) is the cornerstone kernel for a wide range of linear algebra operations. Therefore, it follows that GEMM is one of the fundamental kernels of BLAS.

The literature is full of different implementations and optimizations of high performance GEMM routines. All GEMM implementations rely on the concept of blocking, which we discussed in chapter 2-section 2.3.6, to improve temporal locality and reduce the cache miss ratio. In [29], Goto and Van De Geijn discussed a layered approach to implementing GEMM. Their implementation is widely regarded as the most effective implementation of GEMM. Therefore, we chose their implementation of GEMM for our study.

As we discussed in chapter 3, NVMM provides an opportunity to lower the overhead of in-disk checkpointing required for achieving failure safety. Checkpointing is essential for scientific applications because they run on HPC systems for extended periods of time. Although checkpointing in-NVMM of matrix multiplication has been addressed before [6, 26], but it did not analyze the different implementations of GEMM and the trade-offs between them. These tradeoffs have been extensively studied and optimized for decades. Nevertheless, their performance and write-endurance impact on NVMM while running failure-safe applications were never analyzed in detail. Therefore, re-visiting GEMM implementations in the context of in-NVMM checkpointing is of utmost importance. In chapter, we aim to analyze the persistence, performance tradeoffs that programmers should be aware of. To approach this problem, we perform experiments on different optimizations for the six GEMM implementations from the GotoBLAS library. We make several findings through our study:
• Matrix partitioning or blocking, which has long been the core of all GEMM implementations, can have negative side effect on the write endurance of NVMs when it is used for application-level checkpointing.

• There is a significant trade-off between the different persistence granularities.

Figure 4.1 The different implementations of GEMM. The labels have the form gexy where the letters chosen for x and y indicate the shapes of matrices A and B, respectively.
4.2 Goto Implementation of GEMM

Goto and Van De Geijn discuss a layered approach for decomposing GEMM into kernels. They analyzed six different implementation styles. Of these styles, there are two that decompose GEMM into multiple calls to the GEPP kernel (short for GEneral Panel times Panel). GEPP is further decomposed into one of two smaller kernels: GEBP (short for GEneral Block times Panel multiply) or GEPB (short for GEneral Panel times Block). We will denote the GEBP variation of GEPP as GEPP1 and the GEPB variation as GEPP2. A block is a square-sized sub-matrix while a panel is a rectangular sub-matrix with its long side entirely...
spanning one of the two matrix sides.

Goto and Van De Geijn show that GEPP1 is superior in performance to all other five implementations. For that reason, we chose to show the detailed anatomy of the GEPP1 implementation of GEMM in Figure 4.2 for reference. Figure 4.1 shows the rest of the GEMM implementations. They are basically variations of Panel×Panel, Matrix×Panel and Panel×Matrix kernels. For simplicity, we will use square matrices for our analysis and evaluation. In Figure 4.2, the two input matrices are A and B, the result matrix is C, the matrices dimension is N, and the block size is S. The naming convention for these styles is GEXY where X and Y are the shapes of the input matrices. For example, in figure 4.1, the GEMM1 implementation is basically two variations of Panel x Panel matrix multiplication, which are GEPP1 and GEPP2. The reason is that each outermost loop iteration multiplies a vertical panel from matrix A by a horizontal panel of matrix B. In GEMM2, we see that each iteration multiplies a panel from matrix A with the whole matrix B, and hence the split into two variations of GEneral Panel-Matrix multiplication. For detailed implementation of the five other styles, we refer the reader to the original GotoBLAS paper in [29].

4.3 GEMM Impact on NVMM Write Endurance

From a quick look at the GEPP-based variations in Figure 4.1, we can see that every GEPP iteration touches the whole C matrix. Therefore, the GEPP-based implementations update the C matrix \( \frac{K}{S} \) times to compute the final result. Compared to the naive matrix multiplication (i.e. regular matrix multiply without blocking), which updates the result matrix only once, the number of extra writes is significant.

In each GEPM1 iteration of GEMM2, a panel of the result matrix C is repeatedly updated. If the number of columns in B (i.e. the N dimension) is large enough, it is possible that the cache hierarchy would not fit a block of A, panel of B and another panel of C. In that case, the updated C elements would be evicted from the cache hierarchy, which would result
in a write miss the next time we are updating the $C$ panel. In that case, we would update the panel of $C$ $\frac{K}{X}$ times, which would result in write overhead similar to the GEPP-based implementations. According to Goto, streaming $C$ panel from and to the memory is not expensive if the size of the $C$ panel is much lower than the block size from $A$ matrix. However, streaming updates to the $C$ matrix increases the number of writes to the NVMM, which could hurt the write endurance. To avoid this repeated writes of the $C$ panel in memory, the $B$ and $C$ panels would have to be partitioned into smaller panels such that the cache can fit a panel of $B$, a panel of $C$, and a block of $A$. This limitation potentially inhibits the performance gain we can get from the reuse of the $A$ block. The same applies to GEMP1 except that the matrix $A$ is what would need to be sub-partitioned.

Although GEMP2 and GEPM2 are the lowest performing among the six variations, according to the GotoBLAS paper, they achieve the least number of writes. In these two implementations, each panel of $C$ is touched only once and never updated again. Therefore, the number of writes for these two variations is the least possible.

From the analysis above, we can conclude that although, GEPP1 is superior in performance to the other GEMM implementations, it incurs significant number of extra writes to the NVMM. Therefore, if number of writes is a concern, one of the lowest performance implementations (GEMP2 or GEPM2) should be considered. Otherwise, the GEPM1 and GEMP1 implementations need careful structuring so that they would not incur the same number of writes as GEPP-based implementations.

## 4.4 GEMM with Recompute Checkpointing

The six GEMM implementations of GotoBLAS are similar in that they follow the same layered approach to decomposing the matrices. However they differ in the granularity and memory layout of the element being computed in the inner kernels. Therefore, we will only cover the cost of checkpointing in NVMM for all different variations of GotoBLAS. We choose
the Recompute checkpointing mechanism for checkpointing since it promises the least amount of writes and the lowest performance overhead compared to other checkpointing techniques.

### 4.4.1 Impact of Checkpointing Granularity

The Recompute checkpointing technique requires the application to stop somewhere in the code to save the recent updates to the NVM as well as the matrix indices. The question of where in the code to stop and persist the outstanding updates is critical to both performance and number of extra writes. For the GEPP1 implementation, two reasonable places to stop and take the checkpoint are either at the end of each GEBP iteration or after computing a whole j iteration. We will call the checkpointing at the end of every GEBP and every j iteration, the relaxed-granularity and aggressive-granularity, respectively. We cannot take the checkpoint at the end of a GEPP iteration because if a crash happened, we would have to recompute all data touched. Since every GEPP iteration updates the whole C matrix, we would have to start the GEMM computation from the beginning.

The ii-granularity flushes the whole result panel from GEBP to the NVMM. By the time the ii iteration ends, there is a chance some of the blocks from the updated C panel are already evicted from the LLC. Therefore, the flush instructions latency is likely to be less than or equal to the latency of flushing dirty blocks from the cache hierarchy. Based on the same analogy, the j-granularity would always flush a dirty cache block. Though, that is not a problem since the cache block is not needed until the next kk iteration. In fact, eagerly flushing the dirty cache block can give slight performance improvement since it empties the slot in the cache for useful data. There are two problems with the aggressive-granularity: 1) it negatively impacts any locality for updating the C matrix. 2) it needs more sfence instructions compared to the relaxed granularity. The sfence is a very expensive instruction because it stalls all stores that come after it until all stores and flushes before it are completed (i.e. the Load Store Queue receives an Ack from the L1-Cache). Eventually,
the cache block gets evicted as a result of the normal cache conflict. The cache conflict eviction happens in the background and does not block subsequent stores or flushes from accessing the cache. Therefore, its overhead is amortized. The sfence instruction does block subsequent stores and flushes from accessing the cache. Therefore, the sfence instruction is quite expensive especially when the flushes are for dirty blocks, which is the case with the j-granularity.

The drawback of the ii-granularity is that it comes at the cost of longer recovery time. If a crash happened in ii-granularity Recompute checkpointing, we would have to recompute a whole panel of the C matrix from the beginning of the computations. For very large matrices, this can be a very expensive recovery operation.

Our conclusion is that the larger checkpointing granularity (ii-granularity in case of GEPP1) is better because it incurs fewer sfence instructions. We choose to put the burden on the recovery since it is the rare event compared to normal computations. One way to reduce the recovery cost is to employ a hybrid of less-frequent traditional checkpointing with the Recompute technique as Elnawawy et. al proposed [26].

4.5 Methodology

4.5.1 Simulation configuration

We evaluated GEMM on Gem5 [59], an open source cycle-accurate full system simulator. Our simulator uses x86-64 instruction set architecture (ISA). It models a detailed 4-way out-of-order processor pipeline for each core, with parameters shown in Table 4.1. The table also shows parameters for the 2-level caches that we model, with private per-core L1 caches and an L2 cache shared by all cores. The memory hierarchy model is built on top of Ruby [66], with MESI protocol keeping the L1 caches coherent with respect to one another and with respect to the shared L2 cache. We modeled a Phase Change Memory (PCM) type of NVMM based on the DDR timing parameters shown in the table. This configuration
Table 4.1 The baseline system configuration.

<table>
<thead>
<tr>
<th>Processor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8-core, 2GHz, out-of-order x86-64, 8-wide issue/retire ROB: 192 Fetch/Issue/Store Queues: 32/32/32</td>
</tr>
<tr>
<td>Coherence Protocol</td>
<td>Ruby System, MESI Two Level Protocol</td>
</tr>
<tr>
<td>L1I &amp; L1D Cache</td>
<td>private, 2-cycle access, 64KB, 16-way, 64B block</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared, 20-cycle access, 1MB tile/core, 16-way, 64B block</td>
</tr>
<tr>
<td>MSHRs</td>
<td>32</td>
</tr>
</tbody>
</table>

**DDR-based PCM Main Memory**

<table>
<thead>
<tr>
<th>Capacity</th>
<th>8GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization</td>
<td>1 channel, 2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address mapping, FRFCFS scheduling policy</td>
</tr>
<tr>
<td>PCM latencies</td>
<td>50ns read, 150ns write</td>
</tr>
<tr>
<td>DDR Timing</td>
<td>tRCD 55ns, tWR 150ns, tCL 12.5ns, 64-bit bus width, 800 MHz Clock</td>
</tr>
<tr>
<td>Read &amp; write queues</td>
<td>32-entry RQ &amp; 64-entry WQ, 85% WQ high threshold, 50% low threshold</td>
</tr>
</tbody>
</table>

for PCM is in based on the widely used parameters from Lee et al. [46] and previous work related to NVM [5, 10, 13, 14, 36, 46, 52]. These parameters would yield read and write latency of 60ns and 150ns, respectively.

The DRAM controller we model has a 1KB row buffer size, one channel, two ranks per channel, and eight banks per rank. It maps the addresses to the DRAM based on "RoRaBaChCo" policy where Ch, Ra, Ba, Ro and Co denoting channel, rank, bank, row and column, respectively, and going from MSB to LSB. This mapping ensures interleaving among the different banks to reduce the number of row buffer misses. We note that a row write happens when we are closing a row that has modified data. Therefore, any rows that were only used for reading data for A and B input matrices are not dirty and therefore do not contribute to the total number of row writes.
4.5.2 Workloads

We evaluate the six different variations of the Goto algorithm from figure 4.1. For each variation, we show the speedup and the number of row writes increase relative to the 3-loop naive matrix multiplication. We evaluate regular multiplication with and without checkpointing. For the checkpointing versions, we evaluate two different granularities of Recompute checkpointing: one that checkpoints more frequently (eager) and another one that performs the checkpointing on the outer loop of the multiplication to flush cache lines less frequently, which we call lazy. For each of these two styles of checkpointing, we evaluate two versions with and without packing. For all experiments, we run the matrix multiplication with four worker threads plus one master thread. They run on five cores.

4.6 Evaluation

Figure 4.3 show the execution time and number of row writes overhead for the five different workloads of the Goto variations. All results are relative to the base naive model without any persistence or flushing.

4.6.1 Row Writes Without Checkpointing

As expected, GEPP1 gives the best execution time. Nevertheless, it incurs significant number of row write relative to naive. The big row writes disparity is mainly attributed to naive matrix multiplication not suffering from high number of row writes. The explanation of why naive has number of row writes that is much lower than some of the Goto variations rests within the memory controller; The memory scheduler prioritizes servicing requests from the read queue over write requests because reads are on the critical path of the program execution. Usually, it switches from reads to writes in one of two cases: (1) when the number of requests in the write queue reaches a certain threshold (0.85 of the write queue size,
in our evaluation), or (2) when the read queue is empty. When the scheduler switches to servicing writes, it performs a minimum number of writes before switching back to reads. This number is sixteen, in our evaluation. And since naive matrix multiplication incurs significant number of reads for each write to a cell from the result matrix, most of the time...
the memory controller is busy serving reads. And since the number of writes is much lower than the number of reads, it takes the memory controller long time for it to begin servicing the write requests from the write queue. The longer the writes wait in the write queue, the more likely it is for requests to merge with each other.

Although GEPP1 incurs seventeen times more row writes than naive, it is not the variation with the highest impact. GEMP1 incurs 74 times the row writes as naive. The answer to why GEMP1 and GEPP2 have relatively much higher extra row writes compared to the other variations can be deducted from the layout in figure 4.1. GEMP1 and GEPP2 write to a whole vertical panel of the result matrix in each iteration. Therefore, the writes span more number of rows considering a row-major layout in memory. On other hand, GEPP1 and GEPM1 write to a whole horizontal panel of the result matrix. Since the memory layout is row-major, these subsequent writes in the memory experience higher number of row buffer hits. Therefore, the total number of row buffer writes for them is not as high as GEPP2 and GEMP1.

We notice that GEPM2 and GEMP2 experience the lowest number of row writes. In fact, they reduce the number of row writes compared to the naive matrix multiply. The reason is that the inner kernel of these algorithms, which is GEPDOT, writes to a block from the result matrix and never write to it again. Naive also writes to the result matrix once. Though, the better locality of GEPM2 and GEMP2 coalesces some of the writes together.

4.6.2 Row Writes With Checkpointing

In three of the four runs that use Recompute checkpointing, the row writes increase compared to the base run with no checkpointing at all. The only run that does not show the same behavior is the eager packed run. The reason is that packing improves the locality and increases the number of row buffer hits. When combined with the aggressive flushing of data, the chances of coalescing become much more. On the other hand, when data is packed but for the more relaxed granularity, the benefit of packing is not fully utilized.
4.6.3 Impact of Packing

From the subsection above, we can see that packing improves the number of row writes for all variations compared to their non-packed counterparts. This is expected because packing rearranges the data sequentially in the memory. Therefore, the result is an almost optimal number of row writes compared to naive.

4.7 Related Work

There is a rich body of research that addresses the problem of the lower and upper bounds of I/O data movements. Of these studies, Smith et al.[72] analyze the communication between fast and slow memory where they are either storage and main memory or main memory and caches. They apply their analysis on Goto algorithm, GEPP1. They theoretically reach a lower bound on the memory traffic incurred under Matrix Matrix Multiply and Addition (MMMA), which is basically a variation of GEMM. They show that for a fast memory with capacity of $M$, the lower bound of reads is $\frac{2mnk}{M} - 2M$ elements from slow memory and additionally write at least $mn - M$ elements to slow memory where $m$, $n$, and $k$ are the sizes of the three matrices (Matrix $A$ is $m \times k$, matrix $B$ is $k \times n$, and the result matrix $C$ is $m \times n$). Our work is different in that we analyze all different variations of Goto not just GEPP1. We also characterize the number of row buffer writes in NVMM, which was never a problem with DRAM since writes from the memory controller to the DIMM happened in the background. Furthermore, we studied the effect of adding Recompute checkpointing to matrix multiplication because of the importance and applicability of in-NVMM checkpointing for GEMM kernels.

4.8 Conclusion

In this chapter, we studied GotoBLAS matrix multiply algorithm in its six variations when used with Non-Volatile Main Memory. We have shown that packing data can have significant
positive impact on not only performance but also reducing the number of row buffer writes. We further study the number of row buffer writes when we apply Recompute checkpointing technique to the GotoBLAS algorithms. We have shown that the granularity of checkpointing can change the number of row buffer writes significantly.
5.1 Introduction

From the previous chapters, we could see that whenever applications modify on important data structures, they need to make sure their data modifications are made durable atomically. Chapter 2, we showed example of applying undo logging to matrix multiplication to ensure crash consistency.

In this chapter, we propose PreFlush, a hardware mechanism to reduce the overhead of cache line flush operations by predicting and issuing them early. Our proposal is based on the observation that it is not flush instructions that add performance overhead, it is the fence that exposes their latency to the critical path of the program execution. Therefore, if we could predict when the flush instruction is coming and eagerly issue it without altering
the behavior of the program, we can make the fence instruction retire faster. To the best of our knowledge, this is the first predictor-based technique targeting flush instructions. Our work is transparent to the programmer. It optimizes the cache line flush instruction in the hardware fabric.

Our main contributions in this work are the following:

- We provide a discussion and analysis of the programmer’s perspective when using flush instructions with fences and/or thread synchronization.
- We propose a hardware mechanism that is transparent to the programmer to predict flush instructions and issue them early.
- We evaluate PreFlush on a gem5, a cycle-accurate simulator, and show that for only 3.8KB overhead, we can improve performance by up to 49.8% (22.2% average) over logging-based system.

The rest of the chapter is organized as follows. Section 5.2 gives a brief motivation for why eagerly flushing cache lines could improve performance. In Section 5.3, we discusses the programming model we assume in this work. We then describe our proposed design of the PreFlush technique in detail and we cover the operation algorithm with an example. Later, in Section 5.4, we describe our evaluation methodology and benchmarks. Section 5.5 presents our evaluation experiments and analyses. Section 5.6 summarizes some of the past works related to our proposal. Finally, we conclude in section 5.7.

5.2 Motivation

Figure 5.1, part a, shows the coherence steps to complete flush instructions. As we can see, the flush operation takes long time to complete. Therefore, if we could start early as seen in part b of the figure, we can immediately send the flush Acknowledgment (ACK) to the Load-Store Unit (LSU).
To reduce verbosity in the rest of the chapter, we will use the term flush request to refer to both clflushopt and clwb requests. We note that these are not the only x86 instructions that flush cache lines; we consider them and not the other instructions because they were introduced with programming for NVMM in mind.

5.3 PreFlush Design

In this section, we describe our PreFlush architecture and implementation. We start by defining what it means to speculatively execute a flush, how it can miss-speculate, and how to recover. Next, we describe how we can predict when a flush is needed. Then, we discuss the implementation of our baseline design and optimizations to reduce overheads and support a wider variety of flush patterns in the programs we studied.
5.3.1 Speculating on a flush

The key goal of this work is identifying stores that are likely followed by a flush operation and initiating that flush as early as possible. Then, when the flush executes, it will already have been completed and can immediately retire. Since these flushes are often followed by fences which must wait for the flush to finish, retiring the flush quickly reduces the time it takes for the fence to complete.

For the moment, let us assume that we can identify when a store is the last write to a cache line and we can know that a flush instruction will be executed for the same line, but we have not yet executed it. At this moment, we will let the store complete and then immediately speculate that the line will be flushed. We can consider this operation as equivalent to speculatively executing the flush instruction immediately after the store. We keep a record of this action.

There is little harm in performing the flush early. A flush operation is very similar to a cache write back, and write backs of dirty data can be triggered by the cache at any time due to the replacement policy or coherence actions. In the same vein, a processor can trigger a flush of dirty data in the cache without violating correctness of the program, since dirty data is from stores that must have already completed.

When the actual flush instruction is executed, we can detect that we previously flushed the same line as requested by the flush instruction. At this moment we only need to verify that the requested flush by the instruction is equivalent to the flush speculatively performed. If they are to the same cache line and if there have been no stores to that line since the flush, then the operation is equivalent. Furthermore, the instruction can simply be marked complete and retired since the action was already performed. In the case of single threaded programs, detecting that no intervening stores occurred on the same cache line is a simple matter of monitoring all retiring stores between the speculative flush operation and the flush instruction. If none occur, the operation speculated successfully.
However, on a miss-speculation, we have no choice but to re-perform the flush after enforcing all required dependences. Fortunately, no additional action beyond the flush is needed. In this scenario, the flush instruction simply flows through the pipeline and executes normally without the benefits of speculation.

Figure 5.2 Flush instructions with fences and thread synchronization

5.3.1.1 Single thread example

In the snippet of code in Figure 5.2 part a, instructions \( i_1 \) and \( i_2 \) write to cache lines A and B, respectively. After that, there is a store barrier (sfence) in \( i_3 \) after which we flush cache line A (\( i_4 \)). According to the semantics of the sfence instruction as specified by the X86 ISA, the flush in \( i_4 \) cannot start execution before the stores in \( i_1 \) and \( i_2 \) are complete (i.e. done with their execution and cache access). However, since instruction \( i_1 \) is the last store access to cache line A before the flush in \( i_4 \), our PreFlush predictor could start the flush operation as soon as it sees the store in \( i_1 \) and before the fence in \( i_3 \) is complete. While the fence does delay \( i_4 \), nothing prevents the cache from evicting the line before the fence retires. Hence, this operation is acceptable. Once \( i_4 \) does execute, it will be verified that the speculative operation succeeded, and it is immediately retired.
5.3.1.2 Flush Instructions with Synchronization

In shared memory systems, multiple cores could be working together on the same data. Whenever there is a need to maintain specific order of execution between cores, some type of synchronization is needed. Since our PreFlush technique could change the order of execution by eagerly starting the flush operation, we need to evaluate whether or not this reordering alters the correct behavior of the synchronization points in the program.

Now we consider a more complex example with synchronization between threads. The piece of code in figure 5.2 part b, shows two threads writing to cache line A in a specific order. After these two writes are complete and globally visible, thread 1 flushes A on behalf of both threads. The two threads use flags (t1Done and t2Done) as handshaking synchronization points to achieve the specific order. Let's assume that both flags are initially reset (i.e. false).

At the beginning, thread 2 waits at instruction i7 for thread 1 to set t1Done. Meanwhile, thread 1 writes to the data in cache line A (i1), then it waits at an sfence (i2) until the update to cache line A is complete. When i1 is complete, thread 1 sets t1Done (i3) then it moves to i4 waiting for thread 2 to set t2Done. Since thread 1 already set t1Done, it allowed thread 2 to proceed to i8 to update the data in cache line A. The sfence in i9 guarantees that thread 2 does not set t2Done before the update to cache line A (i8) is complete. Once the update to A is complete, thread 2 passes the sfence and proceeds to i10 to set t2Done. At that point, thread 1 moves from i4 to i5 where it waits at an mfence (memory fence) until all memory accesses preceding it are complete. Only then, thread 1 is ready to flush cache line A (i6).

The semantics of the two X86 flush instructions (i.e. clflushopt and clwb) ensure ordering between the i1 store and the i6 flush, but they do not enforce any ordering between the i8 store and the i6 flush. Nevertheless, the synchronization in this code guarantees that the i1 store completes before the i8 store, which in turn is guaranteed to complete before the flush in i6. Therefore, the flush in i6 is expected to include both updates to cache line A from i1 and i8 (i.e. flush cache line A with value ten).

Our PreFlush technique could potentially alter this behavior. Thread 1 could PreFlush
cache line A as soon as it executes the store in \( i_1 \). Therefore, it would eagerly start the flush before thread 2 gets to update A in \( i_8 \). In that scenario, the speculation is incorrect. We must detect the intervening store on Thread 2 and ensure the flush at \( i_6 \) includes the data from \( i_8 \).

For the case of a single thread, we can easily detect intervening stores using in core structures that monitor retiring stores against a set of speculated flushes. For multi-threaded codes, we can leverage the cache coherence protocol and additional state at the directory to detect intervening stores that happen in parallel.

To deal with this problem, we leverage the coherence protocol. The directory keeps track of cache lines that are PreFlushed to inform the L1 that initiated the PreFlush when the PreFlushed data is accessed between the store and the flush instruction. In section 5.3.7, we explain how we guarantee this functionality.

### 5.3.2 Predicting the Last Store

The other challenge of our design is predicting when a flush is needed so that we can start its execution well in advance of a fence. The idea of our predictor hinges on predicting that a store is the Last Store to a cache line before a flush. Whenever we see this Last Store, we can predict that a flush will soon follow.

Flushes are relatively rare in the workloads we study, so we work backwards from them. When a flush is observed, we need to identify the previous store to the same line and predict it as the Last Store. A naive approach to accomplish this is tagging all cache lines with the PC of the Last Store that modifies them. When the flush occurs, we can look up the Last Store for the line that is flushed. Since flushes are uncommon and since the code paths leading up to flushes are relatively linear, relatively few Last Stores will be detected. These PCs can be enumerated in a small table. Once identified, whenever one of these store instructions executes we predict that a flush is needed.

We will refine this approach more in the next section, and show that adding a simple
confidence counter to this approach can bring accurate predictions. We also show that some loop-based codes like matrix multiply have more complex store-flush patterns, and we extend this basic approach to cover them as well.

Our approach builds on prediction techniques that rely on traces of the Program Counter (PC). The intuition behind using the instruction PC rather than the data address is that addresses of instructions do not change during the program execution while addresses of data being accessed do change with the program execution. Therefore, if a trace of instructions led to an event of interest (\textit{clflushopt} or \textit{clwb}, in our case), matching instruction traces are likely to lead to the same event.

### 5.3.3 Baseline Design

#### 5.3.3.1 Added Structures

We augment each cache line with a \textit{trace}, which is a subset of the instruction PC bits. We update a trace when its corresponding cache line receives a store request. We ignore loads because they do not modify the durable version of a line we flushed to the fail-safe domain.

To update a trace, we XOR the current trace with the store's instruction PC, then we overwrite the current trace with the truncated output. We continue to update traces during a cache line's life time which starts when the L1 allocates a way block for a cache line following a miss until the time the line receives a flush request. However, L1 allocation is not sufficient to know when to start a new trace. The reason is that \textit{clwb} requests do not invalidate cache lines. Nevertheless, any store that follows a \textit{clwb} request should start a new trace. Thus, we add a "Valid Trace" bit (VT bit) to each cache line to explicitly indicate that a trace is valid or not. Accordingly, when a store request accesses a cache line and finds the VT bit is set to zero, it starts a new trace by overwriting the old one with a subset of the request's instruction PC. It also sets the VT bit to one to denote that this trace is now live. In addition to traces, we extend the L1 cache with two new structures: the \textbf{Signatures} table and the \textbf{FlushedTags}
The Signatures table holds signatures, which are traces that ended with a flush request. We use signatures to predict if a live trace will receive a flush request next or not. Traces that end with L1 replacement or Lower-level invalidation are not valid signatures because they don’t lead to flush request. However, we still have to invalidate the trace when its cache line is evicted, regardless of the eviction cause. We use a 2-bit confidence counter per signature to improve the prediction accuracy. The four levels of confidence starting from zero are low-confidence, weak-low-confidence, weak-high-confidence, and high-confidence states, respectively. Any new signature starts with counter value of weak-high-confidence. We also add a bit to each signature to denote the type of the flush (i.e. \texttt{clflushopt} or \texttt{clwb}). Although from persistency point of view it does not matter if the cache line is invalidated or not as long as it is made durable, we chose to PreFlush cache lines using the same flush type that was used when the signature was saved. In other words, if a signature ended with a \texttt{clwb}, we issue \texttt{clwb} when a trace matches it.

The FlushedTags table holds the tags of cache lines that we PreFlush (i.e. flush early as a
result of one of the signatures predicting this cache line will soon receive a flush). It also saves the signature that predicted this PreFlush operation. This structure is needed to train the predictor and to recover from mis-speculation as we will see later.

5.3.3.2 Handling Store Requests

Figure 5.3 shows the execution path for store instructions. Every store request to a cache line updates its trace using a truncated XOR operation, as we discussed above. This operation can be done as a read-modify-write. After updating a trace, we compare it with all entries in the Signatures table to see if there is a match with high or weak-high confidence counter. If we find a match, it means we predict that the line will be flushed soon; therefore, we PreFlush the cache line by injecting the PreFlush request into the L1 cache request queue. At the same time, we save the line's tag as well as the signature that predicted the PreFlush in a new entry in the FlushedTags table. We do not invalidate the PreFlushed line's trace yet. We wait until we receive the actual flush request. The reason is that if our prediction turned out to be wrong, we would want to continue using the trace until it is complete to capture the correct prediction. We would not want to discard a potentially-useful trace.

Besides writing to a cache line and updating its trace, store requests also check if there is a valid entry with a matching tag in the FlushedTags table. If there is, it means we have a mis-prediction because we were expecting that a flush was coming next. In that case, we invalidate the FlushedTags entry because its PreFlush did not include the version of the cache line the new store is about to update. In addition, we use the signature that we saved in the FlushedTags entry along with the tag to check if that signature is still in the Signatures table. If it is, we decrement its confidence counter.

5.3.3.3 Handling Flush Requests

Figure 5.4 shows the execution path for flush instructions. When a flush request arrives from the Load Store Unit (LSU) to the L1 and finds a valid entry with a matching tag in
the FlushedTags table it knows we already flushed the line. This means our prediction was correct. In that case, the L1 sends an acknowledgment (ACK) to the LSU for the flush without having to send the flush request to the directory. Then, we invalidate the entry in the FlushedTags table to prevent subsequent flushes to the same cache line from erroneously thinking the line is already flushed. We also increment the confidence counter of the signature that predicted the flush.

If a flush request arrived from the LSU to the L1 and there is no matching tag in the FlushedTags table, it means we could not predict this flush request was coming. This no-prediction case could be because of one of three scenarios: 1) we have not encountered this trace of instructions before, 2) this trace is in the Signatures table but it has low confidence counter value, or 3) the trace was in the Signatures table but was evicted due to capacity limit. In the case there is a matching signature in the Signatures table, we increment its confidence counter. If there is no matching signature, we store the cache line's trace in the Signatures table as a new signature regardless of whether it was evicted or never seen before.

Figure 5.4 Path of execution for flush instruction.
5.3.4 Last Store Optimization

Using trace-based predictor ensures a high degree of prediction accuracy. Since `clflushopt` and `clwb` are very specific memory requests that happen much less often than stores, we can base our prediction on just the instruction PC of the most recent store instead of recording the whole trace. This design simplification would reduce the number of accesses to the different structures significantly, which saves power and access time. To that end, we replace the traces we added to each cache line with a new structure that saves the instruction PC of the most recent store accessing the line. We call this structure the LastStore table. Each LastStore entry is composed of 35 bits, 32 of which are the least significant bits of the store instruction's PC. The other three bits are the VT bit, the PreFlushed bit, and the type bit.

When a cache line receives a flush request, we move its LastStore entry to the Signatures table. The size of an entry in Signatures table would change to fit the LastStore entry size. Each FlushedTags entry would also change to hold the 32 bits of the new signature.

5.3.5 Sample of Sets Optimization

Instead of tracking store accesses to all cache lines, we can just track a sample of the sets. We would update the predictor when one of the sets from our sample is accessed. The idea is that cache behavior tends to be uniform across all sets. The signatures we save from our sample of sets can be used to predict the flushes for the rest of the cache lines. The benefit of this optimization is significantly lower power and area overhead. The predictor access time would also decrease. The downside of this optimization is that it takes longer to train the predictor. Nevertheless, it has very little negative impact on the prediction accuracy.

5.3.6 Optimized Design Operation

Figure 5.5 shows the complete flow chart of our algorithm. We will go over the flow of execution for all the different outcomes of PreFlush put together. First, we access the
Figure 5.5 PreFlush algorithm.
FlushedTags table in parallel with the L1 tags array. If the memory request is a store request, and it is PreFlushed (i.e. the store's tag matches a valid entry in the FlushedTags table), it means we have a misprediction. In that case, we reduce the confidence of the signature that invoked this misprediction. To do that, we use the signature we saved in the FlushedTags entry to find its match in the Signature table. If the line is not PreFlushed, we check if this store should trigger a PreFlush or not. We check if the store's instruction PC matches any entry from the Signature table with a high or weak-high confidence. If it does, we issue the PreFlush request for that cache line once the store is complete. Following the check for PreFlush prediction, we check if the store request is accessing one of the sets we sample. If it does and the store hits in the cache, we update the LastStore entry.

If the request is a flush not a store, we check for prediction hit (i.e. the line is already PreFlushed). In that case, we do three things: (a) send a flush ACK message to the core, (b) increment the confidence counter of the signature that predicted the correct PreFlush, and (c) invalidate the FlushedTags entry. On the other hand, if the cache line is not PreFlushed, we check if the flush request hits in the cache. If it does, we check if the cache line's corresponding instruction PC from the LastStore table matches any signature from the Signatures table. If it does, we increment that signature's confidence counter before we invalidate the LastStore entry. If there is no match, we save the instruction PC from the LastStore entry in the Signatures table as a new signature before we invalidate the LastStore entry.

5.3.7 Recovering from Ordering Violation

In the previous section, we discussed our programming model assumption when we speculatively flush a cache line early, violating the ordering of synchronization points. In that scenario, we discard the PreFlush entry and perform the flush again. We leverage the coherence protocol to detect when such violation happens. Essentially, the directory keeps record of which core PreFlushed which line until the record is no longer needed. To achieve
Figure 5.6 Sequence of coherence messages that ends with PreFlush invalidation due to ordering violation.
this, we have to differentiate between a regular flush and PreFlush requests. We add a new
table in the directory to hold the tag of the line being PreFlushed and a bitmap to know
which core PreFlushed the line.

Figure 5.6 shows how the directory detects the PreFlush violation and how it responds
to correct it. Processor 0 (P0) has cache line A in shared state. It PreFlushes line A using a
\texttt{clwb} request (i.e. it does not invalidate its local copy) \textsuperscript{1}. When the directory receives the
PreFlush request, it inserts it into the PreFlushers table \textsuperscript{2}. At the same time, it writes back
the cache line to the memory controller (assuming it is dirty). Once the memory controller
sends the writeback ACK to the directory \textsuperscript{3}, the directory sends an ACK for the PreFlush to
P0 \textsuperscript{4}. Later in \textsuperscript{5}, P1 sends an upgrade request to the directory. The directory checks if the
line is PreFlushed, and since it is, it sends a PreFlush NACK to the PreFlusher (P0) \textsuperscript{6}. When
P0 receives the NACK, it discards the FlushedTags entry and decrements the confidence of
the signature that PreFlushed line A \textsuperscript{7}. It also sends the invalidation ACK to the upgrade
requestor. A race scenario could happen when the directory is sending a NACK to the L1
PreFlush issuer at the same time the PreFlush issuer receives the flush request. In that case,
P0 sees that the there is a tag match in the FlushedTags table (i.e. line is PreFlushed), so it
send a flush ACK to the SQ informing it the flush is complete. When the NACK message
reaches P0, the flush ACK has already been sent the SQ. To avoid this race situation, the L1
PreFlush issuer has to pass the regular non-speculative flush request to the directory when it
receives it. Then, it waits for an ACK for the flush request. When the PreFlush issuer receives
the flush ACK from the directory, it safely removes the PreFlush entry from the FlushedTags
table and passes the ACK to the SQ to mark the flush completion. On the directory side,
when it receives the regular flush request from the L1 and finds it is PreFlushed, it knows it
can safely remove the entry from PreFlushers table.

Assuming that the cache line was PreFlushed with a \texttt{clwb} instruction, it would not be
invalidated from all caches. Therefore, if the cache line is shared, other caches can continue
to read the cache line. Since we do not track load requests, this should not be a problem. If
one of the sharers wants to write to the line, it would have to inform the directory by sending an upgrade request. We leverage this upgrade request to inform the L1 that PreFlushed the line that it has been modified.

In the scenario that the clwb requester does not have the cache line in its private cache, it still sends the request to the directory to forward it to the owner (assuming the cache line is in modified state). Since this is a clwb request, the exclusive L1 cache does not need to invalidate its copy. Nevertheless, we force the exclusive L1 to downgrade to shared state to prevent it from modifying the cache line silently. Downgrading to shared state forces all sharers to explicitly request an upgrade from the directory to modify the line. In doing this, the directory is always aware when a PreFlushed cache line is being written to. This downgrade requirement means that any PreFlushed cache line cannot be in exclusive state, which can potentially harm the performance of the cache. On the other hand, allowing the exclusive L1 to continue being exclusive would mean that it could potentially modify the line silently without the directory or the PreFlush issuer knowing. This behavior defies the programming assumption we made in the previous section.

5.3.8 Handling Flushes in Loop-Based Algorithms

Figure 5.7 shows code for part of the inner loops of tiled matrix multiply algorithm. Assuming the matrix elements are of type floats, each cache line would hold sixteen elements of the result matrix $C$. The loop highlighted in blue updates each of the cache lines sixteen times. After that, the loop highlighted in red will flush these cache lines. It will store the PC of the last store to that cache line before the flush. All of these stores have the same instruction PC, which is $PC_k$. Therefore, we insert $PC_k$ in the Signatures table with confidence value of weak-high-confidence. Next time we loop over the blue loop, it will find $PC_k$ in the Signatures table. Therefore, we will erroneously issue PreFlush request. As a result, we will decrement the confidence counter to weak-low-confidence. $PC_k$ would continue to have weak-low-confidence counter until we go into the red-highlighted loop of flushes again.
for (jj=0; jj<N; jj+=block_size) {
    for (i=ii; i<(ii+block_size); i++) {
        for (j=jj; j<(jj+block_size); j++) {
            for (k=kk; k<(kk+block_size); k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
} //end of for j
} //end of for I

for (l=ii; l<(ii+block_size); l++)
    FLUSH(&C[l][jj]);
} //end of for jj

Figure 5.7 Loop-based flushing following loop of stores.
Then it will be incremented to weak-high-confidence. The prediction toggling behavior will continue and we would never correctly predict these flushes.

To deal with this access pattern, we add a counter to each LastStore table entry. We will call this counter the *Occurrences* counter. If an incoming store request has the same instruction PC as the one recorded in the cache line’s LastStore entry, we increment the *Occurrences* counter to denote that we have seen this store before. When we insert the LastStore PC in the Signatures table the first time we encounter the flush, we insert the *Occurrences* counter with it. Later, when we encounter the same loop again, we will find the store instruction PC in the Signatures table. We also check its *Occurrences* counter. If it is greater than one, we know this store instruction had a loop-like pattern. In that case, we do not issue the PreFlush until the counter value from the LastStore entry matches the counter value of the signature.

We note that this optimization does not work with the counter-based optimization above. The reason is that we need to keep track of the most recent store’s instruction PC to know if should increment a counter or not. And since not all cache lines have LastStore entry, we cannot track the counter value for all of them.

### 5.4 Methodology

#### 5.4.1 Simulation configuration

We evaluated our PreFlush technique on a simulator that was built on top of *gem5* [59], an open source cycle-accurate full system simulator. Our simulator uses x86-64 instruction set architecture (ISA). It models a detailed 4-way out-of-order processor pipeline for each core, with parameters shown in Table 5.1. The table also shows parameters for the 2-level caches that we model, with private per-core L1 caches and an L2 cache shared by all cores. The memory hierarchy model is built on top of Ruby [66], with MESI protocol keeping the L1 caches coherent with respect to one another and with respect to the shared L2 cache.
Table 5.1 The baseline system configuration.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8-core, 2GHz, out-of-order x86-64, 8-wide issue/retire ROB: 192 Fetch/Issue/Store Queues: 32/32/32</td>
</tr>
<tr>
<td>Coherence Protocol</td>
<td>Ruby System, MESI Two Level Protocol</td>
</tr>
<tr>
<td>L1I &amp; L1D Cache</td>
<td>private, 2-cycle access, 64KB, 16-way, 64B block</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared, 20-cycle access, 1MB tile/core, 16-way, 64B block</td>
</tr>
<tr>
<td>MSHRs</td>
<td>32</td>
</tr>
<tr>
<td><strong>DDR-based PCM Main Memory</strong></td>
<td></td>
</tr>
<tr>
<td>Capacity</td>
<td>8GB</td>
</tr>
<tr>
<td>Organization</td>
<td>1 channel, 2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address mapping, FRFCFS scheduling policy</td>
</tr>
<tr>
<td>PCM latencies</td>
<td>50ns read, 150ns write</td>
</tr>
<tr>
<td>DDR Timing</td>
<td>tRCD 55ns, tWR 150ns, tCL 12.5ns, 64-bit bus width, 800 MHz Clock</td>
</tr>
<tr>
<td>Read &amp; write queues</td>
<td>32-entry RQ &amp; 64-entry WQ, 85% WQ high threshold, 50% low threshold</td>
</tr>
</tbody>
</table>

We modeled a Phase Change Memory (PCM) type of NVMM based on the DDR timing parameters shown in the table. This configuration for PCM is in based on the widely used parameters from Lee et al. [46] and previous work related to NVM [5, 10, 13, 14, 36, 46, 52]. These parameters would yield read and write latency of 60ns and 150ns, respectively.

We implemented x86 clflushopt and clwb instructions (clflush is implemented but unused). The ordering constraints of clflushopt and clwb are implemented as described in the Intel manual [33]. Specifically, they are ordered only with respect to memory fences (including sfence and mfence), and with respect to older loads/stores to the same cache line address. Similar to stores, clflushopt and clwb access the cache after it is retired from CPU pipeline. Moreover, clflushopt and clwb become durable and the instruction completes when the dirty cache block has been written back to the write buffer in the memory controller.
5.4.2 Workloads

In our evaluation we use seven different workloads. Four of these workloads are from the Whisper library [61] and are based on the NVML and Mneomsyne libraries [21, 74]. The other three workloads are microbenchmarks we developed to evaluate fundamental linear algebra kernels that are frequently used in scientific algorithms. Table 5.2 shows the different workloads we use in our evaluation and their description. For evaluating scientific algorithms, we implemented Tiled Matrix Multiply (TMM) based on the implementation in [26]. We evaluated two of the persistence granularities they proposed ($i_i$ and $j_j$). In addition, we evaluated our LU decomposition algorithm based on the implementation in the Splash-2 benchmark suite [79]. In all of our simulations, we execute at least 450 million instructions to warm up the caches and other structures, and we simulate at least one billion instructions.

Table 5.2 Evaluated workloads. The top four are from Whisper library. The bottom three are loop-based scientific microbenchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redis (RE)</td>
<td>key-value store from Whisper (based on NVML library)</td>
</tr>
<tr>
<td>Vacation (VA)</td>
<td>benchmark from Whisper (based on Mnemosyne library)</td>
</tr>
<tr>
<td>C Tree (CT)</td>
<td>crit-bit tree microbenchmark from Whisper (based on NVML)</td>
</tr>
<tr>
<td>HashMap (HM)</td>
<td>hashmap data structure microbenchmark from Whisper (based on NVML)</td>
</tr>
<tr>
<td>Tiled Matrix Multiply ii (TMM_i)</td>
<td>ii-granularity persistence for 1k-squared tiled matrix multiply</td>
</tr>
<tr>
<td>Tiled Matrix Multiply jj (TMM_j)</td>
<td>jj-granularity persistence for 1k-square tiled matrix multiply</td>
</tr>
<tr>
<td>LU decomposition (LU)</td>
<td>LU factorization algorithm for 1k-square matrices</td>
</tr>
</tbody>
</table>
5.5 Evaluation of PreFlush

5.5.1 Evaluated Designs

We evaluated two different designs of PreFlush: (1) PreFlush with the trace-based predictor we described in section 5.3.3 ($PF_{trace}$) and (2) PreFlush with the last-store optimization ($PF_{LS}$), which uses only the PC of the last store instruction for prediction instead of the full trace of instruction PCs (described in section 5.3.4). For both designs, we apply the counter-mode prediction, which is intended for loop-based workloads (described in section 5.3.8).

For the prediction accuracy study, we evaluated different versions of the PreFlush. The trace-based and last-store predictors do not include the counter-based prediction. The reason is that we wanted to isolate the effect of adding the counter-based predictor to show the performance impact it has on both loop-based and other workloads. We added two more designs, one that has the counter-based predictor applied to the last-store predictor, and another one that applies the sampled-sets optimization, which we described in section 5.3.5, with an underlying trace-based predictor. The reason we did not study the sampled-sets optimization in other experiments is that it cannot be applied with the counter optimization. And since we need the counter-based predictor for improving the loop-based workloads, we opted for excluding the sampled sets from the rest of the evaluation.

All experiments, except for the prediction accuracy, are normalized to a base design that achieves crash-consistency by implementing undo-logging using x86’s `clwb` and `sfence` instructions. Nevertheless, we also compare our designs with an ideal no-crash-consistency model that persists the modified data without any logging or failure-safety persistence. The reason we evaluate this version of the workloads is to compare against the ceiling of improvement any crash consistency software or hardware proposal can ever reach. We note that this is an unrealistic design to match since it does not guarantee crash consistency at all. We refer to the undo-logging design as Base and the no-crash-consistency as Ideal.
All our evaluated designs include the ordering violation guarantees, which we discussed in section 5.3.7. In our experiments, we only use clwb instruction. Our reasoning for not using clflushopt is that the misprediction of clflushopt degrades the performance, compared to clwb, as the data would have to be brought back to the cache from the memory controller.

Table 5.3 The sizes of the different structures we used in our PreFlush designs.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signatures Table</td>
<td>32 entries</td>
</tr>
<tr>
<td>FlushedTags Table</td>
<td>16 entry</td>
</tr>
<tr>
<td>Number of Sampled Sets</td>
<td>8</td>
</tr>
<tr>
<td>Trace Size</td>
<td>16 bits</td>
</tr>
<tr>
<td>Last Store Size</td>
<td>16 bits</td>
</tr>
<tr>
<td>PreFlushers Table</td>
<td>64 entries</td>
</tr>
</tbody>
</table>

5.5.2 Structures Sizing

For all our designs, we sized the structures empirically based on our experiments. Table 5.3 shows the sizes of the different structures. In addition to the structure sizes in the table, we augmented each cache line with an entry for tracking the store accesses. Each entry had sixteen bits for storing a subset of the trace or the last store. The sixteen bits where chosen starting from the least significant bit of the PC. We also added a valid bit to indicate whether the entry is valid or not. From our experiments, we did not see noticeable improvements when we used higher number of bits for tracking PCs.

5.5.3 Prediction Accuracy

Figure 5.8 shows the prediction accuracy of the four different designs of PreFlush. We can observe that the loop-based workloads suffer from very high misprediction rate for the first three designs, which do not employ the counter-based prediction. The last-store design
with the counter predictor, $PF_{LS}$, fixes this big performance hit and achieves 91.91% correct prediction, 3.74% misprediction, and 4.33% not-predicted flushes.

We can also see that the trace-based design ($PF_{trace,noCounter}$) and the Last Store design ($PF_{LS,noCounter}$) are more superior to the sampled-set design ($PF_{SS}$). Nevertheless, the difference between them is relatively small: the sampled set design has 8.63% and 6.2% lower correct prediction compared to $PF_{trace,noCounter}$ and $PF_{LS,noCounter}$, respectively. The reason is that $PF_{SS}$ tracks less number of sets, which means it takes longer time to train the predictor. Therefore, the lower percentage of correct prediction (the blue bar) of $PF_{SS}$ is attributed to the higher "not-predicted" percentage compared to $PF_{trace,noCounter}$ and $PF_{LS,noCounter}$. While the "mispredicted" percentage is comparable between all four designs.

**Figure 5.8** Percentage of clwb flushes that our PreFlush technique predicted, did not predict, and mispredicted. The bars from the left are: the trace-based design without counter-mode prediction ($PF_{trace,noCounter}$), the last-store design without counter-mode prediction ($PF_{LS,noCounter}$), the sampled-set design ($PF_{SS}$), and the last-store design with counter-mode prediction ($PF_{LS}$).
Figure 5.9 The percentage of *clwb* latency savings for our two PreFlush designs normalized to the base undo-logging model.

Figure 5.10 The percentage of performance improvement of our two PreFlush designs normalized to the undo-log model.
5.5.4 PreFlush Performance for Undo-Logging Workloads

We will start by discussing the performance and latency improvement for our two PreFlush designs when running the undo-logging versions of the workloads. Figure 5.9 shows the latency savings for performing the `clwb` operation. We note that these numbers do not include the execution latency inside the core. They only include the latency starting from the moment the request reaches the L1 cache until the time the L1 cache sends the acknowledgment to the Load-Store Unit (LSU) in the core. Our two PreFlush designs, $PF_{\text{trace}}$ and $PF_{\text{LS}}$, achieve a maximum of 81.1% and 75.4% latency savings for the `clwb` instruction, respectively. On average, they achieve 59.1% and 53.8% latency savings, respectively.

Figure 5.10 shows the performance improvement of $PF_{\text{trace}}$ and $PF_{\text{LS}}$ compared to base undo-logging model. Our two PreFlush designs improve performance over the undo-logging base model in all benchmarks. The maximum execution time improvement for $PF_{\text{trace}}, PF_{\text{LS}}$ over base are 49.9% and 47.4%, respectively. The average is 22.2% and 20.6%, respectively.

Overall execution time improvement in the transaction-based microbenchmarks is higher than the linear algebra kernels. We believe this is mainly attributed to the higher number of `sfence` instructions in transaction-style workloads. Nevertheless, we notice that although transaction-based workloads show higher performance improvement, their `clwb` latency savings are lower. From these results, we can conclude two things. First, the overhead of persistency for transaction-style workloads is higher than dense-matrix-based kernels. Second, the instruction latency improvement for `clwb` does not directly translate to performance improvement. We believe the reason for this disparity is twofold: (1) The high number of `clwb` instructions in transaction-based workloads means higher number of `sfence` instructions. Since `sfence` prevents subsequent store and `clwb` instructions from executing until the preceding stores and `clwb` instructions are complete and globally visible, it exposes the latency of both store and `clwb` instructions to the critical path of the program execution. Therefore, the overhead of making data durable at finer-granularity of `clwb` and
*sfence* pairs is higher than grouping these small transactions together and using the *sfence* once. This conclusion is evident in the difference between the two granularities of TMM kernel. The ii granularity (*TMM_ii*) groups more flushes together and have one fence after them while the jj granularity (*TMM_jj*) flushes the cache lines more aggressively with finer granularity. This observation is in line with the analysis in [61], which shows that deferring persistence until the end of the transaction and then committing the transaction in groups is better than performing the persistency in the middle of the transaction. (2) The number of data to persist in transaction-based workloads is relatively higher than it for linear algebra kernels. The reason is that in linear algebra kernels, we have several matrices, but we only persist the data of the result matrix. On the other hand, all the data in the transaction-based workloads are susceptible to persistence.

Another observation from these results is that the latency of *clwb* varies noticeably between the two *TMM* kernels granularity: 81.1% and 52.7% for TMM_jj and TMM_ii, respectively, in *PF* trace. In *PF* LS, 75.5% and 47.8% for TMM_jj and TMM_ii, respectively). The reason is that when we move from a smaller granularity checkpointing (i.e TMM_jj) to larger granularity (TMM_ii), we increase the likelihood of the cache lines to be already evicted from the cache hierarchy. And since writing back or flushing dirty cache line takes longer time than when the cache line is clean in the LLC, the overall benefit from the PreFlush lessens, as well. Nevertheless, we believe the main reason the ii-granularity is faster than the jj-granularity is that the higher granularity means less *sfence* instructions. To confirm our intuition, we ran tests on the two granularities with the same number of *sfence* instructions, and as we expected, the performance difference is primarily due to the *sfence* instructions following every *clwb* or *clflushopt* instruction.

### 5.5.5 PreFlush Execution Time for No-Logging Workloads

Figure 5.11 shows the performance improvement of *PF* _trace_ and *PF* _LS_ over the base ideal, no-crash-consistency version of the workloads. Our two PreFlush designs, *PF* _trace_ and *PF* _LS_,
Figure 5.11 The percentage of performance improvement of our two PreFlush designs normalized to the optimistic no-log model

achieve a maximum execution time improvement of 12.1% and 11.3% over base, respectively. On average, they achieve 7.4% and 5.7% execution time improvement. We note that although these results are relatively modest compared to the logging-based versions of the workloads, these designs are oracle, optimistic designs to see the maximum our schemes can achieve.

5.5.6 Area overhead

The area overhead comes from the 3 added structures. The Signatures table is 32 entries, each entry holds the 16-bit subset of the store PC in case of PF<sub>LS</sub> design or the trace encoding in case of PF<sub>trace</sub>. It also holds the 2-bit confidence counter, and an 8-bit counter-mode prediction counter. Therefore, the Signatures table adds 832 bits (104 bytes). Each entry in the FlushedTags table consists of the signature that triggered the preflush as well as the tag of the cache line. The total overhead of FlushedTags table is 1088 bits (136 bytes). The two designs also add an entry to each cache line. This entry holds the 16-bit last-store or trace-encoding for the cache line. It also has 1 valid bit and 8-bit counter for the counter-
based prediction. Therefore, we add 25 bits for each cache line. For a 64KB L1 cache with 64B cache line size, that overhead accounts for 25600 bits (3200 bytes). We also add the PreFlushers table in the LLC. Each entry in the PreFlushers Table has the cache line tag and the flusher id. For the flusher id, we assume a maximum of 256 cores connected to the last-level cache. Therefore, we need 8 bits for the id. In total, each entry is 60 bits. For the 64 entries that accounts for 3840 bits (480 bytes). The total overhead for our design is 31360 bits (3920 bytes) which is roughly 3.8KB.

5.6 Related Work

In order to maintain data consistency in NVMM, researchers have borrowed traditional storage techniques such as shadow-paging or write-ahead logging and applied them in the context of NVMM. Mnemosyne [74] proposed a library to access NVMM directly and used redo-logging to provide an atomic durable transaction. On the other hand, NV-Heaps [21] used software undo-logging for its atomic transaction implementation and improves its performance by logging at a large granularity instead of logging before every write. Rewind [19] proposed a library to manage NVMM directly from the user application and it also utilized a software undo-log approach for its durable transaction.

Hardware logging approaches for mitigating logging overheads were proposed in multiple published papers. Lu et al. [53] proposed LOC which relaxes the ordering constraint in transactions by using asynchronous redo-logging with background hardware supports. Similar to DudeTM, it requires additional memory accesses and space. Furthermore, it requires redirecting reads to logs or blocking reads during the logging operation until it has been committed to memory, which is generally considered an expensive operation. It also needs additional hardware support for updating memory in the background. Doshi et al. [25] proposed synchronous hardware redo-logging for atomic durability. However, their solution is not optimized for the case with multiple updates to the same address in
one transaction. In this case, they simply create multiple log entries for the same data, which consumes unnecessary memory bandwidth and also degrades performance. Recently, Joshi et al. [36] proposed ATOM which uses synchronous undo logging and includes hardware to help create only one log entry per update per transaction. They also showed better performance compared to a previous redo-logging scheme [25].

Our work is orthogonal to hardware logging support proposals. While these proposals work on reducing the overhead of logging, they still have to flush data out of the cache. Our work accelerates this flush operation by predicting it and eagerly starting it.

There have been a very rich body of research on last-touch and dead-block eviction over the years [40, 41, 44, 45, 48, 50]. These legacy cache optimizations were applied to problems such as prefetching and block replacement. They mostly differ in the prediction mechanism. They can be broadly grouped into four types: trace-based, time-based, cache-burst based, and counting-based predictors. While we do draw from the rich literature in prediction, we note that the design of the predictor is not the core of this work. Our aim is programming for non-volatile memory. We discuss the potential ordering violation that could arise from predicting flushing instruction. None of the previous work had any correctness problem; they only suffered performance in case of misprediction. We discuss and analyze different classes of workloads. Our evaluation analyzes how these workloads flush their data. Furthermore, even when it comes to the predictor implementation, we do not implement any of the legacy proposals on its own. Our predictor and its optimizations draw on the rich body of predictors in computer architecture over the years. We tailored it for our needs by including counter-based and trace-based hybrid predictor. We also had to add new structures to keep track of what we flush for correctness. These extra structures were never needed before because there was never a correctness problem in evicting cache line eagerly.
5.7 Conclusion

We present PreFlush, a lightweight and transparent hardware mechanism that predicts when a cache line flush or write back is needed and speculatively performs the operation early. Since we speculatively perform the flush, we add hardware to determine when it is more profitable to flush, and we also handle cases where the preflush misspeculates and still ensure correct execution without the need for any complex recovery mechanisms. Also, PreFlush requires no modification on existing NVMM-enabled code. Our results show that PreFlush can improve performance by up to 49.8% (22.2% average) for undo-logging workloads.
Non-Volatile Main Memory (NVMM) technologies will open the door for new opportunities for applications in modern and future systems. One of these opportunities is the ability to achieve data persistence in the main memory rather than having to store data in the much-slower storage.

In this dissertation, we shed the light on the opportunity to use NVMM for in-memory checkpointing. We analyzed different schemes to achieve failure safety. We presented a recompute-based failure safety approach and demonstrated its applicability to loop-based code.

Then, we analyzed different implementations of General Matrix Multiply (GEMM) algorithms when checkpointing in NVMM. We showed that there is a design choice to make if the write endurance of NVMM is of more concern than the algorithm execution time. We
also showed that this write-endurance/performance tradeoff can be more important when checkpointing in NVMM.

Finally, we presented PreFlush, a lightweight hardware technique to predict when a cache line flush or write back is needed and speculatively perform the operation early. Since we speculatively perform the flush, we add hardware to determine when it is more profitable to flush, and we also handle cases where the preflush misspeculates and still ensures correct execution without the need for any complex recovery mechanisms. Also, PreFlush requires no modification on existing NVMM-enabled code.
BIBLIOGRAPHY


