Concurrency Implications of Nonvolatile Byte-Addressable Memory

by

Joseph Izraelevitz

Submitted in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Supervised by Professor Michael L. Scott

Department of Computer Science
Edmund A. Hajim School of Engineering and Applied Sciences
Arts, Sciences and Engineering

University of Rochester
Rochester, New York

2018
Dedication

For my parents, my brothers, and, of course, for Lauren.
Table of Contents

Biographical Sketch ix

Acknowledgements xii

Abstract xiii

Contributors and Funding Sources xv

List of Tables xvii

List of Figures xviii

1 Introduction 1

2 Background 4

2.1 NVM Hardware .......................... 5

2.1.1 Power-Backed DRAM .......... 5

2.1.2 PCM .......................... 6

2.1.3 ReRAM .......................... 7
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.4 STT-MRAM</td>
<td>8</td>
</tr>
<tr>
<td>2.1.5 Memory Models and Processor Architectures</td>
<td>9</td>
</tr>
<tr>
<td>2.1.6 NVM Control Logic</td>
<td>12</td>
</tr>
<tr>
<td>2.1.7 Other Nonvolatile Technologies</td>
<td>13</td>
</tr>
<tr>
<td>2.2 NVM in the OS and Drivers</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Wear Leveling</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Persistent Errors</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3 Sharing and Protection</td>
<td>19</td>
</tr>
<tr>
<td>2.3 NVM Software Libraries</td>
<td>21</td>
</tr>
<tr>
<td>2.3.1 Failure-Atomic Updates</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 Persistent Data Structures</td>
<td>29</td>
</tr>
<tr>
<td>2.3.3 File Systems</td>
<td>31</td>
</tr>
<tr>
<td>2.3.4 Garbage Collection</td>
<td>33</td>
</tr>
<tr>
<td>2.4 NVM Software Applications</td>
<td>34</td>
</tr>
<tr>
<td>2.4.1 Databases</td>
<td>34</td>
</tr>
<tr>
<td>2.4.2 Checkpointing</td>
<td>41</td>
</tr>
<tr>
<td>3 Durable Linearizability</td>
<td>45</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>45</td>
</tr>
<tr>
<td>3.2 Abstract Models</td>
<td>50</td>
</tr>
<tr>
<td>3.3 Concrete Models</td>
<td>55</td>
</tr>
<tr>
<td>3.3.1 Basic Memory Model</td>
<td>56</td>
</tr>
<tr>
<td>3.3.2 Extensions for Persistence</td>
<td>57</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

3.3.3 Liveness ................................................. 61

3.4 Implementations ........................................... 63
  3.4.1 Preserving Happens-Before ......................... 65
  3.4.2 From Linearizability to Durable Linearizability .. 66
  3.4.3 Transform Implications .............................. 71
  3.4.4 Persist Points ....................................... 72
  3.4.5 Practical Applications .............................. 74

3.5 Conclusion .................................................. 75

4 Composing Durable Data Structures ...................... 76
  4.1 Introduction ............................................ 76
  4.2 Composition ............................................. 77
  4.3 Query-Based Logging .................................... 79
    4.3.1 The Chronicle ................................... 80
  4.4 Conclusion .............................................. 81

5 Failure Atomicity via JUSTDO Logging ................... 83
  5.1 Introduction ............................................ 83
  5.2 Concepts & Terminology ............................... 88
  5.3 System Model & Programming Model .................... 91
  5.4 Design .................................................. 97
    5.4.1 JUSTDO Log ...................................... 98
    5.4.2 Persistent-Only Accesses ......................... 100
    5.4.3 Register Promotion in FASEs ....................... 101
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.4</td>
<td>Lock Logs</td>
<td>102</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Recovery</td>
<td>103</td>
</tr>
<tr>
<td>5.5</td>
<td>Implementation</td>
<td>105</td>
</tr>
<tr>
<td>5.5.1</td>
<td>jd_root</td>
<td>107</td>
</tr>
<tr>
<td>5.5.2</td>
<td>jd_obj</td>
<td>107</td>
</tr>
<tr>
<td>5.5.3</td>
<td>JUSTDO Routine</td>
<td>108</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Recovery Implementation</td>
<td>109</td>
</tr>
<tr>
<td>5.6</td>
<td>Experiments</td>
<td>112</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Correctness Verification</td>
<td>113</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Performance Evaluation</td>
<td>114</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Recovery Speed</td>
<td>122</td>
</tr>
<tr>
<td>5.6.4</td>
<td>Data Size</td>
<td>122</td>
</tr>
<tr>
<td>5.7</td>
<td>Conclusions</td>
<td>124</td>
</tr>
<tr>
<td>6</td>
<td>iDO Logging: Practical Failure Atomicity</td>
<td>125</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>125</td>
</tr>
<tr>
<td>6.2</td>
<td>Background</td>
<td>128</td>
</tr>
<tr>
<td>6.2.1</td>
<td>System Model</td>
<td>129</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Programming Model</td>
<td>130</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Idempotence</td>
<td>131</td>
</tr>
<tr>
<td>6.3</td>
<td>iDO Failure Atomicity System</td>
<td>132</td>
</tr>
<tr>
<td>6.3.1</td>
<td>The iDO Log</td>
<td>134</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Indirect Locking</td>
<td>135</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.3.3</td>
<td>iDO Recovery</td>
<td>137</td>
</tr>
<tr>
<td>6.4</td>
<td>Implementation Details</td>
<td>139</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Compiler Implementation</td>
<td>139</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Persist Coalescing</td>
<td>141</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Persistent Region Support</td>
<td>142</td>
</tr>
<tr>
<td>6.5</td>
<td>Evaluation</td>
<td>142</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Performance</td>
<td>144</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Scalability</td>
<td>145</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Memory Logging Granularity</td>
<td>150</td>
</tr>
<tr>
<td>6.5.4</td>
<td>Recovery Overheads</td>
<td>151</td>
</tr>
<tr>
<td>6.6</td>
<td>Related Work</td>
<td>153</td>
</tr>
<tr>
<td>6.7</td>
<td>Conclusion</td>
<td>154</td>
</tr>
<tr>
<td>7</td>
<td>Dalí: A Periodically Persistent Hash Map</td>
<td>156</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>156</td>
</tr>
<tr>
<td>7.2</td>
<td>Motivation</td>
<td>160</td>
</tr>
<tr>
<td>7.3</td>
<td>Dalí</td>
<td>162</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Data Structure Overview</td>
<td>162</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Reads</td>
<td>166</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Updates</td>
<td>167</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Further Details</td>
<td>171</td>
</tr>
<tr>
<td>7.4</td>
<td>Correctness</td>
<td>175</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Linearizability</td>
<td>175</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

7.4.2 Buffered Durable Linearizability .................................. 176
7.5 Experiments ........................................................................ 180
7.6 Related Work ..................................................................... 183
7.7 Conclusion ......................................................................... 185

8 Conclusion ............................................................................. 187

A Other Works .......................................................................... 191
A.1 Performance Improvement via Always-Abort HTM .......... 191
A.2 An Unbounded Nonblocking Double-ended Queue .......... 192
A.3 Generality and Speed in Nonblocking Dual Containers ......... 193
A.4 Implicit Acceleration via Unsuccessful Speculation .......... 194
A.5 Interval-Based Memory Reclamation .............................. 195

Bibliography ............................................................................. 197
Biographical Sketch

Joseph (Joe) Izraelevitz received a Bachelor and Master of Science degree in Computer Science, with a second major in History, from Washington University in St. Louis in May 2009. He completed a master’s thesis entitled Automated Archaeological Survey of Ancient Irrigation Canals under the mentorship of Professor Robert Pless. Upon graduation, he received a commission in the US Army as an Armor officer and completed a three-year obligation to the service, including a year-long deployment as a staff officer in Afghanistan.

Joe attended the University of Rochester from Fall 2012 until Fall 2017, receiving a second Master of Science degree in Computer Science in May 2014. He was advised by Professor Michael Scott for the duration. The works he completed over the course of his doctoral work are listed below:


Acknowledgements

Though a thesis has, by tradition, a single name on the cover, this custom misrepresents the work that goes into a doctoral dissertation. I am indebted to all of my co-authors and collaborators whose work is reflected in this document. In no particular order, thank you to Michael L. Scott, Virendra Marathe, Terence Kelly, Aasheesh Kolli, Faisal Nawab, Dhruva Chakrabarti, Charles B. Morrey, Qingrui Liu, Se Kwon Lee, Sam H. Noh, Changhee Jung, Hensen Wen, Wentao Cai, H. Alan Beadle, Matt Graichen, Yossi Lev, Alex Kogan, Hammurabi Mendes, and Lingxiang Xiang for all your help in the development of the ideas presented here. In particular, I would like to thank my adviser, Professor Michael L. Scott, for the expertise and guidance he provided me over the five and some years of my doctorate. I have been in academia long enough to know that I lucked out when he accepted me as his student — I hope I can live up to his example.

I would further like to thank my friends in the computer science department who have worked alongside me for many years. This journey has been the more pleasant because it was not undertaken alone.
Abstract

In the near future, storage technology advances are expected to provide non-volatile byte addressable memory (NVM) for general purpose computing. These new technologies provide high density storage and speeds only slightly slower than DRAM, and are consequently presumed by industry to be used as main memory storage. We believe that the common availability of fast NVM storage will have a significant impact on all levels of the computing hierarchy. Such a technology can be leveraged by an assortment of common applications, and will require significant changes to both operating systems and systems library code. Existing software for durable storage is a poor match for NVM, as it both assumes a larger granularity of access and a higher latency overhead.

Our thesis is that exploiting this new byte-addressable and nonvolatile technology requires a significant redesign of current systems, and that by designing systems that are tailored to NVM specifically we can realize performance gains. This thesis extends existing system software for understanding and using nonvolatile main memory. In particular, we propose to understand
durability as a shared memory construct, instead of an I/O construct, and consequently will focus particularly on concurrent applications.

The work covered here builds theoretical and practical infrastructure for using nonvolatile main memory. At the theory level, we explore what it means for a concurrent data structure to be “correct” when its state can reside in nonvolatile memory, propose novel designs and design philosophies for data structures that meet these correctness criteria, and demonstrate that all nonblocking data structures can be easily transformed into persistent, correct, versions of themselves. At the practical level, we explore how to give programmers systems for manipulating persistent memory in a consistent manner, thereby avoiding inconsistencies after a crash. Combining these two ideas, we also explore how to compose data structure operations into larger, consistent operation in persistence.
Contributors and Funding Sources

The dissertation committee for this work consists of Professors Michael Scott, Chen Ding, and Engin Ipek from the Department of Computer Science at the University of Rochester, and Dr. Virendra Marathe at Oracle Labs in Burlington, MA.

This graduate study was supported, in part, by a Robert L. and Mary L. Sproull Fellowship, the Hopeman Scholarship Fund, and by grants from the National Science Foundation (contract numbers: CCF-1116055, CCF-0702505, CNS-1319417, and CCF-1717712).

The research presented here was additionally supported, in part, by NSF grants CCF-0963759, CNS-1116109, CCF-1422649, and CCF-1337224; by the U.S. Department of Energy under Award Number DE-SC-0012199; by support from the IBM Canada Centres for Advanced Study; by a Google Faculty Research award; and by Hewlett Packard Enterprises and Oracle Corporation.
Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of above named organizations.
List of Tables

2.1 Failure Atomic Systems and their Properties .............. 27

3.1 Instruction Equivalencies for Persistency .................. 58

6.1 Recovery time ratio (ATLAS/iDO) at different kill times ... 151
List of Figures

3.1 Linearization bounds for interrupted operations ........... 47

4.1 Treiber Stack Chronicle Implementation .................... 82

5.1 Two examples of lock-delimited FASEs ..................... 86
5.2 NVM hybrid architecture ..................................... 90
5.3 JUSTDO log format ......................................... 99
5.4 JUSTDO logging example (Globals) ......................... 110
5.5 JUSTDO logging example (JUSTDO Routine) ............... 110
5.6 JUSTDO logging example (main) ............................ 111
5.7 JUSTDO logging example (equivalent transient routine) .... 111
5.8 JUSTDO throughput on workstation ......................... 117
5.9 JUSTDO throughput on server ................................ 118
5.10 JUSTDO throughput on using CLFLUSH ................... 120
5.11 JUSTDO throughput vs. value size ......................... 123
6.1 NVM Hybrid architecture ................................. 128
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>FASEs with different interleaved lock patterns</td>
<td>129</td>
</tr>
<tr>
<td>6.3</td>
<td>iDO log structure and management</td>
<td>133</td>
</tr>
<tr>
<td>6.4</td>
<td>iDO compiler overview</td>
<td>139</td>
</tr>
<tr>
<td>6.5</td>
<td>iDO Throughput on SPLASH3</td>
<td>142</td>
</tr>
<tr>
<td>6.6</td>
<td>iDO Scalability Results</td>
<td>148</td>
</tr>
<tr>
<td>6.7</td>
<td>Performance comparison of iDO with NVThreads</td>
<td>149</td>
</tr>
<tr>
<td>7.3</td>
<td>Incrementally persistent hash map</td>
<td>160</td>
</tr>
<tr>
<td>7.4</td>
<td>Dalí globals and data types</td>
<td>164</td>
</tr>
<tr>
<td>7.5</td>
<td>The structure of a Dalí bucket</td>
<td>164</td>
</tr>
<tr>
<td>7.6</td>
<td>Dalí read method</td>
<td>167</td>
</tr>
<tr>
<td>7.7</td>
<td>Dalí update method</td>
<td>168</td>
</tr>
<tr>
<td>7.8</td>
<td>Lookup table for pointer role assignments</td>
<td>169</td>
</tr>
<tr>
<td>7.12</td>
<td>A sequence of Dalí updates</td>
<td>171</td>
</tr>
<tr>
<td>7.13</td>
<td>Dalí scalability experiments</td>
<td>184</td>
</tr>
<tr>
<td>7.14</td>
<td>Impact of read:write ratio on Dalí throughput</td>
<td>184</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Memory technology trends suggest that nonvolatile byte addressable memory (NVM) will soon be ubiquitous. Compared to existing DRAM, technologies such as PCM [18], ReRAM [185], and STT-MRAM [27] are expected to provide far greater density at similar cost and only slightly slower speed, while also providing durability similar to that of disk.

We expect the availability of fast, reliable NVM as a DRAM replacement to be a significant, disruptive technological advance, calling for the development of software that explicitly exploits both durability and byte addressability. Not surprisingly, existing storage software is a poor match for these new technologies. In wide ranging studies of NVM as a disk replacement [22, 28], researchers found that throughput gains, while impressive, exposed other software inefficiencies, previously masked by storage latency, which limited
improvements.

The availability of NVM is likely to affect the development of a large variety of software. For instance, we expect it to be useful for applications that do frequent durable writes, such as databases or key-value stores. Fast checkpointing of applications is another likely application of NVM. For power-constrained hardware, such as smart phones and laptops, the ability to flush all volatile state to durable storage blurs the line between sleep and hibernate. For large-scale computing, fast checkpointing is critical to prevent data loss during long-running experiments over thousands of machines. For workstations, fast checkpointing can make power outage and hardware failure less likely to result in lost data.

However, NVM comes with its own challenges. Some NVM technologies must be wear–leveled, as each bit has a limited number of writes it can sustain before failing. More significantly, the ability to persist information at the byte level means that errors in software or soft errors in hardware may permanently corrupt nonvolatile state. Operating system protections must also be considered; decisions should be made regarding the extent to which persistent memory can be shared and how its permissions are enforced. Finally, at least in the near future, we expect that caches and processor registers will remain volatile. The loss of an arbitrary subset of the working set in a power failure may result in irreparable damage to the persistent state.

This dissertation explores the various software challenges revealed by the wide availability of NVM as a DRAM replacement, with a specific focus on
concurrent access and the consistency problems created when pairing NVM main memory with volatile caches and registers.

In Chapter 2 we provide necessary background on NVM hardware and software, NoSQL databases, and shared-memory data structures. Chapter 3 introduces an extension to linearizability applicable to data structures designed for NVM, and demonstrates a simple transform for building nonblocking data structures for an NVM-enabled machine. Chapter 4 then discusses strategies for composing NVM-adapted data structures into larger transactions. Chapter 5 then introduces a novel method to implement failure-atomic code regions — that is, code regions whose results are guaranteed to become persistent atomically. Chapter 6 improves on this failure atomicity system by using compiler support to reduce the logging overhead of the technique. Chapter 7 then describes a novel data structure algorithm created for NVM using a novel design paradigm, periodic persistence, and Chapter 8 concludes this work. Finally, Appendix A describes work done over the course of this thesis which explored problems in concurrency without applications to NVM.
Chapter 2

Background

This chapter provides the technical background for this thesis, giving a dis- 
cussion of the current state of NVM device technologies and of some of the 
work that has already been done integrating nonvolatile memory into the sys-
tem stack. Section 2.1 discusses the current state of various NVM hardware 
technologies for DRAM replacement and their expected performance upon 
reaching the commercial market. Section 2.2 reviews driver and kernel-level 
research applicable to NVM, including wear-leveling techniques, operating 
system protections on shared memory, and error correction in pointer-rich 
data structures, while Section 2.3 explores software functionality that we 
would expect to find at the library level. In particular, this section explores 
persistent data structures, transactional memory, garbage collection, and 
NVM file systems. Finally, Section 2.4 discusses some end use cases of the
technology, including database storage and fast durable checkpointing.

2.1 NVM Hardware

A variety of new nonvolatile byte-addressable technologies have already or are expected to enter the market in the coming years. While each technology is expected to provide durability, they vary significantly over critical dimensions such as mean time to failure, expected density, and read and write speeds. We here provide a brief overview of some of the more mature of these technologies and their properties.

2.1.1 Power-Backed DRAM

Traditional DRAM is the most common main memory technology today. Every DRAM bit stores information via a capacitor – whether or not the capacitor is charged indicates the value of the bit. Writes to a bit charge or discharge the capacitor. Reads are destructive; reading a bit discharges the capacitor, necessitating a refresh of the bit after every read. Furthermore, the capacitors leak charge over time, requiring the bit to be periodically refreshed \[41\], \[109\]. This leakage is the reason that traditional DRAM is volatile – power loss will prevent the memory from refreshing itself. Current DRAM technology provides read and write access times around 10 ns — slower than SRAM, but dramatically faster than disk and flash memory.
In the past, battery-backed DRAM has been available to system designers, providing enough energy to drain DRAM information to disk in the event of a power loss. More recent developments have introduced supercapacitor backed DRAM, which replaces the battery with a more reliable large capacitor and reduces drain time by flushing to faster nonvolatile flash memory \cite{84, 146, 194}.

Unfortunately, the scalability of DRAM is unlikely to continue as feature size shrinks. Problems with building narrow enough capacitors, bit flips caused by cosmic rays, and transistor leakage indicate that the technology will have problems scaling to smaller node sizes. As new NVMs do not store charge, these issues will not affect their scaling.

2.1.2 PCM

Phase change memory (PCM) works by changing the electrical properties of a small cell of a phase change material. To write a value, a cell is heated, then cooled. Depending on the temperature to which the cell is heated, and the speed at which it is cooled, the phase change material solidifies into either a crystalline or noncrystalline (amorphous) phase; the difference between these forms results in changes to the electrical resistance of the bit. Reading a value simply measures the electrical resistance of the cell \cite{18}.

As with other new nonvolatile technologies, no power is required to maintain the state of the bit, and depending on temperature cells can retain their
information for decades \cite{166}. Bit endurance is expected to be around $10^8$ writes \cite{59}: better than Flash but worse than DRAM. PCM as a DRAM replacement would require wear leveling \cite{18}.

PCM write performance is in general limited by the crystallization speed of the phase change material, and the search for faster materials is ongoing. Based on a wide ranging survey of PCM technology, writes to PCM are expected to be in the range of 20-200ns, within an order of magnitude of DRAM, while reads are as fast as DRAM \cite{18}. Memory density has surpassed that of state-of-the-art DRAM \cite{122, 181}, achieving cell sizes of 3 nm, and multi-level cells have been developed \cite{101}, which store multiple bits per cell.

Intel and Micron have recently introduced 3DXpoint (believed to be a phase change memory type technology \cite{16}) to the commercial market as a PCIe-based storage device. Though marketed as $1000\times$ faster than NAND Flash \cite{88}, as of the time of this writing, the devices are testing about $10-100\times$ faster \cite{16}. No DIMM cards are yet available for this technology, but they are expected soon.

### 2.1.3 ReRAM

Resistive RAM (ReRAM) is another emerging nonvolatile memory technology (also called Memristor \cite{185}). In some ways similar to PCM, ReRAM operates by modifying the resistance of a cell. Unlike PCM, ReRAM uses a different method to modify the resistance. Two electrodes are separated by a
metal oxide lattice. Applying current one of the electrodes forces oxygen ions in or out or the electrodes and into the metal oxide, forming or destroying conductive filaments across the lattice. Depending on the location of oxygen ions, the cell exhibits varying resistances, representing the cell value. Like PCM, ReRAM can be made into multi-level cells [205].

Unfortunately, like other NVM methods, ReRAM can suffer random bit flips, which occur as conductive filaments spontaneously break or form. Error correcting codes appear to be a necessity for the technology, but with them it can reliably store data for a decade. Wear leveling will still be necessary for this technology: its endurance is slightly higher than PCM, but still not as reliable as DRAM [199].

At this point, ReRAM is slower than other main memory technologies, with reads around 15 ns and writes around 150 ns [205].

### 2.1.4 STT-MRAM

Spin Transfer Torque Magnetic RAM (STT-MRAM) is an emerging non-volatile technology [27] which has already been brought to market [49]. STT-MRAM stores information as the magnetic polarity of three parallel layers. The top layer, called the *reference layer*, is always oriented on a fixed polarity, while the bottom layer, the *free layer*, can switch polarity. A *barrier layer* separates the two magnetized layers. The resistive properties of an STT-MRAM cell depend on whether the magnetic layers are in a parallel or
antiparallel alignment – determining the value of the stored bit \( [116] \).

STT-MRAM can provide competitive read access times, and write speed varies by current – with large amounts of current, write speeds can be comparable with DRAM \( [116] \). Cell endurance is effectively infinite \( [212] \). Unfortunately, STT-MRAM is subject to a higher than normal soft error rate. Read currents can flip cell polarity, bit flips can occur due to the thinness of the barrier layer, and writes can fail to flip the free layer \( [212] \).

STT-MRAM density is affected by these soft errors, which grow as the cell size shrinks \( [212] \). However, working cells have already been shrunk to smaller than DRAM \( [107] \).

### 2.1.5 Memory Models and Processor Architectures

Assuming a volatile cache, software must control and order the write back of cache lines to persistent memory; otherwise it is possible, for instance, to write back a pointer value before its target is persistent. Consequently, carefully synchronized cache flushing must be used to ensure that stores reach memory in the desired order \( [11] \). Key work in discussing the ordering requirements of persistent memory ("memory persistency") was done by Pelley et al., \( [164] \) describing a variety of ordering methods that could be exported to software and a formalization of the concepts. In general memory persistency theory extends memory consistency \( [2] \): we can consider a recovery thread, which reads the state of persistent memory at a crash, and
consider the program to be correctly written if the recovery thread always reads consistent state [164].

A variety of persistency schemes have been developed, with varying levels of hardware adoption. In general, these schemes use three instruction primitives. The first primitive, alternately called a persist, clean, or flush operation, forces data to be written back to nonvolatile storage (and may or may not invalidate the cache line). The second primitive is an ordering instruction, called a persist barrier or persist fence, that ensures any flushes issued before the fence complete before subsequent flushes after the fence. The scope of the barrier — that is, which flushes it applies to, varies by model (for instance, the fence may only apply to flushes issued by the fencing thread). The final operation is called a persist sync, which waits until previously issued flushes have been stored durably before returning. Like fences, syncs can be scoped by the model [164].

Various memory persistency models have been proposed. Epoch ordering [34] groups persistent writes into epochs using epoch barriers. An epoch barrier instruction is a combination of a persist fence and a sync. Writes can be reordered within an epoch, but all writes of a given epoch are guaranteed to be persistent before those of the next one. Alternatively, strict persistency [164], requires that the recovery thread see all writes in the same order as all other threads, making the persistency order the same as the consistency order, but allowing for buffering and coalescing.

The x86 ISA provides growing support for persistency control. The
clflush, clflushopt, and clwb instructions enable a processor to force a cache line back to the memory controller, acting as a persist and, depending on the instruction, evicting the cache line. These instructions are ordered across threads by the sfence instruction, which acts as a persist fence and ensures that the flushes have reached the memory controller [87]. For copying large amounts of data, the nontemporal stores of the movntq type instructions may be useful, as these writes bypass the cache hierarchy entirely [196] and do not allocate cache lines. They are also ordered by sfences, ensuring the writes reach the memory controller [86].

A major concern of earlier persistency models was the potential reordering of writes into persistence within the write pending queue (WPQ) of the memory controller. The WPQ buffers writes as they are written back from the caches into main memory, then issues them to the actual storage medium. If the write pending queue is transient, and issues writes to NVM in a non-FIFO order (as is generally the case for DRAM cards), it is possible for a NVM aware DIMM card (or NVDIMM), through the WPQ, to reorder writes into persistence, even if writes reach the WPQ in the correct order. To solve this problem, Intel announced the asynchronous DRAM refresh (ADR) feature in 2016. ADR guarantees that for any NVDIMM, the attached memory controller ensures that the write queues are automatically flushed into persistence on power failure. This feature thereby prevents the effective reordering of stores into persistence by the memory controller [172].

More recent work has developed and tested processor architectures for
other persistency models. Various schemes and their hardware include epoch persistence [122], buffered epoch persistence [102, 164], explicit epoch persistence [95], delegated persist ordering (DPO) [110], and the hands-off persistence system (HOPS) [150].

2.1.6 NVM Control Logic

The availability of NVM as a possible DRAM replacement necessitates a variety of changes in the control logic of main memory.

Failure Atomicity

The granularity at which writes to NVM are guaranteed to be atomic (called \textit{persist granularity} [164]) is critical to maintaining a consistent persistent state—writing half a byte to persistent memory is almost guaranteed to corrupt state. Atomicity of writes (failure atomicity) has been investigated by Condit et al.; [34] in the case of a power loss, the design uses a tiny capacitor to ensure that a block of eight bytes is guaranteed to be atomic.

Bit Errors

Like DRAM, NVM, especially STT-MRAM, is liable to bit flip errors. Error detection and correction (ECC) for DRAM is a widely studied area with well known solutions. In general, commercially available DRAM provides error correction for one bit error per 64 bit word, and error detection for
two bit errors per word. Popular error correction schemes which add check
bits include Hamming error codes \cite{69} and triple modular redundancy \cite{197}. Improvements are made by displacing the check bits from the associated
data, for instance, in the Chipkill ECC scheme \cite{89}. In general, the overhead
of ECC hardware must be factored into NVM hardware design, as smaller
and more efficient chips tend to incur more errors.

Other error detection systems are more suited to disk storage. Checksums
and duplication (e.g. RAID) are common techniques which, depending on
the hardware, may be amenable to use with NVM.

2.1.7 Other Nonvolatile Technologies

Though this thesis, for the most part, only considers the implications of
NVM as a DRAM replacement, other nonvolatile technologies are available
and may be further integrated into the memory hierarchy in the coming
years. We briefly discuss these advances here.

Storage Class Technologies

Storage class technologies are data storage devices with relatively high la-
tency access times, durable data storage, and a low cost per bit. In re-
cent decades, this market has been dominated by hard disk drives (HDDs),
whereas earlier magnetic tape was used. In the last decade, flash memory
has emerged as a viable nonvolatile storage technology. Flash memory works
using a floating gate design, which traps electrons between two transistors, changing the threshold voltage of the cell. There are two types of flash memory. NAND flash puts cells in series, enabling a dense cell array but slower random access times due to lower address granularity. NOR flash puts every cell on the word and bit lines, giving faster random access. NAND flash serves as a higher performance alternative to disk, though at a higher price point. NOR flash is useful for read–mostly byte addressable storage, such as boot sectors for embedded systems [17]. Both NAND and NOR flash suffer from endurance problems: NAND flash devices typically use a log–structured file system to even wear [171].

All previously mentioned NVM technologies have also been considered for storage class memory, by varying design points to improve density and cost at the expense of latency [18] [118].

**SRAM Replacement**

On the other end of the storage spectrum, NVM could be used as an SRAM replacement for caches and registers. The likely candidate for this transition is STT-MRAM, which provides read times close to current SRAM technology, though generally with slower writes. Possible solutions include mixed SRAM and STT-MRAM caches, with the lower level caches remaining SRAM or, alternatively, relaxing the nonvolatility constraint on STT-MRAM by leaving it more susceptible to soft errors and requiring a refresh operation [180].
CHAPTER 2. BACKGROUND

Battery Backup

At the present time, failure resilience to power outages is generally provided by battery backup (e.g. uninterrupted power supplies (UPS)), which use large batteries to ensure the system is shut off via a safe manner. Unfortunately, UPS's require maintenance, may not be reliable, and are subject to various financial and regulatory burdens. Furthermore, the use of UPS's still requires that software be at least somewhat failure resilient to inconvenient shutdowns, as the machine will be shutdown once the backup battery runs out. Battery backups are also not universally available, whereas if NVM is widely used as a DRAM replacement, it will already be available for use as persistent storage.

For now, it appears that batteries (or supercapacitors) will have a place in NVM chips as part of the ADR system, which drains the write pending queue of the memory controller. By extending the persistence domain through the memory controller, persistent storage in NVM is isolated from both power failures and fail-stop hardware faults in the processor. While extending the persistence domain into the caches would simplify the programming model, and we examine such a system in Chapter 5, it both makes persistence storage vulnerable to hardware faults in the processor and requires significantly more backup power to drain the caches, which reach into the megabytes.
2.2 NVM in the OS and Drivers

As NVM becomes more prevalent, a variety of systems software research is required in order to provide sufficient functionality at or around the operating system level. This section describes some operating system level problems and solutions as explored in the relevant literature.

2.2.1 Wear Leveling

As mentioned previously, NVM technologies, notably PCM and ReRAM, but also flash, lack the endurance of DRAM. It is possible to destroy a memory cell in under a minute if it is flipped constantly at full speed. Consequently, wear leveling of some sort is required to protect against device failure. Wear leveling can be solved at both the hardware and software levels.

At the hardware level, a variety of schemes exist for achieving uniform wear leveling, and we can draw from a wide body of research designed for NAND flash memory [8, 171]. In general, these methods track wear statistics for physical blocks and use an indirection table to move high traffic areas when necessary [105]. Schemes designed specifically for NVM, however, try to minimize tracking and translation overhead as accesses have lower latency than they do in flash.

Perhaps the easiest solution is to reduce the number of writes actually seen by NVM. By using (power–backed) DRAM as a cache for PCM or ReRAM, we can minimize the number of writes actually seen by the lower
endurance storage.

However, a DRAM cache does not alleviate all endurance concerns; we still need to wear level the NVM main memory. One common idea is a rotation scheme. A rotation scheme gradually rotates a cache line (or page) around itself by shifting the line by a small amount at every write \[52, 213\]: this scheme ensures that hot virtual addresses get rotated within the line. Gaps can also be introduced in the cache line to improve the leveling \[168\].

Unfortunately, rotation schemes at the cache line level are generally insufficient: hot spots tend to cover the entire line. Possible solutions include address randomization, which shuffles addresses when pages are mapped into NVM \[168\], and segment swapping, which copies the entire page to a new frame when too hot \[213\]. Another method is to compare the value in memory to the desired new value and avoid rewrites at the bit level \[52, 213\].

Once lines fail, avoiding memory fragmentation can be desirable. By consolidating failed lines into the logical end of pages, hardware can prevent extensive fragmentation of the address space \[53\].

At the software level, some work has been done in both library support and appropriate data structures. Clever memory allocation can reduce the amount of rewriting for a specific location by cycling across the address space during allocation and free \[149\]. Copy–on–write style data structures provide a similar service by avoiding repetitive writes \[191\].

Software can also explicitly take action when lines fail. The operating system would be expected, in general, to copy memory away from faulty
pages. The memory allocator could also help with static failures by never allocating faulty memory. In managed languages, the garbage collector can be used to handle dynamic errors. The garbage collector simply copies the object away from the faulty memory and redirects all pointers to it, then never frees the faulty lines [53].

2.2.2 Persistent Errors

Data consistency in durable storage has focused primarily on file systems. File systems have a significant advantage over byte addressable storage – they can allow unprivileged, poorly written or compromised programs to corrupt files, but access to the file system metadata is protected. When something goes wrong, the file system metadata can be checked using `fsck`, a command that exploits redundancies in the system to fix erroneous values. Often redundancies are built into the file system, using data duplication or checksums.

Using NVM as a byte-addressable device exposes it to a variety of errors not normally seen by disks. Software errors that corrupt persistent memory are extremely difficult to fix. An out-of-control program could trash a significant section of memory before crashing, particularly if persistent metadata is not protected with any sort of memory protection. These issues are much more problematic for nonvolatile storage than volatile—we cannot uncorrupt our data by rebooting and reloading from disk. Also, due to the nature of
NVM, bit flip errors may occur. While single bit flip errors can be corrected in hardware using ECC, double bit flips on a line may permanently corrupt the data.

Avoiding data corruption for NVM can draw on work that tries to prevent memory usage bugs, such as indexing errors and memory allocation errors. Managed memory languages, such as Java or Ruby, provide run-time checking of program execution and can prevent various errors that would otherwise trash memory—for instance, buffer overflows and dangling pointers. Substantial work has also been done in unmanaged memory languages, such as C or C++, to harden software against illegal accesses. For instance, customized memory allocators sparsely allocate objects in the virtual address space [140, 155] or maintain a type-specific pool [4].

### 2.2.3 Sharing and Protection

When memory becomes durable, the extent to which it can be protected and shared becomes important. Most literature assumes that nonvolatile memory segments will be stored as files on disk or other backing store. When a process wants to access a segment, it maps it onto nonvolatile memory, can then use the byte-addressable interface. When a process unmaps the segment, it writes it back to the file system. If the system crashes during this procedure, the operating system and owning process must decide how to recover and possibly unmap the nonvolatile segment from memory. Note that this procedure is
effectively the same as a memory mapped file: the only difference is that the open file will survive a crash since it is stored in NVM.

This procedure creates several problems. For instance, there is no guarantee that the nonvolatile segment will be mapped to the same virtual address every time it is accessed. Pointers that point to volatile memory, or to another nonvolatile segment, will become outdated upon remapping. Strong typing of pointers (including a base address offset) \[32\] or the use of a single address space scheme (where addresses are independent of context) \[25, 56\] can resolve some of these issues. As observed in \[56\] comparable problems and solutions can be seen in the dynamic linking of libraries, which share durable code (instead of data) segments.

By their very nature as saved main memory, nonvolatile segment files are exceptionally good targets for attack. Digital signatures, used in DLLs, are not useful for detecting modifications since the nonvolatile segment files are not read only. Fortunately, they can be explicitly loaded into the data segment of a process, preventing the direct execution of the data, though SQL injection–type attacks are possible by modifying stored code. It is likely that some sort of permissions are required to link trusted programs with certain nonvolatile segment files.

In Aerie \[195\], trusted programs are linked with specific nonvolatile segments to provide fast and secure storage, similar to a traditional file system. These trusted programs have special access to the file system metadata segment, but do not require kernel level privileges. Applications using the
storage communicate with the trusted program via RPC, but can map their files to their own memory. By replacing a system call with RPC, this system provides protected access to the file system metadata without system call overhead.

2.3 NVM Software Libraries

Whereas previous sections focused on the critical components of an NVM enabled system, this next section discusses library-level abstractions that may be used to simplify or speed up the use of the new technologies.

2.3.1 Failure-Atomic Updates

It seems clear that NVM technology will require some sort of transactional semantics: often a programmer will want to modify persistent state in a failure-atomic manner across multiple locations. An incomplete transaction broken by a power loss could permanently corrupt durable storage. Such a requirement exists even in a sequential persistence-enabled program. Fortunately, a large body of work exists regarding transactions, both for byte-addressable memory and for on-disk databases.

Transactions are a widely used synchronization abstraction that simplifies programming concurrent software. A single transaction accesses several data locations at once, but its effects become visible in an “all or nothing” manner.
For instance, to transfer money between two bank accounts, we would need to decrement the payer’s balance and increment the payee’s balance. A system in which only one operation (increment or decrement) is visible would be inconsistent.

Transactions are written as a single piece of sequential code that modifies global state. A correct implementation of transactions ensures the ACID properties, that is:

**Atomicity**  The transaction’s effects should all occur, or none should occur. Also called *failure atomicity*.

**Consistency**  Before and after the transaction, the global state satisfies all application-specific constraints.

**Isolation**  Transactions should observe no changes to program state by other threads during their execution, nor should their intermediate states be visible to other threads.

**Durability**  Transaction effects should become durably stored on commit. This requirement is ignored for volatile systems.

**Software Transactional Memory**

Software transactional memory (STM)\(^{179}\) for volatile systems is a way to provide transactional semantics to the programmer. A large number of high quality implementations exist\(^{142, 143, 186}\) and they vary according to a
variety of design decisions, each of which has impacts on performance \[71\]. We summarize these design parameters here.

Concurrency control refers to the resolution of concurrent accesses to the same data within two separate transactions. The control scheme must resolve these conflicts in order to preserve consistent state, generally by aborting one or more of the conflicting transactions. Pessimistic concurrency control detects and resolves the conflict immediately, often using locks. Optimistic concurrency control delays detection and resolution until later, generally at commit time.

Version management refers to the method by which transactional writes are stored prior to commit. Eager version management (or direct update) directly modifies data, while maintaining an undo log in the case of transaction abort. Lazy version management (or deferred update) waits to update the actual memory location until transactional commit, and maintains a redo log to store its tentative transactional writes.

Conflict detection refers to the method by which conflicting transactions are found. Detection can occur at a variety of points, either at first acquisition of the data (eager detection), at an intermediate validation point, or at commit time (lazy detection). Detection is always done at a larger granularity than the byte level, which means that false conflicts may occur due to collisions.

Correctness of a transactional memory system can be defined in a number of ways. In general, serializability \[162\] is useful for databases: it ensures that
transactional updates satisfy all ACID properties, but may reorder transactions that are otherwise ordered by a happens–before relationship. *Strict serializability* is stronger; it satisfies the ACID properties and also respects happens–before orderings. Correctness of transactional memory must also consider how to handle nontransactional loads and stores, so–called *mixed mode accesses*. *Weak isolation* make no guarantees about how nontransactional accesses interact with concurrent transactions. *Strong isolation* (or *strong atomicity*) \cite{14} respects the ordering of these accesses, effectively upgrading these loads and stores to tiny transactions. Finally, correctness should define what is visible to failed transactions. *Opacity* \cite{65} requires that transactions (even ones which are guaranteed to abort) should never see inconsistent state. In contrast, a *sandboxing* STM system allows transactions to read inconsistent state, as long these transactions are both guaranteed to abort and can never impact the safety of the system (e.g., by crashing the program or doing I/O). \cite{36}

*Nesting* of transactions occurs when a transaction is invoked from within another transaction. The simplest resolution of this scenario is *flattened nesting*, which joins the two transactions together. If either abort, both abort. Alternatively, *closed nesting* allows the inner transaction to abort and restart without affecting the outer one, but when it commits its changes are only visible to the enclosing transaction. In contrast, *open nesting* makes the inner transaction’s writes globally visible before the outer transaction commits. If the outer transaction aborts, the inner transaction’s changes
A special type of nesting, called boosting, allows transactional memory to interact with concurrent data structures. Boosting is a mechanism to raise the level of abstraction, detecting conflicts at the level of semantically non-commutative operations, rather than just loads and stores. Boosting reduces the overhead of tracking accesses and instead records only higher level data structure accesses (e.g. `pop()`). The boosting technique thus gains the benefits of high performing concurrent data structures while still maintaining transactional semantics [72, 78].

**Hardware Transactional Memory**

A long period of research and development into hardware transactional memory (HTM) [80] has resulted in commercial processors such as Intel’s Haswell line [68] and IBM’s Power8 [121] with the feature available. In brief, hardware transactional memory uses the cache coherence layer to isolate ongoing atomic transactions and to detect data conflicts at cache-line granularity. This system significantly reduces the bookkeeping overhead of transactional memory versus STM and provides a useful programming technique for implementing critical section speculation.

However, most current HTM systems are “best effort” only. In particular, HTM may abort for a variety of non-conflict-related reasons. An HTM transaction will abort when the transaction’s working set grows too large, upon the execution of certain instructions (such as I/O instructions or syscalls), the
reception of interrupts, and, of course, on the discovery of a data conflict. System configuration can have a significant impact on HTM performance. For instance, the use of hyperthreading reduces a thread’s effective cache size, raising the abort rate. Hybrid transactional memory \(^{37}\) attempts to integrate more flexible but slower software transactional memory with HTM to solve some of these issues.

**Failure Atomicity Systems for NVM**

Analogous to volatile transactional memory systems, which provide atomicity, isolation, and consistency to volatile programs, are *failure atomicity systems* which provide atomicity, consistency, and durability to programs using NVM. Failure atomicity systems ensure post-crash consistency of persistent data by allowing programmers to delineate failure-atomic operations on the persistent data—typically in the form of transactions \(^{32,111,129,174,196}\) or *failure-atomic sections* (FASEs) protected by outermost locks \(^{24,83,90}\). Given knowledge of where operations start and end, the failure-atomicity system can ensure, via logging or some other approach, that all operations within the code region happen atomically with respect to failure and maintain the consistency of the persistent data. Transactions have potential advantages with respect to ease of programming and (potentially) performance (at least with respect to coarse-grain locking), but can be difficult to retrofit into existing code, due to idioms like hand-over-hand locking and to limitations on the use of condition synchronization or irreversible operations. These
systems vary across a number of axes: Table 2.1 summarizes the differences amongst the systems.

Table 2.1: Failure Atomic Systems and their Properties

<table>
<thead>
<tr>
<th>System</th>
<th>Failure-atomic region semantics</th>
<th>Recovery Method</th>
<th>Logging Granularity</th>
<th>Dependency tracking needed?</th>
<th>Designed for transient caches?</th>
</tr>
</thead>
<tbody>
<tr>
<td>iDO Logging</td>
<td>Lock-inferred FASE</td>
<td>Resumption</td>
<td>Idempotent Region</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Atlas [24]</td>
<td>Lock-inferred FASE</td>
<td>UNDO</td>
<td>Store</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mnemosyne [196]</td>
<td>C++ Transactions</td>
<td>REDO</td>
<td>Store</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>NVThreads [32]</td>
<td>Lock-inferred FASE</td>
<td>REDO</td>
<td>Page</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JUSTIO [50]</td>
<td>Lock-inferred FASE</td>
<td>Resumption</td>
<td>Store</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>NVHeap [32]</td>
<td>Transactions</td>
<td>UNDO</td>
<td>Object</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>NVML [174]</td>
<td>Programmer Delineated</td>
<td>UNDO</td>
<td>Object</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SoftWrAP [58]</td>
<td>Programmer Delineated</td>
<td>REDO</td>
<td>Contiguous data blocks</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Mnemosyne [196], NV-Heaps [32], SoftWrAP [58], and NVML [174] extend transactional memory to provide durability guarantees for ACID transactions on nonvolatile memory. Mnemosyne emphasizes performance; its use of redo logs postpones the need to flush data to persistence until a transaction commits. SoftWrAP, also a redo system, uses shadow paging and Intel’s now deprecated pcommit instruction [87] to efficiently batch updates from DRAM to NVM. NV-heaps, an undo log system, emphasizes programmer convenience, providing garbage collection and strong type checking to help avoid pitfalls unique to persistence—e.g., pointers to transient data inadvertently stored in persistent memory. NVML, Intel’s persistent memory transaction system, uses undo logging on persistent objects and implements several highly optimized procedures that bypass transactional tracking for common functions.

Other failure atomic run-time systems use locks for synchronization and delineate failure atomic regions as outermost critical sections. Atlas [24] is
the earliest example; it uses undo logs to ensure persistence and tracks de-
pendences between critical sections to ensure that it can always roll back
persistent structures to a globally consistent state. Another lock-based ap-
proach, NVThreads [83] operates at page granularity using copy-on-write and
redo logging.

The above failure atomicity systems are nonvolatile memory analogues
of traditional failure atomic systems for disk, and they borrow many tech-
niques from the literature. Disk-based database systems have traditionally
used write-ahead logging to ensure consistent recoverability [148]. Proper
transactional updates to files in file systems can simplify complex and error-
prone procedures such as software upgrades. Transactional file updates have
been explored in research prototypes [163, 182] including some that explored
power-backed DRAM [138]; commercial implementations include Microsoft
Windows TxF [147] and Hewlett Packard Enterprise AdvFS [192]. Trans-
actional file update is readily implementable atop more general operating
systems transactions, which offer additional security advantages and support
scenarios including on-the-fly software upgrades [167]. At the opposite end
of the spectrum, user-space implementations of persistent heaps supporting
failure-atomic updates have been explored in research [209] and incorporated
into commercial products [13]. Logging-based mechanisms in general ensure
consistency by discarding changes from an update interrupted by failure. In
contrast, for idempotent updates, the update cut short by failure can sim-
ply be re-executed rather than discarding changes, reducing required logging
2.3.2 Persistent Data Structures

Persistent storage of data requires the use of some sort of data structure tuned to the performance characteristics of NVM. Persistent data structures provide a means of organizing and protecting durable data.

Consistent and durable data structures (CDDSs) \cite{191} are a style of persistent data structures that use versioning to ensure that updates to the data structure are failure atomic. Updates to the data structure do not change any part of the current structure, but, once all new parts of the structure have been made persistent, the change is committed by incrementing a version number. In this sense, CDDSs are quite similar to Driscoll et al.’s history-preserving data structures \cite{44} (called, confusingly, persistent data structures) which keep a record of all states of the data structure across its entire history. Venkataraman et al. \cite{191} report that CDDSs are quite usable as the main data structures for key-value stores; the authors were able to significantly increase the performance of the Redis NoSQL system using a backing CDDS tree.

NV–trees \cite{207} are another persistent structure which leverage the CDDS work to build even higher performing persistent structures with failure atomicity. Like CDDS trees, NV–trees are updated without modifying the old structure; changes are atomically appended. The key insight of NV–trees
is that persistently stored data does not need to be perfectly sorted within each leaf; we can keep data in unsorted array “bags” at each leaf and use volatile memory, if necessary, to index into the bag. This unordered allows persistent updates to progress quickly as they simply append into the bag array. NV–trees support concurrent updates using multi–version concurrency control (MVCC).

Beyond these early examples, there is growing work in building other concurrent data structures for nonvolatile memory, including hash maps [178], trees [29] [159], and transactional key-value stores [111] [124].

Guerra et al.’s Software Persistent Memory [63] takes a different approach to persistent data structures which does not leverage NVM but rather traditional disk. Persistent data structures are stored on designated “persistent” pages in the application’s virtual memory space. Their library uses strong typing to trace the closure of the pointer-based persistent data structure from its root. When a durability sync is issued, the library moves any data reachable through the persistent data structure to the persistent segment of virtual memory, then flushes any dirty lines back to disk.

Persistent data structures appear to have much in common with concurrent data structures. A common technique for reasoning about persistent data structures is the idea of a recovery thread [164] which is constantly reading the state of persistent memory. The recovery thread reading inconsistent state is equivalent to a poorly timed crash which leaves persistent memory inconsistent. As noted by Nawab et al. [152], this requirement is very nearly
met by standard nonblocking data structures, which, persistence aside, can always be read in a consistent manner by an accessing thread. The trick, of course, is to correctly translate the volatile structure to a persistent one. We return to this idea in Chapter 3.

2.3.3 File Systems

Perhaps the most obvious application of NVM is to host the file system, thereby improving performance using a faster underlying technology. However, unlike previous disk-based file systems, which had to be managed at block granularity, NVM file systems support finer grained access, and can consequently be redesigned to more appropriately leverage NVM storage.

BPFS [34] is the first file system designed explicitly for PCM with a volatile cache. The file system resides entirely in NVM, and relies on epoch ordering of writes with eight byte failure atomicity. In many ways, the file system resembles a persistent tree data structure. Every file is a tree consisting of 4 KB blocks. All data is at the leaf nodes, and every file’s tree is of uniform height. File sizes are stored in the interior nodes, thus specifying which lower nodes are valid. Directory files are simply a mapping between names and inode indexes.

BPFS stores all inodes in a unique *inode file* which is laid out as a single array. An inode contains a pointer to its file’s location in memory. BFPS’s tree structure enables it to support atomic updates to files in several non–
CHAPTER 2. BACKGROUND

traditional ways. In a partial copy-on-write, an operation creates a modified copy of a file or block, then atomically modifies the file system using a pointer swing. In an in-place update, updates smaller than eight bytes can rely on hardware failure atomicity to ensure consistency. Finally, an in-place append appends to a file without moving the original file, then commits the write by incrementing the file size variable.

PMFS [45] is a similar file system designed for NVM. PMFS expands upon the earlier work and explores some design trade-offs. Its layout is similar to that of BPFS, including the tree layout and single inode array file, but it, uses larger blocks that map to the operating system’s page size, simplifying (and consequently requiring) memory mapped access to the files. PMFS also provides an undo log journaling system for metadata updates, reducing the possibly large copy-on-write operations necessitated by BPFS. Finally, the work discusses protection of the file system. All file system metadata is memory mapped by the kernel in read-only mode, protecting it from stray writes from drivers. The region is temporarily upgraded to writable only when a metadata update is required by a system call, and downgraded immediately after. The existing privilege level system prevents user programs from accessing file system metadata, and the paging system prevents unauthorized access to unshared files from concurrent programs.

Shortcut-JFS [123], in contrast, is a file system designed for an NVM device with a traditional block interface. The file system provides two novel ideas. The first is to do differential logging of file modifications: journaling
writes at a finer granularity can reduce both wear and latency on NVM. The second idea is an in-place logging system. In contrast to traditional file systems, in which every append update is written twice (once to the journal, and once to the actual system), in-place journal writes append operations once, then adds the new journal block to the file using an atomic pointer swing. This scheme means that the journal becomes scattered around the file system, a problem for traditional HDD backed file systems, but a non-issue for NVM-backed ones.

2.3.4 Garbage Collection

For software written onto persistent memory, the issue of memory allocation and garbage collection becomes more complex. On loading a persistent memory segment, a persistent memory allocator must determine what memory is in use and which is free.

Memory allocation for volatile memory is traditionally done in two steps. First, the block is marked as occupied. Next, the block is made reachable—that is, some variable in either the stack or heap points to it. With persistent memory, an inopportune crash may come in between these steps, resulting in either a memory leak or a dangling pointer.

This problem is solved by leveraging either transactions or garbage collection techniques. Transactional systems, such as Mnemosyne [196], expect that the two steps are enclosed in a transaction. Alternatively, garbage col-
lection is done upon recovery and loading of the persistent segment, tracing from a designated root object and freeing unreachable memory. This option is used in more tightly managed libraries, such as a CDDS [191] or NV–Heaps [32], and generalized in Makalu [12].

2.4 NVM Software Applications

This final background section discusses applications which could benefit from the use of NVM.

2.4.1 Databases

Databases are an obvious target for NVM technology. Databases are already expected to deliver high performance durable storage, yet are in general optimized to use disk as the backing store. The use of NVM is likely to improve the performance of database management systems (DBMSs) by reducing the overhead of persistent storage and allowing for smaller changes to persistent state. Not surprisingly, existing databases are optimized to avoid costly disk I/O: porting them directly to NVM exposes other inefficiencies incurred due to this avoidance [6, 22, 38, 39]. Even if not using NVM for durable storage, the different access latencies between DRAM and NVM can cause DBMSs to underperform [28].

In particular, certain areas of DBMS development are likely to be im-
proved by the use of NVM. The database log records transactions on the data and is modified for nearly every update to the database. As this log is kept in durable storage, it makes sense to move it to NVM. The buffer cache is used to keep frequently used data in memory to reduce access latency, at the loss of durability, which now must be carefully managed with the help of the log. A persistent buffer cache would eliminate the persistence overhead of one stored in volatile DRAM for small transactions. In-memory databases are also common; they store most of their data in memory instead of on disk, and can thus optimize their structures for random access. NVM databases could leverage these techniques to provide faster software in the future.

**Database Background**

Modern DBMS designs fall generally into two major categories, each with their own utility. The older, more established category is that of relational database systems, which enforce full ACID semantics and the relational algebra of Codd [33]. These databases provide reliability and consistency guarantees suitable for mission-critical data. The more recent category is that of the NoSQL database. These databases tend to have more relaxed semantics and a simplified interface, often corresponding to an enhanced key-value store. NoSQL databases are useful for very large datasets in which data consistency is not of particularly great concern; for instance, machine learning data collections or large read-only sets.
CHAPTER 2. BACKGROUND

Transaction Logs

Database transaction logs, like journals from file systems or logs from transactional memory, are used to enforce atomicity and consistency of database transactions. Logs are a necessity for relational databases. Depending on the strength or weakness of the consistency guarantee of a NoSQL database they may or may not be present.

Relational databases generally use two logs. The first, the archival log, is used as a backup for disk media failure. It records all transactions since the last off-site backup. The other, the temporary log, is used to provide ACID semantics via undo and/or redo logging. Relational databases, due to their disk-oriented design, often use both undo and redo logging in a checkpointing scheme. The database is periodically synchronized between volatile working memory (the buffer cache) and disk in a checkpoint operation. On recovery after a machine crash, transactions that completed after the checkpoint but before the crash are redone, whereas transactions that were interrupted are undone [55].

What is stored in the log can vary from system to system. Physical logging stores a copy of the modified page or a difference entry. In contrast, logical logging stores the operation enacted on the object (effectively boosting). Logical logging, compared to physical logging, reduces logging overhead but makes recovery more complicated and may impose ordering constraints on page eviction from the buffer cache [66]. Combinations of the strategies (physiological logging) seem to provide the best performance—a logical undo
log reduces logging overhead during a transaction, but physical redo logging ensures that no ordering constraints are necessary on page write back [148].

It is important to note that almost all database systems use both an undo and a redo log. This requirement arises because persistence of pages is mostly uncontrolled by the transactional system—to allow transactions to control persistence ordering would impose too much pressure on the buffer cache. Consequently, incomplete transactions may have already had their effects flushed to disk (requiring undo logging) and complete transactions may still reside in volatile main memory (requiring redo logging).

As noted above, transaction logs make a good target for storage in persistent memory. Indeed, exploration of this possibility has already been done for modern NVM [50, 194] and older battery backed DRAM systems (e.g. [17]).

Buffer Cache

The buffer cache is a key component of a database system; it manages the flow of pages between stable disk storage and working volatile memory. Like traditional hardware caches, the buffer cache is managed by an eviction policy (e.g. LRU or clock) and will try to prefetch pages. Unlike traditional caches, however, the buffer cache may be designed to consider persistence requirements. In a no–steal approach, in–progress transactions might “pin” a page to the cache, requiring it to remain in volatile storage. In a force approach, completed pages are always flushed to disk by the transaction before it issues its commit.
The majority of the buffer cache’s responsibilities are dictated simply by the idea that the entire database cannot fit in working memory. Such responsibilities are, of course, unaffected by the availability of NVM. However, the use of NVM will change the persistence requirements of the buffer cache. For instance, the overhead of a traditional “force” operation is significantly reduced: we simply need to mark the page, while still in memory, as durable. A “steal” (an eviction from the buffer cache) also has no effect on persistence.

Alternatively, we can view the CPU caches as effectively replacing the buffer cache, and NVM replacing disk. Viewed this light, CPU caches on an NVM system resemble a stealing, forceable buffer cache [67].

In-Memory Databases

In contrast to disk-resident databases, in-memory databases store the primary copy of their data in RAM. This does not mean, however, that they always ignore durability. Common design techniques going back to the late eighties used small nonvolatile logs to provide a recovery capability [47, 54].

The main memory assumption – that all database data can fit in memory – allows for a number of optimizations which are impractical in a disk resident DBMS. For example, the entire database resides in memory and consequently has no buffer pool [43, 103, 117, 125, 126, 127, 137, 161, 189]. In-memory databases customize their architecture for small random accesses to main memory; they thereby alleviate the performance impacts of slow block-addressed storage accesses and consequently outperform tradi-
tional disk-based architectures [183]. Since transactions are expected to be shorter, locking can be done at a larger granularity, reducing bookkeeping overhead. Indexes are often built differently than for disk, since data does not need to be spatially co-located to index entries for fast access. Pointers can be used freely to avoid duplicate storage of large data items. Sorting data, often a critical step towards ensuring high performance for a disk resident system, is generally unnecessary, since nonsequential access to RAM is still cheap compared to disk. In-memory databases, however, must still persist data to disk for durability. They typically employ a log-based design that writes recovery information to a persistent log [43, 125, 189], writes periodic snapshots to disk at fixed intervals [103, 170], or employs a combination of both [39, 48]. Regardless of the details, nearly all in-memory databases employ a two-copy design that maintains a transient copy in memory and a persistent copy on disk, usually in uncompressed form.

As noted in previous sections, durable main memory storage is vulnerable to additional errors that do not affect disk. Namely, it is more vulnerable to stray writes from buggy software, and is unprotected by RAID type redundancy systems. For power-backed DRAM, the hardware is reliant on an active system working correctly in the face of a crash, which could fail due to hardware malfunction or poor maintenance. These issues seem to have prohibited wide scale reliance on power backed DRAM systems for main database storage [54].
Databases for NVM

Recent research on in-memory databases has also investigated NVM-based durability. For online transactional processing (OLTP) engines not explicitly designed for NVM, NVM-aware logging [31, 50, 85, 198] and NVM-aware query processing [193] can significantly improve performance. Both DeBrabant et al. [38] and Arulraj et al. [6] explore different traditional database designs and how they can be adapted for architectures with NVM.

Other authors present databases designed for NVM from the ground up. Kimura’s FOEDUS [108] proposes dual paging, which keeps a mutable copy of data in DRAM and an immutable snapshot of the data in NVM. A decentralized logging scheme is designed to accommodate dual paging. The use of dual paging and logging makes FOEDUS susceptible to the overheads of log-based multi-copy designs.

Several authors organize their OLTP engines around a central persistent data structure. However, many of the systems that use a persistent data structure still use logs for transactional recovery or atomicity. Numerous authors build engines around custom NVM-adapted B-trees that support atomic and durable updates [29, 159, 165, 191, 207]. Similarly, Chatzistergiou et al. [26] adapt their persistent STM system to build a central AVL tree for their engine, and Oukid et al. [158, 160] organize their engine around persistent column dictionaries. Other authors use batched logging [165], in which log entries are persisted periodically in chunks.
2.4.2 Checkpointing

Another obvious use of NVM is to checkpoint computation. For high performance computing, periodically saving program state is essential to making progress, since large machines have a short mean time to failure (MTTF). Indeed, as machines and computations grow, checkpointing (inherently I/O limited) consumes a larger and larger portion of execution time [46]. Also, as mentioned in the previous section, checkpointing is a critical task in database management systems to ensure that the log does not grow to an unmanageable size, and must be done in a manner that interferes as little as possible with database operation.

Checkpointing techniques vary by system based on expected overhead and reliability concerns. We expect that some of these techniques will be more amenable to NVM than others, and that new techniques will be developed based on the finer granularity interface.

Checkpointing can be done at all levels of the software stack. Applications can manage their own checkpointing manually, though this approach requires application developers to be careful to save and restore all necessary state. Alternatively, user-level libraries can be used. These libraries generally only require applications to link to them, then handle saving the software state periodically. Similarly, the kernel can handle checkpointing, and, indeed, any operating system effectively checkpoints a process automatically during a context switch (though not to persistent storage). In a similar manner, the virtual machine’s hypervisor can handle checkpointing by saving the
entire system state. Finally, cache coherence based checkpointing schemes in hardware can maintain backups automatically. Note that in general, as we go down the software stack, the overhead to the developer lessens, but checkpoints can be less selective in what content they save [112].

The timing of checkpoints is called the checkpoint placement problem and the optimal solution depends on several factors. Obviously, we would like to minimize the size of the checkpoint, so it makes sense to time checkpoints when the working set is small. We would also like to minimize the impact on the program, so it also makes sense to place the checkpoint during periods of read–mostly access. Finally, depending on the expected failure rate, we should tune the checkpoint rate so as to not burden the program excessively.

Certain techniques are useful in reducing the overhead of checkpointing. For instance, we can use incremental checkpointing to only store the difference between checkpoints. Of course, incremental checkpointing requires more complicated recovery mechanisms, is more vulnerable to corruption, and assumes for performance that not all locations are updated within each interval. We can also be more specific in limiting the memory to checkpoint. For instance, unreachable memory is not necessary to checkpoint, and user-level libraries can specify memory that need not be saved (e.g., volatile indices or locations for which the next access is a write). Additionally, to limit the amount of disk I/O, checkpoints can be compressed. Staggering checkpoints across processors may also be useful in order to avoid saturating the I/O device [112].
In distributed systems, coordinating a checkpoint can be difficult, since we must ensure the checkpoint is consistent across all processors. Three basic styles exist for such checkpoints. Uncoordinated checkpointing allows each processor to checkpoint as it needs to—necessitating a more complex recovery which rolls each processor backward until a consistent state is found. Unfortunately, there is no bound on this rollback—we may need to restart the program, a problem called the \textit{domino effect} \[169\]. Alternatively, in a coordinated checkpointing strategy, processors can coordinate via logical clocks or wall clock based methods to ensure that all checkpoints of a given epoch are consistent. Finally, checkpointing can be uncoordinated with a log based strategy. This strategy, called message logging, records every message the processor received or sent, depending on the protocol and its desired resilience, while processors checkpoint themselves as necessary. Message logging can be pessimistic (record every message before handling), optimistic (handle message while recording reception), or casual (the sender and receiver store messages when convenient or when checkpointing) \[46\].

Database checkpointing imposes additional constraints in that processors running the DBMS are expected to maintain high availability and transactional semantics. Consequently, checkpointing for databases requires coordination with the transaction log. In the simplest case, checkpointing should occur when there are no active writing transactions, allowing the buffer cache to write back all modified pages to disk. However, such a constraint is impractical for a highly availability database. Fuzzy checkpointing is a strategy
that spans transactions and writes pages back to storage when possible over a longer period, as necessary recording dirty pages in the also persistent undo log [55].
Chapter 3

Durable Linearizability

3.1 Introduction

When pairing NVM main memory with volatile registers and caches, ensuring a consistent state in the wake of a power outage requires special care in ordering updates to NVM. Several groups have designed data structures that tolerate power failures (e.g., [191, 207]), but the semantics of these structures are typically specified informally; the criteria according to which they are correct remain unclear. This chapter provides a novel correctness condition for machines with nonvolatile memory, and demonstrates that the condition is satisfied by a universal transform on existing nonblocking data structures.

---

1This chapter is based on the previously published papers by Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott: Linearizability of persistent memory objects under a full-system-crash failure model. In: DISC ’16 [95]; and Brief announcement: Preserving happens-before in persistent memory. In: SPAA’16 [94].
In prior proposals for correctness, Guerraoui and Levy have proposed persistent atomicity \cite{guerraoui1997linearizable} (a.k.a. persistent linearizability \cite{berryhill2006reliable}) as a safety condition for persistent concurrent objects. This condition ensures that the state of an object will be consistent in the wake of a crash, but it does not provide locality: correct histories of separate objects, when merged, will not necessarily yield a correct composite history. Berryhill et al. have proposed an alternative, recoverable linearizability \cite{berryhill2006reliable}, which achieves locality but may sacrifice program order after a crash. Earlier work by Aguilera and Frølund proposed strict linearizability \cite{aguilera1998efficient}, which preserves both locality and program order but provably precludes the implementation of some wait-free objects for certain (limited) machine models. The key differences among these safety conditions (illustrated in Figure 3.1) concern the deadlines for linearization \cite{joyeux2003linearization} of operations interrupted by a crash.

Interestingly, both the lack of locality in persistent atomicity and the loss of program order in recoverable linearizability stem from the assumption that an individual abstract thread may crash, recover, and then continue execution. While well defined, this failure model is more general than is normally assumed in real-world systems. More commonly, processes are assumed to fail together, as part of a “full system” crash. A data structure that survives such a crash may safely assume that subsequent accesses will be performed by different threads. We observe that if we consider only full-system crashes (an assumption modeled as a well-formedness constraint on histories), then persistent atomicity and recoverable linearizability are indistinguishable (and
Figure 3.1: Linearization bounds for interrupted operations under a thread reuse failure model. Displayed is a concurrent abstract (operation-level) history of two threads ($T_1$ and $T_2$) on two objects ($O_1$ and $O_2$); linearization points are shown as circles. These correctness conditions differ in the deadline for linearization for a pending operation interrupted by a crash ($T_1$’s first operation). 

- **Strict linearizability** requires that the pending operation linearizes or aborts as of the crash.
- **Persistent atomicity** requires that the operation linearizes or aborts before any subsequent invocation by the pending thread on any object.
- **Recoverable linearizability** requires that the operation linearizes or aborts before any subsequent linearization by the pending thread on that same object; under this condition a thread may have more than one operation pending at a time.

$O_2$ demonstrates the non-locality of persistent atomicity; $T_2$ demonstrates a program order inversion under recoverable linearizability.

thus local). They are also satisfied by existing persistent structures. We use the term **durable linearizability** to refer to this merged safety condition under the restricted failure model.

Independent of failure model, existing theoretical work typically requires that operations become persistent before they return to their caller. In practice, this requirement is likely to impose unacceptable overhead, since persistent memory, while dramatically faster than disk or flash storage, still incurs latencies of hundreds of cycles. To address the latency problem, we introduce **buffered durable linearizability**, which requires only that an operation be “persistently ordered” before it returns. State in the wake of a crash is
still required to be consistent, but it need not necessarily be fully up-to-date. Data structures designed with buffering in mind will typically provide an explicit sync method that guarantees, upon its return, that all previously ordered operations have reached persistent memory; an application thread might invoke this method before performing I/O. Unlike its unbuffered variant, buffered durable linearizability is not local: a history may fail to be buffered durably linearizable even if all of its object subhistories are. If the buffering mechanism is shared across all objects, however, an implementation can ensure that all realizable histories—those that actually emerge from the implementation—will indeed be buffered durably linearizable: the post-crash states of all objects will be mutually consistent.

At the implementation level, prior work has explored the memory persistency model (analogous to a traditional consistency model) that governs instructions used to push the contents of cache to NVM. Existing persistency models assume that hardware will track dependencies and automatically write dirty cache lines back to NVM as necessary [34, 102, 164]. Unfortunately, real-world ISAs require the programmer to request writes-back explicitly [1, 86]. Furthermore, existing persistency models have been explored only for sequentially consistent (SC) [164] or total-store order (TSO) machines [34, 102]. At the same time, recent persistency models [102, 164] envision functionality not yet supported by commercial ISAs—namely, hardware buffering in an ordered queue of writes-back to persistent memory, allowing persistence fence (pfence) ordering instructions to complete without waiting
for confirmation from the physical memory device. To accommodate anticipated hardware, we introduce a memory persistency model, *explicit epoch persistency*, that is both buffered and fully relaxed (release consistent).

Just as traditional concurrent objects require not only safety but liveness, so too should persistent objects. We define two optional liveness conditions: First, an object designed for buffered durable linearizability may provide *non-blocking sync*, ensuring that calls to `sync` complete without blocking. Second, a nonblocking object may provide *bounded completion*, limiting the amount of work done after a crash prior to the completion (if any) of operations interrupted by the crash. As a liveness constraint, bounded completion contrasts with prior art which imposes safety constraints [3, 9, 64] on completion (see Figure 3.1).

We also present a simple transform that takes a data-race-free program (code that uses a set of data-race-free objects) designed for release consistency and generates an equivalent program in which the state persisted at a crash is guaranteed to represent a consistent cut across the happens-before order of the original program. When the original program comprises the implementation of a linearizable nonblocking concurrent object, extensions to this transform result in a buffered durably or durably linearizable object. (If the original program is blocking, additional machinery—e.g., undo logging—may be required. While we do not consider such machinery here, we note that it still requires consistency as a foundation.)

To enable reasoning about our correctness conditions, we extend the no-
tion of linearization points into persistent memory objects, and demonstrate how such persist points can be used to argue a given implementation is correct. We also consider optimizations (e.g. elimination) that may safely be excluded from persistence in order to improve performance.

Summarizing our contributions, we introduce durable linearizability as a (provably local) safety condition for persistent objects under a full-system crash failure model, and extend this condition to (non-local) buffered durable linearizability (Sec. 3.2). We also introduce explicit epoch persistency to explain the behavior of machines with fully relaxed persistent memory systems, while formalizing nonblocking sync and bounded completion as liveness properties for persistence (Sec. 3.3). Next we present automated transforms that convert any linearizable concurrent object into an equivalent (buffered) durably linearizable object, and also introduce persist points for persistent memory objects as a means of proving the correctness of other constructions (Sec. 3.4). We conclude in Sec. 3.5.

3.2 Abstract Models

An abstract history is a sequence of events, which can be: (1) invocations of an object method, (2) responses associated with invocations, and (3) system-wide crashes. We use $O.\text{inv}(m)_t(params)$ to denote the invocation of operation $m$ on object $O$, performed by thread $t$ with parameters $params$. Similarly, $O.\text{res}(m)_t(retvals)$ denotes the response of $m$ on $O$, again performed
by $t$, returning $\text{retvals}$. A crash is denoted by $C$.

Given a history $H$, we use $H[t]$ to denote the subhistory of $H$ containing all and only the events performed by thread $t$. Similarly, $H[O]$ denotes the subhistory containing all and only the events performed on object $O$, plus crash events. We use $C_i$ to denote the $i$-th crash event, and $\text{ops}(H)$ to denote the subhistory containing all events other than crashes. The crash events partition a history as $H = E_0 C_1 E_1 C_2 \ldots E_{c-1} C_c E_c$, where $c$ is the number of crash events in $H$. Note that $\text{ops}(E_i) = E_i$ for all $0 \leq i \leq c$. We call the subhistory $E_i$ the $i$-th era of $H$.

Given a history $H = H_1 A H_2 B H_3$, where $A$ and $B$ are events, we say that $A$ precedes $B$ (resp. $B$ succeeds $A$). For any invocation $I = O.\text{inv}\langle m \rangle_t(\text{params})$ in $H$, the first $R = O.\text{res}\langle m \rangle_t(\text{retvals})$ (if any) that succeeds $I$ in $H$ is called a matching response. A history $S$ is sequential if $S = I_0 R_0 \ldots I_x R_x$ or $S = I_0 R_0 \ldots I_x R_x I_{x+1}$, for $x \geq 0$, and $\forall 0 \leq i \leq x, R_i$ is a matching response for $I_i$.

**Definition 1 (Abstract Well-Formedness).** An abstract history $H$ is said to be well formed if and only if $H[t]$ is sequential for every thread $t$.

Note that sequential histories contain no crash events, so the events of a given thread are confined to a single era. (In practice, thread IDs may be re-used as soon as operations of the previous era have completed. In particular, an object with bounded completion [Sec. 3.3.3, Def. 10 can rapidly reuse IDs.)

We consider only well-formed abstract histories. A completed operation in $H$ is any pair $(I, R)$ of invocation $I$ and matching response $R$. A pending
operation in $\mathcal{H}$ is any pair $(I, \perp)$ where $I$ has no matching response in $\mathcal{H}$. In this case, $I$ is called a pending invocation in $\mathcal{H}$, and any response $R$ such that $(I, R)$ is a completed operation in $\text{ops}(\mathcal{H})$ $R$ is called a completing response for $\mathcal{H}$.

**Definition 2** (Abstract Happens-Before). In any (well-formed) abstract history $\mathcal{H}$ containing events $E_1$ and $E_2$, we say that $E_1$ happens before $E_2$ (denoted $E_1 \prec E_2$) if $E_1$ precedes $E_2$ in $\mathcal{H}$ and (1) $E_1$ is a crash, (2) $E_2$ is a crash, (3) $E_1$ is a response and $E_2$ is an invocation, or (4) there exists an event $\hat{E}$ such that $E_1 \prec \hat{E} \prec E_2$. We extend the order to operations: $(I_1, R_1) \prec (I_2, x)$ if and only if $R_1 \prec I_2$.

Two histories $\mathcal{H}$ and $\mathcal{H}'$ are said to be equivalent if $\mathcal{H}[t] = \mathcal{H}'[t]$ for every thread $t$. We use $\text{compl}(\mathcal{H})$ to denote the set of histories that can be generated from $\mathcal{H}$ by appending zero or more completing responses, and $\text{trunc}(\mathcal{H})$ to denote the set of histories that can be generated from $\mathcal{H}$ by removing zero or more pending invocations. As is standard, a history $\mathcal{H}$ is linearizable if it is well formed, it has no crash events, and there exists some history $\mathcal{H}' \in \text{trunc}(\text{compl}(\mathcal{H}))$ and some legal sequential history $S$ equivalent to $\mathcal{H}'$ such that $\forall E_1, E_2 \in \mathcal{H}' [E_1 \prec_{\mathcal{H}'} E_2 \Rightarrow E_1 \prec_S E_2]$.

**Definition 3** (Durable Linearizability). An abstract history $\mathcal{H}$ is said to be durably linearizable if it is well formed and $\text{ops}(\mathcal{H})$ is linearizable.

Durable linearizability captures the idea that operations become persistent before they return; that is, if a crash happens, all previously completed
operations remain completed, with their effects visible. Operations that have
not completed as of a crash may or may not be completed in some subsequent
era. Intuitively, their effects may be visible simply because they “executed far
enough” prior to the crash (despite the lack of a response), or because threads
in subsequent eras finished their execution for them (for instance, after scan-
ning an “announcement array” in the style of universal constructions \[75\]).
While this approach is simple, it preserves important properties from lin-
earizability, namely \textit{locality} (composability) and \textit{nonblocking progress}.

**Lemma 1** (Locality). Any well-formed abstract history $H$ is durably linear-
able if and only if $H[O]$ is durably linearizable for every object $O$ in $H$.

\textit{Proof.} ($\Rightarrow$) If $H$ is durably linearizable, then $\text{ops}(H)$ is linearizable, and
then $\text{ops}(H[O])$ is linearizable for any object $O$. Therefore, $H[O]$ is durably linearizable, for any object $O$, by definition.

($\Leftarrow$) Fixing an arbitrary object $O$, since $H[O]$ is durably linearizable,
we have that $\text{ops}(H[O])$ is linearizable. Hence, $\text{ops}(H)$ is linearizable, and
therefore $H$ is durably linearizable. \hfill $\square$

**Lemma 2** (Nonblocking). If a history $H$ is durably linearizable and has a
pending operation $I$ in its final era, then there exists a completing response
$R$ for $I$ such that $HR$ is durably linearizable.

\textit{Proof.} For any durably linearizable history $H$, there is a sequential history $S$
equivalent to some history $H' \in \text{trunc}(\text{compl}(\text{ops}(H)))$. If $I$ has a matching
response $R$ in $S$, then $H' \in \text{trunc}(\text{compl}(\text{ops}(HR)))$, so $HR$ must be durably
linearizable. If \( I \) is still pending in \( S \), it must (by definition of sequential) be the final event and, since \( O \)'s methods are total, there must exist an \( R \) such that \( SR \) is legal and thus equivalent to \( H'R \). Otherwise \( I \) is not in \( S \) or \( H' \).

In this case (again, since \( O \)'s methods are total), there exists an \( R \) such that \( SIR \) is equivalent to some \( H'' \in \text{trunc}(\text{compl}(\text{ops}(HR))) \).

Given a history \( H \) and any transitive order \( < \) on events of \( H \), a \( \prec \)-consistent cut of \( H \) is a subhistory \( P \) of \( H \) where if \( E \in P \) and \( E' < E \) in \( H \), then \( E' \in P \) and \( E' < E \) in \( P \). In abstract histories, we are often interested in cuts consistent with \( \prec \), the happens-before order on events.

**Definition 4** (Buffered Durable Linearizability). A history \( H \) with \( c \) crash events is said to be buffered durably linearizable if it is well formed and there exist subhistories \( P_0, \ldots, P_{c-1} \) such that for all \( 0 \leq i \leq c \), \( P_i \) is a \( \prec \)-consistent cut of \( E_i \), and \( P = P_0 \ldots P_{i-1} E_i \) is linearizable.

The intent here is that events in the portion of \( E_i \) after \( P_i \) were buffered but failed to persist before the crash. Note that since \( P_i = E_i \) is a valid \( \prec \)-consistent cut for all \( 0 \leq i < c \), we can have \( P = \text{ops}(H) \), and therefore any durably linearizable history is buffered durably linearizable. Note also that buffered durable linearizability is not in general local: if an operation does not persist before it returns, we will not in general be able to ensure that it persists before any operation that follows it in happens-before order unless we arrange for the implementations of separate objects to cooperate.
3.3 Concrete Models

Concurrent objects are typically implemented by code in some computer language. We want to know if this code is correct. Following standard practice, we model implementation behavior as a set of concrete histories, generated under some language and machine model assumed to be specified elsewhere. Each concrete history consists of a sequence of events, including not only operation invocations, responses, and crash events, but also load, store, and read-modify-write (RMW—e.g., compare-and-swap [CAS]) events, which access the representation of an object. Let $x.\text{ld}_t(v)$ denote a load of variable $x$ by thread $t$, returning the value $v$. Let $x.\text{st}_t(v)$ denote a store of $v$ to $x$ by $t$. We treat RMW events as atomic pairs of special loads and stores (further details below). We refer to the loads, stores, and RMW events as memory events.

Given a concrete history $\mathcal{H}$, the abstract history of $\mathcal{H}$, denoted $\text{abstract}(\mathcal{H})$, is obtained by eliding all events other than invocations, responses, and crashes. As in abstract histories, we use $\mathcal{H}[t]$ and $\mathcal{H}[O]$ to denote the thread and object subhistories of $\mathcal{H}$. The concept of era from Sec. 3.2 applies verbatim. We say that an event $E$ lies between events $A$ and $B$ in a concrete or abstract history $\mathcal{H}$ if $A$ precedes $E$ and $E$ precedes $B$ in $\mathcal{H}$.

**Definition 5** (Concrete Well-Formedness). A concrete history $\mathcal{H}$ is well-formed if and only if

1. $\text{abstract}(\mathcal{H})$ is well-formed.
2. In each thread subhistory of $\mathcal{H}$, each memory event either (a) lies between some invocation and its matching response; (b) lies between a pending invocation $I$ and the first crash that succeeds $I$ in $\mathcal{H}$ (if such a crash exists); or (c) succeeds a pending invocation $I$ if no crash succeeds $I$ in $\mathcal{H}$.

3. The values returned by the loads and RMWs respect the reads-see-writes relation (Def. 7 below).

### 3.3.1 Basic Memory Model

For the sake of generality, we build our reads-see-writes relation on the highly relaxed release consistency memory model \[57\]. We allow certain loads to be labeled as load-acquire ($ld_{acq}$) events and certain stores to be labeled as store-release ($st_{rel}$) events. We treat RMW events as atomic ($ld_{acq}, st_{rel}$) pairs.

**Definition 6** (Concrete Happens-Before). Given events $E_1$ and $E_2$ of concrete history $\mathcal{H}$, we say that $E_1$ is sequenced-before $E_2$ if $E_1$ precedes $E_2$ in $\mathcal{H}[t]$ for some thread $t$ and (a) $E_1$ is a $ld_{acq}$, (b) $E_2$ is a $st_{rel}$, or (c) $E_1$ and $E_2$ access the same location. We say that $E_1$ synchronizes-with $E_2$ if $E_2 = x. ld_{acq}(v)$ and $E_1$ is the closest preceding $x. st_{rel}(v)$ in history order. The happens-before partial order on events in $\mathcal{H}$ is the transitive closure of sequenced-before order with synchronizes-with order. As in abstract histories, we write $E_1 \prec E_2$. 
Note that the definitions of happens-before are different for concrete and abstract histories; which one is meant in a given case should be clear from context.

The release-consistent model corresponds closely to that of the ARM v8 instruction set \cite{1} and can be considered a generalization of Intel's x86 instruction set \cite{86}, where \texttt{st\_rel} is emulated by an ordinary \texttt{st}, and where \texttt{ld\_acq} is emulated with \langle\texttt{mfence;ld}\rangle to force ordering with respect to any previous stores that serve as \texttt{st\_rel}. Given concrete happens-before, we can define the reads-see-writes relation:

\textbf{Definition 7} (Reads-See-Writes). A concrete history $\mathcal{H}$ respects the reads-see-writes relation if for each load $R \in \{x.\texttt{ld}(v), x.\texttt{ld\_acq}(v)\}$, there exists a store $W \in \{x.\texttt{st}(v), x.\texttt{st\_rel}(v)\}$ such that either (1) $W \prec R$ and there exists no store $W'$ of $x$ such that $W \prec W' \prec R$ or (2) $W$ is unordered with respect to $R$ under happens-before.

For simplicity of exposition, we consider the initial value of each variable to have been specified by a store that happens before all other instructions in the history. We consider only well-formed concrete histories here. If case (2) in Def. 7 never occurs in a history $\mathcal{H}$, we say that $\mathcal{H}$ is \textit{data-race-free}.

\subsection{3.3.2 Extensions for Persistence}

The semantics of instructions controlling the ordering and timing under which cached values are pushed to persistent memory comprise a memory \textit{persis-}
Explicit Epoch Persistency

<table>
<thead>
<tr>
<th>Explicit Epoch Persistency</th>
<th>Intel x86 [86]</th>
<th>ARM v8 [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwb addr</td>
<td>CLWB addr</td>
<td>DC CVAC addr</td>
</tr>
<tr>
<td>pfence</td>
<td>SFENCE</td>
<td>DSB</td>
</tr>
<tr>
<td>psync</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Equivalent instruction sequences for explicit epoch persistency.

tency model [164]. Since any machine with bounded caches must sometimes evict and write back a line without program intervention, the principal challenge for designers of persistent objects is to ensure that a newer write does not persist before an older write (to some other location) when correctness after a crash requires the locations to be mutually consistent.

Under the epoch persistency model of Condit et al. [34] and Pelley et al. [164], writes-back to persistent memory (persist operations) are implicit—they do not appear in the program’s instruction stream. When ordering is required, a program can issue a special instruction (which we call a pfence) to force all of its earlier writes to persist before any subsequent writes. Periods between pfences in a given thread are known as epochs. As noted by Pelley et al. [164], it is possible for writes-back to be buffered. When necessary, a separate instruction (which we call psync) can be used to wait until the buffer has drained (as a program might, for example, before performing I/O).

Unfortunately, implicit write-back of persistent memory is difficult to implement in real hardware [34, 102, 164]. Instead, manufacturers have introduced explicit persistent write-back (pwb) instructions [1, 86]. These are typically implemented in an eager fashion: a pwb starts the write-back
process; a \texttt{psync} waits for the completion of all prior \texttt{pwb}s (under some appropriate definition of “prior”).

We generalize proposed implicit persistency models \cite{34,102,164} and real world (explicit) persistency ISAs \cite{1,86} to define our own, new model, which we call \textit{explicit epoch persistency}. Like real-world explicit ISAs, our persistency model requires programmers to use a \texttt{pwb} to force back data into persistence. Like other buffered models, we provide \texttt{pfence}, which ensures that all previous \texttt{pwb}s are ordered with respect to any subsequent \texttt{pwb}s, and \texttt{psync}, which waits until all previous \texttt{pwb}s have actually reached persistent memory. We assume that persists to a given location respect coherence: the programmer need never worry that a newly persisted value will later be overwritten by the write-back of some earlier value. Unlike prior art, which assumes sequential consistency \cite{164} or total store order \cite{34,102,111}, we integrate our instructions into a relaxed (release consistent) model. Table 3.1 summarizes the mapping of our persistence instructions to the x86 and ARM ISAs. Neither instruction set currently distinguishes between \texttt{pfence} and \texttt{psync}, though both may do so at some point in the future. For now, ordering requires that the current thread wait for values to reach persistence.

Returning to concrete histories, we use $x.\texttt{pwb}_t$ to denote a \texttt{pwb} of variable $x$ by thread $t$, $\texttt{pfence}_t$ to denote a \texttt{pfence} by thread $t$, and $\texttt{psync}_t$ to denote a \texttt{psync} by thread $t$. We amend our definition of concrete histories to include these \textit{persistence events}. We refer to any non-crash event of a concrete history as an \textit{instruction}. 
Definition 8 (Persist Ordering). Given events $E_1$ and $E_2$ of concrete history $\mathcal{H}$, with $E_1$ preceding $E_2$ in the same thread subhistory, we say that $E_1$ is persist-ordered before $E_2$, denoted $E_1 \prec E_2$, if

(a) $E_1 = \text{pwb}$ and $E_2 \in \{\text{pfence, psync}\}$;

(b) $E_1 \in \{\text{pfence, psync}\}$ and $E_2 \in \{\text{pwb, st, st_rel}\}$;

(c) $E_1, E_2 \in \{\text{st, st_rel, pwb}\}$, and $E_1$ and $E_2$ access the same location;

(d) $E_1 \in \{\text{ld, ld_acq}\}$, $E_2 = \text{pwb}$, and $E_1$ and $E_2$ access the same location;

or

(e) $E_1 = \text{ld_acq}$ and $E_2 \in \{\text{pfence, psync}\}$.

Finally, across threads, $E_1 \prec E_2$ if

(f) $E_1 = \text{st_rel}$, $E_2 = \text{ld_acq}$, and $E_1$ synchronizes with $E_2$.

To identify the values available after a crash, we extend the syntax of concrete histories to allow store events to be labeled as “persisted,” meaning that they will be available in subsequent eras if not overwritten. Persisted store labels introduce additional well-formedness constraints:

Definition 9 (Concrete Well-Formedness [augments Def. 5]). A concrete history $\mathcal{H}$ is well-formed if and only if it satisfies the properties of Def. 5 and

4. For each variable $x$, at most one store of $x$ is labeled as persisted in any given era. We say the $(x, 0)$-persisted store is the labeled store of
x in $E_0$, if there is one; otherwise it is the initialization store of $x$. For $i > 0$, we say the $(x, i)$-persisted store is the labeled store of $x$ in $E_i$, if there is one; otherwise it is the $(x, i - 1)$-persisted store.

5. For any $(x, i)$-persisted store $W$, there is no store $W'$ of $x$ and psync event $P$ such that $W \preceq W' \preceq P$.

6. For any $(x, i)$-persisted store $W$, there is no store $W'$ of $x$ and $(y, i)$-persisted store $S$ such that $W \preceq W' \preceq S$.

Note that implementations are not expected to explicitly label persisted stores. Rather, the labeling is a post-facto convention that allows us to explain the values returned by reads. The well-formedness rules (#6 in particular) ensure that persisted stores compose a $\preceq$-consistent cut of $H$. To allow loads to see persisted values in the wake of a crash, we augment the definition of happens-before to declare that the $(x, i)$-persisted store happens before all events of era $E_{i+1}$. Def. [7] then stands as originally written.

### 3.3.3 Liveness

With strict linearizability, no operation is left pending in the wake of a crash: either it has completed when execution resumes, or it never will. With persistent atomicity and recoverable linearizability, the time it may take to complete a pending operation $m$ in thread $t$ can be expressed in terms of execution steps in $t$'s reincarnation (see Figure [3.1]). With durable linearizability,
which admits no reincarnated threads, any bound on the time it may take
to complete $m$ must depend on other threads.

**Definition 10** (Bounded Completion). A durably linearizable implementa-
tion of object $O$ has bounded completion if, for each concrete history $\mathcal{H}$ of
$O$ that ends in a crash with an operation $m$ on $O$ still pending, there exists
a positive integer $k$ such that for all realizable extensions $\mathcal{H}'$ of $\mathcal{H}$ in which
some thread in some era of $\mathcal{H}' \setminus \mathcal{H}$ has executed at least $k$ instructions, either
(1) for all realizable extensions $\mathcal{H}''$ of $\mathcal{H}'$, $\mathcal{H}'' \setminus \text{inv} \langle m \rangle$ is buffered durably
linearizable or (2) for all realizable extensions $\mathcal{H}''$ of $\mathcal{H}'$, if there exists a
completed operation $n$ with $\text{inv} \langle n \rangle \in \mathcal{H}'' \setminus \mathcal{H}'$, then there exists a sequential
history $S$ equivalent to $\mathcal{H}''$ with $m \prec_S n$.

Informally: after some thread has executed $k$ post-crash instructions, $m$ has
completed if it ever will.

It is also desirable to discuss progress towards persistence. Under durable
linearizability, every operation persists before it responds, so any liveness
property (e.g., lock freedom) that holds for method invocations also holds
for persistence. Under buffered durable linearizability, the liveness of persist
ordering is subsumed in method invocations.

As noted in Sec. 3.1, data structures for buffered persistence will typically
need to provide a `sync` method that guarantees, upon its return, that all
previously ordered operations have reached persistent memory. If `sync` is
not rolled into operations, then buffering (and `sync`) need to be coordinated
across all mutually consistent objects, for the same reason that buffered durable linearizability is not a local property (Sec. 3.2). The existence of \texttt{sync} impacts the definition of buffered durable linearizability. In Def. \[\text{4}\] all abstract events that precede a \texttt{sync} instruction in their era must appear in \(\mathcal{P}\), the sequence of consistent cuts. For a set of nonblocking objects, it is desirable that the shared \texttt{sync} method be wait-free or at least obstruction free—a property we call \textit{nonblocking sync}. (As \texttt{sync} is shared, lock freedom doesn’t seem applicable.)

### 3.4 Implementations

Given our prior model definitions and correctness conditions, we present an automated transform that takes as input a concurrent multi-object program written for release consistency and transient memory, and turns it into an equivalent program for explicit epoch persistency. Rules \((\text{T1})\) through \((\text{T5})\) of our transform (below) preserve the happens-before ordering of the original concurrent program: in the event of a crash, the values present in persistent memory are guaranteed to represent a \(<\)-consistent cut of the pre-crash history. Additional rules \((\text{T6})\) through \((\text{T8})\) serve to preserve real-time ordering not captured by concrete-level happens-before but required for durable linearizability. The intuition behind our transform is that, for nonblocking concurrent objects, a cut across the happens-before ordering represents a valid static state of the object \[\text{152}\]. For blocking objects, additional recov-
Ch. 3. DURABLE LINEARIZABILITY

ery mechanisms (not discussed here) may be needed to move the cut if it interrupts a failure-atomic or critical section [24, 32, 90, 196].

The following rules serve to preserve happens-before ordering into persist-before ordering and introduce names for future discussion. Their key observation is that a thread \( t \) which issues a \( x.\text{st}._r(v) \) cannot atomically ensure the value’s persistence. Thus, the subsequent thread \( u \) which synchronizes-with \( x.\text{ld}._a(v) \) shares responsibility for \( x \)'s persistence.

(T1) Immediately after store \( S = x.\text{st}_t(v) \), write back the written value by issuing \( \text{pwb}^S = x.\text{pwb}_t \).

(T2) Immediately before store-release \( S = x.\text{st}._r(v) \), issue \( \text{pfence}^S \); immediately after \( S \), write back the written value by issuing \( \text{pwb}^S = x.\text{pwb}_t \).

(T3) Immediately after load-acquire \( L = x.\text{ld}._a(v) \), write back the loaded value by issuing \( \text{pwb}^L = x.\text{pwb}_t \), then issue \( \text{pfence}^L \).

(T4) Handle \text{CAS} \) instructions as atomic \( \langle L, S \rangle \) pairs, with \( L = x.\text{ld}._a(v) \) and \( S = x.\text{st}._r(v') \): immediately before \( \langle L, S \rangle \), issue \( \text{pfence}^S \); immediately after \( \langle L, S \rangle \), write back the (potentially modified) value by issuing \( \text{pwb}^{L,S} = x.\text{pwb}_t \), then issue \( \text{pfence}^L \). (Extensions for other RMW instructions are straightforward.)

(T5) Take no persistence action on loads.
3.4.1 Preserving Happens-Before

In the wake of a crash, the values present in persistent memory will reflect, by Def. 9, a consistent cut across the (partial) persist ordering ($\preceq$) of the preceding era. We wish to show that in any program created by our transform, they will also reflect a consistent cut across that era’s happens-before ordering ($\prec$). Mirroring condition 6 of concrete well-formedness (Def. 9), but with $\prec$ instead of $\preceq$, we have:

**Lemma 3.** Consider a concrete history $\mathcal{H}$ emerging from our transform. For any location $x$ and $(x, i)$-persisted store $A \in \mathcal{H}$, there exists no store $A'$ of $x$, location $y$, and $(y, i)$-persisted store $B \in \mathcal{H}$ such that $A \prec A' \prec B$.

**Proof.** We begin with an intermediate result, namely that for $C = x.st1_i(j)$, $D = y.st2_u(k)$, with $st1, st2 \in \{st, st\_rel\}$, $C \prec D \Rightarrow C \preceq D$. We write $\prec^{(a,\ldots,f)}$ to justify a persist-order statement based on orderings listed in Def. 8. The following cases are exhaustive:

1. If $t = u$ and $x = y$, we immediately have $C \prec^{(c)} D$.

2. If $t = u$ and $st2 = st\_rel$, $C \prec^{(c)} pwb^C \prec^{(a)} pfence^D \prec^{(b)} D$.

3. If $t = u$ but $x \neq y$ and $st2 \neq st\_rel$, it is easy to see that there must exist a $st\_rel$ $S$ (possibly $C$ itself) and $ld\_acq$ $L$ such that $C \prec [S \prec] L \prec D$ (otherwise we would not have $C \prec D$). Moreover these accesses must be sequenced in thread subhistory order. But then $C \prec^{(c)} pwb^C \prec^{(a)} pfence^L \prec^{(b)} D$. 
4. If \( t \neq u \), there must exist an \( S = z.\text{st\_rel}_t(p) \) (possibly \( C \) itself) and an \( L = w.\text{ld\_acq}_u(q) \) such that \( C \prec [S \prec] L \prec D \) (otherwise we would not have \( C \prec D \)). Here \( C \) and \( S \), if different, must be sequenced in thread subhistory order, as must \( L \) and \( D \). Now if \( C = S \), we have \( C \preceq^{(c)} \text{pwb}^C \preceq^{(f)} \ldots \preceq^{(f)} L \preceq^{(e)} \text{pfence}^L \preceq^{(b)} D \), where “…” represents a sequence that carries \( \prec \) through persist orderings (b), (e), and (f). If \( C \neq S \), we have \( C \preceq^{(c)} \text{pwb}^C \preceq^{(a)} \text{pfence}^S \preceq^{(b)} S \preceq^{(f)} \ldots \preceq^{(f)} L \preceq^{(e)} \text{pfence}^L \preceq^{(b)} D \).

Having shown our intermediate result, we observe that \( A \prec A' \prec B \) would imply \( A \preceq A' \preceq B \), a violation of condition 6 of concrete well-formedness.

### 3.4.2 From Linearizability to Durable Linearizability

Unfortunately, preservation of concrete happens-before is not enough to give us durable linearizability: we also need to preserve the “real-time” order of non-overlapping operations (Def. 3 clause 3) in different threads. (As in conventional linearizability, “real time” serves as a stand-in for forms of causality—e.g., loads and stores of variables outside of operations—that are not captured in our histories.)

For objects that are (non-buffered) durably linearizable, we simply need to ensure that each operation persists before it returns:

(T6) Immediately before \( O.\text{res}(m)_t \), issue a \text{psync}.
For buffered durably linearizable objects, we leave out the \texttt{psync} and instead introduce a shared global variable $G$:

(T7) Immediately before $O.\text{res}(m)_t$, issue a \texttt{pfence}, then issue $G.\text{st}_t\text{rel}_t(g)$, for some arbitrary fixed value $g$.

(T8) Immediately after $O.\text{inv}(m)_t$, issue $G.1\text{d_aqc}_t(g)$, for the same fixed value $g$, then issue a \texttt{pfence}.

To facilitate our proof of correctness, we introduce the notion of an \textit{effective history} for $\mathcal{H}$. This history leaves out both the crashes of $\mathcal{H}$ and, in each era, the suffix of each thread’s execution that fails to reach persistence before the crash. We can then prove (Lemma 4) that any effective history of a program emerging from our transform is itself a valid history of that program (and could have happened in the absence of crashes). Moreover (Lemma 5), the (crash-free) abstract history corresponding to the effective history is identical to some concatenation of $\prec$-consistent cuts of the eras of the (crash-laden) abstract history corresponding to $\mathcal{H}$. These two lemmas then support our main result (Theorem 1).

\textbf{Definition 11.} Consider a concrete history $\mathcal{H} = \mathcal{E}_0 \mathcal{C}_1 \mathcal{E}_1 \ldots \mathcal{E}_{c-1} \mathcal{C}_c \mathcal{E}_c$. For any thread $t$ and era $0 \leq i < c$, let $E^t_i$ be the last store in $\mathcal{E}_i[t]$ that either is a persisted store or happens before some persisted store in $\mathcal{E}_i$. Let $B^t_i$ be the last non-store instruction that succeeds $E^t_i$ in $\mathcal{E}_i[t]$, with no stores by $t$ in-between (or, if there is no such instruction, $E^t_i$ itself). For all $0 \leq j < c$, let $\mathcal{P}_j$ be the
subhistory of $E_j$ obtained by removing all persistence events, all “persisted” labels, and, for each $t$, all events that succeed $B^j_t$ in $E_j[t]$. Finally, let $D_i$ be $E_i$ with persistence events and “persisted” labels removed. The effective concrete history of $H$ at era $i$, denoted effective$_i(H)$, is the history $P_0 \ldots P_{i-1}D_i$.

**Lemma 4.** Consider a nonblocking, data-race-free program $P$, and the transformed program $P'$. For any realizable $c$-crash concrete history $H$ of $P'$, and any $0 \leq i \leq c$, effective$_i(H)$ is a realizable concrete history of the original program $P$.

**Proof.** We begin with an intermediate result. For all $0 \leq j < c$, let $Q_j$ be the subhistory of $E_j$ obtained by removing, for each $t$, all events that succeed $B^j_t$ in $E_j[t]$. (Unlike $P_j$, $Q_j$ preserves persistence events and “persisted” labels.) Let cfa$_i(H)$, the “crash-free analogue” of $H$ at era $i$, be the history $Q_0 \ldots Q_{i-1}E_i$, with all “persisted” labels but the last removed for each location. Proceeding by induction on $i$, we argue that cfa$_i(H)$ is a realizable concrete history of the transformed program $P'$.

The base case is trivial: $E_0$ has at most one “persisted” label for each location, and is a realizable concrete history of $P'$. Suppose now that cfa$_{i-1}(H)$ is a realizable concrete history of $P'$, and moreover that $\forall x$, the $(x, 0)$-persisted store in cfa$_{i-1}(H)$ (the only persisted store there is of $x$ in that crash-free history) is the $(x, i - 1)$-persisted store in $H$ (this is also true in the base case).

Consider cfa$_{i-1}(H) \cap E_i$. This is clearly a realizable concrete history of $P'$, and the persisted stores of its final era are clearly the same as those of
the final era in $H_i$. We would like to say the same of $\text{cfa}_i(H) = (\text{cfa}_{i-1}(H) \setminus E_{i-1}) Q_{i-1} \ E_i$. That is, informally, we would like to argue that $Q_{i-1}$ is an acceptable replacement for $E_{i-1}$.

Clearly each thread in $Q_{i-1}$ correctly executes the code of every object (given values read and arguments passed to methods), since we only delete suffixes of thread histories. We preserve in $Q_{i-1}$ the persisted writes of $E_{i-1}$, so reads in $E_i$ will see the same values if $E_i$ is preceded by $Q_{i-1}$ instead of $E_i C$. Any store that happens before a persisted write is also preserved in $Q_{i-1}$, by construction in the choice of $E_t^i$. The only remaining reason why the inductive hypothesis might not hold for $\text{cfa}_i(H)$ would be if the arguments passed to methods in $Q_{i-1}$ were not realizable due to a real-time dependence not captured by happens-before. This possibility, however, is precluded by construction in the choice of $B_t^i$, which arranges for a thread to execute as many non-store instructions (including responses) as possible beyond $E_t^i$.

Having shown our intermediate result, we now observe that $\text{effective}_i(H)$ is simply $\text{cfa}_i(H)$ with persistence events and the remaining “persisted” labels removed. Since $P'$ differs from $P$ only in the addition of persistence instructions (which have no impact on a crash-free history), and since “persisted” labels in histories are merely a syntactic convention to facilitate reasoning, $\text{effective}_i(H)$ is a realizable concrete history of the original program $P$.

Lemma 5. Consider a nonblocking, data-race-free program $P$, and the transformed program $P'$. For any realizable concrete history $H$ of $P'$, and any $0 \leq i \leq c$, the history $\text{abstract}(\text{effective}_i(H))$ is precisely $P_0^a \ldots P_{i-1}^a E_i^a$. 
\[\square\]
where $\mathcal{E}_i^\alpha$ is the $i$-th era of $\text{abstract}(\mathcal{H})$, and $\mathcal{P}_j^\alpha$ is a $\prec$-consistent cut of $\mathcal{E}_j^\alpha$, for any $j < i$.

Proof. Fix an arbitrary $0 \leq i < c$. Since the transform introduces either a psync or a pfence-ed access to a global variable before any response in $\mathcal{H}$, if $(a, R)$ precedes $(I, b)$ in $\mathcal{E}_j$, with $j < i$, then $I \in \text{effective}_j(\mathcal{H})$ implies $R \in \text{effective}_j(\mathcal{H})$. Hence, $\mathcal{P}_j^\alpha$ denotes a $\prec$-consistent cut of $\mathcal{E}_j^\alpha$ for any $j < i$. $\square$

**Theorem 1** (Buffered Durable Linearizability). If a nonblocking, data-race-free program $\mathbb{P}$ is linearizable, the transformed program $\mathbb{P}'$ is buffered durably linearizable.

Proof. Say that $\mathbb{P}$ is linearizable. If $\mathbb{P}'$ is not buffered durably linearizable, there must exist a realizable concrete history $\mathcal{H}$ of $\mathbb{P}'$ where either (1) $\mathcal{A} = \text{abstract}(\mathcal{H})$ is not well-formed; or (2) there exists no $\{\mathcal{P}_0 \ldots \mathcal{P}_{c-1}\}$, where $\mathcal{P}_j$ is an $\prec$-consistent cut of $\mathcal{E}_j$, for $0 \leq j < c$, and $\mathcal{P}(i) = \mathcal{P}_0 \ldots \mathcal{P}_{i-1}\mathcal{E}_i$ is linearizable for all $0 \leq i \leq c$. We assume well-formed concrete histories, so, since $\mathcal{H}$ is well-formed, $\text{abstract}(\mathcal{H})$ is also well-formed. Case (1) is therefore false.

Now say that case (2) is true. By Lemma 4 for all $0 \leq i \leq c$, $\text{effective}_i(\mathcal{H})$ is a concrete realizable history of $\mathbb{P}$, so $\text{abstract(\text{effective}_i(\mathcal{H}))}$ is a realizable abstract history of $\mathbb{P}$. By Lemma 5 for all $0 \leq i \leq c$, $\text{abstract(\text{effective}_i(\mathcal{H}))}$ is of the form of $\mathcal{P}(i)$, stated above. Since by as-
3.4.3 Transform Implications

In addition to the correctness properties of our automated transform, we can characterize other properties of the code it generates. For example, the transformed implementation of a nonblocking concurrent object requires no change to persistent state before relaunching threads—that is, it has a null recovery procedure. Moreover, any set of transformed objects will share a wait-free sync method (a single call to psync).

In each operation on a transient linearizable concurrent object, we can identify some instruction within as the operation’s announce point: once execution reaches the announce point, the operation may linearize without its thread taking additional steps. Wait-free linearizable objects sometimes have announce points that are not atomic with their linearization points. In most nonblocking objects, however, the announce point is the linearization point, a property we call unannounced. This property results in stronger correctness properties in the persistent version when the object is transformed. The result of the transform when applied to an object whose operations are unannounced is strictly linearizable. Perhaps surprisingly, our transform does not guarantee bounded completion, even on wait-free objects. Pending announced operations may be ignored for an arbitrary interval before
eventually being helped to completion \cite{81, Sec. 4.2.5].

### 3.4.4 Persist Points

Linearizability proofs for transient objects are commonly based on the notion of a *linearization point*—an instruction between an operation's invocation and response at which the operation appears to “take effect instantaneously” \cite{76}.

**Theorem 2** (Linearization Points [Herlihy & Wing, restated]). *Suppose, in every realizable effective concrete history $\mathcal{H}$ of object $O$, it is possible to identify, for each operation $m \in \mathcal{H}$, a linearization point instruction $l_m$ between $\text{inv}(m)$ and $\text{res}(m)$ such that $\mathcal{H}$ is equivalent to a sequential history that preserves the order of the linearization points. Then $O$ is linearizable.*

In simple objects, linearization points may be statically known. In more complicated cases, one may need to reason retrospectively over a history in order to identify the linearization points, and the linearization point of an operation need not necessarily be an instruction issued by the invoking thread.

The problem for persistent objects is that an operation cannot generally linearize and persist at the same instant. Clearly, it will need to linearize first; otherwise it will not know what values to persist. Unfortunately, as soon as an operation (call it $m$) linearizes, other operations on the same object can see its state, and might, naively, linearize and persist before $m$ had a
The key to avoiding this problem is for every operation $n$ to ensure that any predecessor on which it depends has persisted (in the unbuffered case) or persist-ordered (with global buffering) before $n$ itself linearizes. To preserve real-time order, $n$ must also persist (or persist-order) before it returns.

**Theorem 3** (Persist Points). Suppose that for each operation $m$ of object $O$ it is possible to identify not only a linearization point $l_m$ between $\text{inv}(m)$ and $\text{res}(m)$ but also a persist point instruction $p_m$ between $l_m$ and $\text{res}(m)$ such that (1) “all stores needed to capture $m$” are written back to persistent memory, and a $\text{pfence}$ issued, before $p_m$; and (2) whenever operations $m$ and $n$ overlap, linearization points can be chosen such that either $p_m \preceq l_n$ or $l_n$ precedes $l_m$. Then $O$ is (buffered) durably linearizable.

The notion of “all stores needed to capture $m$” will depend on the details of $O$. In simple cases (e.g., those emerging from our automated transform), those stores might be all of $m$’s updates to shared memory. In more optimized cases, they might be a proper subset (as discussed below). Generally, a nonblocking persistent object will embody helping: if an operation has linearized but not yet persisted, its successor operation must be prepared to push it to persistence.
3.4.5 Practical Applications

A variety of standard concurrent data structure techniques can be adapted to work with both durable and strict linearizability and their buffered variants. While our automated transform can be used to create correct persistent objects, judicious use of transient memory can often reduce the overhead of persistence without compromising correctness. For instance, announcement arrays [77] are a common idiom for wait-free helping mechanisms. Implementing a transient announcement array [9] while using our transform on the remainder of the object state will generally provide a (buffered) strictly linearizable persistent object.

Other data structure components may also be moved into transient memory. Elimination arrays [74] might be used on top of a durably or strictly linearizable data structure without compromising its correctness. The flat combining technique [73] is also amenable to persistence. Combined operations can be built together and ordered to persistence with a single `pfence`, then linked into the main data structure with another, reducing `pfence` instructions per operation. Other combining techniques (e.g., basket queues [82]) might work in a similar fashion. A transient combining array will generally result in a strictly linearizable object; leaving it persistent memory results in a durably linearizable object.

Several library and run-time systems have already been designed to take advantage of NVM; many of these can be categorized by the presented correctness conditions. Strictly linearizable examples include trees [191, 207].
file systems [34], and hash maps [178]. Buffered strictly linearizable data structures also exist [149], and some libraries explicitly enable their construction [15, 24]. Durably (but not strictly) linearizable data structures are a comparatively recent innovation [90].

3.5 Conclusion

This chapter has presented a framework for reasoning about the correctness of persistent data structures, based on two key assumptions: full-system crashes at the level of abstract histories and explicit write-back and buffering at the level of concrete histories. For the former, we capture safety as (buffered) durable linearizability; for the latter, we capture anticipated real-world hardware with explicit epoch consistency, and observe that both buffering and persistence introduce new issues of liveness. Finally, we have presented both an automatic mechanism to transform a transient concurrent object into a correct equivalent object for explicit epoch persistency and a notion of persist points to facilitate reasoning for other, more optimized, persistent objects.
Chapter 4

Composing Durable Data Structures

4.1 Introduction

Looking beyond individual objects, we should like to be able to compose operations on pre-existing durably linearizable objects into larger failure-atomic sections (i.e., transactions). Composing durable data structures would be useful as most published data structures for NVM meet the durable linearizability criteria [95]; that is, the object ensures that each of its methods, between its invocation and return, (1) becomes visible to other threads atomically and (2) reaches persistence in the same order that it became visible.

\footnote{This chapter is based on the previously published poster abstract by Joseph Izraelevitz, Virendra Marathe, and Michael L. Scott. \textit{Poster presentation: Composing durable data structures.} In: NVMW '17 [93].}
Published objects include trees \[191, 207\] and hash maps \[90, 178\].

Such composability might be seen as an extension of transactional boosting \[78\], which allows operations on linearizable data structures (at least those that meet certain interface criteria) to be treated as primitive operations within larger atomic transactions. In this chapter, we discuss additional interface requirements for durably linearizable data structures in order for them to be atomically composable. We also present a simple, universal, lock-free construction, which we call the *chronicle*, for building data structures that meet these requirements.

### 4.2 Composition

Composition is a hallmark of transactional systems, allowing a set of nested actions to have “all-or-nothing” semantics. The default implementation arranges for all operations to share a common log of writes (and reads, for transactions that provide isolation), which commit or abort together. Unfortunately, this implementation imposes overhead on every memory access, and leads to unnecessary serialization when operations that “should” commute cannot due to conflicting accesses to some individual memory location internally.

Boosting addresses both of these problems by allowing operations on black-box concurrent objects to serve as “primitives”—analogues of read and write—from the perspective of the transactional system. In a system based
on UNDO logs, memory updates are made “in place” and inverse operations are entered in an UNDO log. For a write, the inverse is a write of the previous value. For a higher-level operation, the inverse depends on the semantics of the object (a push’s inverse is a pop). In the event of a transaction abort, the log is played in reverse order, undoing both writes and higher level operations using their inverses. For concurrency control, semantic locks are used to prevent conflicts between operations that do not commute (e.g., puts to different keys commute, but puts to the same key do not; transactions that access disjoint sets of keys can run concurrently).

We aim to extend the boosting of linearizable objects in (transient) transactional memory so that it works for durably linearizable objects in persistent transactional memory. To do so, we must overcome a pair of challenges introduced by the possibility of crashes. First, transactional boosting implicitly assumes that a call to a boosted operation will return in bounded time, having linearized (appeared to happen instantaneously) sometime in between. While we can assume that a durably linearizable object will always be consistent in the wake of a crash (as if any interrupted operation had either completed or not started), we need for composition to be able to tell whether it has happened (so we know whether to undo or redo it as part of a larger operation). Second, transactional boosting implicitly assumes that we can use the return value of an operation to determine the proper undo operation. For composition in a durably linearizable system, we need to ensure that the return value has persisted—so that, for example, we know that the
inverse of $S.\text{pop}()$ is $S.\text{push}(\text{v})$, where $\text{v}$ is the value returned by the $\text{pop}$.

### 4.3 Query-Based Logging

Our method of durable boosting employs what we call “query-based logging,” a technique applicable to both UNDO and JUSTDO logging \[90\]. In our design, the boosted durable data structure is responsible for maintaining sufficient information about interrupted operations to ensure both that their inverses can be computed and that they are executed only once. An interrupted transaction can query the data structure after the crash using a unique ID to gather this information.

The query interface is designed as follows. All the normal exported methods of a boostable data structure take a unique ID for every invocation (e.g., a thread ID concatenated with a thread-local counter). There also exists a query method, which takes a unique ID as argument and returns either NULL, indicating that the operation never completed and never will, or a struct containing the operation’s invoked function, corresponding arguments, and return value.

Boosting using query-based UNDO logging is straightforward. The transaction is executed sequentially, and acquires the appropriate read, write, and semantic locks as needed. Before a boosted operation, we log our intended operation in the UNDO log. After the operation returns, we mark the operation completed in the UNDO log, and, if appropriate, record its return value.
If the operation is interrupted, we can use the query interface to determine if the operation completed and what its return value would be. Using this information, we can complete (or ignore) the UNDO entry, then roll back the transaction in reverse using the normal UNDO protocol and each operation’s inverse. JUSTDO logging works similarly, but rolls forward from the interrupted operation.

### 4.3.1 The Chronicle

To facilitate the use of query-based logging, we present a lock-free construction, called the *chronicle*, that creates a queryable, durably linearizable version of any data structure with the property that each method linearizes at one of a statically known set of compare-and-swap (CAS) instructions, each of which operates on a statically known location. This property is satisfied by, for example, any object emerging from Herlihy’s classic nonblocking constructions [77]. In our construction, each CAS-ed location is modified indirectly through a *State* object. Instead of using a CAS to modify the original location, an operation creates a new global State object and appends it to the previous version. By ensuring that all previous States have been written to persistent storage before appending the new State, we can ensure that all previous operations have linearized and persisted. By attaching all method call data to the State object associated with its linearization point, we can always determine the progress of any ongoing operation.
To demonstrate the utility of the chronicle, Fig. 4.1 presents a variant of the non-blocking Treiber stack [187]. Like the original, this version is linearizable. Unlike the original, it provides durable linearizability and a queryable interface. Figure 4.1 shows its implementation. While the version here flushes the entire chronicle on every operation, simple optimizations can be used to flush only the incremental updates and to garbage collect old entries.

4.4 Conclusion

In summary, this chapter has demonstrated that it is possible to compose durable data structures into larger failure-atomic sections, provided that they conform to our queryable interface. However, in general, durably linearizable data structures cannot be composed, since, on recovery, it may be unclear if an operation has completed (or not). Our queryable interface solves this problem, and our chronicle construction demonstrates that the interface can be met in a universal lock-free manner.


```cpp
class Node {
    Object val;
    // the stored object
    Node* down;
    // the next node down
};

class State {
    State* next;
    // the next State in
    // the chronicle
    Node* head;
    // the head Node
    int method;
    // method invoked
    int uid;
    // a unique id for op
    void* ret;
    // return value of op
};

class Stack {
    State* chronicle;
    Stack() { chronicle = new State(NULL, NULL, INIT, 0, NULL); }
};

State* Stack::flushChronicle(State* fromHereForward) {
    State* s = fromHereForward;
    while (s->next != NULL) {
        clflush(s);
        s = s->next;
    }
    State* realState = s;
    clflush(realState);
    // now chronicle is
    // entirely flushed
    return realState;
}

Object Stack::pop(int uid) {
    State* s = chronicle;
    while (true) {
        s = flushChronicle(s);
        Object x = s->head->val;
        Node n = s->head->down;
        s_new = new State(NULL, n, POP, uid, x);
        if (CAS(&s->next, NULL, s_new)) {
            clflush(s);
            // flush CAS to s->next
            return x;
        }
    }
}

int Stack::push(Object x, int uid) {
    State* s = chronicle;
    while (true) {
        s = flushChronicle(s);
        Node* n = new Node(x, s->head);
        clflush(n)
        s_new = new State(NULL, n, PUSH, uid, SUCCESS);
        if (CAS(&s->next, NULL, s_new)) {
            clflush(s);
            // flush change to s->next
            return SUCCESS;
        }
    }
}
```

---

Figure 4.1: Treiber Stack Chronicle Implementation
Chapter 5

Failure Atomicity via JUSTDO Logging

5.1 Introduction

Eliminating the memory/storage distinction using NVM promises to streamline software and improve performance, but direct in-place manipulation of persistent application data allows a failure during an update to corrupt data. Mechanisms supporting program-defined failure-atomic sections (FASEs) address this concern. Failure-atomicity systems that support FASEs can be implemented as transactional memory with additional durability guarantees as discussed in Section 2.3, or by leveraging applications’ use of mu-

\footnote{This chapter is based on the previously published paper by Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. \textit{Failure-atomic persistent memory updates via JUSTDO logging.} In: ASPLOS’16 [90].
tual exclusion primitives to infer consistent states of persistent memory and guarantee consistent recovery [24]. These prior systems offer generality and convenience by automatically maintaining undo [24, 32] or redo [196] logs that allow recovery to roll back FASEs that were interrupted by failure.

In this chapter, we introduce a new failure atomicity system called JUSTDO logging. Designed for machines with persistent caches and memory (but transient registers), JUSTDO logging significantly reduces the overhead of failure atomicity as compared to prior systems by reducing log size and management complexity.

Persistent CPU caches eliminate the need to flush caches to persistent memory and can be implemented in several ways, e.g., by using inherently non-volatile bit-storage devices in caches [211] or by maintaining sufficient standby power to flush caches to persistent memory in case of power failure. The amount of power required to perform such a flush is so small that it may be obtained from a supercapacitor [198] or even from the system power supply [151]. Preserving CPU cache contents in the face of detectable non-corrupting application software failures requires no special hardware: STORES to file-backed memory mappings persist beyond process crashes [152].

We target persistent cache machines in this chapter as the different NVM device technologies offer different read/write/endurance characteristics and are may be deployed accordingly in future systems. For example, while PCM and Memristor are mainly considered as candidates for main memory, STT-RAM can be expected to be used in caches [211]. Non-volatile caches imply
CHAPTER 5. FAILURE ATOMICITY VIA JUSTDO LOGGING

that stores become persistent upon leaving the CPU’s store buffers. Persistent caches can also be implemented by relying on stand-by power or employing supercapacitor-backed volatile caches to flush data from caches to persistent memory in the case of a failure. Recent technology trends indicate that non-volatile caches are a possibility in the near future, and some failure atomicity systems have already been designed for this machine model.

However even if persistent caches eliminate the cache flushing overheads of FASE mechanisms, the overhead of conventional undo or redo log management remains. A simple example illustrates the magnitude of the problem: Consider a multi-threaded program in which each thread uses a FASE to atomically update the entire contents of a long linked list. Persistent memory transaction systems would serialize the FASEs—in effect, each thread acquires a global lock on the list—and would furthermore maintain a log whose size is proportional to the list modifications. A mutex-based FASE mechanism for persistent memory avoids serializing FASEs by allowing concurrent updates via hand-over-hand locking but must still maintain per-thread logs, each proportional in size to the amount of modified list data.

The key insight behind our approach is that mutex-based critical sections are intended to execute to completion. While it is possible to implement rollback for lock-based FASEs, we might instead simply resume FASEs following failure and execute them to completion. This insight suggests a design that employs minimalist logging in the service of FASE resumption.
CHAPTER 5. FAILURE ATOMICITY VIA JUSTDO LOGGING

Figure 5.1: Two examples of lock-delimited FASEs. Left (lines 1–4): Nested. Right (lines 5–8): Hand-over-hand.

rather than rollback.

Our contribution, JUSTDO logging, unlike traditional UNDO and REDO logging, does not discard changes made during FASEs cut short by failure. Instead, our approach resumes execution of each interrupted FASE at its last store instruction then executes the FASE to completion. Each thread maintains a small log that records its most recent store within a FASE; the log contains the destination address of the store, the value to be placed at the destination, and the program counter. FASEs that employ JUSTDO logging access only persistent memory, which ensures that all data necessary for resuming an interrupted FASE will be available during recovery. As in the Atlas system [24], we define a FASE to be an outermost critical section protected by one or more mutexes; the first mutex acquired at the start of a FASE need not be the same as the last mutex released at the end of the FASE (see Figure 5.1). Auxiliary logs record threads’ mutex ownership for recovery.

Our approach has several benefits: By leveraging persistent CPU caches where available, we can eliminate cache flushing overheads. Furthermore the
small size of JUSTDO logs can dramatically reduce the space overheads and complexity of log management. By relying on mutexes rather than transactions for multi-threaded isolation, our approach supports high concurrency in scenarios such as the aforementioned list update example. Furthermore we enable fast parallel recovery of all FASEs that were interrupted by failure. JUSTDO logging can provide resilience against both power outages and non-corrupting software failures, with one important exception: Because we sacrifice the ability to roll back FASEs that were interrupted by failure, bugs within FASEs are not tolerated. Hardware and software technologies for fine-grained intra-process memory protection [30, 203] and for software quality assurance [20, 21, 60, 175, 210] complement our approach respectively by preventing arbitrary corruption and by eliminating bugs in FASEs.

In this chapter, we describe the design and implementation of JUSTDO logging and evaluate its correctness and performance. Our results show that JUSTDO logging provides a useful new way to implement persistent memory FASEs with improved performance compared with a state-of-the-art system: On five very different mutex-based concurrent data structures, JUSTDO logging increases operation throughput over 3× compared with crash resilience by the state-of-the-art Atlas FASE mechanism [24].

The remainder of this chapter is organized as follows: Section 5.2 presents key concepts and terminology. Section 5.3 presents our assumptions regarding the system on which JUSTDO logging runs and the programming model that our approach supports. Section 5.4 describes the design of JUSTDO
logging, and Section 5.5 presents the details of our current implementation. Section 5.6 evaluates the correctness and performance of our approach, and Section 5.7 concludes with a discussion.

5.2 Concepts & Terminology

Application data typically must satisfy application-level invariants or other correctness criteria. We say that data are consistent if the relevant application-level correctness criteria hold, otherwise the data are corrupt. Failures are events that may corrupt application data; familiar examples include application process crashes, operating system kernel panics, and abrupt power outages. We say that a failure is tolerated if application data consistency either is unaffected by the failure or is restored by post-failure recovery procedures. We distinguish between corrupting and non-corrupting failures; the former preclude successful recovery by corrupting application data directly or by corrupting data necessary for recovery (e.g., logs). A corrupting failure may be caused, for example, by a store through a pointer variable containing an invalid address.

We say that data are persistent if they survive tolerated failures intact and are accessible by recovery code, otherwise the data are transient. Similarly we say that memory locations, memory address ranges, processor cache lines, and other places where data may reside are persistent or transient depending on whether or not the data they contain will be available to recovery code
following any tolerated failure. For example, a *persistent memory region* is a contiguous range of virtual addresses whose contents will survive tolerated failures. Note that persistence does not imply consistency: Failure may render persistent data irreparably corrupt, making recovery impossible.

We reserve the term *non-volatile* for characterizing device technologies that retain data even in the absence of supplied power; examples include memristor, STT-RAM, and PCM. Similarly the term *volatile* characterizes device technologies such as DRAM that require continuously supplied power to retain data. We emphasize that our persistent/transient distinction is orthogonal to volatility. For example, while non-volatile memory (NVM) facilitates the implementation of memory that is persistent with respect to certain kinds of failure, persistent memory also admits alternative implementations that do not involve NVM. Moreover, NVM need not be persistent according to our definition: For example, if thread stacks on a particular computer are reclaimed by the operating system following abnormal process termination, then stack data are not available to recovery code and are therefore transient, even if every byte of memory on the machine is non-volatile.

We distinguish between *partial* and *whole-system* persistence. The latter applies when the entire state of a machine survives tolerated failures, whereas the former describes situations in which some data is persistent and some is transient. Partial persistence results when applications designate only some data as persistent (e.g., a persistent memory region containing long-term application data) and allow the remainder to be transient (e.g., per-thread
CHAPTER 5. FAILURE ATOMICITY VIA JUSTDO LOGGING

Figure 5.2: Hybrid architecture incorporating both conventional volatile CPU registers and DRAM in addition to non-volatile CPU caches and NVM.

function call stacks). Partial persistence is a natural match for future hybrid architectures that incorporate both volatile and non-volatile components, as depicted in Figure 5.2.

We conclude this section by using our definitions to briefly define our hardware and software system model, characterize the failures that JUSTDO logging can tolerate, and describe situations where our approach is likely to offer good performance; all of these topics are covered in greater detail in subsequent sections. JUSTDO logging is designed with future hybrid architectures in mind (Figure 5.2). More specifically, our system model (Section 5.3) and our design (Section 5.4) assume that CPU registers are transient but that both CPU caches and (part of) main memory are persistent, and our programming model assumes partial persistence. JUSTDO logging tolerates non-corrupting failures that were not caused by software bugs within a failure-atomic section. We expect JUSTDO logging to achieve good performance if it is inexpensive to impose ordering constraints on modifications to
persistent data—as would be the case with persistent caches and/or persistent store buffers integrated into the CPU in addition to persistent memory.

5.3 System Model & Programming Model

System Model Figure 5.2 illustrates our system model. As in prior work [139, 211], we consider a system in which both main memory and processor caches are persistent, i.e., their contents survive tolerated failures intact. We place no restrictions on how persistent memory and persistent caches are implemented. A tolerated failure on such a system causes all processor state to be lost but the contents of the caches and memory survive and are available upon recovery. We assume that power failures and non-corrupting fail-stop software failures have these consequences.

If caches are persistent, a store will become persistent once it reaches the coherence layer; release fences force the store into persistence. By comparison, on an x86 system with persistent memory but without persistent caches, stores can be pushed toward persistence using a CLFLUSH. On future Intel systems, new flushing instructions such as CLFLUSHOPT and CLWB will provide fine-grained control over persistence with lower overhead [86]. Flushing instructions will be used with SFENCE to constrain persistence ordering.
Programming Model  Our programming model shares much in common with several recent persistent memory programming approaches. Like NV-heaps \[32\], Mnemosyne \[196\], and Atlas \[24\], JUSTDO logging integrates an existing concurrency control technique with a mechanism for failure atomicity. Whereas NV-heaps and Mnemosyne extend transactional memory, Atlas and JUSTDO logging extend conventional mutex-based concurrency control.

The Atlas system, which we compare against in our evaluation (Section \[5.6\]), illustrates the tradeoffs among convenience, compatibility, generality, and performance that confront any implementation of FASEs. Atlas employs per-thread UNDO logging to ensure the atomicity of FASEs. An UNDO log entry is created for every STORE to persistent memory that is executed by a thread in a program. The log entry must be made persistent before the corresponding STORE can occur. Unlike the isolated transactions of NV-heaps and Mnemosyne, the outermost critical sections that constitute Atlas FASEs may be linked by dependencies: Sometimes an outermost critical section that has completed must nonetheless be rolled back during recovery. Reclaiming UNDO log entries no longer needed for recovery is therefore a non-trivial task in Atlas and is performed in parallel by a separate helper thread. Because dependencies between FASEs must be explicitly tracked, Atlas requires persistent memory updates to be synchronized via locks, which precludes the use of atomics (in the sense of C++11) in the current version of Atlas. Atlas emphasizes generality, programmer convenience, and compatibility with
conventional lock-based concurrency control; a sophisticated infrastructure is required to support these advantages, and the requisite UNDO logging and log pruning carry performance overheads. Specifically, the size of UNDO logs is proportional to the amount of data modified in the corresponding FASE, and the complexity of tracking dependencies for log reclamation can grow with the number of FASEs.

Like Atlas, in JUSTDO logging we expect that failure-atomic modifications to shared data in persistent memory are performed in critical sections delimited by lock acquisitions and releases: A thread that holds one or more locks may temporarily violate application-level consistency invariants, but all such invariants are restored before the thread releases its last lock [24]. Therefore data in persistent memory is consistent in the quiescent state in which no thread holds any locks, and we accordingly equate outermost critical sections with FASEs. Like all FASE implementations, JUSTDO logging guarantees that application data in persistent memory is restored to a consistent state following a failure. As with Mnemosyne and NV-heaps (but not Atlas), JUSTDO applications may safely emit output dependent on a FASE—e.g., acknowledging to a remote client that a transaction has completed—immediately after exiting the FASE. Unlike Atlas, Mnemosyne, and NV-Heaps, which require data-race-free semantics, JUSTDO logging supports unsynchronized read-write (but not write-write) races. Although all of these approaches to failure atomicity are designed for concurrency, all four may also be used in serial code by simply delimiting FASEs as in parallel software.
As with NV-heaps, Mnemosyne, and Atlas, our approach allows the programmer to specify explicitly which data is to be preserved across failure by placing it in persistent memory; such control is useful for, e.g., hybrid architectures that incorporate both DRAM and NVM. In such partially persistent systems that expose both persistent and transient memory to applications, JUSTDO logging requires that FASEs access only persistent memory.

Our current implementation of JUSTDO logging is a C++ library with bindings for C and C++. The library requires FASEs to reside in functions that consolidate boilerplate required for recovery and requires that stores occur via special JUSTDO library calls. Future compiler support could eliminate nearly all of the verbosity that our current prototype requires and could eliminate opportunities for incorrect usage.

Like the aforementioned prior implementations of persistent memory FASEs, JUSTDO logging maintains logs crucial to recovery in the address space of a running application process. Application software bugs or OS bugs that corrupt the logs or the application’s data in persistent memory cannot be tolerated by any of these approaches, but detectable non-corrupting software failures can be tolerated. The main difference between JUSTDO logging and the earlier approaches to persistent memory FASEs is that JUSTDO logging does not tolerate software failures within FASEs: Our approach of resuming execution at the point of interruption is inappropriate for such failures, and our approach does not have the ability to roll back a FASE interrupted by failure.
Fortunately, two promising bodies of active research complement JUSTDO logging by offering protection from corrupting bugs and by offering a high degree of software quality assurance for FASEs. Capability-based memory protection, exemplified by the CHERI system [30, 203], provides fine-grained intra-process memory protection—precisely what modern persistent memory FASE implementations require to protect logs and application data from corruption by application software errors. Symbolic execution techniques offer very high quality assurance for software, but with limited scalability [20, 21, 60]—precisely what JUSTDO logging requires to ensure that FASEs are free of defects.

**Use Cases** Two widespread and important use cases, which we call library-managed persistence and mandatory mediated access, highlight the strengths of JUSTDO logging and its synergies with fine-grained isolation and software quality assurance techniques.

JUSTDO logging can provide the foundation for high-performance thread-safe libraries that manage persistent data structures on behalf of application logic. In such scenarios, exemplified today by the Berkeley Database [157, 206] and similar software, the library assumes responsibility both for orderly concurrent access to shared data in persistent memory and for recovering persistent memory to a consistent state following failures. JUSTDO logging enables expert library developers to write lock-based FASEs in library routines and employ JUSTDO logging to ensure consistent recoverability with
low overhead during failure-free operation. A well-designed JUSTDO-based library will consolidate persistent data updates in small FASEs that lend themselves readily to powerful software quality assurance techniques.

A related use case involves application logic of questionable quality or security that must be constrained to manipulate valuable persistent data only indirectly, via a trusted high-quality intermediary. A widespread example of this pattern is commercial relational database management systems, which mediate application access to database tables while upholding data integrity constraints and preventing arbitrary modifications of the database by buggy, misguided, or malicious application logic. JUSTDO logging provides a new high-performance logging strategy for the intermediary in mandatory mediated access scenarios. OS process boundaries coupled with user permissions can isolate untrusted application code from trusted intermediary software, allowing only the latter direct access to persistent data. However this isolation strategy, widely used today in high-integrity database configurations, requires application logic to communicate with trusted code via heavyweight inter-process communication (IPC) mechanisms. Research on fine-grained intra-process isolation [30], together with JUSTDO logging, suggests a lightweight alternative: Application logic accesses persistent data via a library linked into the same address space as the application, precisely as in the library-managed persistence scenario, but with a crucial difference: The intra-process isolation mechanism protects both the data and the trusted library from untrusted application code. Such a strategy eliminates the over-
head of IPC between application code and the trusted intermediary without weakening protection.

5.4 Design

JUSTDO logging implements lock-delimited FASEs by recording sufficient information during the execution of a FASE such that, if a crash occurs, each FASE can resume at the last STORE it attempted prior to the failure.

The key data structure for our technique is the JUSTDO log, a small per-thread log. This thread-local log contains only a single active entry at any one time, and is written before every STORE within a FASE. The single active log entry contains only the address to be written, the new value to be written there, the size of the write, and the program counter. Immediately after the log entry is completed, the corresponding STORE is performed.

To recover using a crashed program’s set of per-thread JUSTDO logs, we re-enter each interrupted FASE at the program counter indicated in the FASE’s JUSTDO log, re-acquire the appropriate locks, re-execute the idempotent STORE, and continue execution until the end of each FASE.

Successful recovery requires additional steps when writing a JUSTDO FASE. In particular, we must ensure that the instructions in a FASE do not access data that was stored in transient memory, which will not have survived the failure. We satisfy this requirement by mandating that all LOADS and STORES within a FASE access only persistent memory. Furthermore, we must ensure
that instructions in a FASE do not depend on data held only in volatile CPU registers. We satisfy this requirement by preventing register promotion of memory values within FASEs. Finally, the recovery-time completion of each FASE must respect all mutual exclusion constraints present in the program. We ensure this by recording the locks acquired and released in each FASE in thread-local lock logs.

This section describes the design of the JUSTDO log and our auxiliary data structures. For brevity we largely omit release fences from our discussion. We employ release fences as necessary to constrain the order in which stores attain persistence.

5.4.1 JUSTDO Log

The JUSTDO log is updated for every store within a FASE. Our approach transforms every store in the original crash-vulnerable FASE to both a log update and then the store in the JUSTDO-fortified FASE.

Figure 5.3 illustrates the format of the entire thread-local JUSTDO log. The log is implemented as a tightly packed struct where each field holds critical recovery information. To ensure atomic updates to the log, it actually holds two entries, although only one is active at a time. In each entry, we store the destination address, size, and new value. The program counter value is shared between the entries, and we use the high order bits of the program counter to indicate which entry is active. On Intel x86, virtual
addresses are 48 bits, facilitating this tight packing [86]. Additional bits in the size field and indicator bit are reserved for future use (e.g., flags).

To update the log, both the new value and destination address are stored (with the size packed into the high order bits of the address pointer) in the inactive entry, followed by a release fence to ensure that the writes have reached the persistent cache. Subsequently, we store the new program counter (with the indicator bit set for the recently updated entry).

After the log has been successfully updated, we execute a release fence (again to ensure that the updates are persistent), then complete the persistent store by writing the new value to the destination address.
5.4.2 Persistent-Only Accesses

We require that all memory loads and stores within FASEs access only persistent data. This requirement extends to thread-local locations that would ordinarily be transient, such as variables on the stack. By mandating that FASEs can access only persistent data we ensure that no updates in a FASE are dependent on state destroyed by failure.

The persistent-only access requirement means that any thread-local memory locations that might be accessed in the FASE (including those normally stored on the stack) must be moved to persistent memory prior to entering the first critical section of a FASE, and, if desired, moved out of persistent memory at the end of a FASE.

While this “persistent-only access” restriction may appear limiting, we find that it is compatible with familiar design patterns. Consider, for example, the ubiquitous “container” pattern as applied to persistent data: (nearly) all of the container metadata maintained by the library code is persistent; similarly, the data stored in a persistent container is also persistent. User code will ensure that its (possibly large) data values are persistent before passing pointers to them into the library; the library can verify that the data are in persistent memory via range checking. It is straightforward to migrate into persistent memory the relatively small amount of transient data passed on the stack between client and library (e.g., the pointer to persistent data). Unlike physical logging-type systems, our technique only requires the data to be written to persistence once, and is consequently insensitive to data size.
(see Section 5.6.4). The “small transient state property” is typical of the exported methods of shared persistent data structures and their maintenance operations (e.g., rebalancing).

5.4.3 Register Promotion in FASEs

Register promotion is a compiler optimization that eliminates redundant loads from memory by caching memory locations in CPU registers [136, 177]. Register promotion in FASEs is problematic for justdo logging. Consider a value in persistent memory that within a FASE is loaded into a register upon which two subsequent stores depend. If, due to register promotion, the value is not re-loaded from persistent memory prior to influencing the second store, recovery from a crash immediately after the first store is impossible: The crash erases the register containing the value upon which the second store depends.

Our current implementation prevents such anomalies by selective use of the C/C++ “volatile” keyword. We employ a templated type wrapper within FASEs to ensure that loads within FASEs occur via volatile pointers and are therefore not elided by compiler optimization. Note that stores are not affected by this load-specific mechanism. Manual inspection of the assembly code generated for our FASEs confirms that our current approach prevents problematic register promotions without affecting stores.

In practice, the additional loads within FASEs that our current im-
plementation adds do not introduce large performance overheads for the
data-intensive data structures and algorithms that we have considered to
date, because most data structures are limited by LOAD throughput rather
than compute throughput. Disabling register promotion within FASEs and
thereby causing LOADs that would otherwise be elided by an optimizing com-
piler therefore tends to increase the execution times of FASEs by a small
proportion—roughly 2× in our experience. In the future, a justdo-aware
compiler could more selectively disable register promotion in FASEs, allowing
it only where it does not preclude recovery.

5.4.4 Lock Logs

Recovering from failure requires that every recovery thread know which locks
it holds, and furthermore that no locks are held forever. Our design sup-
ports arbitrary lock implementations; our current prototype employs stan-
dard pthread mutexes.

To preserve lock ownership information across crashes, we require that
locks reside in a persistent memory region. Threads maintain two per-thread
persistent logs to facilitate proper restoration of lock ownership during recov-
ery: a lock intention log and a lock ownership log. The purpose of the former
is to speed recovery by obviating the need to inspect all locks in persistent
memory, whereas the latter is used to re-assign locks to recovery threads.

Immediately prior to attempting a lock acquisition, a thread declares its
intent by recording the lock address in the lock intention log. Immediately after acquiring the lock, the thread records the acquisition in the lock ownership log using a JUSTDO store. To unlock a mutex, a thread performs the same operations in reverse order: It first uses a JUSTDO store to remove the lock from its lock ownership log, then unlocks the mutex, and finally removes the lock from the lock intention log. This protocol ensures that following a crash the per-thread lock intention logs collectively record all locks that might be locked, and the lock ownership logs record which thread has locked each lock that is certainly locked.

5.4.5 Recovery

Recovery begins by using the per-thread lock intention logs to unlock all mutexes that might have been locked at the moment of failure. Without lock intention logs, unlocking all mutexes would require inspecting them all or using generational locks in the manner of NV-heaps [32]. The lock intention log enables both arbitrary mutex implementations and fast recovery.

After unlocking all mutexes, recovery spawns one thread per non-empty JUSTDO log; a recovery thread’s duty is to execute to completion a corresponding FASE that had been cut short by failure. Each recovery thread inherits a JUSTDO log and the pair of lock logs left behind by its deceased predecessor.

Recovery threads begin by acquiring all locks in their respective lock
ownership logs, then waiting at a barrier for all other threads to do likewise. Once all locks have been acquired by all recovery threads, each thread re-executes the store instruction contained in its justdo log. Finally, each recovery thread jumps to the program counter value contained in the justdo log and continues execution of the interrupted FASE. Recovery threads track the number of mutexes they hold, and when this count drops to zero the FASE has been completed and the thread exits.

Interestingly, recovery must be executed with an interleaving of instructions (either in parallel or by context switching across recovery threads): Some FASEs may be blocked waiting for other FASEs to release mutexes. This interleaving requirement is actually an advantage, because our approach naturally supports parallel recovery. Furthermore, once our recovery threads have re-acquired all of their locks and passed the barrier, access to shared persistent state is properly synchronized by the appropriate mutexes. Consequently, the resurrected application may spawn ordinary (non-recovery) threads that operate, with appropriate synchronization, upon persistent memory even before our recovery threads have completed the execution of interrupted FASEs. In other words, the restoration of consistency to persistent memory can proceed in parallel with resumed application execution. Section 5.6.3 presents recovery time measurements of crashed processes that manipulated large volumes of persistent data via JUSTDO logging.

Reasoning about the barrier employed by recovery makes it easy to show that our approach tolerates failures during recovery. No persistent memory
state is altered before our recovery threads reach the barrier, so a crash before this point has no effect and recovery may simply be attempted again. After our recovery threads pass the barrier, they execute FASEs under the protection of JUSTDO logging, precisely as in an ordinary execution of the program.

\subsection*{5.5 Implementation}

Our current JUSTDO logging prototype is a C++ library with bindings for both C++ and C. Annotations for JUSTDO-enabled FASEs are a straightforward, if tedious, transformation of transient (non-crash-resilient) code. We hope that future work, integrating compiler support, can automate nearly all of the chores surrounding annotations while also providing additional type safety guarantees to ensure that the “persistent-only accesses” rule is followed within FASEs. In the meantime, however, JUSTDO integration requires every possible code path within a FASE to be identified and annotated at compile time, making JUSTDO integration significantly more complex than other failure atomicity systems. Other systems, such as Atlas, do not need to know all possible FASE code paths at compile time. Compared with prior FASE implementations, our current prototype deliberately trades programmer convenience and generality for performance.

Our library contains three major elements: the \texttt{jd\_root}, the \texttt{jd\_obj}, and the JUSTDO routine. The first two are C++ classes and are entry points
into our library. The justdo routine consolidates the boilerplate required to execute an application-defined FASE under justdo logging. The remainder of this section illustrates the use of these elements in a detailed example shown in Figures 5.4, 5.5, 5.6, and 5.7. Our example code failure-atomically transfers money from $\text{acnt1}$ to $\text{acnt2}$; for clarity we omit type casts and the use of the volatile keyword. Our example code shows the usage of justdo annotations and how to set up a justdo FASE.

By definition, persistent memory outlives the processes that access it. Therefore justdo logging requires mechanisms to enable newly created processes to locate persistent memory containing data of interest and to make the data accessible to application software. At a high level, we follow the same straightforward approach taken by prior research implementations of FASEs and by emerging industry standards for persistent memory [174, 184]: A file system (or the moral equivalent thereof) maps short, memorable, human-readable strings (names) to long persistent byte sequences, and processes use an mmap-like interface to incorporate into their address spaces the persistent data thus located. More specifically, our justdo logging prototype uses the Atlas [24] implementation of persistent memory regions, which supports memory allocation methods \texttt{nv malloc} and \texttt{nv calloc} and contains a header for its root pointer (accessed via \texttt{Get/SetRegionRoot} methods), as shown in our example code.
5.5.1 jd_root

The jd_root object is the main entry point to the justdo library. This object is placed in a well-known location in the persistent region that is accessible by recovery code via GetRegionRoot.

The jd_root object is a global object and is the factory object for jd_objs, which are thread-local. The jd_root maintains a list of the jd_objs that have been allocated to threads.

During recovery, the jd_root object is responsible for unlocking all mutexes and coordinating lock re-acquisitions across recovery threads. Finally, it initiates thread-local recovery, in which recovery threads jump back into their respective FASEs.

5.5.2 jd_obj

The jd_obj is a thread local object for executing a FASE under justdo logging. It contains both the justdo log structure and its associated lock logs. jd_obj exports methods jd_lock, jd_store, and jd_unlock; consequently most lines within a justdo FASE will touch the jd_obj.

The jd_obj object provides a handle to thread-local persistent memory that is used to persist variables normally on the stack; this handle facilitates compliance with the “persistent-only access” rule of Section 5.4.2. In an exception to the “persistent-only access” rule, each thread maintains a reference to its jd_obj on the stack. Following a crash, this reference is correctly re-set in
each recovery thread. This exception allows a recovery thread to share a reference to its \texttt{jd\_obj} with its failed predecessor.

### 5.5.3 JUSTDO Routine

A \texttt{JUSTDO} routine is a function containing a \texttt{JUSTDO FASE}. Such functions have a defined prototype and are annotated to enable recovery. During recovery, the \texttt{JUSTDO} routine’s stack frame provides thread-local scratch space that would be inconvenient to obtain otherwise. The annotations are illustrated in our example code at line \ref{line:transfer_justdo} in \texttt{transfer\_justdo} of Figure \ref{fig:justdo}.

A \texttt{JUSTDO} routine complies with several annotation requirements. It takes three arguments: a \texttt{jd\_obj} and two \texttt{void} pointers for the arguments and return values. We also require that the first line of the \texttt{JUSTDO} routine be a special macro: \texttt{JD\_ROUTINE\_ON\_ENTRY} (line \ref{line:justdo_on_entry}).

There are two ways to execute a \texttt{JUSTDO} routine, corresponding to normal (failure-free) execution and recovery. During failure-free operation, invocation of a \texttt{JUSTDO} routine simply executes the function (and FASE) as written.

During recovery, however, the execution of a \texttt{JUSTDO} routine is different. A recovery thread that has acquired mutexes as described in Section \ref{sec:mutexes} invokes the \texttt{JUSTDO} routine, passing as an argument a reference to the the \texttt{jd\_obj} that it inherits from its failed predecessor thread and NULL for the remaining two arguments, \texttt{args} and \texttt{rets}. The \texttt{JD\_ROUTINE\_ON\_ENTRY} macro
in the JUSTDO routine determines from the \textit{jd\_obj} that it is running in recovery mode and uses the JUSTDO log within the \textit{jd\_obj} to cause control to jump to the last \texttt{STORE} within the FASE executed prior to failure. When a recovery thread unlocks its last mutex, it knows that its assigned FASE has completed and therefore it exits.

### 5.5.4 Recovery Implementation

Having introduced all library constructs and their design, we can now summarize the entire recovery procedure:

1. The application detects a crash and invokes JUSTDO recovery via \texttt{jd\_root}.
2. The \texttt{jd\_root} resets all locks using the lock intention logs.
3. The \texttt{jd\_root} spawns recovery threads for every active \textit{jd\_obj}.
4. Each recovery thread re-acquires locks using its lock ownership logs in its \textit{jd\_obj}, then barriers.
5. Following the barrier, the recovery threads invoke interrupted JUSTDO routines with their inherited \textit{jd\_obj}.
6. Each recovery thread uses the \texttt{JD\_ROUTINE\_ON\_ENTRY} macro to jump to the program counter as indicated by its JUSTDO log.
7. When a recovery thread’s lock count reaches zero, it exits.
CHAPTER 5. FAILURE ATOMICITY VIA JUSTDO LOGGING

```c
struct Root{
    int* accounts;
    lock* locks;
    jd_root* jdr;
};

Root* rt;

struct Args{
    int acnt1, acnt2, amount;
};

struct Returns{
    bool success;
};

struct Locals{
    int acnt1, acnt2;
    bool success;
};

// jd_routine for account transfer
void transfer_justdo(jd_obj* jdo,
    void* args, void* rets){
    JD_ROUTINE_ON_ENTRY(jdo);

    rt->locks[jdo->locs->acnt1];
    jdo->jd_lock(
        rt->locks[jdo->locs->acnt2]);

    // increment first account
    jdo->jd_store(  
        &rt->accounts[jdo->locs->acnt1],
        rt->accounts[jdo->locs->acnt1] +  
        jdo->locals->amount);

    // decrement second account
    jdo->jd_store(  
        &rt->accounts[jdo->locs->acnt2],
        rt->accounts[jdo->locs->acnt2] -  
        jdo->locals->amount);

    // end FASE
    jdo->jd_unlock(  
        rt->locks[jdo->locs->acnt1]);
    jdo->jd_unlock(  
        rt->locks[jdo->locs->acnt2]);

    // outside FASE, can access transient
    rets->success = true;
}
```

Figure 5.4: JUSTDO logging example (Globals)

Figure 5.5: JUSTDO logging example (JUSTDO Routine)
# CHAPTER 5. FAILURE ATOMICITY VIA JUSTDO LOGGING

```c
int main()
{
    int rid =
        LoadPersistentRegion("my_region");
    rt = GetRegionRoot(rid);
    // initialize our root if needed
    if(rt == NULL) {
        rt = nv_malloc(sizeof(Root),rid);
        rt->accounts =
            nv_calloc(sizeof(int),N_ACCTS,rid);
        rt->locks =
            nv_calloc(sizeof(lock),N_ACCTS,rid);
        rt->jdr =
            nv_malloc(sizeof(jd_root),rid);
        new(rt->jdr) justdo_root(jdr);
        SetRegionRoot(rt,rid);
    } else{rt->jdr->recover();}
    // get a thread local jd_obj
    jd_obj* jdo = rt->jdr->new_jd_obj();
    // conduct transfer
    Args args;
    args.acnt1 = 3; // arguments passed
    args.acnt2 = 5; // into FASE
    args.amount = 50; // via jd_routine
    Returns rets;
    transfer_justdo(jdo,args,rets);
    // delete jd_obj
    rt->jdr->delete_jd_obj(jdo);
}
```

Figure 5.6: JUSTDO logging example (main)

```c
// The equivalent transient routine
bool transfer_transient(int acnt1,
    int acnt2, int amount){
    lock(rt->locks[acnt1]);
    lock(rt->locks[acnt2]);
    accounts[acnt1] += amount;
    accounts[acnt2] -= amount;
    unlock(rt->locks[acnt1]);
    unlock(rt->locks[acnt2]);
    return true;
}
```

Figure 5.7: JUSTDO logging example (equivalent transient routine)
5.6 Experiments

We implemented five high-throughput concurrent data structures to evaluate the performance and recoverability of JUSTDO logging. Each data structure is implemented in three variants: a Transient (crash vulnerable) version, a JUSTDO crash-resilient version, and a version fortified with the Atlas crash-resilience system [24]. The five algorithms are the following:

Queue The two-lock queue implementation of Michael and Scott [145].

Stack A locking variation on the Treiber Stack [187].

Priority Queue A sorted list traversed using hand-over-hand locking. This implementation allows for concurrent accesses within the list, but threads cannot pass one another.

Map A fixed-size hash map that uses the sorted-list priority queue implementation for each bucket, obviating the need for per-bucket locks.

Vector An array-based resizable vector in the style of the contiguous storage solution proposed by Dechev et al. [40]. This algorithm supports lookups and updates during re-sizing.

The queue and stack are lock-based implementations of algorithms in the java.util.concurrent library. The vector’s design allows it to exploit atomic STORE instructions, and our transient and JUSTDO versions of the vector take advantage of this feature. Atlas supports only mutex-based syn-
chronization and consequently our Atlas version of the vector uses a reader-writer lock instead, which incurs a non-negligible performance overhead. In all other respects, the three versions of each of our five data structures differ only in the implementation—or non-implementation—of crash resilience.

Note that our implementations of these data structures admit parallelism to varying degrees. Our stack, for example, serializes accesses in a very small critical section. At the other extreme, our hash map admits parallel accesses both across and within buckets. We therefore expect low-parallelism data structures to scale poorly with worker thread count whereas high-parallelism data structures should exhibit nearly linear performance scaling.

5.6.1 Correctness Verification

Conventional hardware suffices for the purposes of verifying the crash-resilience guarantees of justdo logging because both conventional CPU caches and conventional DRAM main memory can be persistent with respect to process crashes: Specifically, stores to a file-backed memory mapping are required by POSIX to be “kernel persistent,” meaning that such stores are guaranteed to outlive the process that issued them; neither msync nor any other measures are required after a store to obtain this guarantee [152].

To test justdo recovery we installed a 128 GB justdo-fortified hash map in a file-backed memory mapping on a server-class machine (described in more detail in Section [5.6.2]). After building the hash table, we used all
sixty of the server’s hardware threads to perform inserts and removes in equal proportion on random keys in the hash table. Our hash buckets are implemented as sorted linked lists, so corruption (if present) will manifest as dangling pointers within a bucket, resulting in a segmentation fault or assertion failure. At intervals of two seconds, we killed the threads using an external SIGKILL. On restarting the process, we performed JUSTDO recovery before continuing execution. This test was conducted for approximately four hours. We constructed similar tests for each of our other four concurrent data structures; these additional tests also injected crashes every two seconds and ran for over twelve hours. No inconsistencies or corruption occurred.

5.6.2 Performance Evaluation

The goal of our performance evaluation is to estimate the overhead of JUSTDO crash resilience compared with high-performance transient (crash-vulnerable) versions of the same concurrent algorithms. We took care to ensure that the transient versions of our five algorithms exhibit quite good performance; these versions provide reasonable performance baselines. For example, running on newer hardware, our transient hash map achieves per-core throughput approaching published results on the state-of-the-art MICA concurrent hash table [128].

Our results are conservative/pessimistic in the sense that our experiments involve small data-intensive microbenchmarks that magnify the overheads of
crash resilience to the greatest possible extent. In real applications, concurrent accesses to shared persistent data structures might \textit{not} be a performance bottleneck, and therefore by Amdahl’s law \cite{5} the overheads of any crash resilience mechanism would likely be smaller. This effect is shown in Section 5.6.4, where the overhead of initializing large data values eliminates the overhead of persistence.

Our tests consist of microbenchmarks with a varying number of worker threads. Tests are run for a fixed time interval using a low overhead hardware timer, and total operations are aggregated at the end. For the duration of microbenchmark execution, each thread repeatedly chooses a random operation to execute on the structure. For our evaluations of queues, stacks, and priority queues, threads choose randomly between \texttt{insert} or \texttt{remove}; these three data structures were sized such that most accesses were served from the CPU caches. Therefore performance for our stack and queues is limited by synchronization.

Our vector and map evaluations drew inspiration from the standard YCSB benchmark \cite{35}. For vectors and maps, the containers are filled to 80\% of the key range, then we perform \texttt{overwrite} operations for random keys in the range. The \texttt{overwrite} operation replaces the value only if it exists, but otherwise does not modify the data structure. We sized our vectors and maps so that the vast majority of these two structures did \textit{not} fit in the CPU caches; keys for accesses were drawn randomly from a uniform distribution. Most accesses miss in the CPU caches, therefore our vector and map are limited
by memory performance.

During each test, threads synchronize only through the tested data structure. To smooth performance curves, pages are prefaulted to prevent soft page faults. For data structures with high memory allocator usage (all except the vector), we implemented a simple thread-local bump pointer block pool to prevent bottlenecking on `malloc` and to minimize the impact of Atlas’s custom memory allocator, which tends to underperform at high thread counts. Variables within the data structures are appropriately padded to prevent false sharing. To generate random numbers, threads use thread-local generators to avoid contention.

Software threads for all experiments are pinned to specific hardware threads. Our thread pinning progression fills all cores of a socket first, then fills the corresponding hyperthreads. Once all cores and hyperthreads are filled, we add additional sockets, filling them in the same order. For all machines, we ran every experimental configuration five times and took the average.

Compilation for the transient and JUSTDO data structures was done using gcc 4.8.4 with the `-O3` flag. Atlas structures were compiled using the clang and llvm-based Atlas compiler, again with the `-O3` flag set.

**“Persistent Cache” Machines** We conducted performance tests on three machines. The first is a single-socket workstation with an Intel Core i7-4770 CPU running at 3.40 GHz. The CPU has four two-way hyperthreaded
cores (eight hardware threads). It has a shared 8 MB L3 cache, with per-core private L2 and L1 caches, 256 KB and 32 KB respectively. The workstation runs Ubuntu 12.04.5 LTS.

Our second machine is a server with four Intel Xeon E7-4890 v2 sockets, each of which has 15 cores (60 hardware threads total). The machine has 3 TB of main memory, with 37.5 MB per-socket L3 caches. L2 and L1 caches are private per core, 256 KB and 32 KB respectively. The server
and the workstation are used to mimic machines that implement persistent memory using supercapacitor-backed DRAM (e.g., Viking NVDIMMs [194]) and supercapacitor-backed SRAM.

Figures 5.8 and 5.9 show aggregate operation throughput as a function of worker thread count for all three versions of our data structures—transient, JUSTDO-fortified, and Atlas-fortified. Our results show that on both the workstation and the server, JUSTDO logging outperforms Atlas for every
data structure and nearly all thread counts. **JUSTDO** performance ranges from three to one hundred times faster than Atlas. **JUSTDO** logging furthermore achieves between 33% and 75% of the throughput of the transient (crash-vulnerable) versions of each data structure. For data structures that are naturally parallel (vector and hash map), the transient and **JUSTDO** implementations scale with the number of threads. In contrast, Atlas does not scale well for our vectors and maps. This inefficiency is a product of Atlas’s dependency tracking between FASEs, which creates a synchronization bottleneck in the presence of large numbers of locks.

Future NVM-based main memories that employ PCM or resistive RAM are expected to be slower than DRAM, and thus the ratio of memory speed to CPU speed is likely to be lower on such systems. We therefore investigate whether changes to this ratio degrade the performance of **JUSTDO** logging. Since commodity PCM and resistive RAM chips are not currently available, we investigate the implications of changing CPU/memory speed ratios by under-clocking and over-clocking DRAM. For these experiments we use a third machine, a single-socket workstation with a four-core (two-way hyper-threaded) Intel i7-4770K system running at 3.5 GHz with 32 KB, 256 KB private L1 and L2 caches per core and one shared 8 MB L3 cache. We use 32 GBs of G.SKILL’s TridentX DDR3 DRAM operating at frequencies of 800, 1333 (default), 2000, and 2400 MHz.

For our tests involving small data structures (queue, stack, and priority queue), the performance impact of changing memory speed was negligible—
which is not surprising because by design these entire data structures fit in the L3 cache. For our tests involving larger data structures deliberately sized to be far larger than our CPU caches and accessed randomly (map and vector), we find that the ratio of JUSTDO logging performance to transient (crash-vulnerable) performance remains constant as the ratio of CPU speed to memory speed varies over a $3\times$ range. Slower memory does not negate the benefits of JUSTDO logging.

Figure 5.10: Throughput on workstation using CLFLUSH (linear scale)
“Transient Cache” Machines To investigate how JUSTDO logging will likely perform on machines without persistent caches, but with persistent main memory, we modified our JUSTDO library to use the synchronous CLFLUSH instruction to push stores within FASEs toward persistent memory. This x86 instruction invalidates and writes back a cache line, blocking the thread until it completes. While Intel has announced higher performance flushing mechanisms in future ISAs [173], this instruction remains the only method available on existing hardware. Our CLFLUSH-ing version uses the CLFLUSH instruction where before it used only a release fence, forcing dirty data back to persistent storage in a consistent order.

We performed CLFLUSH experiments on our i7-4770 workstation and compared with Atlas’s “flush-as-you-go” mode, which also makes liberal use of CLFLUSH in the same way (see Figure 5.10). As expected, JUSTDO logging takes a serious performance hit when it uses CLFLUSH after every store in a FASE, since the reduced computational overhead of our technique is overshadowed by the more expensive flushing cost. Furthermore, the advantage of a JUSTDO log that fits in a single cache line is negated because the log is repeatedly invalidated and forced out of the cache. The cache line invalidation causes a massive performance hit. For the JUSTDO map using four worker threads, the L3 cache miss ratio increases from 5.5% to 80% when we switch from release fences to CLFLUSHes. We expect that the new Intel instruction CLWB, which drains the cache line back to memory but does not invalidate it, will significantly improve our performance in this scenario when
it becomes available.

In contrast to justdo logging, Atlas’s additional sophistication pays off here, since it can buffer writes back to memory and consolidate flushes to the same cache line. Furthermore, these tests were conducted on smaller data sizes to allow for reasonable completion times, so Atlas’s dependency tracking incurs lower overhead. Atlas outperforms the justdo variants by 2–3× across our tested parameters on “transient cache” machines.

5.6.3 Recovery Speed

In our correctness verification test (Section 5.6.1), which churned sixty threads on a 128 GB hash table, we also recorded recovery time. After recovery process start-up, we spend on average 2000 microseconds to mmap the large hash table back into the virtual address space of the recovery process. Reading the root pointer takes an additional microsecond. To check if recovery is necessary takes 64 microseconds. In our tests, an average of 24 FASEs were interrupted by failure, so 24 threads needed to be recovered. It took on average 2700 microseconds for all recovery threads to complete their FASEs. From start to finish, recovering a 128 GB hash table takes under 5 ms.

5.6.4 Data Size

Figure 5.11 shows throughput as a function of data size on the various key-value (hash map) implementations. Tests were run on the server machine
with eight threads, assume a persistent cache, and vary value sizes from a single byte to one kilobyte. For each operation, values were created and initialized with random contents by the operating thread. Allocation and initialization quickly become bottlenecks for the transient implementation. The \texttt{JUSTDO} implementation is less sensitive to data size, since it operates at a slower speed, and value initialization does not begin to affect throughput until around half a kilobyte. At one kilobyte, the allocation and initialization of the data values becomes the bottleneck for both implementations, meaning the overhead for persistence is effectively zero beyond this data size. In contrast to the transient and \texttt{JUSTDO} implementations, the Atlas implementation is nearly unaffected by data size changes: Atlas’s bottleneck remains dependency tracking between FASEs.
Note that only Atlas copies the entire data value into a log; in the case of a crash between initialization of a data value and its insertion, Atlas may need to roll back the data’s initial values. In contrast, JUSTDO logging relies on the fact that the data value resides in persistent memory. After verifying that the data is indeed persistent, the JUSTDO map inserts a pointer to the data. The “precopy” of JUSTDO copies only the value’s pointer off the stack into persistent memory. Consequently, it is affected by data size only as allocation and initialization become a larger part of overall execution. Obviously, the transient version never copies the data value as it is not failure-resilient.

5.7 Conclusions

We have shown that JUSTDO logging provides a useful new way to implement failure-atomic sections. Compared with persistent memory transaction systems and other existing mechanisms for implementing FASEs, JUSTDO logging greatly simplifies log maintenance, thereby reducing performance overheads significantly. Our crash-injection tests confirm that JUSTDO logging preserves the consistency of application data in the face of sudden failures. Our performance results show that JUSTDO logging effectively leverages persistent caches to improve performance substantially compared with a state-of-the-art FASE implementation.
Chapter 6

iDO Logging: Practical Failure Atomicity

6.1 Introduction

While JUSTDO logging performs well if a persistent cache is assumed, the performance drops significantly if we assume a more traditional NVM architecture with transient caches and registers but persistent NVM main memory. On this more traditional architecture, the problem with JUSTDO logging is its requirement that the log be written \textit{and made persistent} before the related

\footnote{This chapter is based on work done by Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung \cite{130}. IDO: Practical failure atomicity with nonvolatile memory. This work was led by our colleagues at Qingrui Liu and Changhee Jung at Virginia Tech, and by Se Kwon Lee and Sam H. Noh at UNIST. We provided assistance writing benchmarks, integrating them with related systems, running experiments, and writing the paper.}
store—a requirement that is very expensive to fulfill on conventional machines. Current ISAs provide limited support for ordering write-back from cache to persistent memory, and these limitations seem likely to continue into the foreseeable future [87]. Typically the only way to ensure that writes reach memory in a particular order is to separate stores with a sequence of instructions commonly referred to as a persist fence. On an Intel x86, the sequence is \langle mfence, clflush, clflush, clflush, ..., mfence \rangle. This sequence initiates and waits for the write-back of a set of cache lines, ensuring that they will be persistent before any future writes. Unfortunately, the wait incurs the cost of round-trip communication with the memory controller.

This chapter demonstrates that recovery via resumption can in fact be made efficient on machines with volatile caches and expensive persist fences. The key is to arrange for each log operation (and in particular each persist fence) to cover not one but many store instructions of the original application. We achieve this coverage via compiler-based identification and maximization of idempotent instruction sequences, which can safely be re-executed an arbitrary number of times without changing program behavior.

This chapter presents iDO, a practical compiler-integrated failure-atomicity system. Like JUSTDO logging, iDO supports fine-grained concurrency through lock-based FASEs, and avoids the need to track dependences by executing forward to the end of each FASE during post-crash recovery. Unlike JUSTDO, however, iDO performs well on machines with volatile caches, outperforming Atlas and NVThreads by substantial margins. Furthermore, iDO does not
require the absence of volatile data which makes JUSTDO impractical.

Instead of logging information at every store instruction, iDO logs (and persists) a slightly larger amount of program state (registers, live stack variables, and the program counter) at the beginning of every idempotent code region within the overall FASE. In practice, idempotent sequences tend to be dramatically longer than the span between consecutive stores—tens of instructions in our benchmarks; hundreds or even thousands of instructions in larger applications [113]. Because it is implemented in the LLVM tool chain [120], our implementation is also able to implement a variety of important optimizations, logging significantly less information—and packing it into fewer cache lines—than one might naively expect. For the sake of convenience we also automatically infer the boundaries of any FASEs contained entirely within a single function (avoiding the need for program annotation), and introduce a new implementation for FASE-boundary locks that requires only a single memory fence, rather than the two employed in JUSTDO.

Following are the major contributions of this chapter:

- We introduce iDO logging, a lightweight logging strategy that leverages idempotence to ensure both the atomicity of FASEs and the consistency of persistent memory in the wake of a system crash. In comparison to existing undo/redo-log–based approaches, iDO requires no log for the memory stores but a lightweight program state checkpoint at the beginning of each idempotent region.
CHAPTER 6. IDO LOGGING: PRACTICAL FAILURE ATOMICITY

- We compare the performance of iDO to several existing systems, demonstrating up to an order of magnitude improvement over Atlas in run-time speed, and dramatically better scaling than transactional systems like Mnemosyne [196].

- We enable fast recovery with iDO which can be orders of magnitudes faster than existing FASEs based failure atomic system, making iDO a practical approach.

- We implement iDO in the LLVM toolchain [120].

This chapter is organized as follows. Section 6.2 gives additional background on failure-atomicity systems and idempotence. Section 6.3 discusses the high-level design of iDO logging; Section 6.4 delves into system details. Performance results are presented in Section 6.5. We discuss related work in Section 6.6 and conclude in Section 6.7.

6.2 Background

Figure 6.1: Hybrid architecture model in which a portion of memory is non-volatile, but the core, caches, and DRAM are volatile.
6.2.1 System Model

iDO (unlike justDO) assumes a near-term hybrid architecture (Fig. 6.1), in which some of main memory has been replaced with nonvolatile memory, but the rest of main memory, the caches, and the processor registers remain volatile. Data in the core and caches are therefore transient, and will be lost on system failure.\(^2\) Portions of main memory are likely to continue to be implemented with DRAM in the short term, due to density, cost, and/or endurance issues with some NVM technologies. As in other recent work, we assume that read and write latencies of NVM are similar to those of DRAM \(^{24}\) and that writes are atomic at 8-byte granularity \(^{31}\). Our failure model encompasses (only) fail-stop errors that arise outside the running application. These include kernel panics, power outages, and various kinds of hardware failure.

\(^2\)In general, we refer to physical memory as *volatile* or *nonvolatile*, and to program memory (data) as *transient* or *persistent*.
6.2.2 Programming Model

As noted in Section 6.1, iDO employs a programming model based on lock-delineated failure-atomic sections (FASEs), primarily because of their ubiquity in existing code. As in JUSTDO logging, a FASE is defined as a maximal-length region of code beginning with a lock (mutex) acquire operation and ending with a lock release, in which at least one lock is always held [12, 24, 83, 90]. Note that the outermost lock and unlock pairs do not necessarily need to be the same (see Figure 5.1).

In general, FASE-based failure-atomicity systems based on UNDO and REDO logging prohibit thread communication outside of critical sections; the concern is that a happens-before dependence between critical sections could be created without the system’s knowledge. An advantage we gain from execute-forward recovery (“recovery via resumption”) is that thread communication outside of critical sections is allowed without compromising correctness (though obviously, FASEs delineated by locks untracked by iDO will not be recovered). With some small caveats, we also support the use of C/C++ atomic variables within critical sections; further details can be found in Section 6.3.3.

Despite its strengths, recovery via resumption has some pitfalls. In order for recovery to succeed, the failure atomic code region must be allowed to be run to completion. For this reason, resumption is infeasible for abortable transactions; no mechanism exists to undo changes already made to the shared state. Consequently, iDO logging is vulnerable to software bugs within
Chapter 6. IDO Logging: Practical Failure Atomicity

FASEs—on recovery, reexecuting the buggy code will not restore consistency. In general, iDO logging is suitable for persistent data accessed by applications that expect to tolerate fail-stop errors, such as kernel panics, hardware faults, or power outages.

6.2.3 Idempotence

An idempotent region is a single-entry, (possibly) multiple-exit subgraph of the control flow graph of the program. In keeping with standard terminology, we use the term inputs to refer to variables that are live-in to a region. An input has a definition that reaches the region entry and a use of that definition after the region entry. We also use the term antidependence to refer to a write-after-read dependence, in which a variable is used and subsequently overwritten. A region is idempotent if and only if it would generate the same output if control were to jump back to the region entry from any execution point within the region (assuming isolation from other threads). To enable such a jump-back, the region inputs must not be overwritten—i.e., there must be no antidependence on the inputs—during the execution of the region.

Idempotent regions have been used for a variety of purposes in the literature, including recovery from exceptions, failed speculation, and various kinds of hardware faults [113]. For any of these purposes—and for iDO—inputs must be preserved to enable re-execution.
6.3 iDO Failure Atomicity System

iDO logging, unlike undo or redo logging, provides failure atomicity via resumption. Once a thread enters a FASE, iDO must ensure that it completes the FASE, even in the presence of failures. At the beginning of each idempotent code region in the body of a FASE, the executing thread logs the inputs to the code region together with the program counter. Since the region is idempotent, the thread never overwrites the region’s inputs before the next log event. Consequently, if a crash interrupts the execution of the idempotent region, we can re-execute the idempotent region from the beginning using the persistent inputs. Once the thread finishes executing the idempotent region, it persists the inputs to the next idempotent region and continues in this fashion until the end of the FASE.

Successful recovery requires some additional care. In particular, if we re-execute a FASE during recovery using a recovery thread, this thread must hold the same locks as the original crashed thread. Tracking this information is the responsibility of the thread’s local lock_array (Section 6.3.1), which is updated at every lock acquisition and release.

Recovery thus comprises the following general steps.

1. iDO detects the crash and retrieves the logs of all interrupted threads.

2. The recovery process creates and initializes a recovery thread for each interrupted thread.

3. Each recovery thread acquires the locks held by its predecessors; the
threads then execute a barrier.

4. After the barrier, each thread loads register values from its predecessor’s log, then jumps to the stored PC which is the beginning of the interrupted idempotent region.

5. Each thread executes to the end of the current FASE, at which point no thread holds a lock. All threads then terminate and recovery is complete.

Elaborating on these steps, the following subsections consider the structure of the iDO log, the implementation of FASE-boundary locks, and the recovery procedure. We also consider the extent to which we can accommodate racy accesses to atomic variables in application code. Compiler details—and in particular, the steps required to identify FASEs and transform the FASEs into idempotent regions—are deferred to Section 6.4.
CHAPTER 6. IDO LOGGING: PRACTICAL FAILURE ATOMICITY

6.3.1 The iDO Log

For each thread, the iDO runtime creates a structure called the \texttt{iDO\_Log}. We manage the per-thread iDO logs using a global linked list whose \texttt{iDO\_head} is placed in a persistent memory location to be found by the recovery procedure (Section 6.4.3). Log structures are added to the list at thread creation. As shown in Figure 6.3, each \texttt{iDO\_log} structure comprises four key fields. The \texttt{recovery\_pc} field points to the initial instruction of the current idempotent region. The \texttt{intRF} and \texttt{floatRF} fields hold live-in register values; each register has a fixed location in its array. The \texttt{lock\_array} field holds \textit{indirect lock addresses} for the mutexes owned by the thread—more on this in Sec. 6.3.2.

The execution of an idempotent region then comprises the following steps:

1. Issue write-back instructions to ensure that all values in \texttt{intRF} and \texttt{floatRF} have persisted, together with any live-in variables in the stack.

2. Update \texttt{recovery\_pc} to point to the beginning of the current idempotent region.

3. Execute the idempotent region, updating the register values when needed for the next idempotent region. Note that an idempotent region will never overwrite its own input registers.

To enforce the order of these steps, the iDO compiler inserts a single \textit{persist fence} between the first step and the second, and again between the second
and the third. After completing the steps, a thread moves on to the next idempotent region. Registers that are live-in to the following region are written to $\text{intRF}$ and $\text{floatRF}$ immediately after their final modification in the current region. Writes-back of stack variables that are live to the following region are likewise initiated immediately after the final write of the current region, though we don’t wait for completion until the fence between steps 1 and 2. In the absence of precise pointer analysis, we can’t always identify the final writes to variables accessed via pointers; these are therefore logged to transient memory and then written back at the end of each idempotent region.

After a crash, the iDO runtime creates a recovery thread for each failed thread. After acquiring any locks owned by its predecessor and executing a barrier with respect to its peers, each thread restores all registers from its log (including the stack pointer, which is almost always live, and possible garbage values for registers that aren’t live), jumps to the idempotent region specified by $\text{recovery.pc}$, and executes through to the end of the current FASE.

### 6.3.2 Indirect Locking

Our discussion thus far has talked mostly about recovering idempotent regions. To recover a full FASE, however, we must introduce lock recovery. In particular, in the wake of a crash, we must reassign locks to the correct
recovery threads and lock them before re-executing a FASE, and we must also ensure that no other locks are accidentally left locked from the previous program execution. Previous approaches [32, 90] persist each mutex. Thus, they have to unlock every mutex on recovery to release it from a failed thread before assigning it to a recovery thread. In JUSTDO logging, this task requires updating a lock intention log and a lock ownership log before and after the lock operation. Each lock or unlock operation then entails two persist fence sequences—a significant expense.

iDO introduces a novel approach that avoids the need to make mutexes persistent. The key insight is that all mutexes must be unlocked after a system failure, so their values are not really needed. We can therefore minimize persistence overhead by introducing an indirect lock holder for each lock. The lock holder resides in persistent memory and holds the (immutable) address of the (transient) lock. During normal execution, immediately after acquiring a lock, a thread records the address of the lock holder in one of the entries of its log’s lock.array. It also sets a bit in the first entry of the array (which is otherwise unused) to indicate which array slots are live. Immediately before releasing a lock, the thread clears both the lock.array entry and the bit. Finally, the iDO compiler inserts an idempotent region boundary immediately after each lock acquire and before each lock release.

Upon system failure, each transient mutex will be lost. The recovery procedure, however, will allocate a new transient lock for every indirect lock holder, and arrange for each recovery thread to acquire the (new) locks iden-
CHAPTER 6. IDO LOGGING: PRACTICAL FAILURE ATOMICITY

Identified by lock holders in its lock array. An interesting side effect of this scheme (also present in JUSTDO logging), is that if one thread acquires a lock and, before recording the indirect lock holder, the system crashes, another thread may steal the lock in recovery! This effect turns out to be harmless: the region boundaries after lock acquire ensure that the robbed thread failed to execute any instructions under the lock.

6.3.3 iDO Recovery

Building on the preceding subsections, we can now summarize the entire recovery procedure:

1. On process restart, iDO detects the crash and retrieves the iDO Log linked list.

2. iDO initializes and creates a recovery thread for each entry in the log list.

3. Each recovery thread reacquires the locks in its lock array and executes a barrier with respect to other threads.

4. Each recovery thread restores its registers (including the stack pointer) from its iDO log, and jumps to the beginning of its interrupted idempotent region.

5. Each thread executes to the end of its current FASE, at which point
no thread holds a lock, recovery is complete, and the recovery process can terminate.

It should be emphasized that, as with all failure atomicity systems, iDO logging does not implement full checkpointing of an executing program, nor does it provide a means of restarting execution or of continuing beyond the end of interrupted FASEs. Once the crashed program’s persistent data is consistent, further recovery (if any) is fully application specific.

Atomic Instructions Most persistent programming models do not support atomic instructions in their FASEs or transactions. Atlas and NV-Threads, for example, mandate that all synchronization happen through locks. Similarly, Mnemosyne allows no atomics inside transactions.

iDO logging can do better. Specifically, C++ atomic variables can be accessed without restriction outside FASEs, and also inside FASEs so long as the intra-FASE accesses never constitute a write-write race (read-write races are ok). These same rules apply to justdo logging. The restriction on write-write races occurs because recovery via resumption may re-execute writes after a crash, and may thus invert the result of a write-write race. For example, consider two threads $t_1$ in a FASE and $t_2$ outside the FASE. Suppose that $t_1$ writes to atomic variable $x$, then $t_2$ writes to $x$, and then the system crashes. The recovery process may re-execute $t_1$’s store to $x$ using a logged value, overwriting the later value written by $t_2$. As $x$’s value might
6.4 Implementation Details

6.4.1 Compiler Implementation

Figure 6.4 shows an overview of the iDO compiler. The compiler is built on top of LLVM. It takes the generated LLVM-IR from the frontend as input. It then applies a three-phase instrumentation to the LLVM IR and generates the executable. We discuss these three phases in the paragraphs below.

**FASE Inference and Lock Ownership Preservation** In its first instrumentation phase, the iDO compiler infers FASE boundaries in lock-base code, and then instruments outermost lock and unlock operations with iDO library calls, on the assumption that each FASE is confined to a single function. As in the technical specification for transactions in C++ [201],...
one might prefer in a production-quality system to have language extensions with which to mark FASE boundaries in the program source, and to identify functions and function pointers that might be called from within a FASE.

**Idempotent Region Formation** In its second instrumentation phase, the iDO compiler identifies idempotent regions. Previous idempotence-base recovery schemes have developed a simple region partition algorithm to guarantee the absence of memory antidependences, making the preservation of live-in variables the only run-time cost. We use the specific scheme developed by De Kruijf et al. [113]. The compiler first computes a set of cutting points for antidependent pairs of memory accesses using alias analysis. It then applies a hitting set algorithm to select the best cutting strategy. On our benchmarks, typical idempotent regions are roughly 30 instructions in length.

**Live-in Variable Preservation** In its third and final instrumentation phase, the iDO compiler ensures that live-in registers and stack variables have persisted at the beginning of each idempotent region, and are not over-written during the region’s execution. For registers, we artificially extend the live interval of each live-in register to the end of the region [114], thereby preventing the register allocator from assigning other live intervals in the region to the same register and reintroducing an antidependence. For stack vari-
ables, we similarly annotate the relevant slots in the stack frame, preventing them from being shared in the stack coloring phase \[120\].

As noted in Section 6.3.1, the only register values that matter are those that are live-in to the next idempotent region; the rest are not needed for correct recovery. The iDO compiler takes advantage of this fact by logging only those registers that are live-in to the following region, and only their final value. The log entries are then persisted (written back) at the end of the idempotent region. Writes-back of live-in stack variables are initiated at the final write of the idempotent region. Writes-back of variables accessed via pointers (e.g., in the heap) are logged in transient memory, and then written back at the end of the region.

6.4.2 Persist Coalescing

As a further optimization, the iDO compiler takes advantage of the fact that register values are small, and do not need to persist in any particular order. A system like Atlas, which logs 32 bytes of information for every store, can persist at most two contiguous log entries in a single 64-byte cache line write-back. In iDO, as many as eight register values can be persisted with a single write-back (\texttt{clflush}). This \textit{persist coalescing} \[164\] is always safe in iDO, even though registers are grouped by name rather than by order of update at run time, because the registers logged in the current region are used only in later regions. If, for example, a running program updates registers $A$, $C$, \ldots
and $B$, in that order, it is still safe to persist the logged values of $A$ and $B$ together, followed by $C$, because the only ordering constraints that matter are between consecutive regions.

### 6.4.3 Persistent Region Support

iDO requires mechanisms to enable processes to allocate regions of persistent memory and make those regions visible to the program. We leverage Atlas’s implementation for this purpose. Atlas’s region manager represents persistent memory regions as files, which processes incorporate into their address space via `mmap`. The mapped regions then support memory allocation methods such as `nv_malloc` and `nv calloc`.

### 6.5 Evaluation

![Graph showing relative performance of failure atomicity systems](image)

Figure 6.5: Relative performance of failure atomicity systems (4 cores, 8 threads).
CHAPTER 6. IDO LOGGING: PRACTICAL FAILURE ATOMICITY

For our evaluation of iDO logging we compared against several alternative failure atomicity runtimes. We employ both real-world applications and computational kernels to explore iDO logging’s performance impact during normal (crash-free) execution. We also employ microbenchmarks to measure scalability. Finally, we report on recovery time.

In these experiments, where applicable, we compared against the following failure atomic runtimes which guarantee crash consistency on a persistent memory machine.

**Atlas** An undo logging system which uses locks for synchronization. Like iDO logging, Atlas defines a failure-atomic region as the outer-most critical section. The undo logging forces Atlas to track dependencies across critical sections and retire persistent updates in the background [24].

**Mnemosyne** A redo based transactional system integrated into the language-level transactions of C and C++ [196].

**NVThreads** A redo logging, lock based approach that operates on the granularity of pages. Critical sections maintain copies of dirty pages and release them upon lock release [83].

**Origin** The original, crash-vulnerable and uninstrumented code.

For clarity, we elided the results of justdo logging [90] as its authors report its performance is dominated by Atlas in all scenarios where the cache is transient (JUSTDO is optimized for persistent cache machines).
All experiments and runtimes were built using the gcc/g++ 4.7 frontend. Atlas, iDO and NVThreads used the LLVM backend, whereas Mnemosyne used the gcc backend due to its reliance on C++ transactions, a feature not yet implemented in LLVM. For all experiments, all runtimes use the same FASEs (but Mnemosyne, as a transactional system, loses concurrency).

### 6.5.1 Performance

To understand iDO’s performance on real-world benchmarks, we integrated it with the SPLASH3 benchmark [176], an upgraded version of the traditional SPLASH2 suite which eliminated data race errors, a critical step for ensuring the correctness of our approach (and of the comparison runtimes). SPLASH3 consists of a variety of applications which were chosen to give a broad sampling of across different levels of concurrency, working set size, and spatial locality, along with computational kernels common in scientific and engineering computing [202]. We view this benchmark suite as a good approximation of systems that could benefit from fast failure atomicity for preserving some portion of their heap state, and that make nontrivial use of multiple data structures and synchronization techniques.

Experiments were run on a single socket Intel i7-4770 desktop with four hyperthreaded cores (eight total hardware threads). The 64 bit processor has thread-private 256KB L1 caches and 1MB L2 caches, with a shared 8MB L3 cache. The machine runs Ubuntu Linux 14.04 LTS.
Figure 6.5 shows the results of our experiments, scaled to the uninstrumented Origin. Our experiments ran SPLASH3 on eight threads and take the average of ten runs for each configuration. All runtimes were integrated into SPLASH3 using the provided M4 scripts; the transactional Mnemosyne required several benchmarks to be reorganized to fit into a transactional framework. Due to an internal allocation error, Mnemosyne failed on two benchmarks (fmm and cholesky), we have not been able to get results for this configuration.

Of note, iDO logging beats Atlas across all benchmarks, averaging about twice as fast. It provides performance comparable to Mnemosyne across these benchmarks, edging out a slightly better mean (about 10%). The two systems perform quite differently on different benchmarks; the critical differentiating factor appears to be the fraction of instructions which are writes [7]. If the write proportion is low, then iDO can enlarge its idempotent sections and significantly reduce the logging overhead. When the write proportion is low, it also means that Mnemosyne’s read redirection and instrumentation (required for REDO logging to ensure that transactions read their own writes) becomes more of a burden.

6.5.2 Scalability

For scalability experiments, we used the same data structure microbenchmarks used in the evaluation of justDO logging [90]. These microbench-
marks perform repeated accesses to a shared data structure across a varying number threads. The data structures we implemented were:

**Stack** A locking variation on the Treiber Stack [187].

**Queue** The two-lock queue implementation of Michael and Scott [145].

**Ordered List** A sorted list traversed using hand-over-hand locking. This implementation allows for concurrent accesses within the list, but threads cannot pass one another.

**Map** A fixed-size hash map that uses the ordered list implementation for each bucket, obviating the need for per-bucket locks.

For testing, we used an Intel machine with two eighteen-core, two-way hyper-threaded Intel Xeon E5-2699 v3 processors at 3.6 GHz (i.e., with up to 72 hardware threads). Every core’s L1 and L2 caches are private to that core (shared between hyper-threads); the L3 cache (45 MB) is shared across all cores of a single processor. The machine runs Fedora Core 19 Linux. Tests were performed in a controlled environment when we were the sole users of the machine. Threads were pinned to cores in a consistent order for all experiments: one thread per physical core on the first processor (1–18), then one thread for each additional hyper-thread on that processor (19–36), then one thread per core (37–54) and one per additional hyper-thread (55–72) on the second processor. Code was written in C++.

These data structures allow varying degrees of parallelism. The stack, for example, serializes accesses in a very small critical section. At the other
CHAPTER 6. IDO LOGGING: PRACTICAL FAILURE ATOMICITY

extreme, the hash map allows concurrent accesses both across and within buckets. We therefore expect low-parallelism data structures to scale poorly with worker thread count whereas high-parallelism data structures should exhibit nearly linear performance scaling. As in justdo logging, our performance results are conservative in that they present the maximum possible throughput of the data structure. In real code, these data structures may not be the overall bottleneck.

Our tests consist of microbenchmarks with a varying number of worker threads. Tests are run for a fixed time interval using a low overhead hardware timer, and total operations are aggregated at the end. For the duration of microbenchmark execution, each thread repeatedly chooses a random operation to execute on the structure. For our evaluations of the queues and stacks, threads choose randomly between `insert` or `remove`. For the ordered list and hash maps, threads choose randomly between `get` or `put` on a random key within a fixed range.

During each test, threads synchronize only through the tested data structure. Variables within the data structures are appropriately padded to prevent false sharing. To generate random numbers, threads use thread-local generators to avoid contention. To smooth performance curves, pages are prefaulted to prevent soft page faults. Performance of the microbenchmarks is much faster without persistence (10x); we elided this result for clarity.

We show our scalability results in Figure 6.6. Similar to the SPLASH3 results, iDO logging matches or outperforms Atlas in all configurations, es-
especially at higher thread counts. In general, iDO logging also scales better than Mnemosyne, showing near perfect scaling on the hash map. This scaling demonstrates the absolute lack of synchronization between threads in the iDO runtime — all thread synchronization is handled through the locks of the original program. In contrast, both Atlas and Mnemosyne quickly saturate their runtime’s synchronization and throttle performance.

The only case in which iDO logging is beaten by Mnemosyne is the ordered list, which uses hand-over-hand locking for traversal. iDO and Atlas
support this style of concurrency, but they require ordered writes to persistent memory at every lock acquisition and release in order to track lock ownership. Mnemosyne, as a transaction system, cannot support hand-over-hand locking, so the entire traversal is done in a single transaction and data is written to persistent memory only once. iDO and Atlas extract more concurrency from the benchmark, but per-thread execution is slowed relative to Mnemosyne. Consequently, at very high thread counts, iDO outperforms Mnemosyne due to extracted parallelism, despite its single thread performance being about 4x slower.

Figure 6.7: Performance between page-level (NVthreads) and byte-level (iDO) memory logging granularity where iDO serves as base.
6.5.3 Memory Logging Granularity

In order to explore the tradeoffs in log granularity sizes, we compared our implementation against NVThreads, a page based redo logging system. Unlike the fine-grained Atlas, Mnemosyne, and iDO, NVThreads logs changes at granularity of pages. This page-level tracking can be efficient if the program touches only a few pages in a large FASE. The updates in each page can be buffered and allow more coalescing, saving the overhead of flushing the same cache line repeatedly. However, if the program touches many pages and the FASEs are small, page-level logging can be significantly more expensive due to write amplification.

As NVThreads makes use of a file system to provide a logging region, its logging overhead can be significantly influenced by the underlying file system’s performance. In order to minimize the filesystem impact, we tested NVThread performance using three kinds of memory-based filesystem—ramdisk mounted with ext2, ramfs, and PMFS (a NVM-dedicated filesystem [45]). Here we report performance for the ramfs case, where NVThreads achieved its best performance.

Figure 6.7 shows overall performance of SPLASH3 benchmarks with eight threads between iDO and NVthreads. We report only eight applications due to crashes in NVThreads. As we can see in Figure 6.7, iDO shows roughly three times better performance than NVThreads on average, and is only outperformed in two cases.

The most important factor that affects the performance difference be-
CHAPTER 6. IDO LOGGING: PRACTICAL FAILURE ATOMICITY

Between iDO and NVThreads is page locality. NVThreads logs at the page granularity and uses copy-on-write to log each modified page and to guarantee consistent state even after a system crash [83]. This logging is done every time a FASE triggers a page fault. Therefore, the overhead from page fault and page granularity logging increases if the update pattern of the application spans multiple pages. For water-s, lu-c, and fft, NVThreads performs nearly as well or better than iDO logging. This result occurs because the stores in these applications are concentrated on hot pages due to a write pattern based on spatial locality [202]. In contrast, NVThreads exhibits far worse performance than iDO in other applications which have low page locality.

6.5.4 Recovery Overheads

Table 6.1: Recovery time ratio (ATLAS/iDO) at different kill times

<table>
<thead>
<tr>
<th>Kill Time</th>
<th>1 s</th>
<th>10 s</th>
<th>20 s</th>
<th>30 s</th>
<th>40 s</th>
<th>50 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>0.7</td>
<td>6.6</td>
<td>14.0</td>
<td>20.7</td>
<td>28.7</td>
<td>34.9</td>
</tr>
<tr>
<td>Queue</td>
<td>0.8</td>
<td>9.0</td>
<td>20.1</td>
<td>31.6</td>
<td>43.3</td>
<td>56.1</td>
</tr>
<tr>
<td>OrderedList</td>
<td>4.1</td>
<td>72.1</td>
<td>162.2</td>
<td>260.9</td>
<td>301.8</td>
<td>424.8</td>
</tr>
<tr>
<td>HashMap</td>
<td>0.3</td>
<td>1.5</td>
<td>2.7</td>
<td>4.2</td>
<td>5.2</td>
<td>6.2</td>
</tr>
</tbody>
</table>

Our final test tested the speed and correctness of recovery by running the microbenchmarks of Section 6.5.2 and killing the process. We interrupt the applications by sending an external SIGKILL signal after the applications have run for 1, 10, 20, 30, 40 and 50 seconds. For the recovery, iDO follows the recovery procedure in Section 6.3.3. As summarized before, iDO needs to
first initialize the recovery threads. Then iDO recovers the live-in variables for the interrupted region, jumps back to the entry of the interrupted region and continues execution till the end of the FASE. During the evaluation, we observed that the recovery time for iDO is constantly around 1 second. Since most of the FASEs in the benchmark are short (generally on the order of a microsecond), the main overhead for iDO recovery comes from mapping the persistent region into the process’s virtual address space and creating the recovery threads, all of which is an approximately constant overhead. In contrast, for Atlas, recovery needs to first traverse the logs and compute a global consistent state following the happens-before order recorded in the logs, then undo any stores in the interrupted FASEs.

Table 6.1 shows the ratio of recovery time between ATLAS and iDO. When the applications run for a short time (1 second) and get killed, ATLAS imposes less recovery overhead as there are only a few logs and ATLAS can quickly traverse and compute a consistent state. In contrast, iDO has to pay the overheads for creating and initializing the recovery threads. However, when the applications run for a longer time (> 10 seconds), ATLAS will create a large number of logs and require much longer time to traverse and compute a consistent state. We can observe up to 400× faster recovery for iDO. From this test, we can observe that iDO enables simple and low-overhead recovery compared to the existing schemes.
6.6 Related Work

iDO logging combines two areas of research: building crash consistent programs on top of byte-addressable nonvolatile memory, and exploiting idempotence. As the related work in nonvolatile memory and failure atomic systems was covered in Chapter 2, we here focus on idempotence.

Over the years, many researchers have leveraged idempotence for various purposes. Mahlke et al. were the first to exploit the idea, which they used to recover from exceptions during speculative execution in a VLIW processor [141]. Around the same time, Bershad et al. proposed restartable atomic sequences for a uniprocessor based on idempotence [10].

Kim et al. leveraged idempotence to reduce the hardware storage required to buffer data in their compiler-assisted speculative execution model [106]. Hampton et al. used idempotence to support fast and precise exceptions in a vector processor with virtual memory [70]. Tseng et al. used idempotent regions for data-triggered thread execution [188].

Recently, researchers have leveraged idempotence for recovery from soft errors—e.g., ECC faults [51, 113]. Also, Liu et al. [133] advanced the state of the art with checkpoint pruning, which serves to remove logging operations that can be reconstructed from other logs in the event of a soft run-time error. Liu et al. [132, 134, 135] also extend the original idempotent processing in the context of sensor-based soft error detectors to ensure complete recovery.

More recently, the energy-harvesting system community has started using
idempotent processing to recover from the frequent power failures that occur in systems without batteries. Xie et al. [204] use idempotence-based recovery and heuristics to approximate minimal checkpoints (logs) to survive power failures. This design revolves around the idea of severing anti-dependences by placing a checkpoint between a load-store pair, in a manner reminiscent of Feng et al. [51] and de Kruijf et al. [113]. Lately, their techniques were used by Woude et al. [190] to highlight both the promise and the limitations of using idempotence to ensure forward progress when multiple power failures occur within a span of microseconds. In a similar vein, Liu et al. [131] highlight the limitations of anti-dependence based idempotence analysis in terms of additional power consumption due to unnecessary checkpoints. Significantly, all of these projects target embedded processors in which out-of-order execution and caches do not exist.

Despite this wealth of related work, iDO is, to the best of our knowledge, the first system to use idempotence to achieve lightweight, fault-tolerant execution of failure-atomic sections in general-purpose programs.

6.7 Conclusion

Fault tolerance is one of the most exciting applications of emerging non-volatile memory technologies. Existing approaches to persistence, however, suffer from problems with both performance and usability. Transactional approaches are generally incompatible with existing lock-based code, and tend
not to scale to high levels of concurrency. Failure-atomic regions (FASEs), by contrast, are compatible with most common locking idioms and introduce no new barriers to scalability. Unfortunately, prior FASE-based approaches to persistence incur significant run-time overhead, consume significant space, and (at least in current instantiations) depend on user annotations.

To address these limitations, we have introduced iDO logging, a compiler-directed approach to failure atomicity. Without requiring user annotation, the iDO compiler automatically identifies FASEs in existing lock-based code. It then divides each FASE into idempotent regions, arranging on failure recovery to restart any interrupted idempotent region and execute forward to the end of the FASE. Unlike systems based on undo or (for transactions) redo logging, iDO avoids the need to log individual program stores, thereby achieving a dramatic reduction in instrumentation overhead. Specifically, across a wide variety of benchmark applications, iDO’s outperforms the fastest existing persistent systems by 10–200% during normal execution, while preserving very fast recovery times.
Chapter 7

Dalí: A Periodically Persistent Hash Map

7.1 Introduction

In current real-world processors, instructions to control the ordering, timing, and granularity of writes-back from caches to NVM main memory are rather limited. On Intel processors, for example, the clflush instruction takes an address as argument, and blocks until the cache line containing the address has been both evicted from the cache and written back to the memory.

\footnote{This chapter is based on the previously published paper by Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey, Dhruva Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In: DISC '17 [154]. This work was led by Faisal Nawab, who developed the algorithm and ran the experiments. We assisted in the development of the algorithm, and by building the proof of correctness, researching related works, and writing the final paper.}
CHAPTER 7. DALÍ: A PERIODICALLY PERSISTENT HASH MAP

controller. When combined with an MFENCE instruction to prevent compiler and processor instruction reordering, CLFLUSH allows the programmer to force a write-back that is guaranteed to persist (reach nonvolatile memory) before any subsequent store. The overhead is substantial, however—on the order of hundreds of cycles. Future processors may provide less expensive persistence instructions, such as the pwb, pfence, and psync assumed in our earlier work [95], or the ofence and dfence of Nalli et al. [150]. Even in the best of circumstances, however, “persisting” an individual store (and ordering it relative to other stores) is likely to take time comparable to a memory consistency fence on current processors—i.e., tens of cycles. Due to power constraints [34], we expect that writes and flushes into NVM will be guaranteed to be failure-atomic only at increments of eight bytes—not across a full 64-byte cache line.

We use the term incremental persistence to refer to the strategy of persisting store \( w_1 \) before performing store \( w_2 \) whenever \( w_1 \) occurs before \( w_2 \) in the happens-before order of the program during normal execution (i.e., when \( w_1 <_{hb} w_2 \)). Given the expected latency of even an optimized persist, this strategy seems doomed to impose significant overhead on the operations (method calls) of any data structure intended to survive program crashes. All the methods previously presented in this thesis (e.g. jUSTDO, iDO, the chronicle) use incremental persistence.

As an alternative, this chapter introduces a strategy we refer to as periodic persistence. The key to this strategy is to design a data structure in
such a way that modifications can safely leak into persistence in any order, removing the need to persist locations incrementally and explicitly as an operation progresses. To ensure that an operation’s stores eventually become persistent, we periodically execute a global fence that forces all cached data to be written back to memory. The interval between global fences bounds the amount of work that can ever be lost in a crash (though some work may be lost). To avoid depending on the fine-grain ordering of writes-back, we arrange for “leaked” lines to be ignored by any recovery procedure that executes before a subsequent global fence. After the fence, however, a known set of cache lines will have been written back, making their contents safe to read. Like naive uninstrumented code, periodic persistence allows stores to persist out of order. It guarantees, however, that the recovery procedure will never use a value $v$ from memory unless it can be sure that all values on which $v$ depends have also safely persisted.

In contrast to checkpointing, which creates a consistent copy of data in nonvolatile memory, periodic persistence maintains a single instance of the data for both the running program and the recovery procedure. This single instance is designed in such a way that recent updates are nondestructive, and the recovery procedure knows which parts of the data structure it can safely use.

In some sense, periodically persistent structures can be seen as an adaptation of traditional persistent data structures [44] (in a different sense of the word “persistent”) or of multiversion transactional memory systems [19],
both of which maintain a history of data structure changes over time. In our case, we can safely discard old versions that predate the most recent global fence, so the overall impact on memory footprint is minimal. At the same time, we must ensure not only that the recovery procedure ignores the most recent updates but also that it is never confused by their potential structural inconsistencies.

As an example of periodic persistence, we introduce Dalí, a transactional hash map for nonvolatile memory. Dalí demonstrates the feasibility of using periodic persistence in a nontrivial way. Experience with a prototype implementation confirms that Dalí can significantly outperform alternatives based on either incremental or traditional file-system-based persistence. Our prototype implements the global fence by flushing (writing back and invalidating) all coherent on-chip caches. Performance results would presumably be even better with hardware support for whole-cache write-back without invalidation.

The remainder of this chapter is organized as follows: Section 7.2 elaborates on the motivation for our work in the context of persistent hash maps. We describe Dalí’s design in Section 7.3 and prove its correctness in Section 7.4. Section 7.5 then presents experimental results. Section 7.6 reviews related work. Section 7.7 summarizes our conclusions.

---

2The name is inspired by Dalí’s painting The Persistence of Memory.
7.2 Motivation

As a motivating example, consider the construction of a persistent hash map, beginning with the nonblocking structure of Schwalb et al.[178]. To facilitate transactional update of entries in multiple buckets, we switch to a blocking design with a lock in each bucket, enabling the use of two-phase locking (and, for atomicity in the face of crashes, undo logging).

This hash map, which is incrementally persistent, consists of an array of buckets, each of which points to a singly-linked list of records. Each record is a key-value pair. Figure 7.1 shows a bucket with three records. For the sake of simplicity, each list is prepend-only: records closer to the head are more recent. It is possible that multiple records exist for the same key—the figure shows two records for the key $x$, for instance, but only the most recent record is used. Deletions are handled by inserting a “not present” record. Garbage collection/compaction can be handled separately; we omit the description here.

Figure 7.1: A bucket containing three records.

Figure 7.2: An example of the write-ordering overhead entailed in updating a data object.

Figure 7.3: A hash map data structure that demonstrates the overhead of write ordering.
Figure 7.2 shows an update to change the value of \( y \) to 4. The update comprises several steps: (1a) A record, \( r_{\text{new}} \) with the new key-value pair is written. The record points to the current head of the list. (1b) A persist of \( r_{\text{new}} \) serves to push its value from cache to NVM. (2a) The bucket list head pointer, \( B \), is overwritten to point to \( r_{\text{new}} \). (2b) A second persist pushes \( B \) to NVM. The first persist must complete before the store to \( B \): it prevents the incorrect recovery state in which \( r_{\text{new}} \) is not in NVM and \( B \) is a dangling pointer. The second persist must complete before the operation that updates \( y \) returns to the application program: it prevents misordering with respect to subsequent operations.

On current hardware, a persist operation waits hundreds of cycles for a full round trip to memory. On future machines, hardware support for ordered (queued) writes-back might reduce this to tens of cycles. Even so, incremental persistence can be expected to increase the latency of simple operations several-fold. The key insight in Dalí is that when enabled by careful data structure design, periodic persistence can eliminate fine-grain ordering requirements, replacing a very large number of single-location fences with a much smaller number of global fences, for a large net win in performance, at the expense of possible lost work. In practice, we would expect the frequency of global fences to reflect a trade-off between overhead and the amount of work that may be lost on a crash. Fencing once every few milliseconds strikes us as a good initial choice.
7.3 Dalí

Dalí is our prepend-only transactional hash map designed using periodic persistence. It can be seen as the periodic persistence equivalent of the incrementally persistent hash map of Section 7.2 and Figure 7.3. As a transactional hash map, Dalí supports the normal get, set, delete, and replace methods. It also supports ACID transactions comprising any number of the above methods.

Dalí updates or inserts by prepending a record to the appropriate bucket; the most recent record for a key is the one closest to the head of the list (duplicates may exist, but only the most recent record matters). Records in a bucket are from time to time consolidated to remove obsolete versions. Dalí employs per-bucket locks (mutexes) for isolation. A variant of strong strict two-phase locking (SS2PL) is used to implement transactions (see Section 7.3.4 for a description).

7.3.1 Data Structure Overview

As mentioned above, Dalí uses a periodic global fence to guarantee that changes to the data structure have become persistent. The fence is invoked by a special worker thread in parallel with normal operation by application threads. We say that the initiation points of the global fences divide time into epochs, which are numbered monotonically from the beginning of time (the numbers do not reset after a crash). Each update (or transactional set of
updates) is logically confined to a single epoch, and the fence whose initiation terminates epoch $E$ serves to persist all updates that executed in $E$. The execution of the fence, however, may overlap the execution of updates in epoch $E+1$. The worker thread does not initiate a global fence until the previous fence has completed. As a result, in the absence of crashes, we are guaranteed during epoch $E+1$ that any update executed in epoch $E−1$ has persisted. If a crash occurs in epoch $F$, however, updates from epochs $F$ and $F−1$ cannot be guaranteed to be persistent, and should therefore be ignored. We refer to epochs $F$ and $F−1$ as failed epochs, and revise our invariant in the presence of crashes to say that during a given epoch $E$, all updates performed in a non-failed epoch prior to $E−1$ have persisted. Failed epoch numbers are maintained in a persistent failure list that is updated during the recovery procedure.

In Dalí, hash map records are classified according to their persistence status. Assume that we are in epoch $E$. Committed records are ones that were written in a non-failed epoch at or before epoch $E−2$. In-flight records are ones that were written in epoch $E−1$ if it is not a failed epoch. Active records are ones that were written during the current epoch $E$. Records that were written in a failed epoch are called failed records. By steering application threads around failed records, Dalí ensures consistency in the wake of a crash.

Dalí’s hash map buckets are similar in layout to those of the incrementally persistent hash map presented in Figure 7.3 Dalí adds metadata to
CHAPTER 7. DALÍ: A PERIODICALLY PERSISTENT HASH MAP

class node:
key k; val v
node* next

class bucket:
mutex lock
int stat<a, f, c, ss> // 2/2/2/58 bits
node* ptrs[3]

class dalí:
bucket buckets[N_BUCKTS]
int list flist
int epoch

Figure 7.4: Dalí globals and data types.

Figure 7.5: The structure of a Dalí bucket.

each bucket, however, to track the persistence status of the bucket’s records. The metadata in turn allows us to avoid persisting records incrementally. Specifically, a Dalí bucket contains not only a singly-linked list of records, but also a 64-bit status indicator and, in lieu of a head pointer for the list of records, a set of three list pointers (see pseudocode in Figure 7.4 and illustration in Figure 7.5). The status indicator comprises a snapshot (SS) field, denoting the epoch in which the most recent record was appended to the bucket, and three 2-bit role IDs, which indicate the roles of the three list pointers. A single store suffices to atomically update the status indicator on today’s 64-bit machines.³

³With 6 bits devoted to role IDs, 58 bits remain for the epoch number. If we start a new epoch every millisecond, roll-over will not happen for 9 million years.

Each of the three list pointers identifies a record in the bucket’s list (or NULL). The pointers assume three roles, which are identified by storing the pointer number (0, 1, or 2) in one the three role ID fields of the status indicator. Roles are fixed for the duration of an epoch but can change in
future epochs. The roles are:

**Active pointer (a):** provided that epoch $SS$ has not failed, identifies the most recently added record (which must necessarily have been added in $SS$). Each record points to the record that was added before it. Thus, the active pointer provides access to the entire list of records in the bucket.

**In-flight pointer (f):** provided that epochs $SS$ and $SS-1$ have not failed, identifies the most recent record, if any, added in epoch $SS-1$. If no such record exists, the in-flight role ID is set to invalid ($\perp$).

**Committed pointer (c):** identifies the most recent record added in a non-failed epoch equal to or earlier than $SS-2$.

To establish these invariants at start-up, we initialize the global epoch counter to 2 and, in every bucket, set $SS$ to 0, all pointers to NULL, the in-flight role ID to $\perp$, and the active and committed IDs to arbitrary values.

Figure 7.5 shows an example bucket. In the figure $SS$ is equal to 5, which means that the most recent record was prepended during epoch 5. The active pointer is Pointer 0. It points to record $e$, which means that $e$ was added in epoch 5, even if we are reading the status indicator during a later epoch. Pointer 1 is the in-flight pointer, which makes $d$ the most recently added record in epoch 4. Because a record points only to records that were added before it, by transitivity, records $a$, $b$, and the prior $a$ were added before or during epoch 4. Finally, Pointer 2 is the committed pointer. This
makes record \( b \) the most recently added record before or during epoch 3. By transitivity, the earlier record \( a \) was also added before or during epoch 3. Both record \( b \) and the earlier record \( a \) are therefore guaranteed persistent (shown in green) as of the most recent update (the time at which \( e \) was added), while the remainder of the records may not be persistent (shown in red).

It is important to note that the status indicator reflects the bucket’s state at \( SS \) (the epoch of the most recent update to the bucket) even if a thread inspects the bucket during a later epoch. For example, suppose that a thread in epoch 10 reads the bucket state shown in Figure 7.5. Given the status indicator, the thread will conclude that all records were written during or before epoch 5 and thus are all committed and persistent (assuming that epochs 4 and 5 are not in the failure list). If one or both epochs are on the failure list, the thread can navigate around their records using the in-flight or committed pointers.

7.3.2 Reads

The task of the read method is to return the value, if any, associated with a given key. A reader begins by using a hash function to identify the appropriate bucket for its key, and locks the bucket. It then consults the bucket’s epoch number \( (SS) \) and the global failed epoch list to identify the most recent, yet valid, of the three potential pointers into the bucket’s linked list.
CHAPTER 7. DALÍ: A PERIODICALLY PERSISTENT HASH MAP

177 // Bucket is assumed locked via SS2PL
178 val bucket::read(key k):
179    node* valid_head =
180    if ss \not\in flist then ptrs[a]
181    elsif ss-1 \not\in flist && f \neq \bot then ptrs[f]
182    else ptrs[c]
183    return search(k, valid_head)

Figure 7.6: Dalí read method.

(Figure 7.6). Call this pointer the valid head. If SS is not a failed epoch, the valid head will be the active pointer, which will identify the most recently added record (which may or may not yet be persistent). If SS is a failed epoch but SS–1 is not, the valid head will be the in-flight pointer. If SS and SS–1 are both failed epochs, the valid head will be the committed pointer.

Starting from the valid head, a reader searches records in order looking for a matching key. Because updates to the hash map are prepends, the most recent matching record will be found first. If the key has been removed, the matching value may be NULL. If the key is not found in the list, the value returned from the read will also be NULL.

7.3.3 Updates

Updates in Dalí prepend a new version of a record, as in the incrementally persistent hash map of Section 7.2. Deletions/overwrites of existing keys and inserts of new keys are processed identically by a unified update method. Like the read method, update locks the bucket. An update to a Dalí bucket comprises several steps:
1. Determine the most recent, valid pointer (as in the read method).

2. Create a new record with the key and its new value (or NULL if a remove).

3. Determine the new pointer roles (if the new and old epochs are different).

4. Retarget the new active pointer to the new record node.

5. Update SS and the role IDs by overwriting the status indicator.

Pseudocode appears in Figure 7.7.

Step 3 is the most important part of the update algorithm, as it is the part that allows the update’s two component writes (the writes to the state word and head pointer) to be reordered. The problem to be addressed is the possibility that writes from neighboring epochs might be written back.
and become mixed in the persistent state. We might, for example, mix the snapshot indicator from the later epoch with the pointer values from the earlier epoch. Given any combination of update writes from bordering epochs, and an indication of epoch success or failure, the read procedure must find a correct and valid head, and the list beyond that head must be persistent.

The details of step 3 appear in Figure 7.8. They are based on the following three rules. First, the new committed pointer was last written at least two epochs prior, guaranteeing that its value and target have become persistent (and would survive a crash in the current epoch). Second, the new active pointer was either previously invalid or pointed to an earlier record than the new committed pointer. In other words, according to both the old and new status indicators, the new active pointer will never be a valid head, so it is safe to reassign. Third, the new in-flight pointer is the most recent valid
CHAPTER 7. DALÍ: A PERIODICALLY PERSISTENT HASH MAP

record set in the previous epoch, or ⊥ if no such record exists. These rules are sufficient to enumerate all entries in the table.

Because each bucket is locked throughout the update method, there is no concern about simultaneous access by other active threads. We assume that each of the two key writes in an update—to a pointer and to the status indicator—is atomic with respect to crashes, but the order in which these two writes persist is immaterial: neither will be inspected in the wake of a crash unless the global epoch counter has advanced by 2.

Figure 7.12 displays two example updates. In Figure 7.9 an update to the bucket has occurred in epoch 5. In Figure 7.10 record \(g\) is added to the bucket in epoch 6. First, we initialize the new record to point to the most recent valid record, \(f\). Then, we change the status indicator to update pointer roles and the epoch number. As we are in epoch 6, the most recent committed record was added in epoch 4 (the previous in-flight pointer). Therefore, pointer 1 is now the committed pointer. The new in-flight pointer is the one pointing to the most recent record added in the previous epoch (pointer 0). The remaining pointer, pointer 2, whose target is older than the new committed pointer, is then assigned the active role and is retargeted to point to the newly prepended record, \(g\).

In Figure 7.11 an additional record, \(h\), is added to the bucket after a crash has occurred in epoch 6 (after the update of Figure 7.10). Because of the crash, epochs 5 and 6 are on the failure list. Records \(e\), \(f\), and \(g\) are thus failed records, because they were added during these epochs and cannot be
Figure 7.9: Initial state in epoch 5.

Figure 7.10: Adding record \( g \) in epoch 6.

Figure 7.11: Adding record \( h \) in epoch 7; epochs 5 and 6 have failed.

Figure 7.12: A sequence of Dalí updates.

relied upon to have persisted. The new record, \( h \), refers to the valid head \( d \) instead. Then, the status indicator is updated. The snapshot number \( SS \) becomes 7. The committed pointer is the one pointing to the most recent persistent record, \( d \). Pointer 1, which points to \( d \), is assigned the committed role. One currently invalid pointer (pointer 2) will point to the newly added record, \( h \). Since the previous epoch is a failed one, there are no in-flight records, so we set the in-flight role as invalid. The net effect is to transform the state of the bucket in such a way that the failed records, \( e \), \( f \), and \( g \), become unreachable.

### 7.3.4 Further Details

**Global Routines.** As noted in Section 7.3.1, our global fences are executed periodically by a special worker thread (or by a repurposed application thread that has just completed an operation). The worker first
increments and persists the global epoch counter under protection of a sequence lock \[119\]. It then waits for all threads to exit any transaction in the previous epoch, thereby ensuring that every update occurs entirely within a single epoch. (The wait employs a global array, indexed by thread ID, that indicates the epoch of the thread’s current transaction, or 0 if it is not in a transaction.) Finally, the worker initiates the actual whole-cache write-back. In our prototype implementation, this is achieved with a custom system call that executes the Intel \texttt{wbinvd} instruction. This instruction has the side effect of invalidating all cache content within a single socket. We hypothesize that future machines with persistent memory will provide an alternative instruction that avoids the invalidation and extends to multiple sockets.

Following a crash, a \textit{recovery procedure} is invoked. This routine reads the value, \(F\), of the global epoch counter and adds both \(F\) and \(F-1\) to the failed epoch list (and persists these additions). The crashed epoch, \(F\), is added because the fence that would have forced its writes-back did not start; the previous epoch, \(F-1\), is added because the fence that would have forced its writes-back may not have finished. Significantly, the recovery procedure does not delete or modify failed records in the hash chains: as illustrated in Figure \[7.11\], recovery is performed incrementally by application threads as they access data.

\textbf{Transactions.} Transactions are easily added on top of the basic Dalí design. Our prototype employs strong strict two-phase locking (SS2PL):
to perform a transaction that includes multiple hash map operations, a thread
acquires locks as it progresses, using timeout to detect (conservatively) dead-
lock with other threads. To preserve the ability to abort (when deadlock is
suspected), it buffers its updates in transient state. When it has completed
its code, including successful acquisition of all locks, it performs the buffered
updates, as described in Section 7.3.3 and releases all its locks.

In-place Updates. A reader executing in epoch $E$ is interested only
in the most recent update of a given key $k$ in $E$. If there are multiple records
for $k$ in $E$, only the most recent will be used. As a means of reducing memory
churn, we modify our update routine to look for a previous entry for $k$ in the
current epoch, and to overwrite its associated value, atomically and in place,
if it is found.

Multiversioning. Because historical versions are maintained, we
can execute read-only operations efficiently, without the need for locking, by
pretending that readers execute two epochs in the past, seeing the values
that would persist after a crash. This optimization preserves serializability
but not strict serializability. It improves throughput by preventing readers
from interfering with concurrent update transactions. To ensure consistency,
read-only transactions continue to participate in the global array that stalls
updates in a new epoch until transactions from the previous epoch have
Garbage Collection. Garbage collection recycles obsolete records that are no longer needed because newer persistent records with the same key exist; it operates at the granularity of a bucket. At the end of an update operation, before releasing the bucket’s lock, a thread will occasionally peruse the committed records and identify any for which there exists a more recent committed record with the same key. Removal from the list entails a single atomic pointer update, which is safe as the bucket is locked. Once the removal is persistent (two epochs later), the record can safely be recycled. If memory pressure is detected, we can use incremental persistence to free the record immediately. Otherwise we keep the record on a “retired” list and reclaim it in the thread’s first operation two epochs hence.

Because the retired list is transient, we must consider the possibility that records may be lost on a crash, thereby leaking memory. Similar concerns arise when bypassing failed records during an update operation, as illustrated in Figure 7.10 and when updating the free list of the memory allocator itself. To address these concerns, we can end the recovery procedure with a sweep of the heap that reclaims any node not found on a bucket list [12]. Since the amount of leakage is likely to be small, this need not occur on every crash.
7.4 Correctness

We here present an informal proof of Dalí’s safety. Specifically, we argue that it satisfies buffered durable linearizability \[95\], an extension of traditional linearizability that accommodates whole-system crashes. For clarity of exposition (and for lack of space), we consider only read and update operations, omitting garbage collection, in-place updates, multiversioning, and transactions. We begin by arguing that a crash-free parallel history of Dalí is linearizable. We then show that the operations preserved at a crash represent a consistent cut of the history prior to the crash, so that when crashes and lost operations are removed from the history, what remains is still linearizable.

7.4.1 Linearizability

The code of Figures 7.6 and 7.7 defines a notion of valid head for a Dalí bucket. Let us say that a bucket is well formed if valid head points to a finite, acyclic list of nodes. We define the valid content of a well-formed bucket to comprise the initial occurrences of keys on this list, together with their associated values.

**Theorem 4.** In the absence of crashes, Dalí is a linearizable implementation of an unordered map.

**Proof.** All Dalí operations on the same bucket acquire the bucket’s lock; by excluding one another in time they trivially appear to take effect atomically
at a point between their invocation and response. While the roles of the various pointers may rotate at epoch boundaries, inspection of the code in Figure 7.7 confirms that, in the absence of crashes, each newly created node in `update` links to `ptrs[a]` (which is always `valid_head`), and `ptrs[a]` is always updated to point to the new node. A trivial induction (starting with initially empty content) shows that this prepending operation preserves both well formedness and the desired sequential semantics.

7.4.2 Buffered Durable Linearizability

Buffered durable linearizability [95] extends linearizability to accommodate histories with “full-system” crashes. Such crashes are said to divide a history into eras, with no thread executing in more than one era[^1]. Information is allowed to be lost in a crash, but only in a consistent way. Specifically, if event $e_1$ happens before event $e_2$ ($e_1 <_{hb} e_2$—e.g., $e_1$ is a store and $e_2$ is a load that sees its value), then $e_1$ cannot be lost unless $e_2$ is also.

Informally, a history is buffered durably linearizable (BDL) if execution in every era can be explained in terms of information preserved from the consistent cut of the previous era. More precisely, history $H$ is BDL if, for every era ending in a crash, there exists a happens-before consistent cut of the events in that era such that for every prefix $P$ of $H$, the history $P'$ is linearizable, where $P'$ is obtained from $P$ by removing all crashes and, in all eras other than the last, all events that follow the cut. A concurrent object

[^1]: With apologies to geologists, eras here are generally longer than epochs.
CHAPTER 7. DALÍ: A PERIODICALLY PERSISTENT HASH MAP

or system is BDL if all of its realizable histories are.

Our BDL proof for Dalí begins with the following lemma:

**Lemma 6.** An epoch boundary in Dalí represents a consistent cut of the happens-before relation on the hash map.

*Proof.* Straightforward: The worker thread that increments the epoch number does so under protection of a sequence lock, and it doesn’t release the lock until (a) no thread is still working in the previous epoch and (b) the new epoch number has persisted (so no thread will ever work in the previous epoch again).

Suppose now that we are given a history $H$ comprising read, update, and epoch boundary events, where some of the epoch boundaries are also marked as crashes. The two epochs immediately preceding a crash are said to have failed; the rest are successful. An update operation is said to be successful if it occurs in a successful epoch and to have failed otherwise. Let us define the “valid content” of bucket $B$ at a point between events in $H$ to mean “a singly linked chain of update records reflecting all and only the successful updates to $B$ prior to this point in $H$.” The following is then our key lemma:

**Lemma 7.** For any realizable history $H$ of a Dalí bucket $B$, and any prefix $P$ of $H$ ending with a successful update $u$, $\text{ptrs}[a]$ will refer to valid content immediately after $u$.

*Proof.* By induction on successful updates. We can ignore the reads in $H$ as they do not change state. As a base case, we adopt the convention that the
CHAPTER 7. DALÍ: A PERIODICALLY PERSISTENT HASH MAP

 initial state of $B$ represents the result of a successful initialization “update.”

The lemma is trivially true for the history prefix consisting of only this single “update,” at the end of which $\texttt{ptrs}[a]$ is $\texttt{NULL}$.

Suppose now that for some constant $k$ and all $0 \leq i < k$, the lemma is true for all prefixes $P_i$ ending with the $i$th successful update, $u_i$. We want to prove that the lemma is also true for $P_k$. First consider the case in which there is no crash between the previous successful update, $u_{k-1}$, and $u_k$. By the same reasoning used in the proof of Theorem 4, $u_k$ will prepend a new record onto the chain at $\texttt{ptrs}[a]$, preserving valid content.

If there is at least one crash between $u_{k-1}$ and $u_k$, there must clearly be at least two failed epochs between them. This means that the valid content as of the end of $u_{k-1}$ will have persisted as of the beginning of $u_k$—its chain will be intact. We wish to show that no changes to the pointers and status indicator that occur between $u_{k-1}$ and $u_k$—caused by any number of completed or partial failed updates—can prevent $u_k$ from picking up and augmenting $u_{k-1}$’s valid content. We do so by reasoning on the transitions enumerated in Figure 7.8.

Let $E_{k-1}$ denote the epoch of $u_{k-1}$ and $E_k$ the epoch of $u_k$. We note that all failed updates between $u_{k-1}$ and $u_k$ occur in epochs numbered greater than $E_{k-1}$. Further, let $v$ denote the value of $a$ (0, 1, or 2) immediately after $u_{k-1}$. Any update that sees the state generated by $u_{k-1}$ will use row 2, 3, or 5 of Figure 7.8 and will choose, as its “new $a$” a value other than $v$. Over the course of subsequent failed updates before $u_k$, $\texttt{ptrs}[v]$’s role may transition.
CHAPTER 7. DALÍ: A PERIODICALLY PERSISTENT HASH MAP

179

at most twice, from a to f to c. As a consequence, the code of Figure 7.7 will never change the value of 

\texttt{ptrs[v]} — that pointer will continue to reference \( u_{k-1} \)'s valid content until the beginning of \( u_k \).

Reasoning more specifically about the ID roles, a status indicator change persisted by a failed update that happens in epoch \( E_{k-1} + 1 \) will, by necessity, make \( \texttt{ptrs[v]} \) the in-flight pointer. A subsequent update that sees this change in epoch \( E_{k-1} + 2 \) or later will by necessity make \( \texttt{ptrs[v]} \) the committed pointer. Alternatively, a failed update in epoch \( E_{k-1} + 2 \) or later, without having seen a previous failed update in epoch \( E_{k-1} + 1 \), will also make \( \texttt{ptrs[v]} \) the committed pointer. A subsequent update that sees this change will leave \( \texttt{ptrs[v]} \)'s role alone. The net result of all these possibilities is that \( u_k \) will chose \( \texttt{ptrs[v]} \) as the \texttt{valid_head} regardless of which failed update's status indicator is read. It will then copy this value to the \texttt{next} field of its new node and point \( \texttt{ptrs[a]} \) at that node, preserving valid content. \qed

\textbf{Theorem 5.} Dalí is a buffered durably linearizable implementation of an unordered map.

\textit{Proof.} Straightforward: Given history \( H \), containing crashes, we choose as our cut in each era the end of the last successful epoch. In the era that follows a crash, the visible content of each bucket (the records that will be seen by an initial \texttt{read} or \texttt{update}) will be precisely the valid content of that bucket. \qed
CHAPTER 7. DALÍ: A PERIODICALLY PERSISTENT HASH MAP

7.5 Experiments

We have implemented a prototype version of Dalí in C/C++ with POSIX threads. As described in Section 7.3.4, we implemented the global fence by exposing the privileged WBINVD instruction to user code using a syscall into a custom kernel module. The WBINVD instruction invalidates all caches within a single socket and blocks all processors within the socket until completion. Since non-volatile memory is not yet widely available, we simulated NVM by memory mapping a tmpfs file into Dalí’s address space. This interface is consistent with industry projections for NVM [184].

As a representative workload for a hash map, we chose the transactional version of the Yahoo! Cloud Serving Benchmark (YCSB) [35, 42]. Each thread in this benchmark performs transactions repeatedly, for a given period of time. Keys are 8 bytes in length, and are drawn randomly from a uniform distribution of 100 million values. Values are 1000 bytes in length. We initialize the map with all keys in the key range.

The tested version of Dalí uses both mentioned optimizations (in-place updates and multiversioning) and our prototype SS2PL transaction processing system; the performance effects of the optimizations are important but not evaluated here. Garbage collection is enabled. Epoch duration is a configurable parameter in Dalí; our experiments use a duration of 100 ms. We compared Dalí with three alternative maps: Silo [189], FOEDUS [108], and an incrementally persistent hash map (IP).
Silo\cite{189} is an open source in-memory database for large multi-core machines.\footnote{https://github.com/stephentu/silo} It is a log-based design that maintains both an in-memory and a disk-resident copy. A decentralized log, maintained by designated logging threads, is used to commit transactions. We configured Silo to use NVM for persistent storage—i.e., Silo writes logs to main memory instead of disk.

**FOEDUS**\cite{108} is an online transaction processing (OLTP) engine, available as open source.\footnote{https://github.com/HewlettPackard/foedus} The engine is explicitly designed for heterogeneous machines with both DRAM and NVM. Like Silo, FOEDUS is a log-based system with both an transient and persistent copy of the data. Unlike Silo, FOEDUS adopts a dual paging strategy in which a logical page may exist in two physical forms: a mutable volatile page in DRAM and an immutable snapshot page in NVM. FOEDUS commits transactions with the aid of a decentralized logging scheme similar to Silo. FOEDUS offers both key-ordered and unordered storage, based respectively on a B-tree variant and a hash map; our experiments use the latter. Like Dalí, both Silo and FOEDUS may lose recent transactions on a crash (their decentralized logs are reaped into persistence in the background).

We also implemented a data store called **IP**, an incrementally persistent hash map\cite{178}, as described in Section 7.2. As in Dalí, transactions in IP are implemented using SS2PL. To ensure correct recovery, per-thread undo logging is employed. In contrast to Dalí, Silo, and FOEDUS, transactions
are immediately committed to persistence.

We benchmarked all four systems on a server-class machine with four Intel Xeon E7-4890 v2 processors, each with 15 cores, running Red Hat Enterprise Linux Server version 7.0. The machine has 3 TB of DRAM main memory. Each processor has a 37.5 MB shared L3 cache, and per-core private L2 and L1 caches of 256 KB and 32 KB, respectively.

Figure 7.13 shows the transaction throughput of Dalí and the comparison systems while varying the number of worker threads from 1 to 60; transactions here comprise three reads and one write. Dalí achieves a throughput improvement of 2–3× over Silo and FOEDUS across the range of threads. The removal of write-ordering overhead in Dalí reduces the time spent blocking per transaction, thereby improving throughput.

Figure 7.14 shows experiments that vary the read-to-write ratio at 60 threads across transactions containing four operations. Dalí’s performance advantages compared to Silo and FOEDUS are larger for workloads with more reads due to the multiversioning optimization, whereas IP’s advantage lies in the reduction in persist instructions at high read percentages.

One possible downside to NVM relative to DRAM is cell endurance. While STT-MRAM is expected to be relatively durable, PCM has endurance capabilities slightly better than flash. Fortunately, periodic persistence, relative to incremental persistence, can reduce cell wear during collisions. In our experiments, the maximum write speed of IP to a single bucket peaks at 1.6 Mops/sec. Assuming all operations access a single bucket, and a PCM en-
durance of $10^8$ writes \[59\], IP would wear out the head pointer’s PCM cell in about 100 seconds — clearly it requires hardware wear-leveling. In contrast, in periodic persistence, the head is updated at most once per epoch. With an epoch duration of 100 ms, the maximum number of writes per second to a single NVM location in Dalí is ten. Again assuming PCM main memory and maximum contention on a single bucket, Dalí would wear out the head pointer’s PCM cell in $10^7$ seconds, or about a third of a year. This degenerate situation could easily be detected and fixed via redirection at some point in that period.

For the mixed workload, assuming minimal hash collisions, Based on these experiments, we can extrapolate with regards to NVM

### 7.6 Related Work

Dalí builds upon years of research on in-memory and NVM-centric designs, and upon decades of research on traditional database and multiversioning algorithms.

Like Dalí, traditional disk-resident databases maintain a single persistent copy of the data (traditionally on disk, but for Dalí in NVM) and must move data into transient storage (traditionally DRAM, but for Dalí CPU caches) in order to modify it. Viewed in this light, CPU caches in Dalí resemble a database’s STEALING, FORCEABLE buffer cache \[66\]. The updating algorithm of the incrementally persistent hash map is similar to
Figure 7.13: Scalability (75% reads).

Figure 7.14: Impact of read:write ratio on Dalí throughput.
tradiotional shadow paging [62, 208], but at a finer granularity. To the best of our knowledge, no prior art in this space has allowed writes to be reordered within an update or transaction, as Dalí does.

The prepend-only buckets of Dalí resemble several structures designed for RCU [144]. Dalí also resembles work on persistent data structures, where “persistent” here refers to the data structure’s ability to preserve its own history [44]. Data structures of this sort are widely used in functional programming languages, where their ability to share space among multiple versions provides an efficient alternative to mutating a single version [156]. In the notation of this field, Dalí resembles a partially persistent data structure—one in which earlier versions can be read but only the most recent state can serve as the basis for new versions [44].

In NVM software, Dalí stands in contrast to various failure atomicity systems (e.g. [24, 26, 32]) and durable data structures (e.g. [26, 29, 159, 178, 207]) that use incremental persistence. A more novel failure atomicity system is SoftWrAP, which uses aliasing to keep both a transient and a persistent copy of data, thus avoiding inconsistencies caused by leaking cache lines [58].

7.7 Conclusion

We have introduced periodic persistence as an alternative to the incremental persistence employed by most previous data structures designed for non-volatile memory. Dalí, our periodically persistent hash map, executes neither
explicit writes-back nor persistence fences within updates; instead, it tracks the recent history of the map and relies on a periodic global fence to force recent changes into persistence. Experiments with a prototype implementation suggest that Dalí can provide nearly twice the throughput of file-based or incrementally persistent alternatives. We speculate other data structures could be adapted to periodic persistence, and that the paradigm might be adaptable to traditional disk based architectures.
Chapter 8

Conclusion

This work has presented several novel designs, concepts, and design philosophies for using nonvolatile memory. It is our hope that they will be useful in the coming years to enable programmers to exploit the promise of the technology.

The engineering effort required to give the application programmer safe, fine-grained, fast, and reliable access to NVM storage is only beginning. Important open topics in NVM include memory safety, language and compiler integration, OS abstractions, and, of course, crash consistency. We here highlight some important open questions for NVM systems software.

**Memory Safety** The most immediate concern in achieving usable byte-addressable NVM is memory safety. Failure atomicity systems can protect
durable data from power outages and other fail-stop errors using ACID semantics, but leave this same data vulnerable to memory corruption from software errors. If we expect the world to use NVM for durable storage, we must be able to protect persistent data from stray writes issued by buggy client applications, while allowing safe access to this same data by (presumably) a trusted user-level library. Since NVM necessitates hardware changes to ensure consistency, what additional hardware primitives should we use to protect persistent memory regions? Or can we leverage existing ISAs to provide high-performance and safe access to these regions by being creative? This problem remains a critical gap in the literature, and is an essential problem to be solved if NVM is to become an acceptable alternative to file I/O.

**Language and Compiler Integration**  Compiler and language awareness of the benefits and pitfalls of NVM is also in its infancy. Some semantic models exist for the ordering and timing of writes-back from caches to NVM, but no in-depth theoretical study exists. What characterizes these “persistency models,” and are some insufficiently strong? Are some persistency models incompatible with certain consistency models? On a more practical level, languages currently interact with NVM via libraries; very little has been done to explore language extensions and compiler-optimized NVM updates. What language-level constructs can be used to distinguish between persistent data stored in NVM and transient data stored in DRAM? Can compilers
reduce the cost of persistent updates by eliminating redundancy, or by using compression? Can compilers automatically generate code to restart the process after a crash? Given that NVM writes are expected to be somewhat slower than reads, what compiler optimizations are worth reinvestigating? Or, since some varieties of NVM tend to have lower write endurance than DRAM, can we use compilers to spread writes across the heap to minimize wear-out? Answering even some of these questions would significantly lower the programming effort needed to begin using NVM, and would allow the technology to be used by all classes of programmers.

**OS Abstractions** Exposing NVM memory regions as an OS abstraction requires the operating system to explicitly manage the region and provide some support to the user. How do we allocate within the region, and should the operating system manage garbage collection after a crash? How do we map the region into the address space, and what do we do about region name or address clashes? How can processes share a region and must they map it to the same address? How can we send persistent regions from one machine to another and ensure compatibility? However an operating system decides to answer these questions, the solutions will have major ramifications on the design and capabilities of user-level software.
**Crash Consistency**  Ensuring consistent NVM state in the wake of a crash is still important, and the development of failure atomicity systems will continue. It is likely worth drawing inspiration from other fields. In particular, it would be interesting to extend the periodic persistence design philosophy into failure atomicity systems.

**Internet of Things**  Looking farther afield, NVM has implications for intermittently powered devices either in the mobile space or as part of the Internet of Things. Devices that harvest energy from their surroundings must be prepared to lose power at any moment, but should be able to make progress regardless. Optimizing energy-aware and failure atomicity systems for these devices is likely to be a critical step in the development of the Internet of Things.
Appendix A

Other Works

Over the course of this dissertation, a fair amount of work was done exploring problems in concurrency without direct applicability to nonvolatile memory. These projects are listed here, with a brief description of the innovations and findings.

A.1 Performance improvement via Always-Abort HTM

Several research groups have noted that hardware transactional memory (HTM), even in the case of aborts, can have the side effect of warming up the branch predictor and caches, thereby accelerating subsequent execution.

\[\text{\footnote{This section represents work published by Joseph Izralevitz, Lingxiang Xiang, and Michael L. Scott. Performance improvement via always-abort HTM. In: PACT '17. [99]}}\]
APPENDIX A. OTHER WORKS

We propose to employ this side effect deliberately, in cases where execution must wait for action in another thread. In doing so, we allow “warm-up” transactions to observe inconsistent state. We must therefore ensure that they never accidentally commit. To that end, we propose that the hardware allow the program to specify, at the start of a transaction, that it should in all cases abort, even if it (accidentally) executes a commit instruction. We discuss several scenarios in which always-abort HTM (AAHTM) can be useful, and present lock and barrier implementations that employ it. We demonstrate the value of these implementations on several real-world applications, obtaining performance improvements of up to 2.5× with almost no programmer effort.

A.2 An Unbounded Nonblocking Double-ended Queue

This work introduces a new algorithm for an unbounded concurrent double-ended queue (deque). Like the bounded deque of Herlihy, Luchangco, and Moir [79] on which it is based, the new algorithm is simple and obstruction free, has no pathological long-latency scenarios, avoids interference between operations at opposite ends, and requires no special hardware support beyond the usual compare-and-swap. To the best of our knowledge, no prior concurrent deque combines these properties with unbounded capacity, or provides

\footnote{This section represents work published by Matthew Graichen, Joseph Izraelevitz, and Michael L. Scott. An unbounded nonblocking double-ended queue. In: ICPP ’16. [61]}
consistently better performance across a wide range of concurrent workloads.

A.3 Generality and Speed in Nonblocking Dual Containers

Nonblocking dual data structures extend traditional notions of nonblocking progress to accommodate partial methods, both by bounding the number of steps that a thread can execute after its preconditions have been satisfied and by ensuring that a waiting thread performs no remote memory accesses that could interfere with the execution of other threads. A nonblocking dual container, in particular, is designed to hold either data or requests. An *insert* operation either adds data to the container or removes and satisfies a request; a *remove* operation either takes data out of the container or inserts a request.

We present the first general-purpose construction for nonblocking dual containers, allowing any nonblocking container for data to be paired with almost any nonblocking container for requests. We also present new custom algorithms, based on the LCRQ of Morrison and Afek, that outperform the fastest previously known dual containers by factors of four to six.

---

3 This section represents work published by Joseph Izraelevitz and Michael L. Scott. *Generality and Speed in Nonblocking Dual Containers.* In: TOPC ’17.
A.4 Implicit Acceleration of Critical Sections via Unsuccessful Speculation

The speculative execution of critical sections, whether done using HTM via the transactional lock elision pattern or using a software solution such as STM or a sequence lock, has the potential to improve software performance with minimal programmer effort. The technique improves performance by allowing critical sections to proceed in parallel as long as they do not conflict at run time. In this work we experimented with software speculative executions of critical sections on the STAMP benchmark suite and found that such speculative executions can improve overall performance even when they are unsuccessful — and, in fact, even when they cannot succeed.

Our investigation used the Oracle Adaptive Lock Elision (ALE) library which supports the integration of multiple speculative execution methods (in hardware and in software). This software suite collects extensive performance statistics; these statistics shed light on the interaction between these speculative execution methods and their effect on performance. Inspection of these statistics revealed that unsuccessful speculative executions can accelerate the performance of the program for two reasons: they can significantly reduce the time the lock is held in the subsequent non-speculative execution of the critical section by prefetching memory needed for that execution; additionally, they affect the interleaving between threads trying to acquire the

---

4This section represents work published by Joseph Izraelevitz, Yossi Lev, and Alex Kogan. Implicit Acceleration of Critical Sections via Unsuccessful Speculation. In: TRANSACT '16. 92
APPENDIX A. OTHER WORKS

lock, thus serving as a back-off and fairness mechanism. This paper describes our investigation and demonstrates how these factors affect the behavior of multiple STAMP benchmarks.

A.5 Interval-Based Memory Reclamation

In this paper we present interval based reclamation (IBR), a new approach to safe reclamation of disconnected memory blocks in nonblocking concurrent data structures. Safe reclamation is a difficult problem: a thread, before freeing a block, must ensure that no other threads are accessing that block; the required synchronization tends to be expensive. In contrast with epoch-based reclamation, in which threads reserve all blocks created after a certain time, or pointer-based reclamation (e.g., hazard pointers), in which threads reserve individual blocks, interval-based reclamation allows threads to reserve all blocks known to have existed in a bounded interval of time. By comparing a thread’s reserved interval with the lifetime of a detached but not yet reclaimed block, the system can determine if the block is safe to free. Like hazard pointers, IBR avoids the possibility that a single stalled thread may reserve an unbounded number of blocks; unlike hazard pointers, it avoids a memory fence on most pointer-following operations. It also avoids the need to explicitly “drop” a no-longer-needed pointer, making it simpler to use.

\footnote{This section represents work to be published by Hensen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-Based Memory Reclamation. In: PPoPP ’18.}
This paper describes three specific interval-based reclamation schemes (one with several variants) that trade off performance, applicability, and space requirements.
Bibliography


