Modernizing Models and Management of the Memory Hierarchy for Non-Volatile Memory

Charles John McGuffey

CMU-CS-21-109

May 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Phillip B. Gibbons, Chair
Guy Blelloch
Nathan Beckmann
Michael Bender (Stony Brook University, New York)
Julian Shun (Massachusetts Institute of Technology)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2021 **Charles John McGuffey**

This research was sponsored by the National Science Foundation under grant numbers CCF-1533858 and CCF-1919223.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Computer Science, Parallelism, Non-Volatile Memories, Cache Replacement

For my family.

Abstract

Non-volatile memory technologies (NVMs) are a new family of technologies that combine near memory level performance with near storage level cost density. The result is a new type of memory hierarchy layer that exists and performs somewhere between the two. These new technologies offer many opportunities for performance improvement, but in order to take advantage of these system design needs to account for their particular characteristics.

In this thesis, we focus on how to design memory management and caching systems for NVMs. Our work is broken into three major categories targeting different primary performance metrics.

1. We study how to design algorithms and memory management to achieve fault tolerance with low cost and efficient recovery using NVMs.
2. We design an extension to the traditional model of caching to account for data writes in order to improve NVM device lifetime and energy consumption.
3. We investigate how to improve throughput in caches by taking advantage of granularity change in the memory hierarchy.

Throughout our work we rely on a blend of theoretical and practical approaches. We provide models for processor faults, cache writebacks, and cache-storage communication that isolate the targeted effects from orthogonal complications. For each model, we show worst case theoretical bounds for our algorithms along with proofs that explain how the benefits are derived. We then take our results and provide empirical evaluations to show their effectiveness in practice. We believe that our ideas and approach provide a solid foundational study on memory hierarchy design in the era of non-volatile memories.

Acknowledgments

Thanks to Phil for the guidance and mentoring he has given me over the years.
Thanks to my collaborators for all of their ideas and enthusiasm.
Thanks to my friends and family for all of their love and support.

Contents

1 Introduction **1**

2 Program Persistence **3**
2.1 Model Definition . . . 5
2.1.1 Single Processor . . . 5
2.1.2 Multiple Processors . . . 6
2.2 Robustness on a Single Processor . . . 6
2.3 Programming for Robustness . . . 9
2.4 Robustness on Multiple Processors . . . 11
2.5 Work Stealing . . . 14
2.5.1 Preliminaries and Overview . . . 14
2.5.2 Proof of the Correctness of Work-Stealing . . . 20
2.5.3 Time Bounds . . . 29
2.6 Fault-Tolerant Algorithms . . . 31
2.7 Chapter Summary . . . 33

3 Writeback-Aware Caching **35**
3.1 Problem Formulation . . . 37
3.1.1 Traditional Caching . . . 37
3.1.2 Writeback-Aware Caching . . . 38
3.2 Writeback-Aware Landlord . . . 38
3.2.1 Algorithm Description . . . 38
3.2.2 Proof of Optimality . . . 40
3.3 Offline Complexity Results . . . 42
3.3.1 NP-Completeness . . . 42
3.3.2 Max SNP-Hardness . . . 43
3.4 Approximations with Theoretical Guarantees . . . 45
3.4.1 Analyzing the Writeback-Oblivious Optimal . . . 45
3.4.2 A 2-Approximation for Savings . . . 46
3.5 Efficient Approximations for Practical Use . . . 47
3.5.1 A Lower Bound for Optimal . . . 47
3.5.2 An Upper Bound for Optimal . . . 47
3.6 Experimental Evaluation . . . 49
3.6.1 Methodology . . . 49

3.6.2 Results 50
3.7 Chapter Summary 52

4 Block-Granularity-Aware Caching **53**
4.1 Problem Formulation 55
4.2 Complexity Analysis 56
4.3 Competitive Lower Bound 58
4.3.1 Item Caches 58
4.3.2 Block Caches 59
4.3.3 Generalizing the Lower Bound 60
4.3.4 Analysis and Discussion 61
4.4 A Competitive Policy for the Block-Granularity-Aware Caching Problem 63
4.4.1 Policy Description 63
4.4.2 The Upper Bound 64
4.4.3 Applying the Bound 69
4.5 Chapter Summary 71

5 Conclusion **73**

Bibliography **75**

List of Figures

2.1 The Parallel Persistent Memory Model 6
2.2 CAM Capsule Example 13
2.3 Fault-tolerant WS-Deque Implementation 19
2.4 Entry State Transition Diagram 19

3.1 Writeback-Aware Landlord Pseudocode 39
3.2 Writeback-Aware Landlord Potential Function 40
3.3 Example 3D Matching to WA Caching Problem Conversion 43
3.4 An Example Trace that Breaks Stack Algorithms 46
3.5 Example WA Caching Problem to MCF Conversion 48
3.6 Cache Costs on MSR Storage Traces 50
3.7 Analyzing Frequency Heuristic 51
3.8 Writeback Cost Sensitivity Analysis 52

4.1 Example Variable-Size Caching to BGA Caching Problem Conversion 58
4.2 A Logical Diagram of IBLP 63
4.3 Adversarial Traces for IBLP 65
4.4 Comparing Bounds in the BGA Caching Problem 70
4.5 BGA Caching Problem Constant Size Partitioning Analysis 71

List of Tables

4.1 BGA Caching Problem Notation 64

Chapter 1

Introduction

In the modern era of computing, the amount of data available and the demands on performance are continuously increasing. This places increasing strain on the memory hierarchy, which is often the main bottleneck in computation [86]. To try and improve memory performance, companies are working on new *Non-Volatile Memories* (NVMs).

NVM technologies typically store data in the physical state of a material. There are currently versions available in both solid state drive (SSD) and memory module (DIMM) form factors [41, 69], and additional technologies are in development [87]. NVMs offer promising performance characteristics: speeds within an order of magnitude of existing random access memory (DRAM), low idle power consumption, large capacity (more bits per unit area than existing random access memory), and the ability to survive power outages and other failures without losing data [22, 29, 45, 69, 109, 110].

In addition to these promising performance characteristics, NVM technologies bring important distinctions from their counterparts in the memory hierarchy.

1. Unlike most memory technologies, data stored in NVMs can survive loss of power or other failures that cause a device to restart.
2. NVM devices require changing the state of the material to perform writes. This means that writes take additional time and energy, and that such devices will wear out after too many data writes.
3. NVM devices expose a cache line interface, but operate on a larger data granularity internally. This causes spatial locality to significantly impact performance.

In this thesis, we investigate how to design caching and memory management systems for NVM technologies. Our work is broken into sections that focus on different metrics that a system might target and how those metrics are affected by NVMs.

Program Persistence. Systems that value fault tolerance can take advantage of NVMs to provide durability at low cost. Since NVMs allow data in memory to survive power failures, data no longer have to travel all the way to storage to persist. This allows progress to be saved at a smaller granularity with less overhead. We study this opportunity and provide a paradigm for algorithms, scheduling, and memory management that provides theoretical guarantees for both consistency and progress while maintaining good performance.

Writeback-Aware Caching. Systems that value power consumption or device lifetime need to

account for the asymmetry between reads and writes if they wish to take advantage of NVMs. Writes to NVMs are more costly than reads in terms of latency, bandwidth, energy, and device lifetime. Unfortunately, traditional models of caching do not account for writes to memory. In this work, we extend the traditional caching model to account for the cost of writebacks, and provide an investigation of the resulting model. Our work provides both a thorough understanding of the theoretical problem and a practical grounding.

Block-Granularity-Aware Caching. Systems that want to maximize their total throughput need to be designed to take advantage of the characteristics of the devices that they run on. One of these currently understudied characteristics is how to deal with device granularity changes. NVMs occupy an important part in this space, sitting at the boundary between storage devices that often operate at a granularity of 4 KB pages and caches that operate on a granularity of 64 byte cache lines. We study how caching can help support this interface transition, developing a model for caches that exist on granularity transitions, and developing a theoretical framework for understanding that problem.

Combining these works into a larger framework results in the following thesis: **When designing systems for non-volatile memory technologies, designing for specific performance metrics and accounting for crucial device characteristics can provide asymptotic theoretical performance improvement and practical improvement to match.**

Chapter 2

Program Persistence

An important consequence of the prevalence and expansion of computing in the modern era is that systems are increasing in size and parallelism. In such systems, the probability that individual components fault is not negligible, with exascale systems expected to have multiple failures each day [28]. In such systems, being able to handle faults gracefully and at low cost is critical to overall performance.

Traditionally, checkpointing and other fault handling techniques managed this by periodically storing data to storage or other redundant systems. However, non-volatile memories provide data persistence in memory. This means that data can be persisted at much lower cost in terms of energy and device bandwidth, and that a large fraction of the active data is persisted by default. This is a huge opportunity to develop new techniques for fault tolerance that reduce both the amount of progress lost to a fault and the cost of maintaining that tolerance.

To investigate this opportunity, we define a parallel computational model, the *Parallel Persistent Memory (Parallel-PM) model*, that consists of a collection of processors with a fast local ephemeral memory of limited size, and sharing a large slower persistent memory. As in the external memory model [5, 7], each processor runs a standard instruction set from its ephemeral memory and has instructions for transferring blocks to and from the persistent memory. The cost of an algorithm is calculated based on the number of such transfers. A key difference, however, is that the model allows for individual processors to fault at any time. If a processor faults, all of its processor state and local ephemeral memory is lost, but the persistent memory remains. We consider both the case where the processor restarts (soft faults) and the case where it never restarts (hard faults). Our model captures a useful view of the interaction of NVM main memory with traditional memory technologies and allows us to provide insight into interesting and theoretically guaranteed fault tolerance.

Our Contributions. In this chapter, we study how to handle processor faults in systems with NVM or other forms of non-volatile main memories. We consider both how an individual processor should restart, and how to run a computation that can continue to make progress despite faults. We make the following contributions:

1. We define the *Persistent Memory Model*, a single processor model for processor faults and their effect on memory. We then define the *Parallel Persistent Memory Model*, which extends the Persistent Memory Model to support multiple processors. We use these models

to study how program progress is affected by faults, and how measures to tolerate faults affect performance in the faultless setting.

2. We identify a paradigm of breaking a computation into "capsules" that have no write-after-read conflicts (writing a location that was read earlier within the same capsule) to provide idempotent behavior in our single processor model. We then use this technique to implement RAM, external memory, and cache-oblivious algorithms [53] asymptotically efficiently in the model.
3. In pursuit of a technique with better performance in practice, we consider a programming methodology in which the algorithm designer can identify capsule boundaries to ensure that the capsules are free of write-after-read conflicts.
4. We extend our ideas to the Parallel Persistent Memory Model, and consider conditions under which programs are correct when the processors are interacting through the shared memory. We identify that if capsules are free of write-after-read conflicts and atomic, in a way that we define, then each capsule acts as if it ran once despite many possible restarts.
5. Our most significant result is a work-stealing scheduler, based on that of Arora, Blumofe, and Plaxton (ABP) [7], that can be used on the Parallel Persistent Memory Model. This scheduler (i) ensures that each stolen task gets executed despite faults, (ii) properly handles faults that cause processors to go down permanently, and (iii) remains efficient in the presence of soft or hard faults. We use our scheduler to show that any race-free, write-after-read conflict free, multithreaded fork-join program can be scheduled in bounded runtime. This bound differs from the ABP result only by a logarithmic factor on the depth term, due to possible faults along the critical path.
6. We apply our techniques to achieve algorithms for prefix-sum, merging, sorting, and matrix multiply that are idempotent in the Parallel Persistent Memory Model. The results for prefix-sums, merging, and sorting are work-optimal, matching lower bounds for the external memory model.

Related Work. When a processor crashes, writes that are still in the cache (have not been recorded in persistent memory) are lost while other writes are not. Prior work includes schemes for encapsulating updates to persistent memory in either *transactions* or *lock-protected failure atomic sections* and using various forms of (undo, redo, resume) logging to ensure correct recovery [17, 19, 30, 31, 32, 37, 43, 52, 55, 57, 62, 64, 65, 66, 77, 78, 80, 82, 88, 96, 111].

The intermittent computing community works on the related problem of small systems that will crash due to power loss [10, 25, 38, 39, 60, 83, 84, 108]. Lucia and Ransford [83] describe how faults and restarting lead to errors that will not occur in a faultless setting. Several of these works [38, 39, 83, 84, 108] break code into small chunks, referred to as *tasks*, and work to ensure progress at that granularity. Avoiding write-after-read conflicts is often the key step towards ensuring that tasks are idempotent. Because these works target intermittent computing systems, which are designed to be small and energy efficient, they do not consider multithreaded programs, concurrency, or synchronization.

In contrast to this flurry of systems research, there is relatively little work from the theory/algorithms community aimed at this setting [42, 67, 68, 93]. David et al. [42] presents concurrent data structures (e.g., for skip-lists) that avoid the overheads of logging. Izraelevitz et al. [67, 68]

presents efficient techniques for ensuring that the data in persistent memory captures a consistent cut in the happens-before graph of the program's execution. Nawab et al. [93] defines periodically persistent data structures, which combine mechanisms for tracking proper write ordering with a periodic flush of all cache lines to persistent memory. None of this work defines an algorithmic cost model, presents a work-stealing scheduler, or provides the provable bounds in this paper.

There is a very large body of research on models and algorithms where processors and/or memory can fault, but to our knowledge, none of it (other than the works mentioned above) fits the setting we study with its two classes of memory (local volatile and shared nonvolatile). Papers focusing on memory faults (e.g., [1, 34, 51] among a long list of such papers) consider models in which individual memory locations can fault. Papers focusing on processor faults (e.g., [9] among an even longer list of such papers) either do not consider memory faults or assume that all memory is volatile.

2.1 Model Definition

2.1.1 Single Processor

Our memory model has two layers: a small fast *ephemeral memory* of size M (in words) and a large slower *persistent memory* of size $M_p \gg M$, which are both partitioned into blocks of B words. We assume standard RAM instructions, as well as an *external read* that transfers a block from persistent memory into ephemeral memory, and an *external write* that transfers a block from ephemeral memory to persistent memory. These assumptions mirror those in the (M, B) external memory model [2].

Our fault model assumes that the processor can *fault* between any two external memory instructions with constant and independent probability f. After faulting, the processor *restarts*, with the ephemeral memory and processor registers in an arbitrary state, but the persistent memory in the same state as immediately before the fault. To enable forward progress, we assume there is a fixed persistent memory location referred to as the *restart pointer location*, containing a *restart pointer*, which is used to set the program counter on restart. The processor can update this pointer during execution. We view the computation as being partitioned into *capsules* that correspond to maximally contiguous sequences of instructions with the same restart pointer. We refer to writing a new restart pointer as *installing* a capsule, and assume that each restart pointer points to the beginning of its capsule. We define the *capsule work* to be the number of external reads and writes in the capsule, and the capsule whose restart pointer is installed as *active*.

Our cost model can be adapted to various instruction costs, but for this work we follow the external memory [2] and ideal cache [53] models in assuming that external reads and writes take unit cost and all other instructions have no cost. We assume that processor faults and restarts have constant cost, since the machine downtime is outside of software control and the restarting process can be made to take a constant number of external memory transfers.

In our analysis, we consider two ways to count the total cost. We say that the *faultless work* (or *work*), W, is the number of external memory transfers assuming no faults. We say that the *total work* (or *fault-tolerant work*), W_f, is the number of external transfers for an actual run

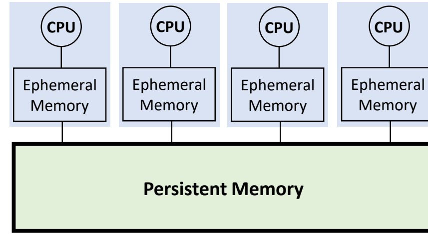


Figure 2.1: The Parallel Persistent Memory Model

including all transfers due to having to restart.

We refer to the result of aggregating these memory, fault, and cost models as the (single processor) (M, B) Persistent Memory Model (PM model).

2.1.2 Multiple Processors

The Parallel Persistent Memory Model (Parallel-PM model) consists of P processors each with its own fast local ephemeral memory of size M, but sharing a single slower persistent memory of size M_p (see Figure 2.1). Each processor works as in the single processor model, and the processors run asynchronously. Any processor can fault between instructions with probability f. If the processor restarts, using its own individual restart pointer location as in the single processor model, it is called a *soft fault*. We also allow for a *hard fault*, in which the processor never restarts—we say that such a processor is *dead*. We allow for concurrent reads and writes to the shared persistent memory, as well as a compare-and-swap (CAS) instruction. All of these operations are assumed to be sequentially consistent.

We consider the same form of multithreaded computations as considered by Arora, Blumofe, and Plaxton (ABP) [7]. In the model, a computation starts as a single thread. On each step, a thread can run an instruction, fork a new thread, or join with another thread. The (faultless) work W is the work summed across all threads in the absence of faults, and the total work W_f is the summed work including faults. In addition, we define the (faultless) *depth* D (and the *fault-tolerant* or *total depth* D_f) to be the maximum work (total work, respectively) along any path in the DAG.

Write-back Caches. Note that while the PM models are defined using explicit external read and external write instructions, they are also appropriate for modeling the (write-back) cache setting. Explicit instructions are used to ensure that external writes get to the persistent memory. Writes to local memory could end up being evicted from the cache and written back to persistent memory, but this does not affect correctness for programs that are race-free and well-formed, as defined in Section 2.2.

2.2 Robustness on a Single Processor

Our goal is to partition the computation into capsules in a way that ensures correctness regardless of faults. Specifically, we want each capsule to look from an external view like it has been run

exactly once after its completion, regardless of the number of times it was restarted. We say that a capsule is *idempotent* if, when it completes, all modifications to the persistent memory are consistent with running once from the initial state.

We say that a capsule has a *write-after-read conflict* if the first transfer from a block in persistent memory is a read (called an "exposed" read), and later there is a write to the same block. Such conflicts cause problems on restarts because the original input data may have been overwritten. We say a capsule is *well-formed* if the first access to each ephemeral word in the registers is a write. Being well-formed means that a capsule will not read the undefined values from registers and ephemeral memory after a fault. We say that a capsule is *write-after-read conflict free* if it is well-formed and had no write-after-read conflicts.

Theorem 1. *With a single processor, all write-after-read conflict free capsules are idempotent.*

Proof. On restarting, the capsule cannot read any persistent memory written by previous faults on the capsule, because we restart from the beginning of the capsule and the exposed read locations are disjoint from the write locations. Moreover, the capsule cannot read the state of the ephemeral memory because a write is required before a read (well-formedness). Therefore, the first time a capsule runs and every time a capsule restarts it has the same visible state, and because the processor instructions are deterministic, will repeat exactly the same instructions with the same results. □

We also show that it is possible to simulate models such as the RAM model, the external memory model, and the ideal cache model efficiently in the persistent memory model.

Theorem 2. *Any RAM computation taking t time can be simulated on the $(O(1), B)$ PM model with $f \leq 1/c$ for some constant $c \geq 2$, using $O(t)$ expected total work, for any B ($B = 1$ is sufficient).*

Proof. The simulation keeps all simulated memory in the persistent memory one word per block. It also keeps two copies of the registers in persistent memory, and the simulation swaps between the two. At the end of a capsule on one copy, it sets the restart pointer to a location just before the other copy. The code at that location, run at the start of the next capsule, copies the other copy of the registers into the current copy, and then simulates one instruction given by the program counter, by reading from the other copy of the registers, and writing to the current copy of registers (typically just a single register). The instruction might involve a read or write to the simulated memory, and an update of the program counter either to the next simulated instruction, or if a jump, to some other instruction. Once the instruction is done, the other copy of the registers is installed. This repeats. The capsules are write-after-read conflict free because they only read from one set of registers and write to the other, and the simulated memory instructions do a single read or write. Every simulated step takes a constant number of reads and writes to the persistent memory. Since the capsule work is constant, it can be bounded by some k. If $kf \leq 1/2$ then the probability of a capsule faulting is bounded by $1/2$ and therefore the expected total work on any capsule is upper bounded by $2k$. Setting $c = 2k$ gives the stated bounds. □

Theorem 3. *Any (M, B) external memory computation with t external accesses can be simulated on the $(O(M), B)$ PM model with $f \leq B/(cM)$ for some constant $c \geq 2$, using $O(t)$ expected total work.*

Proof. The simulation consists of rounds each of which has a *simulation* capsule and a *commit* capsule. It maps the ephemeral memory of the source program to part of the ephemeral memory, and the external memory to the persistent memory. It keeps the registers in the ephemeral memory, and keeps space for two copies of the simulated ephemeral memory and the registers in the persistent memory, which it swaps back and forth between.

The simulation capsule simulates some number of steps of the source program. It starts by reading in one of the two copies of the ephemeral memory and registers. Then during the simulation all instructions are applied within their corresponding memories, except for writes from the ephemeral memory to the persistent memory. These writes, instead of being written immediately, are buffered in the ephemeral memory. This means that all reads from the external memory have to first check the buffer. The simulation also maintains a count of the number of reads and writes to the external memory within a capsule. When this count reaches M/B, the simulation "closes" the capsule. The closing is done by writing out the simulated ephemeral memory, the registers, and the write buffer to persistent memory. For ephemeral memory and registers, this is the other copy from the one that is read. The capsule finishes by installing a commit capsule.

The commit capsule reads in the write buffer from the closed capsule to ephemeral memory, and applies all the writes to their appropriate locations of the simulated external memory in the persistent memory. When the commit capsule is done, it installs the next simulation capsule.

This simulation is write-after-read conflict free because the only writes during a simulation capsule are to the copy of ephemeral memory, registers, and write buffer. The write buffer has no conflicts since it is not read, and the ephemeral memory and registers have no conflicts since they swap back and forth. There are no conflicts in the commit capsules because they read from write buffer and write to the simulated external memory. The simulation is therefore write-after-read conflict free.

To see the claimed time and space bounds, we note that the ephemeral memory need only be a constant factor bigger than the simulated ephemeral memory because the write buffer can only contain M entries. Each round requires only $O(M/B)$ reads and writes to the persistent memory because the simulating capsules only need the stored copy of the ephemeral memory, do at most M/B reads, and then do at most $O(M/B)$ writes to the other stored copy. The commit capsule does at most M/B simulated writes, each requiring a read from and write to the persistent memory. Because each round simulates M/B reads and writes to external memory at the cost of $O(M/B)$ reads and writes to persistent memory, the faultless work across all capsules is bounded by $O(t)$. Because the probability that a capsule faults is bounded by the maximum capsule work, $O(M/B)$, when $f \leq B/(cM)$, there is a constant c such that the probability of a capsule faulting is less than 1. Since the faults are independent, the expected total work is a constant factor greater than the faultless work, giving the stated bounds. □

Theorem 4. *Any (M, B) ideal cache computation with t cache misses can be simulated on the $(O(M), B)$ PM model with $f \leq B/(cM)$ for a constant $c \geq 2$, using $O(t)$ expected total work.*

Proof. The simulation is similar to our external memory simulation, using rounds consisting of a *simulation* capsule and a *commit* capsule. During each simulation capsule a simulated cache of size $2M/B$ blocks is maintained in the ephemeral memory. The capsule starts by loading the

registers, and with an empty cache. During simulation, entries are never evicted, but instead the simulation stops when the cache runs out of space, i.e., after $2M/B$ distinct blocks are accessed. The capsule then writes out all dirty cache lines (together with the corresponding persistent memory address for each cache line) to a buffer in persistent memory, saves the registers and installs the commit capsule. The commit capsule reads in the buffer, writes out all the dirty cache lines to their correct locations, and installs the next simulation capsule. The simulation is write-after-read conflict free.

We now consider the costs of the simulation. $O(M)$ ephemeral memory is sufficient to simulate the cache of size $2M$. The total faultless work of a simulation capsule (run once) is bounded by $O(M/B)$ because we only have $2M/B$ misses before ending, and then have to write out at most $2M/B$ dirty cache blocks. Accounting for faults, the total cost is still $O(M/B)$— given constant probability of faults. The size of the cache and cost are similarly bounded for the commit capsule. We now note that over the same simulated instructions, the ideal cache will suffer at least M/B cache misses. This is because the simulation round accesses $2M/B$ distinct locations, and only M/B of them could have been in the ideal-cache at the start of the round, in the best case. The other M/B must therefore suffer a miss. Therefore each simulation round simulates what were at least M/B misses in the ideal cache model with at most $O(M/B)$ expected cost in the PM model. As in the previous proofs, and since the capsule work is bounded by $O(M/B)$, the probability of a capsule faulting can be bounded by $1/2$ when $f \leq B/(cM)$, for some c. Hence the expected total work can be bounded by twice the faultless work. □

2.3 Programming for Robustness

The simulation of external memory is not completely satisfactory because its overhead, although constant, could be significant. Here we describe one protocol to program directly for the model, which can greatly reduce the overhead.

Our protocol is designed so capsules begin and end at the boundaries of certain function calls. We call functions with capsule boundaries *persistent function calls* and those without ephemeral functions. We assume that all user code between persistent boundaries is write-after-read conflict free, or otherwise idempotent. In addition to the persistent function call we assume a `commit` command that forces a capsule boundary at that point. As with a persistent call, this command requires a constant number of external reads and writes.

We assume that all user code between persistent boundaries is write-after-read conflict free, or otherwise idempotent. This requires a style of programming in which results are copied instead of overwritten. For sequential programs, this increases the space requirements of an algorithm by at most a factor of two. Persistent counters can be implemented by placing a commit between reading the old value and writing the new. In the algorithms that we describe in Section 2.6, this style is very natural.

Implementing Persistent Calls. The standard stack protocol for function calling is not write-after-read conflict free and therefore cannot be used directly for persistent function calls. We therefore describe a simple mechanism based on closures and continuation passing in functional programming [4]. The convention also serves to clearly delineate the capsule boundaries, and

will be useful in the discussion of the parallel model. We then discuss how this can be implemented on a stack and can be used for loops.

We use a contiguous sequence of memory words, called a *closure*, to represent a capsule. The restart pointer for the capsule is the address of the first word of the closure. A closure consists of an instruction pointer in the first position (the start instruction), local state, arguments, and a pointer to another closure, which we refer to as the *continuation*. Typically a closure is constant size and points indirectly (via a pointer) to non-constant sized data, although this is not required. Once a closure is filled, it can be installed and started. Any faults while it is active will restart it. Since the base of the closure is loaded into the base pointer when starting, the instructions have access to the local state and arguments. A closure can be thought of as a stack frame, but need not be allocated in a stack discipline. Indeed, as discussed later, allocating in a stack discipline requires some extra care. The continuation can be thought of as a return pointer, except it does not point directly to an instruction, but rather another closure (perhaps the parent stack frame), with the instruction to run stored in the first location.

A persistent function call consists of creating two closures, a continuation closure and a callee closure, and then installing and starting the callee closure. The continuation closure corresponds to what needs to be run when returning from the callee. It consists of a pointer to the first instruction to run on return, any local variables that are needed on return, an empty slot for the return result of the callee, and the continuation of the current closure. The callee closure consists of a pointer to the first instruction to run in the called function, any arguments it needs, and a pointer to the continuation closure in its continuation. The return from a persistent call consists of writing any results into the closure pointed to by the continuation (the continuation closure), and then installing and starting the continuation closure. Note that if the processor faults after installing the continuation closure, then a computation will only back up as far as the start of the continuation. Therefore persistence occurs at the boundaries (in and out) of persistent function calls. Because a capsule corresponds to running a single installed closure, all capsules correspond to the code run between two persistent function boundaries. We note that if a function call is in tail position (i.e., it cannot call another function), then the continuation closure is not necessary, and the continuation pointer of the current active capsule can be copied directly to the new callee closure before installing it (this is the standard tail call optimization).

Our calling mechanism is write-after-read conflict free. This is because we only ever write to a closure when it is being created, and read when it is being used in a future capsule. The one exception is writing results into a closure, but in this case the callee does the writes, and the caller does the reads after the return and in a different capsule. A loop can be made persistent by using tail recursive function calls. To avoid allocating a new closure for each, the implementation could use just two closures and swap back and forth between the two.

Closures can be implemented in a stack discipline by allocating both the callee and continuation closures on the top of the stack, and popping the callee closure when returning from the called function, and the continuation closure when returning from the current function. Standard stack-based conventions, however, will not be write-after-read conflict free because they are based on side-affecting the current stack, e.g. by changing the value of local variables. Also the return address is typically stored in the child (callee) frame. Here it is important it is kept in the continuation closure so that the move to a new closure can be done atomically by swinging the restart-pointer (changing the start instruction address and base pointer on the same instruction).

A *commit* command can be implemented in the compiler, or by hand, by putting the code after the commit into a separate function body, and making a tail call to it.

To implement closures we need memory allocation. This can be implemented in various ways in a write-after-read conflict free manner. One way is for the memory for a capsule to be allocated starting at a base pointer that is stored in the closure. Memory is allocated one after the other, using a pointer kept in local memory (avoiding the need for a write-after-read conflict to persistent memory in order to update it). In this way, the allocations are the same addresses in memory each time the capsule restarts. At the end of the capsule, the final value of the pointer is stored in the closure for the next capsule. For the Parallel-PM model, each processor allocates from its own pool of persistent memory, using this approach. In the case where a processor takes over for a hard-faulting processor, any allocations while the taking-over processor is executing on behalf of the faulted processor will be from the pool of the faulted processor.

2.4 Robustness on Multiple Processors

With multiple processors our previous definition of idempotent is inadequate since the other processors can read or write persistent memory locations while a capsule is running. We therefore consider a stronger variant of idempotency that adds the requirement that capsules act as if they ran atomically. Atomicity is not necessary for correctness, but provides a simple definition that is sufficient for our purposes.

We consider the history of a computation, which is an interleaving of the persistent memory instructions from each of the processors. The history includes the additional instructions due to faults (i.e., it is a complete trace of instructions that actually happened). A capsule within a history is *invoked* at the instruction it is installed and *responds* at the instruction that installs the next capsule on the processor.

We say that a capsule in a history is *atomically idempotent* if all its instructions can be moved in the history to be adjacent somewhere between its invocation and response without violating the memory semantics (atomicity), and the instructions are idempotent at the spot they are moved to (idempotency).

We say that two persistent memory instructions on separate processors *conflict* if they operate on the same block and one is a write. For a capsule within a history we say that one of its instructions has a *race* if it conflicts with another instruction that is between the invocation and response of that capsule.

Theorem 5. *Any capsule that is write-after-read conflict free and race free in a history is atomically idempotent.*

Proof. Because the capsule is race free we can move its instructions to be adjacent at any point between the invocation and response without affecting the memory semantics. Once moved to that point, the idempotence follows from Theorem 1 because the capsule is write-after-read conflict free. □

This property is useful for user code if one can ensure that the capsules are race free via synchronization. We use this extensively in our algorithms. However, races are required for

program synchronizations. We therefore consider other conditions that are sufficient for atomic idempotence.

Racy Read Capsule. We first consider a *racy read capsule*, which reads one location from persistent memory and writes its value to another location in persistent memory. The capsul can have other instructions, but none of them that depend on the racy read are allowed to have races. A racy read capsule is atomically idempotent if all its instructions except for the read are race free. This is true because we can move all instructions of the capsule, with possible repeats due to faults, to the position of the last read. The capsule will then properly act like the read and write happened just once. Because races are allowed on the read location, there can be multiple writes by other processors of different values to the read location, and different such values can be read anytime the racy read capsule is restarted. However, because the dependent instructions are race free, no other processor can "witness" these possible writes of different values to the write location. Thus, the racy read capsule is atomically idempotent. A racy read capsule is a useful primitive for copying from a volatile location that could be written at any point into a processor private location that will be stable once copied. Then when the processor private location is used in a future capsule, it will stay the same however many times the capsule faults and restarts. We make significant use of this in our work-stealing scheduler.

Racy Write Capsule. We also consider a *racy write capsule*, for which the only instruction with a race is a write instruction to persistent memory, and the instruction races only with either read instructions or other write instructions, but not both kinds. Such a capsule can be shown to be atomically idempotent. In the former case (races only with reads), then in any history, the value in the write location during the capsule transitions from an old value to a new value exactly once no matter how many times the capsule is restarted. Thus, for the purposes of showing atomicity, we can move all the instructions of the capsule to immediately before the first read that sees the new value, or to the end of the capsule if there is no such read. Although the first time the new value is written (and read by other processors) may be part of a capsule execution that subsequently faulted, the effect on memory is as if the capsule ran just once without fault (idempotency). In the latter case (races only with other writes), then if in the history the racy write capsule is the last writer before the end of the capsule, we can move all the instructions of the capsule to the end of the capsule, otherwise we can move all the instructions to the beginning of the capsule, satisfying atomicity and idempotency.

Compare-and-Modify (CAM) Capsule. We now consider idempotency of the CAS instruction. Recall that we assume that a CAS is part of the machine model. We cannot assume the CAS is race free because the whole purpose of the operation is to act atomically in the presence of a race. Unfortunately efficiently simulate a CAS at the user level is non-trivial when there are faults. The problem is that a CAS writes two locations, the two that it swaps. In the standard non-faulty model one is local (a register) and therefore the CAS involves a single shared memory modification and a local register update. Unfortunately in the Parallel-PM model, the processor could fault immediately before or after the CAS instruction. On restart the local register is lost and therefore the information about whether it succeeded is lost. Looking at the shared location does not help since identical CAS instructions from other processors might have been applied to the location, and the capsule cannot distinguish its success from their success.

Instead of using a CAS, here we show how to use a weaker instruction, a *compare-and modify*

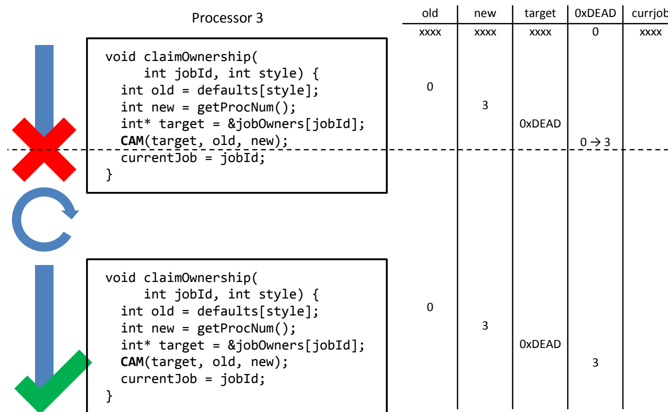


Figure 2.2: CAM Capsule Example. In CAM capsules, earlier faulting runs of the capsule may perform work that is visible to the rest of the system.

(CAM). A CAM is simply a CAS for which no subsequent instruction in the capsule reads the local result (i.e., the swapped value).[1] Furthermore, we restrict the usage of a CAM to protect against the ABA problem. For a capsule within a history we say a write w (including a CAS or CAM) to persistent memory is *non-reverting* if no other conflicting write between w and the capsule's response changes the value back to its value before w. We define a *CAM capsule* as a capsule that contains one non-reverting CAM and may contain other write-after-read conflict free and race free instructions.

Theorem 6. *A CAM capsule is atomically idempotent.*

Proof. Assume that the CAM is non-reverting and all other instructions in the capsule are write-after-read conflict free and race free. Due to faults the CAM can repeat multiple times, but it can only succeed in changing the target value at most once. This is because the CAM is non-reverting, so once the target value is changed, it could not be changed back. Therefore if the CAM ever succeeds, for the purpose of showing atomicity, in the history we move all the instructions of the capsule (including the instructions from faulty runs) to the point of the successful CAM. This does not affect the memory semantics because none of the other instructions have races, and any of the other CAMs were unsuccessful and therefore also have no affect on memory. At the point of the successful CAM the capsule acts like it ran once because it is write-after-read conflict free—other than the CAM, which succeeded just once. If the CAM never succeeds, the capsule is conflict free and race free because the CAM did not do any writes, so Theorem 5 applies. □

The example CAM capsule in Figure 2.2 shows one of the interesting properties of idempotence: unlike transactions or checkpointing, earlier runs that faulted can make changes to the memory that are seen or used by other processes. Similarly, these earlier runs can affect the results of the successful run, as long as the result is equivalent to a non-faulty run.

A CAM can be used to implement a form of test-and-set in a constant number of instructions. In particular, we will assume a location can either be *unset*, or the value of a process identifier or

[1]Some CAS instructions in practice return a boolean to indicate success; in such cases, the boolean cannot be read either.

other unique identifier. A process can then use a CAM to conditionally swap such a location from *unset* to its unique identifier. The process can then check if it "won" by seeing if its identifier is in the location. We make heavy use of this in the work-stealing scheduler to atomically "steal" a job from another queue. It can also be used at the join point of two threads in fork-join parallelism to determine who got there last (the one whose CAM from *unset* was unsuccessful) and hence needs to run the code after the join.

An alternative to the CAM approach is the work of Ben-David et al. [15], who examine the performance of CAS in more detail and provide a way to construct an idempotent version. Their construction could be applied in place of a CAM operation at the cost of added complexity and additional constant-factor overhead.

Racy Multiread Capsule. It is also possible to design capsules that are idempotent without the requirement of atomicity. By way of example, we discuss the *racy multiread capsule*. This capsule consists of multiple racy read capsules that have been combined together into a single capsule. Concurrent processes may write to locations that the capsule is reading between reads, which violates atomicity. Despite this, a racy multiread capsule is idempotent since the results of the final successful run of the capsule will overwrite any results of partial runs. We make use of the racy multiread capsule in the work-stealing scheduler to reduce the number of capsules required. It is not needed for correctness.

These capsules provide a strong foundation for building program synchronizations. In fact, our work-stealing scheduler is constructed using these capsules, illustrating that they provide for the majority of common use cases.

2.5 Work Stealing

2.5.1 Preliminaries and Overview

We provide an efficient work stealing scheduler (WS) for the Parallel-PM model based on the work-stealing scheduler of Arora, Blumofe, and Plaxton (ABP) [7]. Our scheduler, like theirs, uses a work-stealing double ended work queue and works in a multiprogrammed environment where the number of active processors can change.

However, supporting processor faults causes some crucial differences. First, our scheduler will not use the CAS operation, for reasons described in Section 2.4, and instead will make use of the CAM operation. Second, our scheduler has to handle soft faults anywhere in either the scheduler or the user program. This requires some care to maintain idempotence. Third, our scheduler has to handle hard faults. In particular it has to be able to steal from a processor that hard faults while it is running a thread. It cannot restart the thread from scratch, but needs to start from the previous capsule boundary (a thread can consist of multiple capsules). These faults can increase both the total work and the total depth. The expected work is only increased by a constant factor, which is not a serious issue. However, for total depth, expectations cannot be carried through the maximum implied by parallel execution. To deal with this, we consider high probability bounds.

Machine Assumptions. As in the ABP work, we require some assumptions about the machine. The schedule is a two-level scheduler in which the work-stealing scheduler, under our control,

maps threads to processes, and an adversarial operating system scheduler maps processes to processors. The OS scheduler can change the number of allocated processors and which processes are scheduled on those processors during the computation, perhaps giving processors to other users. The number of processes and the maximum number of processors used is given by P. The average number that are allocated to the user is P_A.

ABP assume that the quanta for scheduling is at least the time for two scheduling steps where each step takes a small constant number of instructions. In our case we cannot guarantee that the quanta is big enough to capture two steps since the processor could fault. However it is sufficient to show that with constant probability two scheduling steps complete within the quanta, which we can show.

The available instruction set contains a yield-to-all instruction. This instruction tells the OS that it must schedule all other processes that have not hard faulted before (or at the same time) as the process that executes the instruction. It is used to ensure that processors that are doing useful work have preference over ones who run out of work and need to steal.

The Scheduler Interface. For handling faults, and in particular hard faults, the interaction of the scheduler and threads is slightly different from that of ABP. We assume that when a thread finishes it jumps to the `scheduler`.[2] When a thread forks another thread, it calls a `fork` function, which pushes the new thread on the bottom of the work queue and returns to the calling thread. When the scheduler starts a thread it jumps to it (actually a capsule representing the code to run for the thread). Recall that when the thread is done it jumps back to the scheduler. These are the only interactions of threads and the scheduler—i.e. jumping to a thread from the scheduler, forking a new thread within a thread, and jumping back to the scheduler from a thread on completion. All of these occur at capsule boundaries, but a thread itself can consist of many capsules. We assume that at a join (synchronization) point of threads whichever one arrives last continues the code after the join and therefore that thread need not interact with the scheduler. The other threads that arrive at the join earlier finish and jump to the scheduler. In our setup, therefore, a thread is never blocked, assuming the `fork` function is non-blocking.

The Work-Stealing Deque. A work-stealing deque (WS-deque) is a concurrent deque supporting a limited interface. Here we used a similar interface to ABP, including the functions `popTop`, `pushBottom`, and `popBottom`. Any number of concurrent processors can execute `popTop`, but only the owner of a deque can execute either `pushBottom` or `popBottom`. The deque is linearizable except that `popTop` can return empty even if the deque is not-empty. This can only happen if another concurrent `popTop` call succeeds with a linearization point when the original `popTop` is live, i.e., from invocation to response.

We provide an implementation of a idempotent WS-deque in Figure 2.3. Our implementation maintains an array of tagged entries that refer to threads that the processor has either enabled or stolen while working on the computation. The tag is simply a counter that is used to avoid the ABA problem [59]. An *entry* consists of one of the following states:

- *empty*: An empty entry is one that has not been associated with a thread yet. Newly created elements in the array are initialized to empty.
- *local*: A local entry refers to a thread that is currently being run by the processor that owns

[2]Note that jumping to a thread is the same as installing a capsule.

this WS-Deque. We need to track local entries to deal with processors that have a hard fault (i.e., never restart).

- *job*: A job entry is equivalent to the values found in the original implementation of the WS-Deque. It contains a thread (i.e., a capsule to jump to start the thread).
- *taken*: A taken entry refers to a thread that has already been or is in the process of being stolen. It contains a pointer to the entry that the thief is using to hold the stolen thread, and the tag of that entry at the time of the steal.

The transition table for the entry states is shown in Figure 2.4.

In addition to this array of entries, we maintain pointers to the top and the bottom of the deque, which is a contiguous region of the array. As new threads are forked by the owner process, new entries will be added to the bottom of the deque and the bottom pointer will be updated using the `pushBottom` function. The top pointer will move down on the deque as threads are stolen. This implementation does not delete elements at the top of the deque, even after steals. This means that we do not need to worry about entries being deleted in the process of a steal attempt, but does mean that maintaining P WS-Deques for a computation with span T_∞ requires $O(PT_\infty)$ storage space.

Our implementation of the WS-Deque maintains a consistent structure that is useful for proving its correctness and efficiency. The entires of our WS-Deque are always ordered from the top to the bottom of the array as follows:

1. A non-negative number of taken entries. These entries refer to threads that have been stolen, or possibly in the case of the last taken entry, to a thread that is in the process of being stolen.
2. A non-negative number of job entries. These entries refer to threads that the process has enabled that have not been stolen or started since their enabling.
3. Zero, one, or two local entries. If a process has one local entry, it is the entry that the process is currently working on. Processes can momentarily have two local entries during the `pushBottom` function, before the earlier one is changed to a job. If a process has zero local entries, that means the process has completed the execution of its local work and is in the process of acquiring more work through `popBottom` or stealing, or it is dead.
4. A non-negative number of empty entries. These entries are available to store new threads as they are forked during the computation.

We can also relate the top and bottom pointers of the WS-Deque (i.e. the range of the deque) to this array structure. The top pointer will point to the last taken entry in the array if a steal is in process. Otherwise, it will point to the first entry after the taken entries. At the end of a capsule, the bottom pointer will point to the local entry if it exists, or the first empty entry after the jobs otherwise. The bottom pointer can also point to the last job in the array or the earlier local entry during a call to `pushBottom`.

Scheduling Algorithm Overview. Each process is initialized with an empty WS-Deque containing enough `empty` entries to complete the computation and both top and bottom pointers set to the first entry. The process that is assigned the root thread installs the first capsule of this thread, and sets its first entry to `local`. All other processes isntall the `findWork` capsule.

During computation, processes that have work to do will perform that work while marking

```
1  P = number of procs
2  S  = stack size

4  struct procState {
5    union entry = empty
6                | local
7                | job of continuation
8                | taken of ⟨entry*,int⟩

10   ⟨int,entry⟩ stack[S];
11   int top;
12   int bot;
13   int ownerID;

15   inline int getStep(i) { return stack[i].first; }

17   inline void clearBottom() {
18     stack[bot] = ⟨getStep(bot)+1, empty⟩; }

20   void helpPopTop() {
21     int t = top;
22     switch(stack[t]) {
23       case ⟨_, taken(ps,i)⟩:
24         // Set thief state.
25         CAM(ps, ⟨i,empty⟩, ⟨i+1,local⟩);
26         CAM(&top, t, t+1);    // Increment top.
27    } }

29   // Steal from current process, if possible.
30   // If a steal happens, location e is set to "local"
31   // & a job is returned. Otherwise NULL is returned.
32   continuation popTop(entry* e, int c) {
33     helpPopTop();
34     int i = top;
35     ⟨int, entry⟩ old = stack[i];
36     commit;
37     switch(old) {
38       // No jobs to steal and no ongoing local work.
39       case ⟨j, empty⟩: return NULL;
40       // Someone else stole in meantime.  Help it.
41       case ⟨j, taken(_)⟩:
42          helpPopTop(); return NULL;
43       // Job available, try to steal it with a CAM.
44       case ⟨j, job(f)⟩:
45         ⟨int, entry⟩ new = ⟨j+1, taken(e,c)⟩;
46         CAM(&stack[i], old, new);
47         helpPopTop();
48         if (stack[i] != new) return NULL;
49         return f;
```

```
50        // No jobs to steal, but there is local work.
51        case ⟨j, local⟩:
52          // Try to steal local work if process is dead.
53          if (!isLive(ownerID) && stack[i] == old) {
54            commit;
55            ⟨int, entry⟩ new = ⟨j+1,taken(e,c)⟩;
56            stack[i+1] = ⟨getStep(i+1)+1, empty⟩;
57            CAM(&stack[i], old, new);
58            helpPopTop();
59            if (stack[i] != new) return NULL;
60            return getActiveCapsule(ownerID);
61          }
62          // Otherwise, return NULL.
63          return NULL;
64    }  }

66    void pushBottom(continuation f) {
67      int b = bot;
68      int t1 = getStep(b+1);
69      int t2 = getStep(b);
70      commit;
71      if (stack[b] == ⟨t2, local⟩) {
72        stack[b+1] = ⟨t1+1, local⟩;
73        bot = b + 1;
74        CAM(&stack[b], ⟨t2, local⟩, ⟨t2+1, job(f)⟩
75      } else if (stack[b+1].second == empty) {
76        states[getProcNum()].pushBottom(f);
77      }
78      return;
79    }

81    continuation popBottom() {
82      int b = bot;
83      ⟨int, entry⟩ old = stack[b-1];
84      commit;
85      if (old == ⟨j, job(f)⟩) {
86        CAM(&stack[b-1], old, ⟨j+1,local⟩);
87        if (stack[b-1] == ⟨j+1, local⟩) {
88          bot = b-1;
89          return f;
90        }   }
91      // If we fail to grab a job, return NULL.
92      return NULL;
93    }
```

```
94    ˆ findWork() {
95      // Try to take from local stack first.
96      continuation f = popBottom();
97      if (f) GOTO(f);
98      // If nothing locally, randomly steal.
99      while (true) {
100       yield();
101       int victim = rand(P);
102       int i = getStep(bot);
103       continuation g = states[victim].popTop(&stack[bot],i);
104       if (g) GOTO(g);
105     }
106   }
107 }

109 procState states[P]; // Stack for each process.

111 // User call to fork.
112 void fork(continuation f) {
113   // Pushes job onto the correct stack.
114   states[getProcNum()].pushBottom(f);
115 }

117 // Return to scheduler when any job finishes.
118 ˆ scheduler() {
119   // Mark the completion of local thread.
120   states[getProcNum()].clearBottom();
121   // Find work on the correct stack.
122   GOTO(states[getProcNum()].findWork());
123 }
```

Figure 2.3: Fault-tolerant WS-Deque Implementation. Jumps are marked as `GOTO` and functions that are jumped to and do not return (technically continuations) are marked with a ˆ. All CAM instructions occur in separate capsules, similar to function calls.

		New State			
		Empty	Local	Job	Taken
Old State	Empty	-	✓		
	Local	✓	-	✓	✓
	Job		✓	-	✓
	Taken				-

Figure 2.4: Entry State Transition Diagram
Entry state transition diagram.

the bottom of their deque as `local`. Processes that do not have work will first try to take a thread from their own deque using `popBottom`. If `popBottom` fails, the process will pick a random victim and try to steal work from the victim using `popTop`. In a faultless setting, our work-stealing scheduler fuctions like that of ABP.

We use the additional information stored in the WS-Deques and the configuration of capsule boundaries to provide fault tolerance. We provide correctness in a setting with soft faults using idempotent capsules. Each capsule in the scheduler is an instance of one of the capsules discussed in Section 2.4. This means that processes can fault and restart without affecting the correctness of the scheduler.

Providing correctness in a setting with hard faults is more challenging. This requires the scheduler to ensure that work being done by processes that hard fault is picked up in the same capsule that the fault ocurred during by exactly one other process. We handle this by allowing thieves to steal `local` entries from dead processes. A process can check whether another process is dead using a liveness oracle `isLive(procId)`.

One way to construct the liveness oracle would be by implementing a counter and a flag for each process. Each process updates its counter after a constant number of steps (this does not have to be synchronized). If the time since a counter has last updated passes some threshold, the process is considered dead and its flag is set. If the process restarts, it can notice that it was marked as dead, clear its flag, and enter the system with a new empty WS-Deque. Constructing such an oracle does not require a global clock or tight synchronization.

By handling these high level challenges, along with some of the more subtle challenges that occur when trying to provide exactly-once semantics in the face of both soft and hard faults, we are able to achieve a correct scheduler.

2.5.2 Proof of the Correctness of Work-Stealing

We now provide a proof of the correctness of our work-stealing scheduler. Throughout our proof of correctness, we will refer to the code of the work-stealing scheduler shown in Figure 2.3.

We begin by stating some definitions and assumptions. We assume that at least one process will not hard fault during the computation. If this is not true, the computation will have no processes performing work and will never finish. The local continuation of a process can be queried using the function `getActiveCapsule`. This function may be persistent or ephemeral. Any process can query whether another process has hard faulted through the ephemeral function `isDead`, which makes use of the liveness oracle. We define the owner of a WS-Deque to be the process that has the same process number as the `ownerID` field of that WS-Deque. We consider a `popBottom` to be successful if the CAM at Line 86 is successful. We consider a `popTop` to be successful if either of the CAM operations at Lines 46 or 57 are successful.

The first property we prove about our implementation is that the bottom of a WS-Deque can only be operated on by one process at any time.

Lemma 1. *For a given WS-Deque, only one process can call* `pushBottom`*,* `popBottom`*, or* `clearBottom` *at any time. This process is the owner of the WS-Deque unless the owner hard faults in the middle of a* `pushBottom` *or* `popBottom` *invocation.*

Proof. We first consider the `pushBottom` function. All calls to `pushBottom` are made from

the `fork` function. These calls are always made to the WS-Deque chosen by the `getProcNum` function. Since this function returns the ID of the process that is running it and there is no capsule boundary between the call to `getProcNum` and the call to `pushBottom`, the process running the `fork` will always invoke `pushBottom` on its own WS-Deque. Since the `pushBottom` function is part of the scheduler rather than the algorithm code, it is never pushed onto the WS-Deque as a job. This means that it can only be stolen from the local state in the event of a process hard fault. Similarly, all calls to `popBottom` and `clearBottom` are made using the `getProcNum` function inside the `findWork` and `fork` functions respectively. Therefore, the same argument holds. Since only the owner can invoke these functions and it will run them to completion before calling any other functions, we know that at most one of these function invocations can exist at any time. □

From this property, we find the related result.

Corollary 1. *For a given WS-Deque, only one process can update the bottom pointer at any time. This process is the owner of the WS-Deque unless the owner hard faults in the middle of a `pushBottom` or `popBottom` invocation.*

Proof. The only functions that update the bottom pointer are `pushBottom`, `popBottom`, or `clearBottom`. Applying Lemma 1 gives the desired result. □

We then use this property about the bottom of WS-Deques to show that user level threads that are being worked on by processes are tracked with local entries.

Lemma 2. *Every process that is working on user level threads will have a local entry that is pointed to by the bottom pointer of their WS-Deque.*

Proof. All user threads are initiated by the `findWork` function at Line 97 or Line 104.

If the thread is started at Line 97, it means that `popBottom` returned that continuation. The if statement at Line 87 requires a local entry to exist at `stack[b-1]` in order to return a non-NULL value. The bottom pointer is then set to this location before the return. Corollary 1 tells us that bottom pointer will not be modified by any other process. The entry pointed to by the bottom pointer can only be modified from local by calls to `pushBottom` or `popTop`. We know from Lemma 1 that `pushBottom` cannot be running concurrently. We show that `popTop` cannot concurrently modify the entry by observing that `popTop` will only modify a local entry for a process that hard faulted, and a process cannot return a value after it hard faults. Therefore, the values that exist at Line 87 must still exist upon jumping to the continuation.

If the thread is started at Line 104, it means that `popTop` returned that continuation. The `popTop` function can return a non-NULL value at Line 49 or Line 60. In either case, the return is preceded by a call to the `helpPopTop` function. This function ensures that the entry pointed to by the newly taken entry is set to local. This newly taken entry was set by the CAM at Line 46 if the return happened at Line 49 or the CAM at Line 57 if the return happened at Line 60. Both of these CAMs set the entry pointer in the taken to the argument passed to `popTop`. Looking at Line 103, we see that this is the pointer to the bottom of the thief's WS-Deque. Therefore, that is the entry that will be set to local. We know that the bottom entry and pointer will not be modified between the call to `helpPopTop` and the jump to the continuation because the owner

process is the one running the calls to popTop and findWork and the jump to the thread, and can therefore not hard fault or make other calls to pushBottom, clearBottom, or popTop.

In both cases where user threads are started, a local entry exists on the bottom of the WS-Deque owned by the process starting that thread. It then remains to show that this local entry is not deleted before the process ceases working on that thread. Local entries are only modified by the clearBottom, pushBottom, and popTop functions. We know from Lemma 1 that unless the process hard faults, only the owner can run pushBottom or clearBottom. If the owner calls clearBottom, it must have done so from the scheduler function. This function is only called when the user level thread completes, meaning the process is no longer working on it. Calls to pushBottom may modify the local entry that existed prior to the call if the CAM in Line 74 succeeds, but Lines 72 and 73 will create a local entry at the new bottom before this can happen. Calls to popTop will never modify a local entry unless the owner has hard faulted. In this case, the local entry will be set to taken by the CAM at Line 57. Once this CAM is successful, the taken entry will point to the bottom of the thief's WS-Deque, which will be an empty entry. The first helpPopTop call on the victim's WS-Deque that resolves Line 25 will change the empty entry to local. Since the thief must complete a call to helpPopTop between Line 74 and the return from popTop, the local entry will be created before the thief begins working on the thread. □

Since a process can never work on multiple user level threads, we prove that there are never multiple local entries visible to steal if the process crashes.

Lemma 3. *At most one local entry can be successfully targeted by a popTop on a WS-Deque. Calls to the popTop function of that WS-Deque after the successful steal completes will target an empty entry.*

Proof. In order for a local entry to be stolen the top pointer of the WS-Deque must point to that entry. Since this entry is a local entry, any thief will execute the case beginning at Line 51. In this case Line 56 will be executed prior to any CAM operation. This will set the entry below the top pointer to empty. Once the local entry has been stolen, the top pointer will be changed to point to the empty entry by the helpPopTop function. No popTop targeting an empty entry will succeed, or perform any modifications to the WS-Deque at all. As long as the entry remains empty, no popTop on that WS-Deque can succeed. Empty entries can only be modified by the pushBottom function. The code that performs this modification is enclosed in the if statement at Line 71. The condition in this if statement will always fail since the CAM at Line 57 removes the remaining local entry from the WS-Deque. Since the empty entry pointed to by the top pointer will never be modified, no further popTop calls can be successful. □

Having completed these useful structural lemmas, we can begin to prove the correctness of our functions. We focus first on proving correctness in the face of soft faults and leave hard faults for later in the proof.

Lemma 4. *Any popBottom function targeting a job entry will be successful unless a concurrent popTop function targeting the same entry is successful or the process hard faults.*

Proof. If at any point during the findWork function the process hard faults, then the lemma is vacuously true. This means that we can ignore hard faults for the sake of the proof.

The entry targeted by a popBottom invocation is the entry immediately above the bottom pointer. If this entry is a job, the CAM at Line 86 will succeed unless the entry is changed before the CAM happens. Job entries are only modified by successful invocations of popBottom or popTop, so if neither of these functions concurrently succeed on the target entry, the CAM will succeed, and therefore the popBottom will succeed. □

Lemma 5. *Any popTop function targeting a job entry or a local entry on a process that hard faulted will be successful unless a concurrent popBottom or popTop function targeting the same entry is successful or the process hard faults.*

Proof. If at any point during the findWork function the process hard faults, then the lemma is vacuously true. This means that we can ignore hard faults for the sake of the proof.

We first consider the case when the top pointer points to a job entry. In this case the CAM at Line 46 will succeed unless the entry is changed before the CAM happens. Job entries are only modified by successful invocations of popBottom or popTop, so if neither of these functions concurrently succeed on the target entry, the CAM will succeed, and therefore the popTop will succeed.

We next consider the case when the victim has hard faulted and the top pointer points to a local entry. In this case, the CAM at Line 57 will succeed unless the entry is changed before the CAM happens. Local entries are only modified by successful invocations of popTop or pushBottom. We know from Lemma 1 that pushBottom functions can only be run by the owner of the WS-Deque or a thief if the owner of the WS-Deque hard faulted. The owner has hard-faulted, so it cannot run the pushBottom function. Since pushBottom is a scheduler function, it can only be stolen from a local entry, rather than a job entry. By applying Lemma 3 we find that it is impossible for a popTop to target a local entry after the pushBottom function is stolen. By applying Lemma 3 we find that if a popTop invocation targeting a local entry on a process that hard faulted is running concurrently with the pushBottom function for that process' WS-Deque then the entry targeted by the popTop invocation was the target of another successful popTop invocation that ran concurrently with the original popTop invocation. This means that a popTop invocation targeting a local entry on a process that hard faulted will either succeed or be concurrent with another popTop invocation that succeeds at targeting the same entry.

Since we have proven the lemma for both possible cases, the proof is complete. □

When proving the correctness of pushBottom we consider how hard faults affect the implementation of the function and the interleavings that result. We also connect the user level interface function fork to the scheduler.

Lemma 6. *Every continuation will be added to a WS-Deque as a job exactly the number of times fork was called on it.*

Proof. Job entries are only added to a WS-Deque via the pushBottom function. This function is only ever invoked by the fork function. Each call to fork directly calls pushBottom exactly once. It therefore suffices to show that each call to pushBottom other than recursive calls result in the passed argument being added to a WS-Deque exactly once. We show that each call to pushBottom will have exactly one of the following results: the argument is added to

the associated WS-Deque exactly once or the owner hard faults and `pushBottom` is recursively called with the same argument on a different WS-Deque whose owner has not hard faulted. Since we know that not all processes can hard fault, this is sufficient.

We begin by assuming that the process does not hard fault while running `pushBottom`. We know that the bottom entry of the WS-Deque is a local entry by Lemma 2 and that it cannot be concurrently modified by Lemma 1. This means that all statements inside the if block that begins at Line 71 are executed at least once and that the first execution of the CAM at Line 74 will succeed. This adds the continuation to the WS-Deque as a job entry. The tag before the entry prevents the CAM from succeeding more than once. We are then left to show that soft faults will not result in additional calls to `pushBottom` being made. Until the CAM succeeds, we know that the capsule will always enter the if block at Line 71. In order for this CAM to succeed, Line 72 must be completed, setting the entry below the old bottom pointer to local. Local entries can only be modified by the `pushBottom`, `clearBottom`, or `popTop` functions. The current instance of `pushBottom` will not change this entry, and Lemma 1 states that no other instance of `pushBottom` or `clearBottom` can be running concurrently. We observe that `popTop` will only modify a local entry on a process that hard faulted. This lets us conclude that the local entry will not be modified if the process does not hard fault, preventing the process from executing the recursive `pushBottom` call. This means that the continuation is added to the WS-Deque exactly once.

We then consider the case when the process hard faults while running `pushBottom`. If the hard fault occurs prior to the CAM at Line 74 succeeding, then the CAM will not succeed on this invocation of `pushBottom` and the thief that steals this thread will recursively call `pushBottom` on its own WS-Deque. In this case, the owner hard faulted, so the local entry at `stack[b]` will not be modified until it is stolen by a call to `popTop`. During this `popTop`, the top pointer will point to `stack[b]`. This means that the thief will set `stack[b+1]` to empty in Line 56 prior to completing the `popTop`. Furthermore, when the CAM at Line 57 succeeds, it changes the entry at `stack[b]` from local to taken. When the thief begins runs the `pushBottom` capsule, it will bypass the if block starting at Line 71 in favor of the else block. Since the if block is not taken, the CAM will never be attempted. The else block recursively calls `pushBottom` with the same argument on the thief's WS-Deque.

If the hard fault occurs after the CAM succeeds, then the continuation has been added to the WS-Deque and it must not be added again. The tag before the entry prevents the CAM from succeeding more than once. Since the CAM was successful, the entry at `stack[b]` has been set to job. This means that in order for the `pushBottom` function to be restarted, a thief had to steal the local entry set at `stack[b+1]` during Line 72. In order for the steal to occur, the CAM at Line 57 had to succeed, which would change the entry from local to taken. Since taken entries are never modified, we know that `stack[b+1]` must be a taken entry for the `pushBottom` function to be resumed. This means that when the thief restarts the capsule, it can never reach the invocation of the `pushBottom` function.

We have proven that each call to `pushBottom` will add the argument to the associated WS-Deque exactly once or recursively call `pushBottom` with the same argument on a different WS-Deque whose owner has not hard faulted. This proves that each call to fork results in the argument being added to a WS-Deque as a job exactly once, completing the proof. □

Now that we have shown that user work is correctly added to the scheduler, we show that each process will try to perform the work that has been added.

Lemma 7. *Every call to* `findWork` *results in a successful* `popTop` *or a successful* `popBottom` *unless the process hard faults or the computation ends.*

Proof. If at any point during the `findWork` function the process hard faults, then the lemma is vacuously true. This means that we can ignore hard faults by the process calling `findWork` for the sake of the proof.

The `findWork` function begins by calling the `popBottom` function. Lemma 4 shows that the `popBottom` call will be successful unless unless all job entries on the WS-Deque are stolen prior to the CAM at Line 86.

If the `popBottom` function is not successful then the `findWork` function will proceed to the while loop that performs steal attempts. This loop selects a victim process at random, and the performs the `popTop` function on that victim. We know from Lemma 5 that `popTop` will succeed if the top entry of the victim's WS-Deque is a job entry or if the victim has crashed and the top entry of its WS-Deque is a local entry unless a concurrent `popTop` targeting the same entry succeeds.

In order for the computation to complete, each user thread must be run to completion. This means that if the computation is not complete there is a positive number of user threads that have not been run to completion. Since user threads can only be enabled by other user threads, at least one of these threads must be enabled. Lemma 6 states that this thread had a job entry created for it. Since the thread has not been completed, it must have a process working on it or its job entry must be in a WS-Deque. If the job entry is in a WS-Deque, then the top pointer of that WS-Deque must point to that entry, or another job entry above it. We know from Lemma 5 that if the thief calls `popTop` on this WS-Deque, it will succeed unless a concurrent call to `popBottom` or `popTop` successfully targets that entry. If a process is working on the thread, Lemma 2 states that the WS-Deque has a local entry pointed to by the bottom. If the process does not hard fault, it will eventually call `fork` with a new continuation or complete its user level thread. If `fork` is called, it will result in a new job entry which may be targeted by `popBottom` or `popTop`. If the user level thread is completed, either there exists another enabled user level thread that this analysis applies to, or the computation is complete. If the process hard faults at any time, then its top entry is a valid `popTop` target and Lemma 5 applies.

At any time, progress is being made towards the end of the computation or there exists a target that the process running `findWork` may call `popBottom` or `popTop` on successfully. Since the `findWork` function will make attempts to call `popTop` repeatedly until it succeeds, it will eventually either succeed, or the computation will finish. □

We extend the proof of processor effort to show that between all of the available processes, all of the work that is added to the scheduler is found without inadvertently duplicating any of that work.

Lemma 8. *Each job entry in a WS-Deque will be the target of a successful* `popBottom` *or* `popTop` *exactly once.*

Proof. To show that a job entry cannot be affected by a successful `popBottom` or `popTop` more than once, we note that either successful function is caused by a successful CAM operation

on the associated entry. Such a CAM changes the entry to either local or taken, depending on which function was successful.

In order for the computation to complete, each user thread must be run to completion. This means that while there are job entries in any WS-Deque, the scheduler will continue to run. Job entries are located in a WS-Deque above the bottom pointer of the WS-Deque and at or below the top pointer of the WS-Deque. The structure of a WS-Deque means that for any job entry is not directly above the bottom pointer, all entries between it and the bottom entry are job entries. Similarly, for any job entry is not at the top pointer, then all entries between it and the top entry are job entries.

If a process that does not have user level work on its WS-Deque does not hard fault, it will perform one failed `popBottom` call, and then repeatedly make `popTop` attempts until one is successful and it becomes a process that has user level work. These `popTop` attempts are made on random processes, ensuring that each process will eventually be chosen as a victim.

If a process that has user level work on its WS-Deque does not hard fault, it will repeatedly run any local work that it has, then call the `popBottom` function. Lemma 4 tells us that this function will succeed unless it is targeting a non-job entry or a concurrent `popTop` call targeting the same entry succeeds. If `popBottom` succeeds, the bottom pointer is set to the next lowest entry and the process is repeated. If `popBottom` is not targeting a job entry, the structure of a WS-Deque tells us that there are no job entries on that WS-Deque. If a concurrent `popTop` call targeting the same entry succeeds that entry becomes taken and the top pointer of the WS-Deque must point to that entry. This also means that the WS-Deque contains no job entries. Once the WS-Deque contains no job entries and the local entry (if any) finishes, the process becomes a process that does not have user level work.

If a process that has user level work on its WS-Deque does hard fault, it will have some non-negative number of job entries above the bottom pointer of its WS-Deque. We rely on thief processes to pop these entries from the WS-Deque. Lemma 5 tells us that every `popTop` attempt on the WS-Deque will be successful unless a concurrent `popTop` or `popBottom` is successful. Lemma 1 states that `popBottom` cannot be run on a WS-Deque owned by a process that hard faulted unless it is stolen. Since `popBottom` is a scheduler function, it can only be stolen by a local entry. The structure of the WS-Deque means that local entries cannot be stolen until there are no job entries on the WS-Deque. This means that the top entry will be the target of a successful `popTop` call from a thief. Since a successful `popTop` call results in the top pointer being lowered, this process will repeat until all job entries have been targeted by successful `popTop` calls.

As long as there exists at least one process that does not hard fault, it will switch between having user level work that it completes to not having user level work and making `popTop` attempts until it finds some. The end result of this process is that no job entries will remain in any WS-Deque. Since job entries are only modified by successful calls to `popBottom` or `popTop`, each job entry must have been removed by a successful `popBottom` or `popTop` call. □

Once the scheduler has assigned threads to various processes, the processes must complete the work. The following two lemmas show that each thread that is assigned has computation begun on it, which is sufficient to show completion in the face of soft faults.

Lemma 9. *Every continuation that is affected by a successful `popTop` is jumped to at least once.*

Proof. We consider a continuation to be affected by a successful `popTop` if the WS-Deque entry associated with that continuation is targeted by a successful CAM operation inside of the `popTop` function. We consider an entry and a continuation to be associated if the entry is a job containing the continuation or the entry is local while the continuation is being run by the process that owns the WS-Deque the entry resides in. After the successful CAM, the target entry has been set to a taken entry that contains a pointer to the bottom of the thief. Since taken entries are never changed, we know that the if statement will succeed if and only if the CAM succeeded during the current capsule. Therefore if the process does not hard fault then the continuation will be returned to `findWork`, which jumps to that continuation. Soft faults may cause some of the instructions to be re-run, but will not change the resulting memory state (due to idempotency). If the process hard faults at any point between the successful CAM and the jump, it relies on other thieves calling the `helpPopTop` function to ensure that there is a local entry at the location pointed to in the taken entry, which is the bottom of its WS-Deque. This entry will eventually be stolen by some other thief. That thief will restart the capsule that the original thief hard faulted during. Since we know that not all processes hard fault, at some point a process will complete the `popTop` function and jump to the continuation inside the `findWork` function. □

Lemma 10. *Every continuation that is affected by a successful `popBottom` is jumped to at least once.*

Proof. We consider a continuation to be affected by a successful `popBottom` if the WS-Deque job entry containing that continuation is targeted by a successful CAM operation inside of the `popBottom` function. After the CAM is successful, the target entry has been set to local. This local entry can only be changed by a call to `clearBottom`, or a call to `popTop` after the process hard faults. By Lemma 1, we know that `clearBottom` cannot run concurrently with `popBottom`. This means that the if statement will succeed if and only if the CAM succeeded during the current capsule. Therefore if the process does not hard fault, then the continuation will be returned to `findWork`, which jumps to that continuation. Soft faults may cause some of the instructions to be re-run, but will not change the resulting memory state. If the process hard faults at any point between the successful CAM and the jump, then then a local entry will exist at the bottom of its WS-Deque until that entry is stolen. Lemma 9 shows that once the entry is stolen, it will be jumped to at least once. Jumping to either `popBottom` or `findWork` during the specified window will maintain the local variables, including the continuation that will then be jumped to in `findWork`. □

We now show that hard faults do not prevent any computation from being completed.

Lemma 11. *Any user thread on a process that hard faulted will be be affected by a successful `popTop` and the capsule that was in process will be restarted.*

Proof. In order for the computation to complete, each user thread must be run to completion. This means that while there are unfinished user threads, the scheduler will continue to run. Lemma 2 states that any process that is working on a user level thread has a local entry that

is pointed to by its bottom pointer. Lemma 7 states that processes that run out of work will eventually perform a successful `popBottom` or `popTop` unless they hard fault or the computation ends. Using Lemma 8, we know that the number of successful calls that target a job entry is limited. Since successful `popBottom` calls can only target job entries and successful `popTop` calls can only target job or local entries, all other successful `popTop` calls must occur on user threads on processes that have hard faulted. We combine Lemma 9 with the fact that `popTop` calls `getActiveCapsule` when stealing a local entry to finish the proof. □

We have shown that all work created at the user level is completed and that no user level threads are created by the scheduler. We conclude the proof by showing that the scheduler does not over-execute user threads.

Lemma 12. *No capsule in user level code will be run to completion more times than the number of times it is invoked by user level code.*

Proof. A capsule is considered run to completion when all of its instructions have been completed and the restart pointer for the subsequent capsule has been installed. We assume that capsules are handled as discussed in Section sec:pers-single-proc-robust, which describes how to ensure that soft faults during direct runs will not cause a capsule to run to completion multiple times. We are then left to show that the scheduler does not result in extra invocations of user level code. This might happen in two ways: threads might be added to WS-Deques more times than they were enabled or entries on WS-Deques may be run multiple times.

We first show that threads are not added to WS-Deques more times than they are enabled. Threads are only enabled as job entries through calls to the `fork` function. We show in Lemma 6 that the number of job entries added for a thread is exactly the number of times fork is invoked on that thread.

We next show that although WS-Deque entries can be run multiple times, this will not result in any capsule being run to completion multiple times. Lemma 8 states that each job entry will be the result of a successful `popBottom` or `popTop` exactly once. In Lemmas 10 and 9, we prove that in either of these cases we will jump to the beginning of the thread.

If the thread is run to completion without hard faulting, it will complete the user level work normally, possibly make calls to `fork`, and then call the scheduler function. We know that local work is not stolen from a process unless that process hard faulted, so we do not have to consider steal attempts at this time. The user level work does not interact with the scheduler, and therefore cannot affect the entries on the WS-Deque. If `pushBottom` is run to completion without any hard faults, then the original entry that corresponded to the user thread will be replaced with a job entry containing the newly enabled thread continuation. A new local entry corresponding to the thread will be created below the original entry. The scheduler function calls `clearBottom`, sets the local entry at the bottom of the WS-Deque to empty. Once this has been completed there is no longer an entry corresponding to the thread, so it cannot be jumped to again unless it is later re-enabled. Since the process has not hard faulted, no thief will ever steal the local entry associated with the thread.

We then consider what happens if the process running the thread hard faults. If the process hard faults during the user level code, then it will be stolen regularly. Since the `popTop` function returns the active capsule when stealing a local entry (Line 60) rather than the entire thread, the

thief will start on the first capsule that has not been run to completion rather the beginning of the thread. The thief will also set the local entry that corresponded to the thread to empty during Line 57, preventing it from being stolen by any other thief. These facts are true for any steal on a process that has hard faulted. We consider two cases for a hard fault during `pushBottom`: before the CAM at Line 74 is run at all, and after it has been run at least once. If the hard fault occurs before the CAM is run then the entry at `stack[b]` will remain local until it is stolen. During the `popTop` call when this entry is stolen, the thief will set `stack[b+1]` to empty during Line 56. Since this occurs before the CAM at Line 57, it will occur before the top pointer can be changed to `stack[b+1]`. Since this entry will be set to empty before any `popTop` calls can see it, it will never be stolen. When the thief restarts the active capsule in `pushBottom`, the entry at `stack[b]` will have been set to taken and the entry at `stack[b+1]` will have been set to empty, so the thief will call `pushBottom` on its own WS-Deque. That call can be analyzed in the same manner as the original call. If the hard fault occurs after the CAM at Line 74 has been run then the entry at `stack[b]` has been set to job. In this case, the current thread will not be stolen until the top pointer is set to point to `stack[b+1]`. This entry was set to local by Line 72. When the thief restarts the active capsule, the state of the WS-Deque will cause it to bypass both if clauses and immediately return to the user thread without further modifying the WS-Deque. If the process hard faults during the scheduler function, the hard fault will either occur before `clearBottom` finishes, in which case it will be stolen and restarted normally, or it will occur after the `clearBottom` finishes, in which case the entry will be set to empty and therefore never stolen.

We have now proven that threads are not added to WS-Deques more times than they are enabled and that WS-Deque entries being run multiple times will not cause a user level capsule inside threads associated with those entries to be run to completion multiple times. This completes the proof. □

We combine these lemmas to prove our correctness Theorem.

Theorem 7. *The implementation of work stealing provided in Figure 2.3 correctly schedules work according to the specification in Section 2.5.*

Proof. We know from Lemma 6 and Lemma 8 that every enabled user thread will be scheduled on to an active process. Lemma 9, Lemma 10, and Lemma 11 combine to prove that every scheduled thread is run to completion. Lemma 8 and Lemma 12 show that no work is duplicated or re-executed. Since all work is scheduled and run to completion following the computation dependencies, the implementation is correct. □

2.5.3 Time Bounds

We now analyze bounds on runtime based on the work-stealing scheduler. As with ABP, we consider the total amount of work done by a computation, and the depth of the computation, also called the critical path length. In our case we have W, the work assuming no faults, and W_f, the work including faults. In algorithm analysis the user analyzes the first, but in determining the runtime we care about the second. Similarly we have both D, a depth assuming no faults, and D_f, a depth with faults.

We take a similar approach to ABP to prove the time bounds. In particular, as in their algorithm, our `popTop`, `popBottom`, and `pushBottom` functions all take $O(1)$ work without faults. With our WS-deque, these operations take expected $O(1)$ work (including faults). Also as with their version, our `popTop` is unsuccessful (returns Null when there is work) only if another `popTop` is successful during the attempt. The one place where their proof breaks down in our setup is the assumption that a constant sized quanta can always capture two steal attempts. Because our processors can fault multiple times, we cannot guarantee this. In their proof this is needed to show that for every P steal attempts, with probability at least $1/4$, at least $1/4$ of the non-empty deques are successfully stolen from ([7], Lemma 8). In our case a constant fraction $(1 - O(1) \cdot f)^2$ of adjacent pairs of steal attempts will not fault at all and therefore count as a steal attempt. For analysis we can assume that if either steal in a pair faults, then the steal is unsuccessful. This gives a similar result, only with a different constant, i.e., with probability at least $1/4$, at least $(1 - O(1) \cdot f)^2/4$ of the non-empty deques are successfully stolen from. We note that hard faults affect the average number of active processors P_A. However they otherwise have no asymptotic affect in our bounds because a hard fault in our scheduler is effectively the same as forking a thread onto the bottom of a work-queue and then finishing.

ABP show that their work-stealing scheduler runs in expected time $O(W/P_A + DP/P_A)$. To apply their results we need to plug in W_f for W because that is the actual work done, and D_f for D because that is actual depth. While bounding W_f to be within a constant factor of W is straightforward, bounding D_f is trickier because we cannot sum expectations to get the depth bound (the depth is a maximum over path lengths). Instead we show that with some high probability no capsule faults more than some number of times l. We then simply multiply the depth by l. By making the probability sufficiently high, we can pessimistically assume that in the unlikely event that any capsule faults more than l times then, the depth is as large as the work. This idea leads to the following theorem.

Theorem 8. *Consider any multithreaded computation with W work, D depth, and C maximum capsule work (all assuming no faults) for which all capsules are atomically idempotent. On the Parallel-PM with P processors, P_A average number of active processors, and fault probability bounded by $f \leq 1/(2C)$, the expected total time T_f for the computation is*

$$O\left(\frac{W}{P_A} + D\left(\frac{P}{P_A}\right)\lceil \log_{1/(Cf)} W \rceil\right).$$

Proof. We must account for faults in both the computation and the work-stealing scheduler. The work-stealing scheduler has $O(1)$ maximum capsule work, which we assume is at most C. Because we assume all faults are independent, the probability that a capsule will run l or more times is upper bounded by $(Cf)^l$. Therefore if there are κ capsules in the computation, including the capsules executed as part of the scheduler, the probability that any one runs more than l times is upper bounded by $\kappa(Cf)^l$ (by the union bound). If we want to bound this probability by some ϵ, we have $\kappa(Cf)^l \leq \epsilon$. Solving for l and using $\kappa \leq 2W$ gives $l \leq \lceil \log_{1/(Cf)}(2W/\epsilon) \rceil$. This means that with probability at most ϵ, $D_f \leq D \log_{1/(Cf)}(2W/\epsilon)$. If we set $\epsilon = 2/W$, then $D_f \leq 2D \log_{1/(Cf)} W$. Now we assume that if any capsule faults l times or more that the depth of the computation equals the work. This gives $(P/P_A)(2/W)W + (1 - 2/W)2D\lceil \log_{1/(Cf)} W \rceil)$ as the expected value of the second term of the ABP bound, which is

bounded by $O((P/P_A)D\lceil\log_{1/(Cf)} W\rceil)$. Because the expected total work for the first term is $W_f \leq (1/(1-Cf))W$, and given $Cf \leq 1/2$, the theorem follows. □

This time bound differs from the ABP bound only in the extra $\log_{1/(Cf)} W$ factor. If we assume P_A is a constant fraction of P then the expected time simplifies to $O(W/P + D\lceil\log_{1/(Cf)} W\rceil)$.

2.6 Fault-Tolerant Algorithms

In this section, we outline how to implement several algorithms for the Parallel-PM model. The algorithms are all based on binary fork-join parallelism (i.e., nested parallelism), and hence fit within the multithreaded model. We state all results in terms of faultless work and depth. The results can be used with Theorem 8 to derive runtime bounds for the Parallel-PM. Recall that in the Parallel-PM model, external reads and writes are unit cost, and all other instructions have no cost (accounting for other instructions would not be hard). The algorithms that we use are already race-free. Making them write-after-read conflict free simply involves ensuring that reads and writes are to different locations. All capsules of the algorithms are therefore atomically idempodent. The base case for each of our variants of the algorithms is done sequentially within the ephemeral memory.

Prefix Sum. Given n elements $\{a_1, \cdots, a_n\}$ and an associative operator "+", the prefix sum algorithm computes a list of prefix sums $\{p_1, \cdots, p_n\}$ such that $p_i = \sum_{j=1}^{i} a_j$. Prefix sum is one of the most commonly-used building blocks in parallel algorithm design [71].

We with the standard prefix sum algorithm [71]. The algorithm consists of two phases—the up-sweep phase and the down-sweep phase, both based on divide-and-conquer. The up-sweep phase bisects the list, computes the sum of each sublist recursively, adds the two partial sums as the sum of the overall list, and stores the sum in the persistent memory. After the up-sweep phase finishes, we run the down-sweep phase with the same bisection of the list and recursion. Each recursive call in this phase has a temporary parameter t, which is set to 0 for the initial call. Then within each function, we pass t to the left recursive call and $t + LeftSum$ to the right recursive call, where $LeftSum$ is the sum of the left sublist computed from the up-sweep phase. In both sweeps the recursion stops when the sublist has no more than B elements, and we sequentially process it using $O(1)$ memory transfers. For the base case in the down-sweep phase, we set the first element p_i to be $t + a_i$, and then sequentially compute the rest of the prefix sums for this block. The correctness of p_i follows from how t is computed along the path to a_i.

This algorithm fits the Parallel-PM model in a straightforward manner. We can place the body of each function call (without the recursive calls) in an individual capsule. In the up-sweep phase, a capsule reads from two memory locations and stores the sum back to another location. In the down-sweep phase, it reads from at most one memory location, updates t, and passes t to the recursive calls. Defining capsules in this way provides write-after-read conflict-freedom and limits the maximum capsule work to a constant.

Theorem 9. *The prefix sum of an array of size n can be computed in $O(n/B)$ work, $O(\log n)$ depth, and $O(1)$ maximum capsule work, using only atomically-idempotent capsules.*

Merging. A merging algorithm takes two sorted arrays A and B of size l_A and l_B ($l_A + l_B = n$), and returns a sorted array containing the elements in both input lists. Our algorithm is based on the classic divide-and-conquer approach [20].

The first step of the algorithm is to allocate the output array of size n. Then the algorithm conducts dual binary searches of the arrays in parallel to find the elements ranked $\{n^{2/3}, 2n^{2/3}, 3n^{2/3}, \ldots, (n^{1/3} - 1)n^{2/3}\}$ among the set of keys from both arrays, and recurses on each pair of subarrays until the base case when there are no more than B elements left (and we switch to a sequential version). We put each of the binary searches into its own capsule, as well as each base case. These capsules are write-after-read conflict free because the output of each capsule is written to a different subarray. Based on the analysis in [20] we have the following theorem.

Theorem 10. *Merging two sorted arrays of total size n can be done in $O(n/B)$ work, $O(\log n)$ depth, and $O(\log n)$ maximum capsule work, using only atomically-idempotent capsules.*

Sorting. We outline a samplesort algorithm based on Blelloch et al. [20]. The sorting algorithm first splits the set of elements into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and recursively sorts each of the subarrays. The recursion terminates when the subarray size is less than M, and the algorithm then sequentially sorts within a single capsule. Then the algorithm samples every $\log n$'th element from each subarray. These samples are sorted using mergesort, and $\sqrt{n}$ pivots are picked from the result using a fixed stride. The next step is to merge each $\sqrt{n}$-size subarray with the sorted pivots to determine bucket boundaries within each subarray. Once the subarrays have been split, prefix sums and matrix transposes are used to determine the location in the buckets where each segment of the subarray is to be sent. After that, the keys need to be moved to the buckets, using a bucket transpose algorithm. We can use our prefix sum algorithm and the divide-and-conquer bucket transpose algorithm from [20], where the base case is a matrix of size less than M, and in the base case the transpose is done sequentially within a single capsule (note that this assumes $M > B^2$ to be efficient). The last step is to recursively sort the elements within each bucket. All steps can be made write-after-read conflict free by writing to locations separate than those being read. By applying the analysis in [20] with the change that the base cases (for the recursive sort and the transpose) are when the size fits in the ephemeral memory, and that the base case is done sequentially, we obtain the following theorem.

Theorem 11. *Sorting n elements can be done in $O(n/B \cdot \log_M n)$ work, $O((M/B + \log n) \log_M n)$ depth, and $O(M/B)$ maximum capsule work, using only atomically-idempotent capsules.*

Matrix Multiplication. Consider multiplying two square matrices A and B of size $n \times n$ (assuming $n^2 > M$) with the standard recursive matrix multiplication [40] based on the 8-way divide-and conquer approach.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$
$$= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Note that every pair of submatrix multiplications shares the same output location. This leads to write-after-read conflicts since a straightforward implementation will read the value from the

output cell, add the computed value, and finally write the sum back. Therefore, we modify the algorithm to allocate two copies of temporary space for the output in each recursive subtask, which allows the subtasks to write to different output spaces (avoiding conflicts), and eventually sum the values from the temporary space into the original output space.

If we stack-allocate the memory for each processor, a straightforward upper bound for the total extra storage is $O(pn^2)$ on p processors using the standard space bound under work-stealing. A more careful analysis can tighten the bound to $O(p^{1/3}n^2)$. This extra storage can be further limited to $O(n^2)$ by modifying the order of the recursive calls, assuming that the main memory size is larger than the total size of all private caches.

When this algorithm is scheduled by a randomized work-stealing scheduler, the computation is race-free. All multiplications that run at the same time have different output locations. The summations are independent of each other, and applied after all of the associated multiplications have completed.

If we put each arithmetic operation in a separate capsule, the algorithm incurs $O(n^3)$ memory accesses, which is inefficient. Hence, we mark a capsule anytime the recursion reaches a subtask that can entirely fit into the ephemeral memory. This happens when the matrix size is smaller than $c'\sqrt{M}$ for a constant $c' < 1$. Within these capsules, we run the algorithm sequentially . For the matrix additions, we similarly mark a capsule boundaries such that each capsule can fit into the ephemeral memory. This does not affect the overall work. We obtain the following theorem.

Theorem 12. *Multiplying two square matrices of size n can be done in $O(n^3/(B\sqrt{M}))$ work, $O(M/B + \log^2 n)$ depth, and $O(M/B)$ maximum capsule work, using only atomically-idempotent capsules.*

2.7 Chapter Summary

In this chapter, we describe the Parallel Persistent Memory Model, which characterizes faults as loss of data in individual processors and their associated volatile memory. We consider an external memory model view of algorithm cost, but the model could easily be adapted to support other traditional cost models. We also provide a general strategy for designing programs based on capsules that perform properly even when faults occur. We specify a condition of being atomically idempotent that is sufficient for correctness, and provide examples of atomically idempotent capsules that can be used to generate more complex programs. We use these capsules to build a work-stealing scheduler that can run programs in a parallel system while tolerating both hard and soft faults with only a modest increase in the total cost of the computation. We also provide several algorithms designed to support fault tolerance using our capsule methodology. We believe that the techniques in this paper can provide a practical way to provide the desirable quality of fault tolerance without requiring significant changes to hardware or software.

Chapter 3

Writeback-Aware Caching

The long history of papers on caching problems [3, 11, 12, 13, 14, 35, 36, 46, 47, 50, 70, 72, 75, 85, 99, 102, 116, 117] has largely overlooked an increasingly important cost in real caches: the cost of *writebacks*. Any data item that has been modified since being fetched into the cache (i.e., a *dirty* item) must be written back to memory on eviction. In contrast, a data item that has *not* been updated since being fetched (a *clean* item) can simply be discarded from the cache on eviction.

Traditional memory systems were designed to minimize response time, with replacement policies designed to maximize the number of cache hits. Modern processors, however, have greatly increased their instruction throughput by increasing parallelism (number of cores) rather than increasing clock frequency. For memory-intensive programs, the number of concurrently in-flight memory requests grows linearly with the number of cores, such that the available memory bandwidth is often the primary performance bottleneck. Moreover, these additional requests combined with the end of Dennard scaling [23, 44] has caused power consumption to become critical for computing systems ranging from exascale computing [107] to microcomputing [39]. The practical importance of these metrics has been underscored by a significant amount of systems research [79, 105, 112]. Unlike the traditional metric of response time, memory system bandwidth and power consumption are significantly impacted by writebacks [61, 79].

Non-volatile memory technologies further emphasize the importance of writebacks in real memory systems. Writing data into these memories requires more time and energy than reading data, sometimes by an order of magnitude or more [63, 69, 76, 100, 109]. Furthermore, these technologies have limited write endurance, so reduced writes mean increased device longevity.

It is therefore imperative to have an understanding of how to manage caches in order to minimize writebacks. Motivated by the lack of research in this blossoming performance consideration, we study the effects of writebacks in caching. We extend traditional caching models to account for this new variable and show a variety of theoretical and practical results.

Our Contributions. In this chapter, we initiate a general exploration of writeback-aware caching, seeking to bridge the gap between real caching systems and the theoretical understanding of caches. We make the following contributions:

1. We define and study the *Writeback-Aware Caching Problem*, which generalizes traditional caching problems by adding writeback costs: Given a sequence of reads and writes to data items and a specified cache size, the goal is to minimize the sum of the miss and writeback

costs when servicing the sequence in order. For our algorithms, we allow data items to have variable sizes, variable miss costs, and variable writeback costs. For our hardness results, we assume data items have unit size, unit miss cost, and any fixed positive writeback cost.

2. Our main result is an online algorithm, called Writeback-Aware Landlord, and an analysis showing that it achieves the optimal bound.

   Our algorithm and analysis are a careful generalization of the well-studied Landlord algorithm [117] to properly account for the distinction between clean and dirty items. Compared to Blelloch et al.'s [22] partitioning-based algorithm, our new algorithm uses a completely different approach/analysis (no cache partitioning), handles general sizes and costs, and improves the bound from 3-competitive with $3\times$ more cache to 2-competitive with $2\times$ more cache.

3. Although we prove a competitive ratio between our algorithm and the offline optimal algorithm, computing that optimal algorithm is hard. We extend Farach-Colton and Liberatore's NP-completeness proof to show that the Writeback-Aware Caching Problem is NP-complete regardless of the items' writeback cost(s) and miss cost(s). We further show the Writeback-Aware Caching Problem is Max-SNP hard, using a reduction from the 3D-matching problem.

4. Because finding an exact solution is difficult, we turn to approximations. We show that for a trace with maximum writeback to load cost ratio ω, Furthest-in-the-Future, the optimal deterministic policy when ignoring writebacks, is only a $(\omega+1)$-approximation to optimal. We also provide an algorithm that is a 2-approximation of the cost difference between a cache of size zero and the optimal cache.

5. Furthermore, we provide practical algorithms for bounding the offline optimal cost from above and below. Although there are no formal guarantees of their accuracy, we show they work reasonably well for large real-world traces that would otherwise be difficult to analyze.

6. Finally, we perform a detailed experimental study using real-world storage traces. Our main finding is that Writeback-Aware Landlord outperforms state-of-the-art online replacement policies when writebacks are expensive, reducing the total cost by 14% on average across these traces. This illustrates the practical gains of explicitly accounting for writebacks.

Related Work. Theoretical work on the caching problem traditionally begins with the offline version of the problem. Different variants of the problem have been introduced and widely studied, including optimal solutions, complexity analyses, and approximation schemes [3, 11, 13, 24, 35, 36, 85].

Initial work comparing the offline and online versions of caching problems was done by Sleator and Tarjan [102]. They provided a lower bound for the competitive ratio and showed that several deterministic algorithms had matching upper bounds in that model. Fiat et al. [50] provided a similar bound for randomized policies and showed ways of approximating online policies using other online policies. Young [116] found that the 'greedy-dual' algorithm for the variable cost model matched Sleator and Tarjan's bound. He later generalized this algorithm to the variable sizes and obtained a matching upper bound [117] using the Landlord algorithm.

Even et al. [47] considered a model where the cost and size of an item can change when it is accessed. Although this has some similarities to the model we introduce, neither the model nor their online algorithm can accurately model writebacks.[1]

The effects of writebacks have been well studied at the storage layer. Some of this work [54, 101] studies how to schedule writebacks to disk in order to minimize cost. Other work studies using write caches in front of storage to achieve sequential rather than random performance [18, 103]. These works provide many useful ideas that could be used to extend this work, but ignore the issues that arise with cache workloads containing mixed reads and writes.

With the emergence of highly asymmetric memory technologies, the systems community has begun to investigate the effects of writebacks on cache performance. Zhou et al. [120], motivated by phase-change memory technology, explicitly considered writebacks and proposed a partitioning scheme to reduce the effect of writes to main memory. Wang et al. [113] and Qin and Jin [98] provided similar techniques for reducing writebacks to memory by keeping track of frequently written items. These replacement policies lack worst-case bounds, and in fact it is not hard to construct request traces that yield arbitrarily bad performance.

On the theory side, we are aware of only two prior works. Back in 2000, Farach-Colton and Liberatore [49] studied a local register allocation problem that is a special case of writeback-aware caching with unit size data items, unit miss cost and unit writeback cost. They showed the offline decision problem is NP-complete using a reduction from set cover. Second, Blelloch et al. [22] provided a writeback-aware online algorithm that is 3-competitive to offline optimal when given $3\times$ the cache size, for the setting with unit size, fixed miss cost and fixed writeback costs. Their algorithm partitions the cache into a dirty half and a clean half, and applies Sleator and Tarjan's analysis [102] to each half.

3.1 Problem Formulation

3.1.1 Traditional Caching

The widely studied *caching problem* focuses on a single level of the memory hierarchy (cache), with capacity k, that must serve a *trace*, which is a sequence of requests for data. A request is considered to have been served when the cache contains or loads the data *item* associated with that request. Associated with each item e is a size $S(e)$ and a load cost $L(e)$. In order to load e, the cache first evicts items from the cache as needed in order to have $S(e)$ available space, and then pays $L(e)$ to load the item. Solutions to the caching problem, known as *replacement policies*, are strategies for selecting items to evict in order to minimize the total cost of the loads. *Offline* policies are given the entire trace in advance, whereas *online* policies observe the next request in the trace only after serving the previous request.

Variants. For the *generalized* caching problem (*generalized model*), the cost and size of an item may be arbitrary positive functions. Simpler versions include, for all items e: (i) the *basic model* in which items have unit size and cost: $S(e) = L(e) = 1$, (ii) the *bit model* in which cost equals size: $S(e) = L(e)$, (iii) the *cost model* in which items have unit size: $S(e) = 1$, and (iv) the *fault model* in which items have unit cost: $L(e) = 1$ [3].

[1] Personal communication with Guy Even at SPAA'18.

3.1.2 Writeback-Aware Caching

We modify the caching problem to account for writebacks by identifying each request in the trace as either a read or a write. An item in the cache is *dirty* if either (i) it was loaded as a result of a write request or (ii) there has been a write request for the item since it was loaded. All other items in the cache are *clean*. Because clean items have no changes that need to be propagated to memory, evicting them has no cost. However, dirty items need to be written back to memory upon eviction. The *Writeback-Aware Caching Problem* (WA Caching Problem for short) adds a writeback cost $V(e)$ for evicting an item e that is dirty, and modifies the goal to be minimizing the sum of the miss and writeback costs.

Definition 1. *In the (generalized)* Writeback-Aware Caching Problem*, we are given* (i) *a cache size k,* (ii) *an (online or offline) trace σ of requests, where each request consists of an item e and a flag indicating whether it is a read or write, and* (iii) *each item e has an associated size $S(e) > 0$, miss cost $L(e) > 0$ and writeback cost $V(e) > 0$. Starting and ending with an empty cache, the goal is to minimize the sum of the miss and writeback costs while serving all the requests in σ.*

Since none of the original parameters of the caching problem are changed, any variant of the original problem can be made writeback-aware. In fact, the original caching problem is equivalent to setting the writeback costs to zero, i.e. $V(e) = 0 \ \forall \ e$. Unless stated otherwise, when we refer to the WA Caching Problem, we mean the generalized variant defined above.

Frontloading Writeback Accounting. Caches in a writeback-aware setting pay costs at two different times: upon retrieving an item that is not in the cache, and upon evicting a dirty item. Having to consider costs upon eviction increases the complexity of analysis and encourages online policies to maintain dirty items past the point of usefulness in order to delay paying the eviction cost. To prevent these issues, when calculating the cost of a policy run on a prefix of a trace, we will charge the cost of writebacks to the write access that dirtied the item. Writes to items that are already dirty are not charged, because they do not result in additional writebacks. In other words, write accesses are charged both for loading the item (if not already in the cache) and writing it back (if it is not already dirty). This does not affect a policy's total cost for the full trace, because each charged writeback will happen later when the item is eventually evicted (recall that the cache must be empty at the end of the trace).

3.2 Writeback-Aware Landlord

We present a deterministic online algorithm called Writeback-Aware Landlord, and show that it achieves the optimal competitive ratio for deterministic algorithms.

3.2.1 Algorithm Description

Our algorithm is based on the classic Landlord algorithm [117]. In Landlord, there is a credit assigned to each item that is used to determine how long the item will remain in the cache. When an item e is accessed, its credit is set to its load cost $L(e)$. Whenever items must be evicted to make space in the cache, Landlord decreases the credit of each item in proportion to the item's size until an item reaches zero credit. This item (or items) may then be evicted.

```
1  def WritebackAwareLandlord(item e, bool write):
2      if e is not in cache:
3          # make space for the item
4          while freeSpace < e.size:
5              # find victim
6              minRank, victim = infinity, none
7              for f in cache:
8                  credit = f.wbCredit + f.loadCredit
9                  if credit / f.size < minRank:
10                     minRank = credit / f.size
11                     victim = f
12             evict(victim)
13             # decrease other items' credit
14             for f in cache:
15                 delta = f.size * minRank
16                 # decrease wb credit first
17                 if delta > f.wbCredit:
18                     f.loadCredit -= (delta - f.wbCredit)
19                     f.wbCredit = 0
20                 else:
21                     f.wbCredit -= delta
22         # add the item to the cache
23         insert(e)
24     # update requested item's credit
25     e.loadCredit = e.loadCost
26     if write:
27         e.wbCredit = e.wbCost
```

Figure 3.1: Writeback-Aware Landlord assigns each item two *credit* values: one for loads and one for writebacks. On access, an item's credits are updated to the cost of the request (i.e., writeback cost for writes). When needed, the item with the least credit per size is evicted, and all other items' credits are reduced in proportion.

To adapt Landlord to the writeback-aware setting, we must account for writeback costs. In particular, we must determine how to balance loads and writebacks in a way that leads to an optimal competitive ratio. Our algorithm, called *Writeback-Aware Landlord* and shown in Figure 3.1, maintains two separate credits that are increased independently. In particular, accessing (including writing) an item e sets (increases) its load credit to $L(e)$ and writing e sets its writeback credit to $V(e)$. The algorithm described in Figure 3.1 chooses to decrease the writeback credit before decreasing load credit, but the algorithm maintains optimality regardless of how the decrease is spread between the two types of credits.

$$\Phi = (h-1) \times \sum_{f \in WALL} \big(credit_l(f) + credit_w(f)\big)$$
$$+ k \times \sum_{f \in OPT} (cost_l(f) - credit_l(f))$$
$$+ k \times \sum_{f \in OPT \text{ and dirty}(f)} (cost_w(f) - credit_w(f))$$

Figure 3.2: The potential function used to prove the competitive ratio for Writeback-Aware Landlord. Here, $WALL$ refers to the contents of the cache for Writeback-Aware Landlord and OPT refers to the contents of the cache of the offline optimal policy. The first term is the sum of the credits of each item in WALL's cache. The second and third terms are the difference between how much cost was paid for an item to enter OPT's cache and how much credit that item retains in WALL.

3.2.2 Proof of Optimality

We show that Writeback-Aware Landlord has a competitive ratio matching the lower bound proven by Sleator and Tarjan [102] for the basic variant of the traditional caching problem. This means that Writeback-Aware Landlord is an optimal deterministic online writeback-aware policy, and that adding consideration of writebacks does not reduce the competitiveness of online policies compared to offline policies.

Theorem 13. *Writeback-Aware Landlord with size k has a competitive ratio of $k/(k-h+1)$ compared to the optimal (offline) algorithm with size $h \leq k$.*

Proof. We consider the contents of two caches: the first is size h and makes optimal caching decisions (OPT), and the second is size k and runs Writeback-Aware Landlord (WALL). Both caches serve the same request trace. For the purposes of the analysis, we say that OPT uses its cache to serve the request first, and then WALL serves the request using its own cache. In Figure 3.2 we define a potential function Φ, which is carefully designed to capture both how resistant WALL is to change, and how far it is from the state of OPT.

We show that: **(i)** Φ is zero at the beginning of the trace, **(ii)** Φ is never negative, **(iii)** Each cost c paid by Writeback-Aware Landlord can be charged to a unique decrease in Φ of at least $(k-h+1)c$, **(iv)** Φ can only ever increase by an amount kc when the optimal algorithm pays a cost c. Combining invariants (i) and (ii) means that Φ can never have decreased more than it has increased. Invariant (iii) upper bounds the cost paid by WALL as a function of the decrease in potential. Invariant (iv) upper bounds the increase in potential as a function of the cost paid by OPT. These results combined show that proving the invariants will suffice to prove the theorem.

We now provide a proof for each invariant.

(i). At the beginning of a trace, the cache is empty. Therefore, each summation is empty and Φ is zero.

(ii). Each negative credit term corresponds to a positive cost term with the same constant. Credit values can never exceed the associated cost of that item. Therefore, Φ is always non-negative.

(iii). Consider any access that causes charges to WALL. Such an access must target an item that is not in the cache, or dirty a clean item in cache. If the item is not in the cache, WALL performs eviction(s) to clear space, and then loads the item. Evicting an item with no credit has no effect on Φ. We then apply a modified form of Young's analysis [117] to the combined credit to show that Φ does not increase when WALL reduces credit.

Young's analysis is applied to the potential function used to analyze Landlord (LL), which is similar to that of WALL, but does not contain any terms involving writebacks. The analysis compares the total size of items in LL that decrease in credit to the total size of such items in OPT. Since decreasing credit only occurs in order to make space for a requested item and OPT processes requests before LL, we know that the requested item is in OPT's cache but not LL's at the time of the access. This means that the size of items in OPT that have their credit decreased by LL is at most the size of OPT minus the size of the requested item. Furthermore, since LL is evicting items to make space, it must contain a total size greater than its size minus the size of the requested item. The ratio of LL's effected object size to OPT's effected object size is thus greater than the ratio of the two cache sizes, which means that the decrease in potential due to the first term outweighs the increase in potential due to the second term.

When we apply Young's analysis to WALL, we see that the aggregrate credit decrease in the first term will remain the same. However some of this decrease will occur in writeback credit rather than load credit. For items that are clean in OPT's cache, this decrease in credit will not show up in the second and third terms of Φ. Since these omitted reductions are to negative terms, Φ will decrease overall.

We then consider the change in credits for the accessed item. If the item was not in the cache, its load credit changes from zero to its load cost. If the access dirtied the item, the writeback credit changes from zero to the writeback cost. This means that the total credit increase i of the item is at least the cost c charged to WALL by the access. Since the item has just been accessed (and OPT serves requests before WALL), it must also be in OPT and be dirty if the access was a write. This means that the second and third terms cause Φ to decrease by ki, while the first increases Φ by $(h-1)i$. Since $i \geq c$, and the potential change due to eviction is not positive, any access that causes a charge c to WALL causes Φ to decrease by at least $(k-h+1)c$.

(iv). We now show that any increase in Φ can be charged to costs paid by the optimal algorithm. Φ can increase due to changes in credits, OPT loading items, or items in OPT becoming dirty. Credits only decrease when WALL is evicting items, which we have already shown does not increase Φ. The credit for an item is only increased when WALL serves an access to that item. In such cases the item must also be in OPT. Thus, the decrease in Φ due to the second and third terms outweigh the increase due to the first. When an item is loaded into or becomes dirty in OPT, Φ increases by kc where c is the load cost, writeback cost, or their sum, depending on the transition. However, c is exactly the amount that OPT is charged to load these items. □

This proof shows that Writeback-Aware Landlord can perform no worse than Sleator and Tarjan's [102] lower bound. Writeback-Aware Landlord therefore achieves the optimal competitive ratio for online deterministic policies.

3.3 Offline Complexity Results

In 2000, Farach-Colton and Liberatore (FL) showed that the basic offline writeback-aware decision problem (the basic variant of the traditional caching problem from Section 3.1 with unit cost writebacks) is NP-complete using a reduction from set cover [49]. We extend their results to cover the more general Offline WA Caching Problem and further show that problem is MaxSNP-Hard.

3.3.1 NP-Completeness

The FL Reduction. The set cover problem is: given a set of elements and non-empty subsets of that set, find a collection of subsets (a cover) of minimum cardinality such that the union of the collection equals the original set of elements. The FL reduction generates an instance of the basic offline writeback-aware problem from an instance of set cover. The cache size is set to the number of subsets. The reduction uses one item in the trace for each element and each subset in the set cover instance. We refer to items associated with elements as element items and items associated with subsets as subset items. For each element, we refer to the subsets that contain it as adjacent subsets, and other subsets as non-adjacent subsets.

The generated trace consists of a write to each subset item, followed by a subtrace for each element. The subtrace for an element consists of a write to the associated element item, followed by a read of the element item and the non-adjacent subset items. This read pattern is repeated a total of four times.

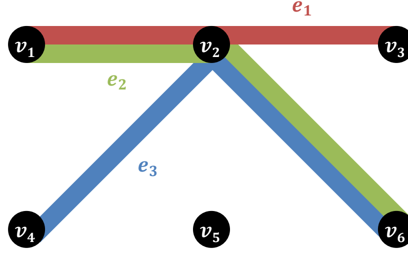
The FL reduction shows that any solution to the set cover problem maps to a solution to the caching problem, and that any optimal solution to the caching problem can be converted to a solution to the set cover problem. The high-level idea is that writing back a subset item corresponds to choosing that subset for the cover.

Updating the Reduction. There are two primary differences between the FL model and ours. The first is that the FL model assumes that both loads and writebacks have unit cost for all items, while we support different costs for each item and access type. The second is that in the FL model data does not need to be written back to memory if it is not evicted prior to its last use, whereas we assume all dirty items must eventually be propagated to storage.

To adapt the FL reduction to our data propagation model, we replace each write to an element item in the generated trace with a read to the same item, and we add a write to each set item at the end of the trace.

To support general writeback costs, we define the parameter ω as the maximum writeback cost to read cost ratio for any item. We modify the FL reduction such that the read pattern of the subtrace is repeated $\lfloor \omega + 3 \rfloor$ times rather than four times. This ensures that repeated reads in a subtrace are more valuable than the single writeback that could be saved by forgoing them.

Making these adjustments allows the FL reduction to reduce the set cover problem to the Offline WA Caching Problem. Since set cover is NP-Complete, this suffices to show that the Offline WA Caching Problem is also.



$\mathbf{G_1}$	$\mathbf{G_2^1}$			$\mathbf{G_3}$	$\mathbf{G_2^2}$			$\mathbf{G_3}$	$\mathbf{G_2^3}$			$\mathbf{G_3}$	$\mathbf{G_2^4}$			$\mathbf{G_3}$	$\mathbf{G_2^6}$			$\mathbf{G_3}$	$\mathbf{G_1}$
$W(e_1)$				e_1				e_1				e_1	e_1	e_1	e_1	e_1	e_1	e_1	e_1	e_1	$W(e_1)$
$W(e_2)$				e_2				e_2	e_2	e_2	e_2	e_2	e_2	e_2	e_2	e_2				e_2	$W(e_2)$
$W(e_3)$	e_3	e_3	e_3	e_3				e_3	e_3	e_3	e_3	e_3				e_3				e_3	$W(e_3)$
	$v_{1,1}$	$v_{1,1}$	$v_{1,1}$		$v_{2,1}$	$v_{2,1}$	$v_{2,1}$										$v_{6,1}$	$v_{6,1}$	$v_{6,1}$		
					$v_{2,2}$	$v_{2,2}$	$v_{2,2}$														

Figure 3.3: Example 3D Matching to WA Caching Problem Conversion. Performing the conversion on the hypergraph above results in the trace below. The trace should be read column-wise from left to right, where each column is read from top to bottom. Requests are reads unless otherwise specified. Gadgets are marked above the trace.

3.3.2 Max SNP-Hardness

We prove that the Offline WA Caching Problem is max SNP-hard using a reduction from bounded three-dimensional (3D) matching [73].

The 3D Matching Problem. Consider a hypergraph $G = (V, E)$. We say that $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices in G and $|V| = n$. Similarly, $E = \{e_1, e_2, ..., e_m\}$ is the set of hyperedges in G and $|E| = m$. Each hyperedge e_i consists of a subset of vertices from V that it connects. For each vertex, we refer to edges that contain that vertex as adjacent edges, and other edges as non-adjacent edges. A hypergraph G is tripartite if the vertices can be divided into three disjoint sets $V = \{V_1 \cup V_2 \cup V_3\}$, $V_1 \cap V_2 = V_2 \cap V_3 = V_3 \cap V_1 = \emptyset$ such that no edge contains more than one vertex from any set. A hypergraph G is three-uniform if each hyperedge is incident upon exactly 3 vertices. A hypergraph G is B bounded if no vertex has degree greater than B.

The maximum bounded 3D matching problem, given a bounded three-uniform tripartite hypergraph G, is to find the largest cardinality set of edges such that no edges in the subset share vertices. More formally, we define M to be a matching of $G = (V, E)$ if $M \subseteq E$ and $\forall\, e_i, e_j \in M, e_i \cap e_j = \emptyset$. We say a matching M of G is maximum if all other matchings M' of G contain at most as many edges as M, i.e. $|M| \geq |M'|$. The decision version of this problem is: given a hypergraph G and an integer k, decide if there exists a matching of cardinality k. This problem is known to be NP-Complete [74] and max SNP-hard [73].

Generating the Caching Instance. Given a 3D matching instance G, we will construct an instance P of the Offline WA Caching Problem such that any valid matching in G corresponds to a solution to P.

An example bounded three-uniform tripartite hypergraph and the generated trace are shown in Figure 3.3. We will make use of this example to to illustrate points throughout the construction.

Without loss of generality, we discard every vertex from the graph with degree zero.

The cache size of the generated instance will be equal to the number of edges m. The trace will use one ***edge item*** e_i for each edge $\bar{e}_i \in E$ and $d-1$ ***filler items*** $v_{(i,j)}, j \in [1, d-1]$ for each vertex $\bar{v}_i$, where d is the degree of $\bar{v}_i$. All items share a load cost of one and a writeback cost of ω, which can be any positive real. In the example, we set the cache size to three and use three edge items and four filler items (one each for $\bar{v}_1$ and $\bar{v}_6$, and two for $\bar{v}_2$). We also set $\omega = 0.5$.

Like the trace generated by the FL reduction, the trace we generate consists of a prefix and suffix, with a subtrace for each vertex in G. We will refer to both the prefix and suffix as gadget $\mathbf{G_1}$. This gadget consists of one write to each edge item. This gadget is shown in the first and last column in the example ($\{W(e_1); W(e_2); W(e_3)\}$.

Each subtrace will be composed of two gadgets. Gadget $\mathbf{G_2^i}$ will be created based on the vertex $\bar{v}_i$. This gadget will contain reads of every non-adjacent edge item and every filler item for $\bar{v}_i$. This read pattern will repeat a total of $\lfloor \omega \rfloor + 3$ times. For the example, the gadget $\mathbf{G_2^1}$ generated for vertex 1 would look like $\{R(e_3); R(v_{(1,1)}); R(e_3); R(v_{(1,1)}); R(e_3); R(v_{(1,1)})\}$ for $\omega < 1$. The second gadget in the trace, $\mathbf{G_3}$, consists of reads to each edge item.

Mapping Solutions. Consider any maximum matching for G. We generate a solution for the caching instance as follows: For any time during $\mathbf{G_1}$ or $\mathbf{G_3}$, all m active items can all be kept in the cache. During $\mathbf{G_2^i}$, the $m-1$ items that are being read during the gadget are kept in cache. In addition, if an edge adjacent to $\bar{v}_i$ is in the matching (there can be at most one), that edge item is kept in the cache during the gadget. Otherwise, any of the remaining items can be chosen to remain in cache. The cost of the resulting solution is $7m - 2n + 2m\omega - \omega k$ for a matching of size k.

We now show that the solution generated for the maximum matching is the optimal solution to the caching instance. Because the only cache contention is during $\mathbf{G_2^i}$, we can ignore $\mathbf{G_1}$ and $\mathbf{G_3}$. During $\mathbf{G_2^i}$, retaining each read item for the entirety of the gadget saves $\lfloor \omega \rfloor + 2$. Retaining items that are not read during the gadget can save at most $\omega + 1$ (one read and one writeback) per item. It is thus optimal to retain all read items and one adjacent edge item. An edge item can only avoid a writeback if it is retained across all vertex subtraces. Because the matching solution will retain the edge items for each edge in the matching at all times, the matching with the most edges will have the greatest writeback savings.

To generate a solution to the matching problem from the caching solution, simply take for the matching every edge associated with an item held during the entire trace.

Lower Bounding the Size of the Matching. For any 3-uniform tripartite hypergraph G with maximum vertex degree B and m edges, the size of the maximum 3D matching $k \geq m/(3B-2)$.

Consider any edge e in the input graph G. Because G is 3-uniform, e must be incident upon exactly 3 vertices. Each of these vertices can have at most $B-1$ edges other than e incident upon them. e can be adjacent to at most $3(B-1)$ other edges. For any maximum matching M, if none of the edges adjacent to e are in M, then e must be in M. Because we consider any edge in the input graph, there can be at most $3(B-1)$ edges not in M for each edge in M. Dividing the m edges in G by the ratio of edges in the matching finishes the proof.

Generating an Approximation Algorithm. Assume there exists a $1+\epsilon$ approximation algorithm A for the Offline WA Caching Problem. Consider an instance M of the 3D matching problem with maximum matching size k. Let x and x' be the cost of the optimal solution and the solution generated by A, respectively, for the Offline WA Caching instance generated by applying the

process above to M. We know from the solution mapping that $x = 7m - 2n + 2m\omega - \omega k$. By algebra, we see that $k = (7m - 2n + 2m\omega - x)/\omega$ and the same relation holds for k' and x'.

When we subtract the k' equation from the k equation, we see that $k - k' \leq (x' - x)/\omega$. By plugging in the relationship between x and x', we get $k - k' \leq \epsilon x/\omega$. By bounding x as a function of m and using the bound relating m and k above, we see that $k - k' \leq \epsilon(7 + 2\omega)k(3B - 2)/\omega$. Rearranging, we get $k' \geq k(1 - \epsilon(7 + 2\omega)(3B - 2)/\omega)$. As ω becomes large, this becomes $k' \geq k(1 - \epsilon 2(3B - 2))$.

This means that any constant approximation algorithm for the Offline WA Caching Problem can be used as a constant approximation to Bounded 3D Matching. Since the matching problem is max SNP-complete, the Offline WA Caching Problem is max SNP-hard.

3.4 Approximations with Theoretical Guarantees

Since we have shown that solving the Offline WA Caching Problem in polynomial time is not possible, we turn to approximation schemes.

3.4.1 Analyzing the Writeback-Oblivious Optimal

The Furthest-in-the-Future (FitF) policy [13, 85], which evicts the item accessed furthest in the future, optimally solves the basic version of the traditional offline caching problem. However, it cannot distinguish between reads and writes. This means that there are traces where FitF will use the minimum number of loads, but each load will also cost a writeback, whereas the optimal policy can achieve the same number of loads while avoiding the writebacks.

Theorem 14. *FitF is an $\omega + 1$ approximation to the basic Offline WA Caching with maximum writeback cost ω. This bound is tight.*

Proof. Consider a basic Offline WA Caching instance. Let L_B and L_A be the number of loads in the solution generated by FitF and algorithm A, respectively. Because FitF minimizes loads, $L_B \leq L_A$. The number of writebacks an algorithm suffers cannot be greater than the number of loads it suffers, so $W_B \leq L_B$. Through substitution: $Cost_B = L_B + \omega W_B \leq (1 + \omega)L_B \leq (1 + \omega)L_A \leq (1 + \omega)Cost_A$.

We now provide a family of traces where the solution generated by FitF has $\omega + 1 - \epsilon$ times the cost of the optimal solution for arbitrarily small values of ϵ. For a cache of size k, we generate a trace T using $k - 1$ dirty items and $k - 1$ clean items. T consists of a read to each clean item, followed by a write to each dirty item. The family F of traces consists of each trace that is generated by an integral number of repetitions of T.

Because FitF loads the clean items first and makes eviction decisions when they are closer to reuse than the single dirty item in cache at the time, FitF will retain all clean items for the duration of the trace. The optimal solution is to retain all dirty items for the duration of the trace. In each iteration after the first, FitF will suffer $k - 1$ loads and $k - 1$ writebacks, while the optimal solution will suffer only $k - 1$ loads. Thus the ratio of costs for all iterations after the first will be $\omega + 1$. □

$(A, \mathbf{W})$, $(B, \mathbf{W})$, $(F, \mathbf{R})$, $(B, \mathbf{W})$, $(C, \mathbf{W})$, $(D, \mathbf{R})$, $(G, \mathbf{R})$, $(D, \mathbf{R})$, $(A, \mathbf{W})$, $(E, \mathbf{W})$, $(H, \mathbf{R})$, $(E, \mathbf{W})$, $(C, \mathbf{W})$

Figure 3.4: An Example Trace that Breaks Stack Algorithms.

Aside: Stack Algorithms. One reason that FitF is not optimal is that it is a so-called *stack algorithm* [85]. Stack algorithms are replacement policies where the content of a larger cache is always a superset of the content of a smaller cache serving the same trace.

Stack algorithms are useful for several reasons. On an intuitive level, they make the problem easier to reason about, because each cache decision can be considered individually. Stack algorithms can be more easily computed using greedy algorithms or dynamic programming. They are also easy on system designers, as multiple cache sizes can be simulated on a trace simultaneously [85].

To the best of our knowledge, stack algorithms have not been investigated in any model other than the basic caching model. Here, we show that no stack algorithm is optimal in the presence of multiple item costs or multiple item sizes.

Consider the Offline WA Caching instance shown in Figure 3.4. The optimal solution for a cache of size 2 is to hold items B, D, and E in the cache. When the cache size increases to 3, a stack algorithm must keep each of these items. However, the optimal solution is to hold A, B, C, and E in the cache, dropping D. This means that the optimal solution is not a stack algorithm.

Such bad cases are not limited to small cache sizes, or to a single change in cache size. It is possible to construct an example where the optimal solution for a cache of size k is not a subset of the optimal solution for a cache of size $k + 1$ for any value of k by modifying the trace in Figure 3.4 to replace each item and request in the trace with $k - 1$ items and one request for each of the replacement items, respectively. The optimal solution for a cache of size k retains all replacement items for B, D, and E, while the optimal solution for size $k + 1$ will replace one of the D items with one of the A items and one of the C items. Furthermore, as the cache increases in size from k to $2k$, the D items will gradually be replaced with A and C items.

Our construction holds for any variant of caching with multiple costs. As long as each request interval for A, B, C, and E provide more potential savings than the request intervals for D, then the cache will switch from D to A or C as soon as the space becomes available.

It is also straightforward to construct traces with multiple item sizes where stack algorithms are non-optimal. An example is having multiple items share the same time period with access frequency proportional to the square of item size. As the cache becomes large enough to accommodate larger items, these items will displace the lesser-used smaller items.

Because we have constructed examples that break stack algorithms for varying costs and varying sizes, we claim the following.

Theorem 15. *For any caching problem with multiple costs or multiple sizes, no stack algorithm is an optimal solution.*

3.4.2 A 2-Approximation for Savings

We give an algorithm that provides a 2-approximation of the *savings* of the optimal solution. We define the savings of a solution as the difference between the cost of the solution and the cost of

loading and then immediately evicting each item accessed by the trace.

Our algorithm considers loads and writebacks separately. Although running any writeback-oblivious optimal algorithm on the trace is an $\omega + 1$ approximation of the cost (see above), it will provide an upper bound for the savings that can be obtained due to loads. A similar bound for the savings due to writebacks can be found by running the same algorithm on a modification of the original trace that treats reads as having load cost zero and writes as having load cost equal to their writeback cost. As the eviction decisions of both of these algorithms are valid solutions to the original problem, we choose the one with greater savings as the approximate solution. Because the optimal savings must lie between the larger of the savings and the sum, we can be off by at most a factor of two.

This technique will likely perform well when the savings available in the trace are dominated by either loads or writebacks, but will perform poorly when the two metrics contribute evenly to the total savings.

3.5 Efficient Approximations for Practical Use

3.5.1 A Lower Bound for Optimal

We compute a practical lower bound for the cost of the optimal solution by considering the relaxed view of time introduced in Berger et al. [16]. In this view, the solution has capacity equal to the size of the cache multiplied by the length of the trace. Intervals between consecutive accesses to an item take up space equal to the product of the item size and interval length, and have cost equal to the savings obtained by holding the item in cache for the entire interval. By packing the cache with intervals of highest density, the ratio of interval cost to space, a solution is generated that reflects a cache with the same average size, but that can change size over the course of the trace.

To make this scheme writeback-aware, we add into consideration intervals between consecutive writes to the same item. These intervals are assigned cost equal to the sum of the costs of the load intervals to the item during the interval's time period and the item's writeback cost.

The addition of the writeback intervals also affects the packing scheme. While the writeback-oblivious version could simply choose intervals while it had space, the aware version must update the dependent intervals of each interval it selects. Chosen writeback intervals invalidate load intervals for the same item that occur during their time period. Chosen load intervals cause the writeback interval (if any) that shares an item and time period with them to decrease in cost and space by the corresponding values of the load interval. Despite these complications, the result is a lower bound for the optimal offline solution that is accurate and efficient for many real-world traces. Following the naming convention of Berger et al., we call this algorithm writeback-aware practical flow-based offline optimal - lower (WAPFOO-L).

3.5.2 An Upper Bound for Optimal

We similarly adapt the ideas of Berger et al. [16] to create a practical upper bound. Their bound relies on converting the instance of the caching problem to an instance of the minimum-cost flow

$$(A, \mathbf{W}), (B, \mathbf{R}), (A, \mathbf{R}), (A, \mathbf{R}), (A, \mathbf{W}), (B, \mathbf{R}), (C, \mathbf{R}), (C, \mathbf{W}), (C, \mathbf{R}), (A, \mathbf{W})$$

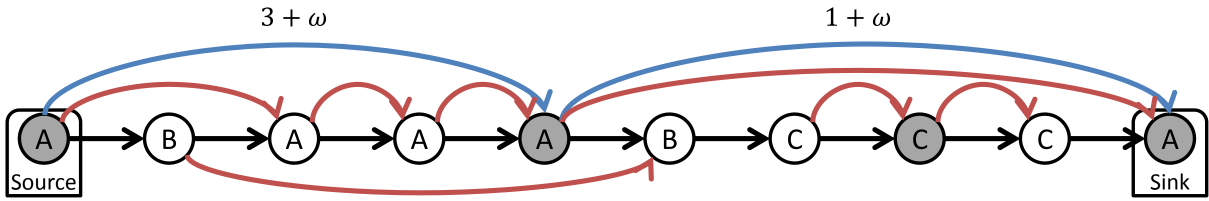


Figure 3.5: Example WA Caching Problem to MCF Conversion. The trace above is converted to the MCF problem below. All items are said to have load cost 1 and writeback cost ω. Black edges have cost 0 and capacity equal to the cache size. Red edges have cost -1 and capacity 1. Blue edges have labeled cost and capacity 1.

(MCF) problem. In the writeback-oblivious setting, this transformation completely captures the caching problem instance. However, computing the MCF for instances generated from large traces is prohibitively expensive. To make this more practical, Berger et al. consider subsets of edges at a time, breaking the graph into bite-size chunks and reducing the processing complexity. By applying the same principles used to make the lower bound writeback-aware, we can achieve the same result for the upper bound.

Minimum Cost Flow. The *minimum cost flow* (MCF) problem [94] consists of a directed graph $G = \{V, E\}$ and an amount of flow f. One vertex $s \in V$ is designated as the source vertex and another vertex $t \in V$ is designated as the sink. Each edge $e \in E$ has both a cost per unit flow $c(e)$ and flow capacity $u(e)$ associated with it. The goal of the problem is to route f units of flow from the source to the sink while minimizing the total cost. Each vertex other than the source and sink must have the same amount of flow leaving and entering.

Converting Between Problems. An example trace and the generated MCF problem are shown in Figure 3.5. The transformation creates one vertex in the graph for each request in the trace. For simplicity, we will refer to vertices as if they were the requests they represent. The first and last requests are chosen as the source and sink, respectively. To simulate empty cache space between requests, we generate an edge from each request to the next with cost 0 and capacity k. For modeling load savings, we generate an edge between subsequent requests to the same item with cost equal to the item's load cost and capacity 1. We model writeback savings with an edge between each write and the subsequent write to the same item. Edges representing writebacks have cost equal to the item's writeback cost plus the sum of the costs of load intervals for that item that overlap with the writeback interval. In the example, we show edges representing loads and writebacks in red and blue, respectively. We set the flow from source to sink to be equal to the size of the cache. The result is a directed acyclic graph (DAG) that approximates the cost savings that can be found in the instance of the basic WA Caching Problem.

Solving the generated MCF problem provides a close approximation to the solution of the original WA Caching Problem. It is not exact, because a solution to the MCF problem can obtain savings from an item twice during same time period. To correct this, after solving the

problem, we remove from the solution all load intervals that overlap in the access sequence with a writeback interval corresponding to the same item that is also in the solution.

3.6 Experimental Evaluation

To demonstrate that the theory behind Writeback-Aware Landlord holds up well in practice, we evaluate it against several state-of-the-art replacement policies on real-world storage traces [92]. This study shows that Writeback-Aware Landlord is effective in the presence of significant read-write asymmetry, reducing total cost to cache the trace by 41% over LRU and by 24% over GDS [27]. We further study how Writeback-Aware Landlord's performance varies for different writeback costs, performance metrics, and additional heuristics, and analyze from where its benefits come.

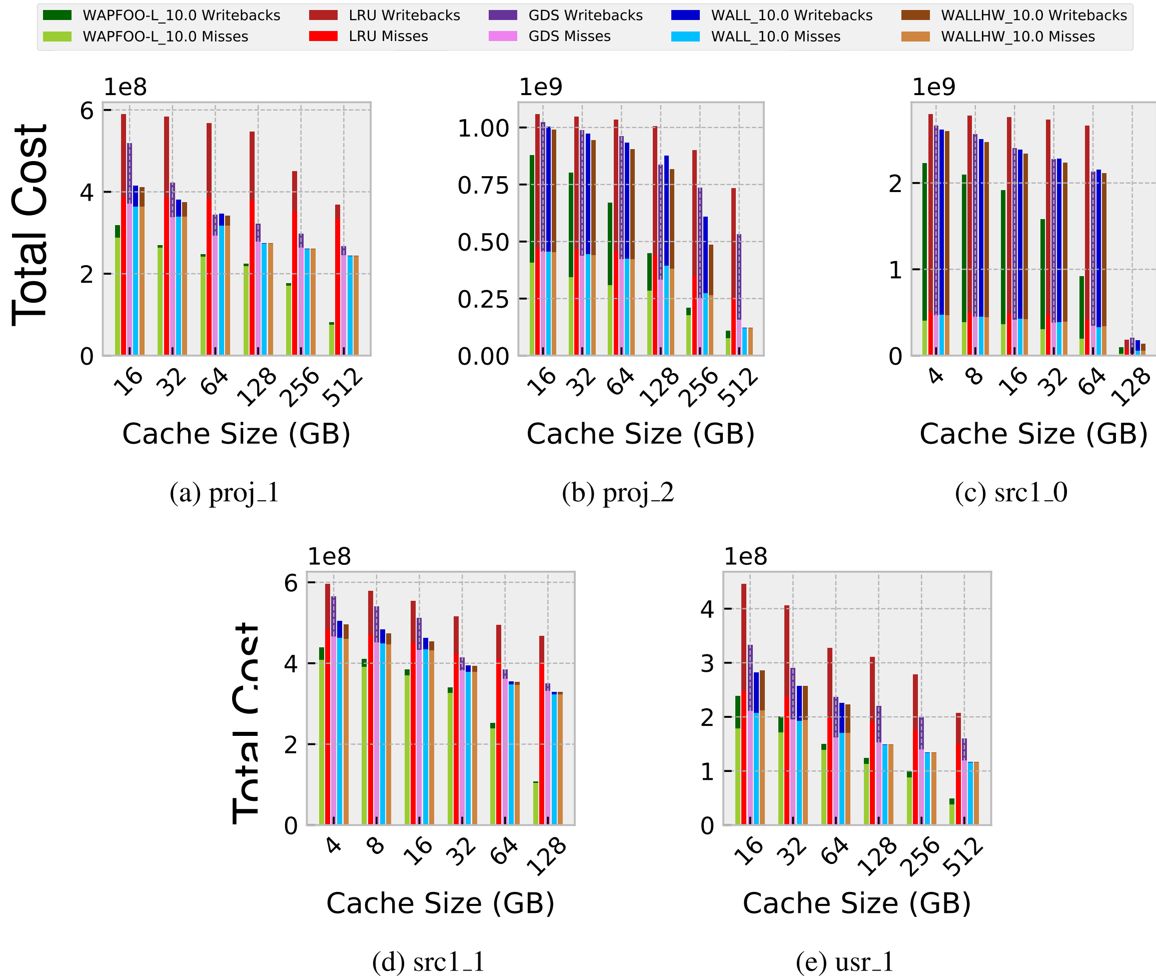
3.6.1 Methodology

Workloads. Our simulations make use of block traces from Microsoft Research (MSR) [92], which represent access patterns experienced by MSR servers, and represent many commonly seen behaviors. They are distributed in a format that specifies the time, type, offset, and size of the request. We use the size as specified and treat the offset as a request ID. We evaluate 512 M requests for each trace, replaying the trace if necessary.

Metrics. We compare policies on their total cost over the trace, as defined in Section 3.1.2. Because the traces do not specify cost, we consider each item to have unit load cost and writeback cost 10. We choose this value to be between the read-write asymmetries of emerging technologies like Intel Optane [41] and flash memory [56]. Choosing to have the same cost for each item regardless of size represents a system where the cost of communication between the cache and storage is largely independent of the amount of data being communicated, i.e., where latency trumps bandwidth.

Competing Policies. We compare Writeback-Aware Landlord against LRU, GDS, and WAPFOO-L. LRU is the simplest policy commonly used in practice, and works well on traces with high temporal locality. GDS is an efficient implementation of (non-writeback-aware) Landlord that considers item cost and size when making decisions. Both LRU and GDS have theoretical worst-case bounds on their performance similar to Writeback-Aware Landlord in the basic and generalized model, respectively (Section 3.1.1 describes these models). WAPFOO-L is the lower bound described in Section 3.5.1. Comparing against these policies allows us to isolate the importance of accounting for writebacks as well as see how much potential for improvement exists.

Implementation. The version of Writeback-Aware Landlord described in Figure 3.1 simplifies the theoretical analysis, but requires work proportional to the number of cached items for each eviction. Because this is impractical, we implement WALL in an equivalent fashion that requires only logarithmic work per eviction. Our implementation, based on Greedy Dual Size (GDS) [27], uses a global "inflation value" L rather than reducing credit and a min-heap for victim selection.



(a) proj_1 (b) proj_2 (c) src1_0

(d) src1_1 (e) usr_1

Figure 3.6: Total cost (misses + weighted writebacks) for different replacement policies on the five storage traces at cache sizes 4–512 GB.

We also test a version of WALL, which we refer to as WALLHW, that reduces load credit before writeback credit. As mentioned in Section 3.2, this does not affect the optimality of the algorithm.

3.6.2 Results

Figure 3.6 shows the total cost for the chosen caching algorithms across five different MSR traces for cache sizes from 4 to 512 GB. Each cost bar is split between cost due to misses, and costs due to writebacks.

The performance difference is fairly uniform across all traces, excluding *src1_0*. *src1_0* is an outlier: in this trace, 43% of accesses are writes, and the number of bytes written is an even larger fraction. Worse, these writes are distributed across a large number of distinct items, making it impossible for Writeback-Aware Landlord to signficantly reduce writebacks. The other traces have write percentages ranging from 5–12%, providing few enough writes for the extra credit they receive to be meaningful. WAPFOO-L follows the same general trends as the other policies,

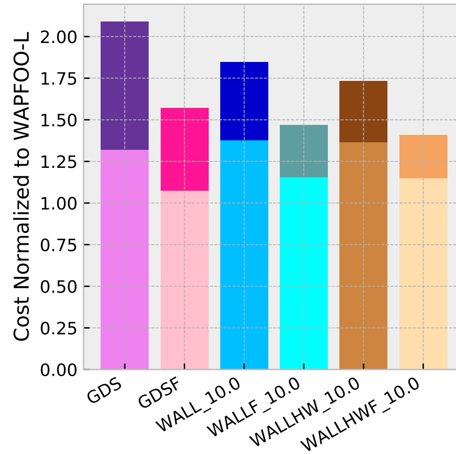


Figure 3.7: Total cost for different frequency-based replacement policies normalized to WAPFOO-L, averaged across all traces at 128 GB. The light and dark portions of each bar show the cost due to misses and writebacks, respectively.

but performs meaningfully better. This gap shows that there are still potential gains to be made by better replacement policies.

The arithmetic mean across traces and cache sizes of the miss cost of WALL is only 3.2% greater than that of GDS. However, WALL reduces writeback cost relative to GDS by 47%, significantly saving on writebacks without significantly harming hit rate. The result is that WALL's total cost is, on average, 88% of GDS, 72% of LRU, and 156% of WAPFOO-L. WALLHW performs even better, increasing miss rate by 2.6% for a 51% writeback cost reduction (compared to GDS). This results in average total cost that is 86% of GDS, 70% of LRU, and 151% of WAPFOO-L.

WALL Benefits from Additional Heuristics. It is common practice for systems to augment replacement policies with heuristics. Among the most popular heuristic is frequency, which says that items that have been requested frequently will be requested again.

GDSF [6] modifies GDS to account for frequency by multiplying an item's credit by the number of hits it has received while in the cache. Although this algorithm actually has worse theoretical guarantees than GDS, it performs well on real traces. We make a similar modification to Writeback-Aware Landlord, which we call WALLF.

Figure 3.7 shows the effect of the frequency heuristic on the costs incurred by GDS and WALL at a cache size of 128 GB. Costs are averaged across traces and, to avoid biasing results towards a particular trace, are normalized to WAPFOO-L. We see that considering frequency reduces the number of both misses and writebacks for all considered policies. These results suggest that writebacks share many of the locality patterns seen in loads, and that frequency is a useful indicator of utility. Both GDS and WALL see improvements from the frequency heuristic, although they are more pronounced in GDS.

The benefits of adding frequency to writeback-aware caches may be less than adding it to writeback-oblivious caches. This could be explained by the fact that both frequency and writeback-awareness are weighting particular items more heavily, which becomes less impactful as it affects more items.

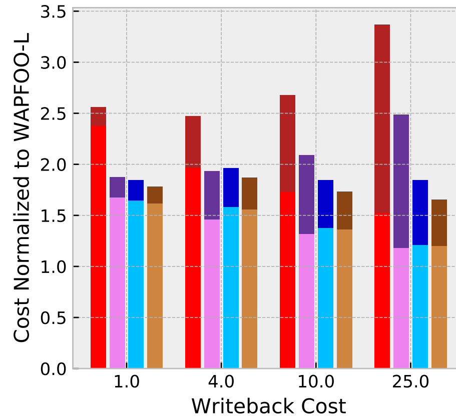


Figure 3.8: Total cost normalized to WAPFOO-L, averaged across all traces at 128 GB. Writeback-Aware Landlord's benefits improve as writeback cost varies from 1–25.

Sensitivity to Writeback Cost. Our previous results have assumed that writebacks are $10\times$ as expensive as reads. This cost asymmetry may have a large impact on caching decisions and the resultant costs. Figure 3.8 shows how the system changes with different writeback costs from $\omega = 1 \text{to} 25$. This represents a reasonable range from bandwidth-sensitive DRAM systems through storage technologies with heavy read-write asymmetry [56].

GDS does not consider writebacks, so its number of hits and writebacks remain constant. However, because these results are normalized to WAPFOO-L, these trends are seen as an increasing fraction of cost spent on writebacks.

WALL's results are more interesting, and show how it trades off misses and writebacks. Overall, WALL's total cost decreases relative to LRU and GDS as writeback cost increases, primarily because it manages to reduce the number of writebacks as they become more valuable. This is at the cost of additional loads, which can be seen in the miss costs for WALL rising relative to GDS as the writeback cost increases. These results show that WALL effectively accounts for cost asymmetry to reduce total cost.

3.7 Chapter Summary

As modern systems continue to evolve and make increasing use of NVM technologies, considerations like bandwidth, energy, and device lifetime will become more and more important. Prior work in caching, which focused on minimizing response time, has not considered writebacks, a crucial driver of performance in these metrics. We introduced the Writeback-Aware Caching Problem to fill this gap in our understanding. We showed that optimally solving Writeback-Aware Caching is hard even in the simplest setting and developed an online replacement policy with strong theoretical guarantees and good empirical performance. We believe that these results will help build a foundation for further theoretical and empirical work in caching on systems that provide improved lifetime, energy, or bandwidth.

Chapter 4

Block-Granularity-Aware Caching

Caches are pervasive in computer systems and often determine a system's end-to-end performance. A vast literature has studied caching, yielding important insights in the design of algorithms and computer systems. However, this literature has thus far neglected a critical component of many storage systems; the change in *access granularity* across levels of the storage hierarchy.

The ideal storage system would consist of a single large, fast level of memory. Given that such technology does not exist, real systems approximate this ideal via a hierarchy ranging from small, fast memories to larger and slower storage devices, starting from small SRAM caches (KBs), to larger SRAM caches (MBs), to off-chip DRAM memory (GBs), to persistent storage elements like NVM, flash, or disk (TBs). Each level of the storage hierarchy organizes its data in chunks of a specific size to simplify management and reduce overheads. For example, SRAM caches typically consist of 64 B "lines"; DRAM of "rows" of 2-4 KB; NVM of "blocks" ranging from 256 B to 4 KB; and flash and disk of "pages" of 4 KB [58].[1]

As a result, access granularity changes as one proceeds through the storage hierarchy. E.g., to satisfy a 64 B cache-line miss, DRAM must load a 2 KB row. But once the row is loaded, the other lines in the row can be read at considerably lower cost than the first line. So *what should caches do with the rest of the row?*

Most caches today ignore granularity change, and only load the smallest possible data granularity to service a request. But this ignores an opportunity to load some or all of the larger granularity at minimal cost. Such prefetching could be used to avoid subsequent cache misses on the loaded data, as long as it is possible to prevent the upper cache levels from becoming polluted with useless data. This observation is the critical question that our work addresses: *What caching opportunities and challenges are introduced by granularity change?*

We seek to answer this questions by developing a model of caching that accounts for granularity change and provides a strong framework for future inquiry.

Our Contributions. In this chapter, we initiate a general exploration of block-aware caching, seeking to make use of data that is already in-flight to improve cache performance. We make the following contributions:

1. We define the *Block-Granularity-Aware Caching Problem*, which generalizes the tradi-

[1] In fact, there are often different granularities for reads and writes, e.g., "erase blocks" in flash can be many MBs. We focus on reads in this work.

tional caching problem by grouping items into blocks that can be loaded together: Given a sequence of accesses to data items, a specified cache size, and an assignment of data items to blocks, the goal is to minimize the cost of loading blocks when servicing the sequence in order.

2. We analyze the complexity of the Offline Block-Granularity-Aware Caching Problem, showing that it is NP-Complete using a reduction from the variable-size caching problem.
3. We provide a lower bound on the competitive ratio of deterministic replacement policies in the Block-Granularity-Aware Caching Problem that shows the extra power available to caches in this model. We also provide a tighter lower bound for deterministic policies that operate entirely on a single data granularity. Using these bounds, we are able to show how the size of the cache being compared against affects relative policy performance. In particular, we show that caches that operate on the smaller granularity compete poorly against small caches, while caches that operate on the larger granularity compete poorly against large caches.
4. Our main result is an online algorithm, called Item-Block Layered Partitioning, and an analysis showing that it achieves a nearly optimal competitive ratio.

   Our algorithm uses a layered partitioning scheme to combine the best aspects of both small and large granularity caching. Compared to the baseline algorithms, Item-Block Layered Partitioning competes well for all common cache sizes while avoiding the performance cliffs that such policies suffer.

Related Work. The original caching problem is well studied and well understood from a theoretical perspective. The first major results were when Belady [13] and Mattson [85] separately devised optimal solutions for caching with unit size and cost items. Sleator and Tarjan [102] performed the initial work comparing the performance of *online* caches, which must make decisions as requests arrive, against *offline* caches, which are allowed to view the entire trace when making decisions. They provided a lower bound for the cost ratio of any deterministic online algorithm compared to the optimal offline algorithm for a worst-case trace in the unit size and cost items model (and therefore any model). Furthermore, they showed that several deterministic algorithms had matching upper bounds in that model. Fiat et al. [50] provided a lower bound for randomized online replacement policies compared to optimal and a randomized policy that matches that bound; they also showed ways of approximating online policies using other online policies.

Other variants of the problem have been considered, and considerable work has been done on complexity and algorithms for these variants [3, 11, 24, 35, 47, 116]. In particular, caching with variable-size items appears to be a problem quite similar to ours, since one could think of different subsets of items from the same block as different size items. It would be fortuitous to be able to apply the breadth of work on this problem [16, 36, 117]. However, there are important differences that make this impractical. In the variable-size problem, items are accessed in a manner where any quantity less than the whole is useless, whereas the items of a block can be accessed, loaded, and evicted individually. This results in a setting where choosing what items to load is an additional dimension with significant impact on performance.

To our knowledge, there is no theoretical work on accounting for granularity change. Some

work [20, 21] includes parameters for granularities at different levels of cache, but these works either assume that the granularity is constant throughout the hierarchy or ignore the effects on cache performance.

The systems community has taken several approaches to deal with the issue of granularity change. There is work on designing memory controllers at granularity boundaries that schedule accesses to increase locality [48, 90, 91, 119, 121, 122]. Another way to achieve this result is address mapping techniques, which attempt to place data such that data that are requested at similar times are located together [81, 106, 115, 118]. Another research target is how to manage swapping between rows in order to balance capturing a larger fraction of locality and latency for switching to different rows [89, 95, 104, 114]. There is also work that looks at how to allocate items to blocks when granularity changes to improve performance [8, 26, 33, 97].

Having caches understand and use granularity changes is complementary to these techniques. Caching decisions can benefit from the increased locality caused by memory controller or address mapping, further improving performance. Even better, moving the responsibility for spatial locality from the storage management to the cache provides the same benefits without the tension of switching between rows, at the cost of a small effective reduction in cache space.

4.1 Problem Formulation

Like traditional caching problems (see Section 3.1.1), the Block-Granularity-Aware Caching Problem (BGA Caching Problem) consists of a single level of memory (cache) that receives a series of requests for data. Each request, which we refer to as an access, is associated with one data item. If the item is in the cache, then the request is served and the cache is not charged. If the item is not in the cache, then the cache must load the item from the subsequent layer of memory or storage. If this load causes the amount of data in cache to exceed the cache size k, then items must be evicted from the cache to remedy the situation.

What makes the BGA Caching Problem different from traditional caching problems is that some data beyond the accessed item can be accessed at no additional cost. To represent this, we say that the universe of items is partitioned into *blocks* of size less than or equal to the block size $B \geq 1$. When the cache loads data from storage, it can load any subset of a block for unit cost. The BGA Caching Problem generalizes the caching problem. In particular, when each item is in a different block, this model exactly matches the traditional caching model.

Definition 2. *In the* Block-Granularity-Aware Caching Problem *(BGA Caching Problem), we are given* (i) *a cache of size k,* (ii) *an (online or offline) trace σ of requests, where each request consists of an unit size item e, and* (iii) *a partitioning of the items into disjoint sets (blocks) such that no partition contains more than B items. Starting with an empty cache, the goal is to minimize the number of times that a subset of a block (up to and including the entire partition) is loaded into the cache while serving all the requests in σ.*

The blocks that the items are partitioned into represent the larger granularity data chunks that the subsequent layer of the memory hierarchy operates on. In such systems, there is typically a small amount of memory, known as a buffer [58], that is used to handle data as it is being read or written. The cost of moving data from the mass storage into the buffer is typically large relative to the cost of operating on data in the buffer. This suggests that items that are brought into the

buffer can be accessed easily, motivating our model.

Assumptions. For this work, we assume that each item has unit size and each block has unit cost. Extending the model to support variable size items or variable cost blocks could be done, but is outside the scope of this work.

We also assume that the caches we study are much larger than the block size, ie $k > h \gg B$. In addition, we focus on deterministic policies for this work.

Baseline Policies. We consider two baseline cache designs that are commonly used in caches at a granularity boundary. An *Item Cache* admits only the item that is accessed on any request, ignoring the remaining items in the block. By contrast, a *Block Cache* admits all items in the requested block, and performs evictions on every item in a block at the same time. In other words, a Block Cache treats each request as if it is requesting the entire block rather than an individual item. These cache designs result from simply choosing one data granularity at the boundary, rather than adapting the cache design to account for granularity change.

4.2 Complexity Analysis

In this section, we investigate the complexity of the Block-Granularity-Aware Caching Problem. Using a reduction from the variable-size caching problem (also known as fault model, see Section 3.1.1), we are able to show that the Block-Granularity-Aware Caching Problem is NP-Complete, even with unit block costs and unit item sizes.

Theorem 16. *The Offline BGA Caching Problemis NP-Complete.*

Proof. Our proof relies on a reduction from the fault model of caching, which is known to be NP-complete [36]. We begin by showing how to create items for the BGA Caching Problem and then assign them to blocks. We then use these blocks to generate a trace where the cost paid by the optimal cache is the same as the optimal cost for the variable-size caching problem.

The first step of the reduction is to scale the variable-size caching problem to have integral item sizes. This can be done by multiplying the size of each item and the cache size by the same value (assuming the sizes are rational numbers). After the sizes are all integral, the items of the BGA Caching Problem can be created and partitioned. For each item in the variable-size caching problem, we create one block in the BGA Caching Problem. The maximum size of these blocks can be chosen to be any value greater than or equal to the size of the largest item in the scaled variable-size problem. For each of these blocks we will use only the first z items, where z is the size of the corresponding item in the scaled problem. We refer to these as the *active set* for that block.

The idea for trace generation is to create a trace for the BGA Caching Problem that has accesses to the same amount of cache space as in the variable-size problem. For each access in the variable size trace, the BGA Caching trace replaces it with consecutive accesses to the active set of the corresponding block. Each item is accessed a number of times equal to the number of items in the active set, in round robin order. The ordering of the variable-size trace is maintained, so that the ordering of the blocks in the BGA Caching trace is the same as the ordering of the

items in the variable-size trace. The size of the cache is set to be the same as that of the scaled variable-size problem.

We are left to show that the optimal cost of the BGA Caching Problem that we generate is the same as the optimal cost of the variable size caching problem. First, we show that scaling sizes does not affect the result. Since the cache size was scaled by the same factor as the items, the fraction of the cache space that each item takes remains unchanged.

Beyond this, we show that there is an optimal solution for the generated BGA Caching Problem instance that loads and evicts the entire active set of a block at the same time. To do this, we rely on the fact that an optimal solution can load all items from the active set that are not in cache for unit cost. This means that any consecutive series of requests to a single block can be served for that unit cost. However, unless the cache contains the entire active set, it must pay at least unit cost to serve the block.

Since the active sets corresponding to different items in the variable size problem are in distinct blocks, there is no way to load an active item without paying unit cost for that block. When combined with the fact that the active items remain unchanged for each block, we can show that evicting a single active item increases the cost paid by the cache the same as evicting all active items for that block.

If we assume that the cache evicts all active items for a block at the same time, then upon the first consecutive access to a block, either the entire active set will be in cache or none of it will be. If the set is in cache, no loads are required. If the set is not in cache, then loading less than the entire set will cause the need for additional loads. In particular, the maximum amount of cache space used for the active set multiplied by the number of loads must exceed the active set.

We use the repeated accesses to the active set to show that the benefits of such additional loads cannot outweigh the drawbacks. Since any item can be loaded for unit cost, the benefit of using less cache space for one set of consecutive accesses to the active set cannot exceed the amount of cache space reserved. Due to the repeated nature of the accesses, loading less than the entire active set will result in at least as many additional loads as the size of the active set. This means that loading the entire active set upon the first miss is optimal.

Since we have shown that an optimal solution is to load and evict entire active sets at a time, any access that immediately follows an access to the same active set will automatically be a hit. Putting these results together, the optimal solution to the generated trace is the same as the optimal solution to the original trace. □

Figure 4.1 shows an example of the reduction. We are able to simulate variable size items through the use of multiple items from the same block accessed consecutively. By repeating these access sequences, we force the optimal solution to load all items that are used in the block. The optimal solution to the input instance can easily be translated into an optimal solution to the generated instance.

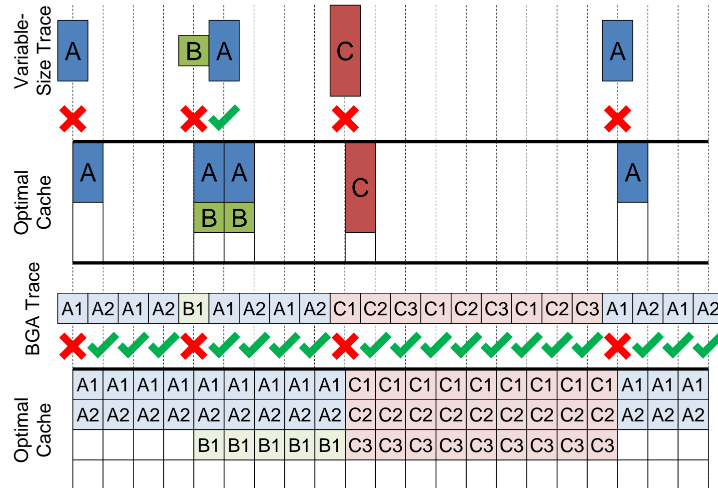


Figure 4.1: A diagram showing how to transform an instance of the variable-size caching problem into an instance of the Block-Granularity-Aware Caching Problem with the same cost. Notice that storing item A1 with block C would not decrease the number of misses, since A2 would still suffer a miss. The repeated accesses force the optimal solution to load the entire block in order to minimize misses.

4.3 Competitive Lower Bound

In this section, we will show how to adapt techniques for lower bounding competitive ratios in caching to the BGA Caching Problem. We will start by providing a lower bound for Item Caches and Block Caches, and then use the insights we gain to devise a more general lower bound.

4.3.1 Item Caches

We start by examining how Item Caches perform in the BGA Caching Problem. These policies are widespread and well studied, so they serve as a logical starting point for our investigation.

The lower bound for Item Caches in traditional caching comes from Sleator and Tarjan [102], and is defined as follows. We define k to be the size of the online cache and $h \leq k$ to be the size of the optimal cache.

1. Assume both the optimal and online caches are full.
2. Access $k - h + 1$ items that have not been seen before.
3. Create a set of items containing the items in the optimal cache during step one and the items accessed during step two. This set contains $k + 1$ items.
4. Access the item from the set that is not in the online cache. Repeat this process a total of $h - 1$ times.
5. To generate a longer trace, return to step one. Note that the assumption will be satisfied.

In these traces, the online policy never hits, as the items in step two are newly accessed, and the items in step four are chosen to be outside of that cache. The optimal policy also misses on every

access in step two. Since it knows the future accesses, it can perform evictions so as to store each item that will accessed in step four, allowing it to achieve hits on all of these accesses. This results in a competitive ratio of $k/(k-h+1)$.

The BGA Caching Problem problem modifies traditional caching by allowing hits to be achieved through spatial locality (two different small granularity items reside in the same large granularity item). This means that it is possible for the optimal policy to miss on only one out of every B accesses in step two if the items are chosen such that an entire block is accessed. The cost of this modification is that the optimal cache uses B space rather than unit space in step two, and thus step four must be shortened to $h-B$ accessses. This provides the following result:

Theorem 17. *Any Item Cache has a competitive ratio of at least $B(k-B+1)/(k-h+1)$ where k is the size of the online cache and $h \leq k$ is the size of the optimal cache.*

Proof. Create a trace according to the following procedure:

1. Assume both the optimal and online caches are full.
2. Choose a block that has not been seen before. Access each item from that block. Repeat this process until $k-h+1$ items have been accessed in this step.
3. Create a set of items containing the items in the optimal cache during step one and the items accessed during step two. This set contains $k+1$ items.
4. Access the item from the set that is not in the online cache. Repeat this process a total of $h-B$ times.
5. To generate a longer trace, return to step one. Note that the assumption will be satisfied.

Since the online policy does not load any item that is not accessed, it will miss on each access in step two. The accesses in step four are chosen to ensure that the online policy misses on each.

The optimal policy will load each block on its first access in step two, resulting in $(k-h+1)/B$ misses. It can use its remaining $h-B$ space to store the items that will be accessed in step four, hitting on each.

Taking the ratio of these misses provides the bound. □

This competitive ratio is nearly a multiplicative B factor worse than traditional lower bounds under the assumption that $k \gg B$. Since the eviction decisions of the policies, and hence their miss rates, have not changed, this shows the increased power of the optimal algorithm in this model.

4.3.2 Block Caches

Although Item Caches do not perform well in the BGA Caching Problem, perhaps Block Caches will. These policies are naturally suited to handle the sorts of access sequences that Item Caches suffered on, and may therefore offer a better realization of the potential performance improvements.

This intuition proves accurate for the trace design scheme described in the previous section. Online Block Caches will be able to hit on all subsequent accesses to a block in step two, providing the same number of misses as the optimal policy on that step. As long as the number of accesses in the second step dominates the number in the third, these policies will perform well.

The problem with Block Caches is that when a small number of items in a block are accessed, the remaining items will pollute the cache, reducing the available space to serve accessed items. In particular, when only one item from a block is used at a time, the cache is effectively B times smaller. We can apply this insight along with the traditional bound to provide a bound for Block Caches.

Theorem 18. *Any Block Cache has a competitive ratio of at least $k/(k - B(h-1))$ where k is the size of the online cache and $h \leq k/B$ is the size of the optimal cache. When $h > k/B$, the competitive ratio is infinite.*

Proof. Create a trace according to the following procedure:

1. Assume both the optimal and online caches are full and that each item in the optimal cache is from a different block.
2. Access one item from each of $\lceil k/B \rceil - h + 1$ blocks that have not yet been accessed.
3. Create a set of items containing the items in the optimal cache during step one and the items accessed during step two. This set contains $\lceil k/B \rceil + 1$ items, each of which are from a different block.
4. Access the item from the set that is not in the online cache. Repeat this process a total of $h - 1$ times.
5. To generate a longer trace, return to step one. Note that the assumption will be satisfied.

The online policy has not previously seen any of the blocks accessed in step two, so it will miss on each access. Because it loads and evicts at block granularity, it will store exactly $\lceil k/B \rceil$ blocks at a time. This means that step four can always choose one item that is not in the online cache, resulting in misses for all accesses in that step.

The optimal policy will miss on each access in step two, resulting in $\lceil k/B \rceil - h + 1$ misses. Since it knows what items will be accessed in step four, it can choose to keep those items and hit on each such access.

With the assumption that B evenly divides k, taking the ratio of the misses results in the target bound.

If $h > k/B$, then step four can be repeated infinitely, since the optimal policy can store the entire set of items. □

This result shows that Block Caches perform very poorly on traces that do not take advantage of the spatial locality that is available in the BGA Caching Problem. In particular, they suffer a performance penalty where the effective cache size is reduced by a factor equal to the ratio between the size of the block and the average number of items used per block.

4.3.3 Generalizing the Lower Bound

We would like to have a more general lower bound for policies beyond Item Caches and Block Caches. Here, we provide such a bound.

Like the worst-case traces for Item Caches and Block Caches, we will first access new items until the online cache can no longer store the entire active set, and then repeatedly ac-

cess whichever item the cache chooses not to store. Unlike the previous bounds, we cannot make assumptions about the granularity of loads or evictions.

When adding new items to the active set, we take advantage of the block structure to allow the optimal cache to outperform the online cache. For an accessed block, the worst-case trace can access any item from that block that is not in the online cache. This can continue until the online cache loads every item in the block. The online cache will miss on each access, while the optimal cache can load each accessed item on the first access to the block. Using this insight, we categorize policies using a parameter a, which is the number of distinct consecutive accesses to a block that must occur before the policy loads the entire block. Later on, we will remove this parameter to provide a more general bound.

Our trace construction is similar to what we use for Item Caches, with the differences being that the online policy will incur a misses on each block in step two and the optimal policy will need to use a space to service these requests, leaving only $h - a$ space available for step four.

Lemma 13. *For any deterministic replacement policy that requires $a \leq B$ consecutive distinct accesses to a block to load all of it, the competitive ratio of that policy is at least $(a(k-h+1) + B(h-a))/(k-h+1)$ where k is the size of the online cache and $h \leq k$ is the size of the optimal cache. When $h > k$, the competitive ratio is infinite.*

Proof. Create a trace according to the following procedure:

1. Assume both the optimal and online caches are full.
2. For $\lceil (k-h+1)/B \rceil$ blocks that have not yet been accessed:
   While there exists an item from that block that the online cache has not yet loaded, access that item. This will occur a times per block.
3. Create a set of items containing the items in the optimal cache during step one and the items in the blocks accessed during step two. This set contains at least $k+1$ items.
4. Access an item from the set that is not in the online cache. Repeat this process a total of $h - a$ times.
5. To generate a longer trace, return to step one. Note that the assumption will be satisfied.

As with our other generation schemes, the online policy will miss on every access. This causes $a\lceil (k-h+1)/B \rceil$ misses in step two and $h - a$ in step four.

The optimal policy will load each of the a items that will be accessed in a block on its first access in step two, resulting in $\lceil (k-h+1)/B \rceil$ misses. It can use its remaining $h - a$ space to store the items that will be accessed in step four, hitting on each.

With the assumption that B evenly divides $k - h + 1$, taking the ratio of the misses (and simplifying) results in the target bound.

If $h > k$, then step four can be repeated infinitely, since the optimal policy can store the entire set of items. □

4.3.4 Analysis and Discussion

Having devised lower bounds for deterministic policies in the BGA Caching Problem, we can use the insights we have developed to learn more about the original problem.

In order to minimize the lower bound, we need to consider the a parameter. In Theorem 19, the a parameter shows up in one positive term and one negative term, both in the numerator. When $k - h + 1 > B$, then the positive term dominates, and minimizing a also minimizes the competitive ratio. Otherwise, the negative term dominates and maximizing a minimizes the ratio. Thus, to achieve the best competitive ratio, one should load either an entire block or a single item, and nothing in between. This result is true even when we allow the a parameter to be non-constant, resulting in the following Theorem:

Theorem 19. *The competitive ratio of any deterministic replacement policy is at least* $(k+(B-1)(h-1))/(k-h+1)$ *where* k *is the size of the online cache and* $h \leq k - B + 1$ *is the size of the optimal cache.*

Proof. To show this, we first consider a single cycle of the trace used in Lemma 13. The online policy suffers misses equal to the sum of its chosen a parameters across each block accessed. The optimal cache, meanwhile, will suffer one miss on each block, using space equal to the maximum number of accesses for a single block. Since this space is forced to store particular items, it cannot be used in step four of the generation.

Combining these observations shows that the a parameter should be minimized for all blocks except for one. This one block provides the maximum value that appears in step four. When we analyze the miss ratio, we see that it is actually independent of this choice. The number of misses that the optimal cache suffers is not affected. For the online policy, it suffers a number of misses equal to the chosen value of the parameter in step two and h minus that value in step four. Thus, setting the a parameter to one minimizes the lower bound. Plugging that value into the bound from Lemma 13 provides the result. □

Since the only distinguishing characteristic of items in a block are whether they have been accessed or not, it makes sense that we would either want to bring in all or none of the non-accessed items after an access. However, it is less clear why an approach similar to that of the ski-rental problem, where the policy waits to amortize the cost of loading the entire block against individual item accesses is not correct. The answer appears to lie in the fact that the number of possible "rentals" is bounded by B and that the decision to "purchase" additional items can easily be changed when a different block is accessed.

The intuition for choosing between 1 and B for the a parameter can be found in the relative costs due to the types of locality. For caches where the online and offline compared are roughly equal size, the online cache needs as much space as possible to compete on traditional worst-case traces, and using cache space to serve spatial locality is more harmful than helpful. By contrast, when the online cache is much larger than the offline cache, the marginal benefit of the extra lines is small, making them more useful when devoted to handling spatial locality. In these situations, policies that load the entire block on access perform better.

We can also use these bounds to gain insights into eviction. The lower bound for Block Caches shows that evicting at the block granularity is inefficient. This means that efficient policies need to be able to classify items in a block into different priority levels. We achieve this by distinguishing between items that have been accessed, and those that have not (only another item in their block has been accessed).

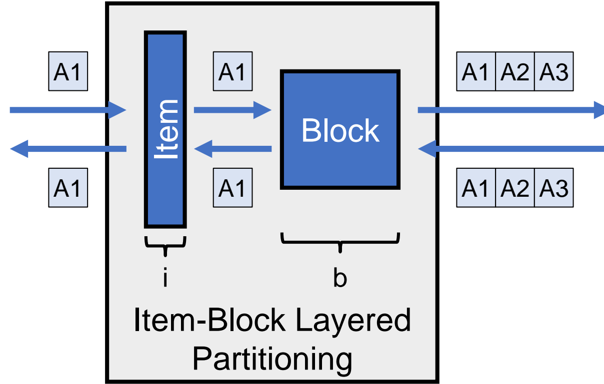


Figure 4.2: A logical diagram of IBLP, consisting of an Item Cache partition running LRU in front of a Block Cache partition running LRU.

Putting this all together, we see that in order to maximize performance, policies should load the entire block on an access (unless comparing against an optimal with similar size, where they should load individual items), to take advantage of spatial locality, but perform evictions in a manner that treats items that have been accessed differently from those that have not.

We can also gain broader insight about the overall problem. The lower bound we prove is much greater than the lower bound for the granularity-oblivious problem, meaning that the gap between online and offline policies is larger. The difference in the bounds starts at nearly a multiplicative factor of B when $k \approx h$ and tapers off, hitting 2 when $k \approx Bh$. In prior models, the augmentation factor equals the competitive ratio at 2. By contrast, in the BGA Caching Problem, $k = 2h$ has a competitive ratio of $\approx 2 + B$ and a competitive ratio of 2 requires $k \approx Bh$. The meeting point of the augmentation factor and the competitive ratio occurs when $k \approx \sqrt{B}h$. This helps illustrate the greater power available to offline policies in our model.

4.4 A Competitive Policy for the Block-Granularity-Aware Caching Problem

In this section, we provide a policy that performs well in the BGA Caching Problem. Our policy, Item-Block Layered Partitioning (IBLP), combines elements of Item Caches and Block Caches in a manner that effectively handles the corner cases that arise in either alone. We begin by describing how IBLP works, then move to proving the upper bound on its competitive ratio. We then analyze how to select the parameters for the policy and then conclude by comparing this policy to alternatives.

4.4.1 Policy Description

Our policy, known as Item-Block Layered Partitioning, divides the available space into two different layers of cache. The first layer, which serves each access to the cache, loads only the *items* that are accessed and performs evictions using the Least-Recently Used (LRU) replacement pol-

k	Online Cache Size
h	Optimal Cache Size
B	Block Size
i	Item Layer Size
b	Block Layer Size
$k = i + b$	IBLP partitioning

Table 4.1: BGA Caching Problem notation.

icy. The second layer, which only serves accesses that miss in the first layer, also uses the LRU policy for evictions, but loads and evicts at the granularity of *entire blocks* at a time. In other words, IBLP organizes the cache as an Item Cache backed by a Block Cache. We refer to these layers as the item layer and block layer, respectively, and define their sizes as i and b. A diagram of IBLP can be found in Figure 4.2.

Although this is relatively straightforward in description and implementation, it includes some subtle design choices. The cache is split into two different layers to handle the two types of locality, with the item layer handling temporal locality and the block layer handling spatial locality. The ordering of the two layers is important to ensure that accesses with high temporal locality do not reorder blocks in the LRU list of the block layer. Allowing such reorderings would cause blocks with a small number of frequently accessed items to pollute the block layer, reducing its effective space for worst-case traces. Note that the block layer is *neither* inclusive nor exclusive of the item layer. If the block layer was inclusive of the item layer (every item in the item layer must also reside in the block layer), the item layer would have no contribution to the overall hit rate. By contrast, having an exclusive policy (where no item is duplicated across layers), would provide improved performance, but such a policy would require a more complicated method of tracking to ensure that no item has a shorter lifetime than they would in this policy. For our initial exploration, we decided to use the simpler policy over the more complicated one with limited benefits. Even with this simpler policy, choosing partition sizes is involved. We build up to this decision by analyzing each layer individually and then combining the analysis, with the results discussed in Section 4.4.3.

4.4.2 The Upper Bound

In order to prove our upper bound on the competitive ratio of IBLP, we introduce a new linear programming technique to analyze how the optimal cache makes use of cache space. Using this technique, we will first consider how each layer performs separately against adversarial approaches to the type of locality that it targets. We will then provide an analysis of the combined problem to ultimately prove the upper bound.

Our Analysis Technique. Our analysis visualizes the optimal cache's performance on a trace as a rectangle, with one axis representing the time in units of accesses, and the other representing cache space. When an item is brought into cache, it take up one unit of space and a number of units of time equal to the number of accesses between when it is loaded and when it is evicted.

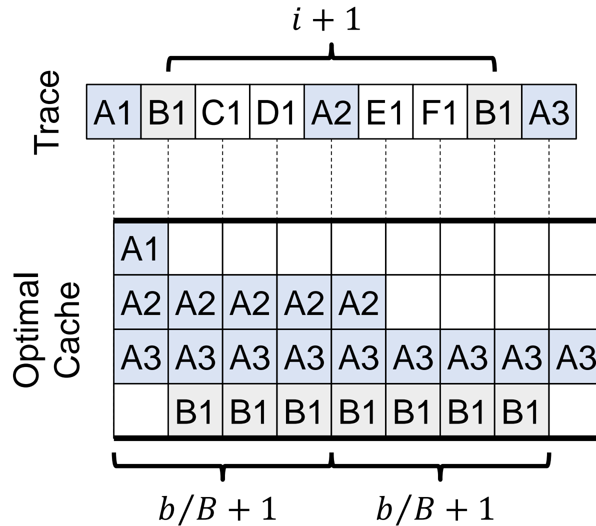


Figure 4.3: The pattern of accesses resulting in worst-case traces and the resulting cache space used in the optimal cache over time. The accesses to block A show the worst-case spatial locality, while the accesses to block B show the worst-case temporal locality.

A visualization of this method can be found in Figure 4.3.

In order to use this method, we assume that each access is chosen so as to cause the online policy that we compare against to miss.[2] Therefore, if we choose a rectangle that accurately captures the average long-term behavior of the trace, then the miss ratio is the length of the time dimension of the rectangle divided by the number of hits the optimal cache can achieve during the rectangle subtracted from the length.

There are two constraints that limit the number of hits the optimal cache can achieve. The first constraint is that of cache usage. This means that the total number of units of space used by the optimal cache cannot exceed the area of the rectangle. To reduce clutter, we will ignore the need to leave one unit of cache space available for accesses that the optimal cache misses on in this analysis.[3] The second constraint is that of accesses. This means that the number of accesses that occur cannot exceed the length of the rectangle in the time dimension. We will make these constraints more concrete in the analyses below.

Both of these constraints are necessary to fully specify the problem, but they still provide the optimal cache degrees of freedom that are not available in the original caching problem. The constraints refer to the total amount of the resource (cache usage or accesses) used, but not where these resources are located. This allows the optimal cache to make use of solutions that are not viable in the original caching problem by having multiple accesses occur at the same time or overdrawing cache space at one time by underutilizing it at a different time. Since this looseness empowers the optimal cache it can only hurt the resulting bounds. In other words, our bound is correct, but perhaps not as tight as possible.

[2]This assumption is accurate as long as hits have limited ability to change eviction decisions.
[3]The effect on the analysis is limited to replacing h with $h - 1$ in some places.

Temporal Locality. We begin our analysis by comparing how the item layer and the optimal cache perform on adversarial temporal locality. More specifically, we ignore any hits caused by spatial locality. In this setting, an access can be a miss for the item layer and a hit for the optimal cache if and only if there have been at least i distinct items accessed since the item was last accessed. This means that any such hit requires at least i units of cache space. Each time this occurs, the access that hits is constrained to be to the same item as the access that caused the load. The accesses to block B in Figure 4.3 illustrate this pattern.

Turning these constraints into a linear program and solving it provides an upper bound for the competitive ratio of the item layer on temporal locality.

Theorem 20. *When considering hits due to only temporal locality, the item layer of IBLP has a competitive ratio upper bounded by* $\frac{i}{i-h}$ *where* i *is the size of the item layer cache and* $h < i$ *is the size of the optimal cache.*

Proof. We define r to be the fraction of accesses that the optimal cache hits on due to temporal locality. The competitive ratio is $\frac{1}{1-r}$. Since there is h cache space in the rectangle and each hit requires i cache space, the cache space constraint is $h \geq ri$. Since each hit forces one access to a particular item, the accesses constraint says that the number of hits must be less than the number of accesses, i.e., $1 \geq r$. Although in this problem, this constraint is loose, it becomes critical for later versions that include spatial locality.

This results in the following linear program:

$$\text{Maximize: } \frac{1}{1-r}$$

subject to:

$$h \geq ri$$

$$1 \geq r$$

where r is the free variable.

Solving this linear program provides the desired result. □

This result matches the upper bound from Sleator and Tarjan [102] for traditional LRU caches.[4] Since we are focusing only on temporal locality, this behavior is to be expected, as the item layer behaves exactly like an LRU cache of size i.

Spatial Locality. We next consider how the block layer compares to the optimal cache in the face of adversarial spatial locality. Similar to the analysis above, we will ignore any hits not due to our chosen form of locality. However, spatial locality introduces several important variations.

For spatial locality, hits to an item cannot be caused by previous accesses to that item, only misses to a different item in their block that causes the original item to be loaded. This means that each item can cause at most one hit per time that it is loaded. Furthermore, since the number of items loaded at a time is upper bounded by the block size and one of them is the item that was just missed on, the maximum number of hits that can be caused by each miss is at most $B - 1$.

[4]The lack of a negative one in the denominator is due to the issue of space used by misses as discussed earlier.

We now consider what adversarial spatial locality looks like for the block layer. In order for the block layer to miss on an access, there must have been b/B distinct other blocks accessed since the last access to that block. This means that in order for the optimal cache to achieve multiple hits from the same load operation, each additional item loaded must be stored in cache for $b/B + 1$ accesses more than the previous item. This results in a triangle-like cache usage pattern, as shown for the A block in Figure 4.3.

The dimensions of this pattern provide an interesting set of tradeoffs in the design space. Since the optimal cache must miss at least once in order to load items to hit on, the competitive ratio is upper bounded by the number of items loaded at a time. However, the cache usage of each item increases with the number of items loaded.

The design of the linear program is based on the number of items t that the optimal cache chooses to load on each miss and the number of misses s that cause the optimal cache to perform loads. The optimal cache achieves $s(t-1)$ hits, resulting in a competitive ratio of $\frac{1}{1-s(t-1)}$. Since the total cache usage due to each miss is $\sum_{j=0}^{t-1} 1 + j(b/B + 1)$, the cache usage constraint says that $h \geq s(\sum_{j=0}^{t-1} 1 + j(b/B + 1))$. Similarly, the accesses constraint is $1 \geq st$ since each load causes the specific miss and each subsequent hit to be fixed accesses. Solving the resulting linear program provides the resulting bound.

Theorem 21. *When considering hits due to only spatial locality, the block layer of IBLP has a competitive ratio upper bounded by min*$\Big(B, (b + 2Bh - B)/(b + B)\Big)$ *where* b *is the size of the block layer cache and* h *is the size of the optimal cache.*

Proof. We define s to be the fraction of accesses where the optimal cache misses and performs loads and t to be the number of items that the optimal cache chooses to load on each miss. The optimal cache achieves $s(t-1)$ hits, resulting in a competitive ratio of $\frac{1}{1-s(t-1)}$. Since the total cache usage due to each miss is $\sum_{j=0}^{t-1} 1 + j(b/B + 1)$, the cache usage constraint says that $h \geq s(\sum_{j=0}^{t-1} 1 + j(b/B + 1))$. Similarly, the accesses constraint is $1 \geq st$ since each load causes the specific miss and each subsequent hit to be fixed accesses. This results in the following maximization problem:

$$\text{Maximize: } \frac{1}{1 - s(t-1)}$$

subject to:

$$h \geq s\Big(\sum_{j=0}^{t-1} 1 + j(b/B + 1)\Big)$$

$$1 \geq st$$

where s and t are the free variables.

Solving this maximization problem provides the second term in the theorem. The first term comes from the fact that t cannot exceed B combined with the second constraint in the linear program. □

Combining the Localities. We now show how to combine the two methods of achieving hits to obtain an upper bound for the entirety of IBLP for general traces. Since IBLP hits on an access if either of its partitions hits on that access, the restrictions on an access in order for it to be a miss for IBLP must be at least as strict as the union of the previous restrictions. This allows us to formulate a linear program for the entire policy by combining the hits and the constraints of the previous two versions.

The resulting linear program is too complex to solve directly. To deal with this, we modify the spatial locality problem to take the number of hits due to temporal locality as an input. We then use the result of this problem to choose the number of temporal locality hits that maximizes the competitive ratio. The result of this is shown below in Theorem 22.

Theorem 22. *The competitive ratio of IBLP is upper bounded by:*

$$\begin{cases} \frac{(b+B(2i-1))^2}{8B(B+b)(i-h)} & i \leq \frac{2Bb-b+2B^2+B}{2B} \\ \frac{2Bi-Bb+b-B^2-B}{2i-2h} & i > \frac{2Bb-b+2B^2+B}{2B} \end{cases}$$

where $i \geq h$ is the size of the item layer, b is the size of the block layer, and h is the size of the optimal cache.

Proof. As in the previous proofs, we define r, s, and t to be the fraction of accesses that the optimal cache hits on due to temporal locality, the fraction of accesses that the optimal cache misses on and loads items for spatial locality, and the number of items loaded for spatial locality, respectively.

The total number of hits that optimal cache achieves is equal to the sum of the hits from the individual localities. This results in a competitive ratio of $\frac{1}{1-r-s(t-1)}$. We combine the amount of cache space used and the number of accesses forced in a similar way. This results in the following linear program:

$$\text{Maximize: } \frac{1}{1-r-s(t-1)}$$

subject to:

$$h \geq ri + s\Big(\sum_{j=0}^{t-1} 1 + j(b/B+1)\Big)$$

$$1 \geq r + st$$

where r, s, and t are the free variables.

Unfortunately, we were unable to solve this linear program directly (one hour of computation time in Wolfram Mathematica). To obtain a solution, we break the program into smaller chunks that can be solved individually. We start by modifying the linear program to compute the values of s and t that maximize the competitive ratio given a particular r value. This results in the following values for s and t:

$$s = \frac{(B+b)(1-r)^2}{b-br+B(2h-1+r-2ir)}$$

$$t = \frac{b - br + B(2h - 1 + r - 2ir)}{(B + b)(1 - r)}$$

When we plug in these values and solve the resulting maximization problem, we achieve results for both r and the competitive ratio:

$$r = \frac{b + B(4h - 2i - 1)}{b + B(2i - 1)}$$

$$\text{Ratio} = \frac{(b + B(2i - 1))^2}{8B(B + b)(i - h)}$$

This result is a valid upper bound, but fails to account for the constraint that t cannot exceed the block size B. By using the expressions for the values of r and t, we find that this occurs when $i > \frac{2Bb - b + 2B^2 + B}{2B}$. In this region, we know from above that t maxes out at B. We apply this change to the prior analysis, with the following results:

$$r = \frac{2Bh - Bb + b - B^2 - B}{2Bi - Bb + b - B^2 - B}$$

$$\text{Ratio} = \frac{2Bi - Bb + b - B^2 - B}{2i - 2h}$$

Putting these results together finishes the proof. □

4.4.3 Applying the Bound

Having proved a bound on the competitive ratio for IBLP as a function of the layer sizes, we can now consider how to partition the cache space. This task is complicated by the fact that the optimal partitioning depends on the size of the optimal cache being compared against.

Known Optimal Size. When the size of the optimal cache is known, then the optimal partitioning can be directly computed. For the first part of the piecewise function (smaller i), the minimum competitive ratio and item layer size when it occurs are:

$$\text{Ratio} = \frac{(k + B - 1)(k - h + B(2h - 1))}{(k - h + B)^2}$$

$$i = \frac{k^2 + 4Bhk - hk + 4B^2h - 3Bh - B^2}{2Bk + k + 2Bh - h + 2B^2 - 3B}$$

For the second part (larger i), setting $i = k$ provides the minimum competitive ratio of:

$$\frac{2Bk - B^2 - B}{2(k - h)}$$

By solving for the point of equality for the competitive ratios, we see that the smaller i value should be chosen when $k \geq \frac{3Bh - h - B^2 - B}{B - 1}$ (the online cache is large relative to the optimal cache), and the larger i value should be chosen otherwise.

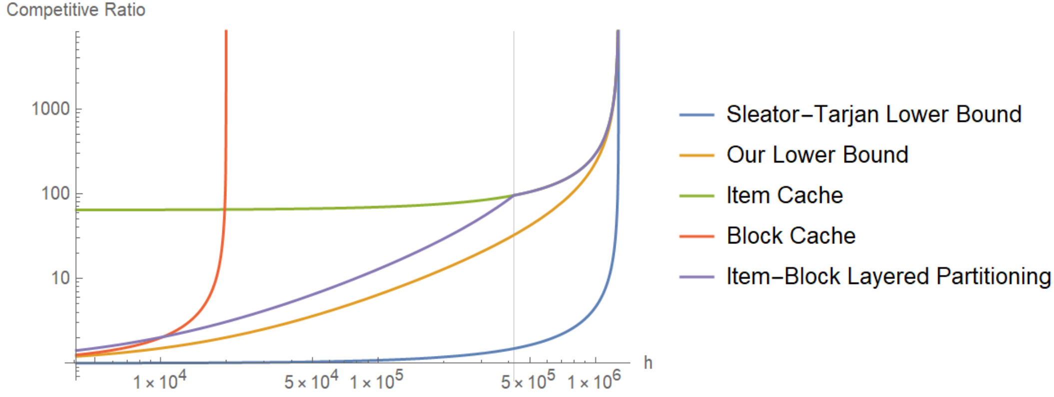


Figure 4.4: A graph comparing the upper bound for IBLP against the Sleator-Tarjan bound, our lower bound, an LRU Item Cache, and an LRU Block Cache. The x-axis is the size of the optimal cache and the y-axis is the competitive ratio (lower is better). In this graph, the online cache size $k = 1.28M$ and block size $B = 64$.

Figure 4.4 shows how the performance of IBLP compares to the lower bound, as well as an Item Cache and a Block Cache of the same size running LRU, for one set of parameters. This general shape is consistent for most parameter settings, with IBLP outperforming the Item Cache for $k \approx 3h$ and larger, and outperforming the Block Cache for $k \approx 4Bh$ and smaller. In addition, IBLP performs close to optimal for all values of k, whereas the performance of the baselines degrade severely outside of their ideal performance conditions.

With the assumptions we make regarding the relative sizes of k, h, and B, the upper bound for IBLP differs from the lower bound by a multiplicative factor of at most 3, and often significantly less. Comparing against the points of interest from the lower bound, we see that the competitive ratio is $\approx 2B$ when $k = 2h$, $k \approx Bh$ yields a competitive ratio of ≈ 3, and the meeting point occurs when $k \approx \sqrt{2Bh}$.

Unknown Optimal Size. When comparing against an optimal cache of unknown size, a unique aspect of our problem arises. The optimal partitioning strategy depends on the size of the optimal cache being compared against. As shown in Figure 4.5, for any fixed partition size, the competitive ratio will match the bound for the optimal choices at one partition size but show sigificant dropoff for larger values of h and limited improvement for smaller values of h.

Having the optimal online strategy depend on the size of the optimal cache being compared against is unique amongst caching problems. Even writeback-aware caching, which would appear to have the same issue of multiple types of costs, does not exhibit this phenomenon. The difference appears to lie in the fact that unlike in other caching problems, the two types of costs (temporal and spatial locality) are different functions the cache size. This means that for different sizes of the optimal cost, the relative performance of traces changes depending on the amount of spatial and temporal locality that they contain. Accordingly, the importance of performing well on these traces changes in the competitive ratio. Dealing with this issue would require a different metric of cache cost that accounts for the varying value of traces. We leave such questions for

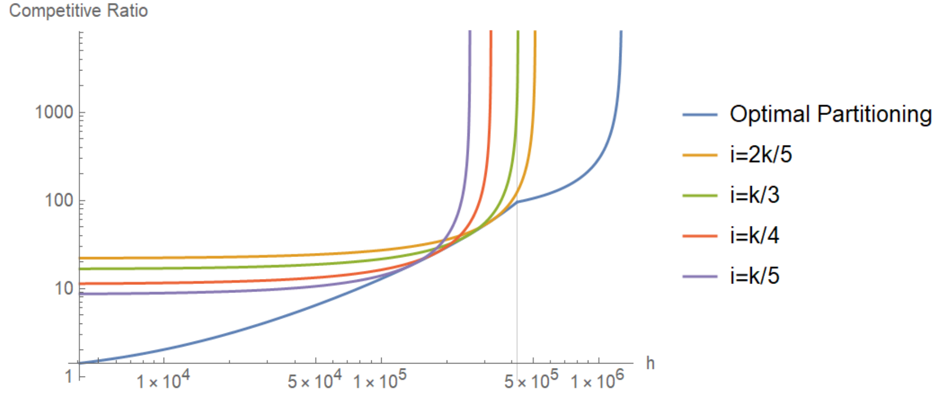


Figure 4.5: A graph showing how the upper bound of IBLP with constant layer sizes performs compared to the optimal layer sizes. The x-axis is the size of the optimal cache and the y-axis is the competitive ratio (lower is better). In this graph, the online cache size $k = 1.28M$ and block size $B = 64$.

future work.

4.5 Chapter Summary

Demands on the modern memory hierarchy continue to increase as more data become available and need to be served. In order for system throughputs to increase to match these demands, they will need to take every advantage offered by the underlying technology. We provide an opportunity to unlock one such advantage by studying granularity change in the memory hierarchy. In our investigation, we introduce the Block-Granularity-Aware Caching Problem, which generalizes the caching problem to account for the benefits of spatial locality at granularity boundaries. We provide a strong theoretical framework for this problem, including an analysis of the complexity of the problem, and both upper and lower bounds on the competitiveness of deterministic policies. We believe that these results provide crucial insights on the potential available in this area, and how it can be harnessed to improve performance.

Chapter 5

Conclusion

In this work, we have begun an investigation into how to design caching and memory management systems for non-volatile memory technologies and the new performance characteristics that they have. In particular, we have studied: (i) how to take advantage of NVM persistence to improve fault tolerance, (ii) how to account for writebacks in caching systems in order to improve overall device utilization, and (iii) how caching systems can take advantage of data granularity changes to obtain additional data at low cost.

For each of these areas, our work provides a strong theoretical framework closely tied with practice in order to develop insights and techniques that can be applied directly to real systems. Combined, these results support the thesis that **when designing systems for non-volatile memory technologies, designing for specific performance metrics and accounting for crucial device characteristics can provide asymptotic theoretical performance improvement and practical improvement to match.**

However, there are still many important questions for future research. These include extending our theoretical understanding beyond the initial frameworks that this research generates. For fault tolerance, this includes generating a minimum sufficient definition for idempotence, generating lower bounds for cost, and extending our work to more of the vast world of algorithms that could benefit from it. For writeback-aware caching, our results could be extended to consider randomized algorithms and forms of analysis beyond worst-case. These problems are even more pressing for granularity-aware caching, given the significant limitations that we found for deterministic policies in that model.

There is also significant practical research to be done. For the work on fault tolerance we provide an implementation of our scheduler, but an implementation of closures and experimental results have not been performed to our knowledge. For the work on caching, extending our experiments to a broader range of traces and cost metrics would help understand the applicability of the results. Furthermore, implementing real caches that use these eviction policies rather than simulations would be informative.

Even beyond these questions, there is still important research on NVMs to be done. Perhaps we can design an interface for NVMs that allows for better performance. There are currently two different methods in use for layering DRAM and NVM, but better options may exist. The error correcting codes thought to be used inside NVM technologies could be used in conjunction with other aspects of a system. All of these questions and more exist to be explored in the pursuit of

performance.

As NVMs become more prevalent and the memory hierarchy continues to evolve with the demands placed upon it, questions like these will become more and more important to solve. In order for computing to continue to evolve as we desire, so too must our understanding of the technologies we use to implement it.

Bibliography

[1] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared memory. In *PODC*, 1992. 2

[2] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988. 2.1.1

[3] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *SODA*, volume 99, pages 31–40. Citeseer, 1999. 3, 3, 3.1.1, 4

[4] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL*, 1989. 2.3

[5] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2008. 2

[6] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):3–11, 2000. 3.6.2

[7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2), Apr 2001. 2, 5, 2.1.2, 2.5.1, 2.5.3

[8] Hagit Attiya and Gili Yavneh. Remote memory references at block granularity. In *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. 4

[9] Yonatan Aumann and Michael Ben-Or. Asymptotically optimal PRAM emulation on faulty hypercubes. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991. 2

[10] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1), 2015. 2

[11] Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. *Journal of the ACM (JACM)*, 48(5):1069–1090, 2001. 3, 3, 4

[12] Nathan Beckmann and Daniel Sanchez. Maximizing cache performance under uncertainty. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 109–120. IEEE, 2017. 3

[13] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM*

Systems journal, 5(2):78–101, 1966. 3, 3, 3.4.1, 4

[14] Laszlo A. Belady and Frank P. Palermo. On-line measurement of paging behavior by the multivalued min algorithm. *IBM Journal of Research and Development*, 18(1):2–19, 1974. 3

[15] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2019. 2.4

[16] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proc. ACM Meas. Anal. Comput. Syst. (SIGMETRICS'18)*, 2018. ISSN 2476-1249. doi: 10.1145/3224427. URL `http://doi.acm.org/10.1145/3224427`. 3.5.1, 3.5.2, 4

[17] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016. 2

[18] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. Borg: Block-reorganization for self-optimizing storage systems. In *FAST*, volume 9, pages 183–196. Citeseer, 2009. 3

[19] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Notices*, volume 51, pages 677–694. ACM, 2016. 2

[20] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2010. 2.6, 2.6, 4

[21] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–366, 2011. 4

[22] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–12. ACM, 2015. 1, 2, 3

[23] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007. 3

[24] Mark Brehob, Stephen Wagner, Eric Torng, and Richard Enbody. Optimal replacement is NP-hard for nonstandard caches. *IEEE Transactions on computers*, 53(1):73–76, 2004. 3, 4

[25] Michael Buettner, Ben Greenstein, and David Wetherall. Dewdrop: an energy-aware runtime for computational RFID. In *NSDI*, 2011. 2

[26] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 139–149, 1998. 4

[27] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997. 3.6, 3.6.1

[28] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.: Int. J.*, 1(1), April 2014. 2

[29] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. Write-avoiding algorithms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 648–658. IEEE, 2016. 1

[30] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*, 2014. 2

[31] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMove: Helping programmers move to byte-based persistence. In *INFLOW*, 2016. 2

[32] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015. 2

[33] Trishul M Chilimbi, Mark D Hill, and James R Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 1–12, 1999. 4

[34] B. S. Chlebus, A. Gambin, and P. Indyk. PRAM computations resilient to memory faults. In *European Symposium on Algorithms (ESA)*, 1994. 2

[35] Marek Chrobak, Howard J. Karloff, T. H. Payne, and Sundar Vishwanathan. New results on server problems. In *SIAM Journal on Discrete Mathematics*, pages 172–181, 1991. 3, 3, 4

[36] Marek Chrobak, Gerhard J. Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard-even in the fault model. *Algorithmica*, 63(4):781–794, 2012. 3, 3, 4, 4.2

[37] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011. 2

[38] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. *OOPSLA*, 2016. 2

[39] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 116–127. ACM, 2018. 2, 3

[40] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009. 2.6

[41] Intel Corporation. Optane SSD DC P4800X series, 2018. Retrieved online on 11 Jan 2019 at https://ark.intel.com/products/97161/Intel-Optane-SSD-DC-P4800X-Series-375GB-2-5in-PCIe-x4-3D-XPoint. 1, 3.6.1

[42] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. EPFL Technical Report, 2017. 2

[43] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *PLDI*, 2012. 2

[44] Robert H. Dennard, Fritz H. Gaensslen, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. 3

[45] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for NVRAMs. *PVLDB*, 13(9), 2020. 1

[46] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 389–400. IEEE, 2012. 3

[47] Guy Even, Moti Medina, and Dror Rawitz. Online generalized caching with varying weights and costs. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 205–212. ACM, 2018. 3, 3, 4

[48] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Memory controller policies for DRAM power management. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 129–134, 2001. 4

[49] Martin Farach-Colton and Vincenzo Liberatore. On local register allocation. *Journal of Algorithms*, 37(1):37–65, 2000. 3, 3.3

[50] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991. 3, 3, 4

[51] Irene Finocchi and Giuseppe F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). In *ACM Symposium on the Theory of Computing (STOC)*, 2004. 2

[52] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018. 2

[53] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999. 2, 2.1.1

[54] Binny S Gill and Dharmendra S Modha. WOW: wise ordering for writes-combining spatial and temporal locality in non-volatile caches. In *FAST*, 2005. 3

[55] David Grove, Sara S. Hamouda, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Olivier Tardieu. Failure recovery in resilient X10. Technical Report RC25660 (WAT1707-028), IBM Research, Computer Science, 2017. 2

[56] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009. 3.6.1, 3.6.2

[57] Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *Inter. Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2004. 2

[58] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 4, 4.1

[59] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012. 2.5.1

[60] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proc. ACM Conference on Embedded Network Sensor Systems*, 2017. 2

[61] Mark Horowitz. Computing's energy problem (and what we can do about it). In *Proc. of the IEEE Intl. Solid-State Circuits Conf. (ISSCC)*, 2014. 3

[62] Terry Ching-Hsiang Hsu, Helge Bruegner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *EuroSys*, 2017. 2

[63] IBM. www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014. 3

[64] Intel. Intel NVM library. https://github.com/pmem/nvml/. 2

[65] Intel. Intel architecture instruction set extensions programming reference. Technical Report 3319433-029, Intel Corporation, April 2017. 2

[66] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS*, 2016. 2

[67] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, 2016. 2

[68] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Brief announcement: Preserving happens-before in persistent memory. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2016. 2

[69] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019. 1, 3

[70] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 78–89. IEEE, 2016. 3

[71] J. JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992. 2.6

[72] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr, and Joel Emer. High performance

cache replacement using re-reference interval prediction (RRIP). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010. 3

[73] Viggo Kann. Maximum bounded 3-dimensional matching is max SNP-complete. *Inf. Process. Lett.*, 37(1):27–35, 1991. 3.3.2

[74] Richard M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68, 1975. 3.3.2

[75] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 245–250. IEEE, 2007. 3

[76] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. *ACM Transactions on Storage (TOS)*, 10(4):15, 2014. 3

[77] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: exploiting NVRAM in write-ahead logging. In *ASPLOS*, 2016. 2

[78] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *ASPLOS*, 2016. 2

[79] Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical report, U.T. Austin, 2010. 3

[80] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2017. 2

[81] Wei-Fen Lin, Steven K Reinhardt, and Doug Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Transactions on Computers*, 50(11):1202–1218, 2001. 4

[82] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS*, 2017. 2

[83] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *PLDI*, 2015. 2

[84] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *OOPSLA*, 2017. 2

[85] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970. 3, 3, 3.4.1, 3.4.1, 4

[86] Sally A McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162, 2004. 1

[87] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale research letters*, 9(1):526,

2014. 1

[88] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *EuroSys*, 2017. 2

[89] Seiji Miura, Kazushige Ayukawa, and Takao Watanabe. A dynamic-SDRAM-mode-control scheme for low-power systems with a 32-bit RISC CPU. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 358–363, 2001. 4

[90] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160. IEEE, 2007. 4

[91] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *2008 International Symposium on Computer Architecture*, pages 63–74. IEEE, 2008. 4

[92] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4 (3):10, 2008. 3.6, 3.6.1

[93] Faisal Nawab, Joseph Izraelevitz, Ternece Kelly, Charles B. Morrey III, and Dhruva R. Chakrabarti amd Michael L. Scott. Dali: A periodically persistent hash map. In *DISC*, 2017. 2

[94] James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations research*, 41(2):338–350, 1993. 3.5.2

[95] Seong-Il Park and In-Cheol Park. History-based memory mode prediction for improving memory performance. In *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS'03.*, volume 5, pages V–V. IEEE, 2003. 4

[96] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ISCA*, 2014. 2

[97] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. *ACM SIGPLAN Notices*, 37(1):101–112, 2002. 4

[98] Hanfeng Qin and Hai Jin. Warstack: Improving LLC replacement for NVM with a writeback-aware reuse stack. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pages 233–236. IEEE, 2017. 3

[99] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007. 3

[100] Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. Phase change memory: From devices to systems. *Synthesis Lectures on Computer Architecture*, 6(4):1–134, 2011. 3

[101] Jiri Schindler, John Linwood Griffin, Christopher R Lumb, and Gregory R Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST*, volume 2,

pages 259–274, 2002. 3

[102] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985. 3, 3, 3.2.2, 3.2.2, 4, 4.3.1, 4.4.2

[103] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010. 3

[104] Vladimir V Stankovic and Nebojsa Z Milenkovic. DRAM controller with a close-page predictor. In *EUROCON 2005-The International Conference on" Computer as a Tool"*, volume 1, pages 693–696. IEEE, 2005. 4

[105] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. The virtual write queue: Coordinating DRAM and last-level cache policies. *ACM SIGARCH Computer Architecture News*, 38(3):72–82, 2010. 3

[106] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *ACM SIGARCH Computer Architecture News*, 38(1):219–230, 2010. 4

[107] Qinghui Tang, Sandeep Kumar S. Gupta, and Georgios Varsamopoulos. Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1458–1472, 2008. 3

[108] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *OSDI*, 2016. 2

[109] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory I/O primitives. In *International Workshop on Data Management on New Hardware*, pages 12:1–12:7, 2019. 1, 3

[110] Stratis D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014. 1

[111] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011. 2

[112] Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. Improving writeback efficiency with decoupled last-write prediction. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 309–320. IEEE Computer Society, 2012. 3

[113] Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A. Jiménez. WADE: Writeback-aware dynamic cache management for NVM-based main memory system. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10 (4):51, 2013. 3

[114] Ying Xu, Aabhas S Agarwal, and Brian T Davis. Prediction in dynamic SDRAM controller policies. In *International Workshop on Embedded Computer Systems*, pages 128–138. Springer, 2009. 4

[115] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A Harding, and Onur Mutlu. Row buffer locality aware caching policies for hybrid memories. In *2012 IEEE*

30th International Conference on Computer Design (ICCD), pages 337–344. IEEE, 2012. 4

[116] Neal Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11 (6):525–541, 1994. 3, 3, 4

[117] Neal E. Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002. 3, 2, 3, 3.2.1, 3.2.2, 4

[118] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 32–41, 2000. 4

[119] Hongzhong Zheng, Jiang Lin, Zhao Zhang, Eugene Gorbatov, Howard David, and Zhichun Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 210–221. IEEE, 2008. 4

[120] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):53, 2012. 3

[121] Zhichun Zhu and Zhao Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *11th International symposium on high-performance computer architecture*, pages 213–224. IEEE, 2005. 4

[122] Zhichun Zhu, Zhao Zhang, and Xiaodong Zhang. Fine-grain priority scheduling on multi-channel memory systems. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 107–116. IEEE, 2002. 4