Parallel Execution Using Memory Versioning And Renaming

Eran Gilad
Parallel Execution Using Memory Versioning And Renaming

Research Thesis

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

Eran Gilad

Submitted to the Senate of the Technion — Israel Institute of Technology Tamuz 5778 Haifa July 2018
This research was carried out under the supervision of Prof. Yoav Etsion, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Title</th>
<th>Conference/Proceedings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eran Gilad, Eric W. Mackay, Mark Oskin, and Yoav Etsion</td>
<td>O-structures: Semantics for versioned memory.</td>
<td>Workshop on Memory Systems Performance and Correctness (MSPC), 2014.</td>
</tr>
<tr>
<td>Eran Gilad, Tehila Mayzels, Elazar Raab, Mark Oskin, and Yoav Etsion</td>
<td>Architectural support for unlimited memory versioning and renaming.</td>
<td>International Parallel &amp; Distributed Processing Symposium (IPDPS), 2018.</td>
</tr>
</tbody>
</table>

ACKNOWLEDGEMENTS

I dedicate this dissertation to my daughters Roni, Lihi and Moran, in hope that they too will find their passion one day and be able to pursue it.

This has been a long journey, and I enjoyed almost every moment. First, I’d like to thank my advisor, Prof. Yoav Etsion. Working with him was a pleasure, both professionally and personally. I’d also like to thank Prof. Mark Oskin, who was a close collaborator throughout my studies. It was a wonderful experience to be part of the Technion Computer Engineering (TCE) team, whose members – students and administrators – were both colleagues and close friends. I’ve also had a great time working with dedicated teaching staffs and associating with fellow CS students. I thank them all, and in particular Tehila Mayzels and Nadav Amit.

To my wife Dana, who has supported me on the hard times and shared my joy on the good times – I could not have done this without you. Thank you for being there with and for me on this journey! Finally, I’d like to express my greatest appreciation to my parents, Carol and David, my siblings, Danit, Gadi and Sivan, and my in-laws, the Keidar family. Your help and support were invaluable.

The generous financial help of the Technion and the Hasso Plattner Institute is gratefully acknowledged.
# Contents

## List of Figures

## Abstract

## 1 Introduction

1.0.1 State management ........................................... 5
1.0.2 Memory system extensions .............................. 6
1.0.3 Programming models ........................................ 6
1.0.4 Parallel execution using memory versioning and renaming ... 7
1.0.5 Parallelizing dynamic data structures ................. 8
1.0.6 Summary ...................................................... 8

## 2 O-structures: Semantics for Versioned Memory

2.1 Introduction ........................................................... 9
2.2 Motivation and background ................................... 10
  2.2.1 Existing memory interfaces ............................ 10
  2.2.2 Limitations of existing memory interfaces .......... 11
  2.2.3 O-structures ................................................. 12
2.3 Semantics .............................................................. 13
2.4 Task-Based Execution and Versioned Memory ............. 15
  2.4.1 Background: task-based execution models.......... 15
  2.4.2 How task-based models can use O-structures .... 16
2.5 Managing data structures ...................................... 17
  2.5.1 Static data structures .................................. 17
  2.5.2 Dynamic data structures ................................. 17
2.6 Evaluation .......................................................... 18
  2.6.1 Simulation framework .................................. 18
  2.6.2 Experimental results .................................... 19
2.7 Conclusions ....................................................... 21

## 3 Deterministic Fine-grained Task Ordering using Multi-Viewed Memory

3.1 Introduction .......................................................... 23
3.2 Task-based programming and memory versioning ........ 25
### 3.2.1 The need for memory versioning

### 3.3 O-structures

#### 3.3.1 High-level interface

### 3.4 Execution Model

#### 3.4.1 Scheduling strategies

#### 3.4.2 Task queue ordering

#### 3.4.3 Task identifiers

#### 3.4.4 Spawning order

### 3.5 Handling irregular data structures

#### 3.5.1 Unipath, Semi-unipath and Multipath code

#### 3.5.2 Parallelizing using versioning, renaming and locking

#### 3.5.3 Mutating unipath operations

#### 3.5.4 Read-only unipath operations

#### 3.5.5 Parallelizing Semi-unipath operations

### 3.6 Evaluation

#### 3.6.1 Methodology

#### 3.6.2 Speedup analysis

#### 3.6.3 TLP and MLP

### 3.7 Related Work

#### 3.7.1 Non-speculative task-based programming models

#### 3.7.2 Thread-level speculation (TLS)

#### 3.7.3 Fine-grained pipelining

#### 3.7.4 Memory versioning

### 3.8 Conclusions

### 4 Architectural support for unlimited memory versioning and renaming

#### 4.1 Introduction

#### 4.2 O-structures

##### 4.2.1 Semantics and instruction set

##### 4.2.2 Programming model

##### 4.2.3 Practical considerations

#### 4.3 Architectural support

##### 4.3.1 Caching

##### 4.3.2 Reducing communication across caches

#### 4.4 Garbage collection

##### 4.4.1 Program cooperation

##### 4.4.2 Version shadowing

##### 4.4.3 Version sorting

##### 4.4.4 Operation

##### 4.4.5 Garbage collector lists structure

##### 4.4.6 Overhead
4.4.7 Allocating and Freeing O-structures .............................................. 65
4.5 Evaluation ................................................................. 65
  4.5.1 Methodology ......................................................... 65
  4.5.2 Versioning only — regular data structures ......................... 67
  4.5.3 Versioning and renaming — snapshot isolation ....................... 67
  4.5.4 Versioning, renaming and locking – task-based execution ........... 68
  4.5.5 Size and latency sensitivity ........................................ 69
  4.5.6 Garbage collection overhead ....................................... 71
4.6 Related Work ........................................................... 71
  4.6.1 Dataflow memory ................................................... 71
  4.6.2 Implicit memory versioning ....................................... 72
  4.6.3 Loosely related mechanisms ..................................... 73
4.7 Conclusions ............................................................. 73

5 Snapshot-based Synchronization: A Fast Replacement for Hand-over-Hand Locking .................................................. 75
  5.1 Introduction ............................................................ 75
      5.1.1 Snapshot-based Synchronization .................................. 76
  5.2 Snapshot-based Synchronization Design .................................... 77
      5.2.1 Interface and algorithms ....................................... 78
      5.2.2 Locking granularity ........................................... 82
  5.3 Optimized Implementation ............................................... 82
      5.3.1 Using SIMD capabilities ...................................... 83
      5.3.2 Copying snapshots ............................................. 83
      5.3.3 Deferring snapshot creation by trailing ....................... 84
      5.3.4 NUMA awareness ............................................. 84
      5.3.5 Reader synchronization ...................................... 85
      5.3.6 Putting it all together ....................................... 86
  5.4 Evaluation ............................................................ 86
      5.4.1 Experimental setup ............................................. 86
      5.4.2 Scalability ..................................................... 87
      5.4.3 Effect of data structure size ................................ 88
      5.4.4 Effect of read-write ratio ................................... 89
      5.4.5 Entrance bottleneck analysis ................................ 89
      5.4.6 Memory overhead of synchronization ......................... 90
  5.5 Related Work .......................................................... 91
  5.6 Conclusions ........................................................... 92

6 Conclusion and future work ...................................................... 95
  6.1 Future work ........................................................... 97
      6.1.1 Automatic unipath detection .................................. 97
# List of Figures

1.1 Various parallel processing technologies ........................................... 4  
2.1 Parallelizing using O-structures ...................................................... 13  
2.2 O-structures task-based execution model vs. others ............................ 17  
2.3 Memory level parallelism of linked list insertions ............................... 19  
2.4 Memory level parallelism of binary tree insertions ............................. 20  
2.5 Memory level parallelism of dense matrix multiplication ..................... 20  
2.6 Memory level parallelism of sparse matrix multiplication .................... 21  
3.1 Task-based program decomposition mapping onto virtual time ............... 25  
3.2 Using versioning to parallelize sparse matrix operations .................... 27  
3.3 Multi-versioned memory fine-grained dataflow .................................. 31  
3.4 Scheduling strategies – work-first vs. help-first ............................... 32  
3.5 Task identifiers in nested spawning ............................................... 35  
3.6 Unipath traversal technique ............................................................ 40  
3.7 Speedup of parallel versioned vs. sequential unversioned code ............ 43  
3.8 Scalability of linked list operations ................................................. 44  
3.9 Scalability of binary tree operations .............................................. 45  
3.10 Scalability of hash table operations ........................................ ..... 46  
3.11 Scalability of red-black tree operations ........................................ 47  
3.12 Scalability of regular algorithms ................................................... 48  
4.1 Hardware support for O-structures .................................................. 54  
4.2 Memory layout of a version block ................................................... 56  
4.3 Chained list of version blocks ....................................................... 56  
4.4 Memory layout of compressed version blocks ................................... 58  
4.5 O-structures storage example ........................................................ 60  
4.6 Version shadowing example .......................................................... 62  
4.7 Garbage collector lists structure ..................................................... 64  
4.8 Speedup of parallel benchmarks .................................................... 66  
4.9 Scalability of parallel benchmarks .................................................. 67  
4.10 Renaming snapshot isolation vs. read-write lock .............................. 68  
4.11 Impact of L1 cache size ................................................................. 70
4.12 Impact of versioning latency ........................................... 71
5.1 Synchronization using a snapshot ......................................... 77
5.2 The need to fully recreate snapshots upon collision ....................... 81
5.3 Locking nodes vs. locking pointers ......................................... 82
5.4 Thread ordering optimized for NUMA ....................................... 84
5.5 Scalability using 2 NUMA nodes ........................................... 87
5.6 Scalability using 1 NUMA node ........................................... 88
5.7 Effect of initial size ......................................................... 89
5.8 Effect of read-write ratio .................................................. 90
Abstract

As the number of cores in modern computers increases, the most practical way to efficiently utilize all cores is still designing parallel algorithms, which are hard to reason about, and implementing them in parallel programs, which are hard to debug. High-level programming models that aim to ease parallel programming either rely on a task-based specification, in which the programmer defines small, minimally dependent, sequences of code, or merely eliminate technical aspects such as data partitioning and thread creation. Unfortunately, without revisiting the interaction between software and hardware, easier programming comes at the price of less parallelism.

The memory interface is a striking example of a hardware/software interface that is ill-suited for parallelism, and specifically for parallel access to the program’s state. Even on modern computers, that interface follows the design described in the von Neumann architecture more than 70 years ago. This interface design is fundamentally sequential and suited for serial instruction execution. Indeed, modern memory systems allow multiple cores to read and write state concurrently, and the underlying implementation can even handle out-of-order accesses done by a single thread. Still, the mainstream memory interface merely supports parallel execution; it is not designed to ease parallel programming or be tightly integrated with high-level programming models.

This dissertation describes an extended memory system, whose main features are versioning and renaming. Versioning allows programs to encode dependencies along with the state, enabling the memory systems to enforce proper access ordering. Renaming allows multiple versions of the same memory location to coexist, eliminating false dependencies. When leveraged by a task-based programming model, the extended memory system facilitates efficient parallel execution of common sequential algorithms, even when their state is managed by dynamic data structures. The outcome of the execution is deterministic and identical to a sequential execution of the same code. A hardware implementation of the memory extension is also detailed, along with practical requirements such as an effective garbage collection.
Chapter 1

Introduction

For more than a decade, computer systems include an increasing number of processors. This trend was driven by the diminish of Dennard scaling [DGR+74], which prevented processors frequencies from being increased. Moore’s law [Moo65], however, still provided an increasing number of transistors per processor with every technology generation, allowing new processor to contain more compute resources. Unfortunately, getting the additional transistors to improve single processor performance without increasing clock speed is much harder. Instead, processor manufacturers took a different path and moved to multi-core processor designs.

Hardware with multiple processing units can only be used by code that embodies a sufficient amount of parallelism. In practice, neither extracting potential parallelism from a sequential specification nor developing parallel algorithms from scratch is easy. Moreover, while inherently parallel algorithms tend to scale better as the number of processing units increases, sequential algorithms are still much easier to reason about and debug. As the performance gain provided by increasing processor frequency is no longer available — free lunch is over [Sut05] — fully exploiting modern hardware is challenging.

The gap between ease of programming and modern hardware architecture has motivated a plethora of proposed solutions, spanning the range between micro-architecture and high-level algorithm design. Figure 1.1 classifies a few noticeable technologies allowing parallel processing. Not all technologies target the same workloads and use the same amount of resources, hence cannot be placed on absolute scales of performance and ease of use. Still, general trends can be observed:

- High-level software abstractions make programming easier but obtain less parallelism. Cilk [BJK+95], for instance, allows easy switching between parallel and sequential code sections, but the runtime is provided with little information. On the other hand, lock-free programming relies on atomic operations and the architecture’s raw memory model. Verifying such algorithms is extremely difficult, but when done right, they can exploit most parallelism provided by the hardware.

- Micro-architectural solutions such as out-of-order execution cores can parallelize sequential code, allowing easy programming. However, parallelization is at the instruction level,
and parallel resources are within a single core. The amount of parallelism is thus very restricted.

- Traditional dataflow architectures [DM75, AN90] use non-imperative programming models, which require state copying and cannot leverage locality. While the dataflow model can in theory exploit all parallelism in the program, in practice the overhead of parallelism management (mostly tracking dependencies) can overwhelm the system.

- Speculative execution can use hardware support for better performance and offer an easy programming model. Transactional memory [HM93] is used with parallel code hence is somewhat harder to program, while Thread-level speculation [SBV95] parallelizes sequential code and only requires minimal hints.

The von Neumann interface between hardware and software is a very convenient abstraction but also makes it harder for each side to fully exploit the other. For instance, on the x86 architecture, writes to a location that is used by a single processor and writes to a location that is used for cross-processor communication are done using the same instructions. The hardware thus has no way to determine whether one processor’s write should be pro-actively pushed to another processor’s cache, reducing the latency of the next read done by the other processor. On the other end, parallel algorithms implemented in high-level programming languages use software synchronization abstractions, built on top of the low-level hardware interface. While common abstractions such as a mutex are easy to understand and use, hardware designed for parallel workload can provide much more than atomic operations and a coherence protocol.

---

Figure 1.1: Various parallel processing technologies. The star marks the target of this work.
1.0.1 State management

A central entity in every program execution is its state. A computation is guaranteed to be correct only if, at any given moment, the memory system presents the program with a state consistent with the current computation step. In the traditional von Neumann model, instructions appear to execute one by one, communicating with the memory system via a single channel. When extended to parallel machines running multi-threaded programs, a sequentially consistent memory interface becomes impractical. Instead, the hardware allows a certain amount of discrepancies between the state different threads can observe at the same time. Those discrepancies are precisely described in the memory model specification. In order to ensure the state’s consistency, multi-threaded programs must take the memory model into consideration, and use dedicated instructions to eliminate state discrepancies where they violate the computation’s correctness.

In the von Neumann model, parallel memory systems merely provide the appearance of a unified memory system, and minimally order concurrent accesses. Namely, the memory system is unaware of the semantics of the program, the use of the state it stores and in particular how threads use the state to communicate with each other. If one thread is expected to consume state generated by another thread, the program must construct mechanisms that will ensure this ordering. Those mechanisms will be implemented in software and extend the program state. While the combination of von Neumann model and software synchronization abstractions is used by the vast majority of systems, dedicated memory systems can simplify software and speed up parallel computations.

The dataflow model takes a very different approach toward state management. Instead of considering code execution and the memory system as two separate entities, the dataflow model drives execution based on the availability of the state. A dataflow program is not specified by a sequence of operations, but rather by a graph of dependencies among different operations. Non-dependent operations can thus be scheduled to execute at any possible order (as well as in parallel) regardless of the order in which they appear in the program’s source code. The state in the dataflow model is not stored in a central repository, but rather directly transferred between dependent operations. Consequently, the runtime system that schedules execution must be able to infer the state produced and consumed by every operation.

The dataflow model is inherently parallel. However, dependency management is a burden. First, each operation must clearly indicate the state it produces and consumes in a way which allows dependencies among operations to be calculated. Further, tracking dependencies at runtime becomes harder as parallelism increases. Lastly, the dataflow model prohibits operations from having side-effects, namely state modifications that are visible by other operations but are not identified as dependencies. While functional programming has recently been regaining popularity, it is by far less common than the dominant imperative programming paradigm.
1.0.2 Memory system extensions

The limited support for parallelism in the von Neumann model and the overheads of the dataflow model have motivated various attempts to extend the memory interface. The common goal of many extensions is to bind ordering or dependency information to the program’s state, allowing the memory system to actively affect execution progress by stalling premature accesses to data. For instance, Cray MTA/XMT machines [ACC+90] employed full-empty bits to mark whether a particular piece of information was ready for consumption or not. This strategy was also used in I-structures [ANP89], allowing immutable state to be safely shared among separate operations on dataflow machines, saving data copying. The single bit was extended to a numeric version identifier in ROKO [SA12], allowing finer control over dependencies.

The memory system described in this dissertation further extends the traditional interface, offering 3 powerful capabilities:

1. **Versioning:** Each datum is tagged with a version number, allowing read-after-write dependencies to be enforced. Like full-empty bits versioning prevents reading data that was not created yet. Like M-structures [BNA91], versioning allows a memory location to be modified, yet without a restrictive access interleaving.

2. **Renaming:** Also known as multi-versioning, renaming adds a dimension to the flat memory layout, allowing each location to hold multiple versions of data at the same time. When read-after-write dependencies are properly enforced by versioning, renaming eliminates write-after-read and write-after-write dependencies.

3. **Locking:** Synchronization is all about protecting the program’s state. Providing hardware-supported synchronization primitives (namely, locks) that are tied to the state is both useful and efficient.

Those extensions are visible to the program via the instruction set, allowing semantic information to be delivered to the hardware. The extended interface allows the software to leverage additional hardware capabilities and allows the hardware to make more informed decisions based on provided semantic information.

1.0.3 Programming models

Leveraging extended memory systems requires appropriate programming models. Those models can be sequential or parallel, emphasizing either ease of programming or performance, respectively. In general, dataflow-based models rely on sequential specifications, augmented by programmer annotations or compiler analysis. On the other hand, models that are driven by control flow rely on inherently parallel specifications.

As systems are becoming more parallel, the task-based programming model is gaining popularity. A task is an abstraction representing a sequence of instructions performing a distinct part of the computation, typically a function or loop body. As opposed to threads, which abstract a processing unit, tasks are not related to the underlying hardware — one thread (or core) can
execute multiple tasks, and a task can migrate across threads (or cores). The limited amount of computations and clear bounds of every task makes it easy to reason about and specify tasks inputs and outputs. This makes tasks a common building block of dataflow-based systems, which schedule processing work based on the availability of the work’s input.

Task-based models can be designed to guarantee the \textit{statically sequential} property [GS11]. Given this property, the tree of tasks created by a program incorporates the exact ordering of a sequential execution, as specified in the source code. Consequently, the outcome of a parallel execution will be identical to the outcome of a sequential run, regardless of the amount of parallelism exhibited at run time. Such models can thus offer both ease of programming (code can be debugged on a sequential execution) and the ability to leverage parallel hardware.

1.0.4 Parallel execution using memory versioning and renaming

This dissertation describes a memory element called \textit{O-structure}, which adds versioning, renaming and locking to state management. O-structures design (and name) is inspired by the dataflow semantics of I-structures and M-structures, but the extended memory system is meant to be used by imperative code. O-structures expose versioning, renaming and locking via dedicated instructions, allowing programs to use the \textit{Ordering} capabilities to safely execute in parallel.

The complete set of features provided by O-structures is best used by task-based programs. When dependencies are encoded in memory as versions, true dependencies are enforced, false dependencies are eliminated, and execution is statically sequential. Further, when shared state is maintained in widespread pointer-based data structures such as trees, a parallel statically-sequential execution is achievable even if dependencies cannot be fully reasoned about.

The extended memory interface is also useful for models other than task-based execution. Versioning can be used to simply indicate the availability of some data, much like I-structures and full-empty bits. Versioning can thus allow multi-threaded programs to safely share immutable data. Renaming can also be used to provide snapshot isolation, ensuring consistent views of data structures that are modified by concurrent threads. Such features can also be implemented in software, but hardware support makes O-structures much faster than any software implementation.

O-structures hardware implementation described in this dissertation supports an unlimited number of versioned locations, unlimited number of versions per location, and efficient garbage collection. Most of the logic is added to the caches subsystem, without modifying the main memory. Further, any memory location can be treated as an O-structure if accessed using the dedicated instructions (but private state can and should be managed using the conventional memory interface). In order to support unlimited storage, dynamic data structures must be used by the system, possibly incurring an overhead higher than the use of limited buffers. However, this design decision relieves both the system and the programmer from worrying about exceeding system limitations and handling such cases.
1.0.5 Parallelizing dynamic data structures

Some common pointer-based data structures provide a useful property: algorithms operating on them always enter the data structure via the same pointer (namely, root or head), can reach a certain node using a single path, and never revisit a node after moving to another. This property is called Unipath and is related to the dominator concept [Pro59] of graph theory. Using O-structures, operations on Unipath data structures can be efficiently parallelized. Moreover, an extended compiler merely needs a few hints (provided via programmer annotations) to parallelize a series of sequential operations on a Unipath data structure.

One technique used in Unipath parallelization is hand-over-hand locking. Hand-over-hand is a well known fine-grained locking scheme, mostly implemented in software running on conventional hardware. However, such implementations suffer from the overhead of a per-node lock, modified every time the node is accessed. Fortunately, the Unipath pattern can be leveraged to provide the appearance of hand-over-hand locking using a completely different mechanism: snapshot-based synchronization. By storing a private recording of the locations of all other threads, a thread can check locally whether an access to a certain node is safe, and communicate with other threads only if a potential collision is detected. Using snapshot-based synchronization to parallelize Unipath operations thus outperforms traditional hand-over-hand locking.

1.0.6 Summary

This dissertation describes a solution that eases the burden of parallel programming while achieving an efficient parallel execution. The core of the proposed solution is combining state and dependency management using an extended memory interface, supporting versioning, renaming and locking. The unique interaction between software and hardware introduces both advantages and challenges on both ends; empiric evaluation shows the former prevails.

This dissertation is organized as follows: Chapter 2 describes the extended memory interface and an evaluation of the parallelism it unlocks. Chapter 3 describes the task-based programming model that relies on the proposed memory interface. Chapter 4 describes efficient hardware support and additional programming models it can serve. Lastly, Chapter 5 describes a software-only optimization of the central synchronization pattern.
Chapter 2

O-structures: Semantics for Versioned Memory

2.1 Introduction

This chapter introduces $O$-structures, a new type of storage element designed to aid the parallel execution of sequential programs. We choose the name “$O$-structure” because it provides ordered access to multiple versions of program state for the same memory address. $O$-structures are inspired by I-structures [ANP89] and M-structures [BNA91] for dataflow machines, but carry semantics designed for task-based execution models of imperative, not dataflow programs. Like I-structures, $O$-structures only allow versions to be written once, like M-structures, they provide lock-like semantics to access memory versions across tasks. Novel to $O$-structures, however, is their ability to support ordered access to multiple values stored within them.

$O$-structures are designed to be coupled with task-based execution models, where a single imperative program thread is carved via software or hardware into multiple program tasks. Tasks are expected to be ordered, yet crucially it is not expected that tasks are spawned in sequential order [RTL+05]. Like the von Neumann model, in task-based execution models, state is manipulated sequentially within a task and persists indefinitely. Like the dataflow model, however, tasks that write state essentially create new versions of that state, rather than overwrite the existing data in memory which may be needed for the completion of tasks earlier in program order; and also like the dataflow model state can be created in parallel across program tasks. Finally, like the multithreaded von Neumann model, there exists a precise set of semantics for how memory operations interact between tasks.

$O$-structures are inspired by register renaming. Each time a task writes an address a new version of that memory location is created. Throughout this text, we denote the pairing of an address and a version of data at that address by $<\text{address:version}>$. The intent is, just as with register renaming, accesses across tasks are not serialized because of false output-address dependencies. When multiple stores are in flight to the same memory address, those stores will be directed to different versions and can thus proceed in parallel. When multiple reads are in flight to the same address, those reads will be satisfied by the appropriate version written by a
previous store, or held until the matching store is complete.

By resolving the ambiguity in naming different generations of a single memory datum, memory versioning increases task parallelism by allowing tasks to both spawn out of program order and generate state concurrently. Without memory versioning, correct association of data producers and consumers requires either conservative code generation, which inhibits memory-level parallelism [SMSO03], speculation [OHL+97], or restricted, in-order spawning of tasks [BPBL06, ECR+10].

The focus of this chapter is the semantics of O-structures, their interface and their interaction with task execution environments. While not describing a concrete implementation, we argue for the use of O-structure semantics as a building block and discuss how O-structures can be used to construct compound data structures. We demonstrate the potential parallelism extracted by O-structures using an execution-model simulator.

We begin in the next section by motivating the need for a new memory element. Next in Section 2.3 we formally define the behavior of O-structures. In Section 2.5 we demonstrate how O-structures can be used to build common data structures. Section 2.6 evaluates the MLP unlocked by O-structures. Finally, we conclude in Section 2.7.

2.2 Motivation and background

2.2.1 Existing memory interfaces

von Neumann systems

For single threaded von Neumann-based execution model machines, memory semantics are quite straightforward. Read operations must supply an address and the result is whatever the value of the previously written Write operation to that same address was. At the hardware level (below the operating system and user process abstraction) a Read sent to an address that has no prior Write returns an undefined value. There is no accepted name for a memory location on a von Neumann machine, so as a shorthand in this chapter we will call them V-structures.

When multiple processors (or cores) are sharing access to the same physical memory, additional semantics are enforced. This is known as the consistency model and it defines how Read and Write operations are interleaved across cores. Additional operations, fences, are typically provided in order to allow programmer control of the interleaving. In addition, atomic operations, such as Atomic-Compare-and-Swap, or Test-and-Set, are introduced for applications to synchronize their threads of control with changes in memory state.

Dataflow systems

With dataflow computers [DM75] the primary memory system is the token store. Conceptually, the token store is an associative array. Programs name data values with a tag. The \(<\text{tag:}\text{value}\>\) is inserted into the token store by the result of an instruction. Dataflow machines have been designed and built with a wide variety of tag and matching schemes, but from a high level
perspective tags are used to name the inputs to a particular dynamic instance of an instruction to be executed. Once all inputs are available (a tag match), the instruction is executed.

Early on researchers recognized the inefficiency of building large token store memories [CSvE93] and developed alternative “dataflow-compatible” memory systems. Perhaps the earliest and most widely known is the I-structure [ANP89]. An I-structure is a memory location that can be written only once but read multiple times. When a Read operation is dispatched to an address, if no Write to that address has occurred, the Read request is queued. When a Write operation occurs to an address, any pending Read requests are dequeued and provided with the written value. Subsequent Read operations to the address are satisfied with the result of the Write. A subsequent Write to the same address is an error. The I-structure was followed up by the M-structure [BNA91]. An M-structure is subtly different than an I-structure, and conceptually an M-structure is a set data-structure. A Write (called a put in M-structure parlance) inserts a value to this set. A Read (get) removes a value. When the set is empty, a pending Read is queued until it can be satisfied by a Write.

**Exotic systems**

In addition to V-structures, token stores, I-structures and M-structures, systems have been built with other novel memory semantics. The Cray MTA [ACC+90] had full/empty bits on each memory location. These bits eased inter-thread synchronization by providing more sophisticated read and write operations, such as Read-when-Full, and Write-when-Empty. Several systems have also incorporated basic type information into the memory system [DCF+92, Org83]. Machines have been built that support in-memory operations such as Fetch-and-Add. When these operations are associative they can even be combined [GGK+82]. Finally, several researchers have proposed more sophisticated in-memory computation logic, most notable in the late 1990’s with the intense amount of research into “Intelligent Memory” architectures [OCS98, KPP+97, KHY+99, CHS+99, HKK+99].

**2.2.2 Limitations of existing memory interfaces**

Our goal is to have a memory system that supports highly concurrent execution of single-threaded programs written in imperative languages. These execution systems, typically task-based [KT99, SBV95, RSL93, FLR98, BPBL06], have two defining characteristics to note here: (1) their goal is to execute with high concurrency, hence ideally multiple memory operations to the same address will be in flight simultaneously; and (2) because instructions (or tasks) come from an imperative language program thread, there is an ordering between them. This means the outstanding memory requests issued by those tasks have an ordering as well, and this ordering must be respected for correct program execution.

A V-structure (von Neumann memory system) is not a good match for these execution models, because only one value for a given address can be stored at a time. Moreover, there is no ordering between Read and Write operations. Without additional logic, a Write later in program execution could erroneously satisfy an earlier Read. The MultiScalar team recognized this issue
and developed the Speculative Versioning Cache [GVSS98]. Speculative Multithreading systems also rely on processor caches [KT99] to maintain different versions of data for the same address. Recently, non-speculative versioning systems, which rely on programmer annotations of task read and write sets, have been proposed. Task superscalar [ECR+10] uses memory renaming to eliminate output dependencies in task-based execution. ROKO [SA12] maintains serial semantics using read and write timestamps for every shared address, but only a single version of the data is kept. While simplifying versioning implementation, this does not eliminate output and anti-dependencies, and requires that read operations update a versioning data structure.

I-structures are also a poor fit for task-based execution models because they can only be written once. I-structures are really not designed for programs written in imperative languages. While it is theoretically possible to compile such programs to a system that uses I-structures for memory storage, the effect will be to grossly waste resources.

M-structures have some of the necessary aspects of a memory structure suitable for task-based execution models. M-structures can be written to more than once and can pair individual Write operations to Read operations. Read operations to an empty M-structure also block, which is one of the necessary semantics needed in order to stall Read requests from later tasks from completing until earlier Write’s are dispatched. However, M-structures lack several key semantic elements. Write and Read operations are not ordered. For task-based execution models, this means a Read operation may end up being satisfied by a Write later in program order — again, clearly an error. Moreover, it’s important in these models that potentially multiple Read operations be satisfied by the same Write operation.

2.2.3 O-structures

In the next section we provide a precise description of the semantics of an O-structure. Here, we will describe its operation informally, and motivate its use by way of example.

An O-structure supports multiple versions of the data stored within it. These versions are ordered. For example, version #1 is before version #5. Versions can be written out of order because they are independent of each other. For example, version #5 can be written before version #1.

Programs can read and write a specific version of data. The read operation implicitly acquires access to the requested version and obtains the actual data only when it is available. The write operation creates an empty version, stores the actual data and implicitly releases the new version, making it available to other tasks.

Let’s consider now an extremely simple example (Figure 2.1). In this example, there are four tasks, some of which may execute concurrently. Given the semantics of O-structures these tasks compute the correct value for x, y and z as if the program code was executed sequentially. No other synchronization is required between these tasks. Task 1 writes a new value to variable x at version #1. Task 3 also creates a new value for variable x but with a different version #3. Task 2 reads version #1 of variable x and produces the variable y in version #2. Because Task 2 explicitly asks for the value of x created by Task 1, if Task 3 were to finish before Task 1
\begin{align*}
  x &= 5 & \text{task}_1: \text{store-version}(x, 5, \#1) \\
  y &= x + 1 & \text{task}_2: \text{reg}_1 = \text{load-version}(x, \#1) \\
  & & \text{store-version}(y, \text{reg}_1 + 1, \#2) \\
  x &= 6 & \text{task}_3: \text{store-version}(x, 6, \#3) \\
  z &= x + 2 & \text{task}_4: \text{reg}_2 = \text{load-version}(x, \#3) \\
  & & \text{store-version}(z, \text{reg}_2 + 2, \#4)
\end{align*}

(a) Code \hspace{1cm} (b) “Compiled” code to use O-structures

Figure 2.1: Simple example code and how a task-based execution model can use O-structures to parallelize it.

was even started. Task 2 would still block on the attempt to read version \#1 of the variable \( x \). Similarly, if Task 4 were to execute before Task 3 it would block on the attempt to read version \#3 of the variable \( x \).

### 2.3 Semantics

An O-structure is a memory element that maintains multiple versions of the datum. These versions are ordered. For example, given versions 1, 2 and 3, version 2 is ordered after version 1 and before version 3. We will see later that this order is expected to be used to reflect original program order for task-based execution models. Programs must specify what version of the memory location they wish to load or store. Initially, all versions exist in an \textit{uncreated} state. Loads directed to a version that is not yet created, will block. All created versions are available simultaneously for loading. For instance, if versions 1 and 2 have both been stored to, then they can both be loaded from. Versions can also be created out of sequence. For example, version 2 may be stored to and loaded from, before version 1 is ever created. In such a situation, loads directed to version 1 will stall, while those directed to version 2 will succeed. We also introduce per-version locks and the ability to load the latest version, which enable ordering when dependencies cannot be statically analyzed in full. For brevity, we omit the address from the following list of O-structure operations. All operations, however, would also take an address just as with ordinary loads and stores. O-structures support the following operations:

- \textbf{LOAD-VERSION}: Given a version \( v \), the call returns the value of that version. If the version has not been created yet or is locked by an earlier task \( u < v \), the call stalls. If it is locked by a later task \( w > v \), then the call returns the value successfully. Allowing the read of a locked version enables writers (here, \( w \)) to bypass earlier readers (here, \( v \)), without affecting the readers’ progress.

- \textbf{LOAD-LATEST}: Given a version \( v \), the call returns the value of the highest created version that is smaller than (or equal to) \( v \). If no such version exists or it is locked by an earlier
A version can be in one of observable 3 states: uncreated, locked and unlocked. The result of accessing a locked version depends on the relation between the locking task and the accessing task. We say task $T_1$ is earlier than task $T_2$ if on a sequential run, $T_1$ is executed before $T_2$; respectively, $T_2$ is later than $T_1$. Task $T_2$ will block if trying to load a version locked by $T_1$, yet $T_1$ can read a version locked by $T_2$. An implementation should also cater for garbage collection of unneeded versions, as described in Chapter 4. However, a discarded version must not be accessible to the program, hence its state is not observable. Table 2.1 summarizes the effect of performing the supported operations on a version in each of the states. UNLOCK-VERSION is omitted as it can only be called by the task that locked the version; like in conventional locking, any other option is clearly an error.

Operations accessing the latest version have two conceptual stages. First, the O-structure is searched for the highest version equal or below the required one. Second, an equivalent to LOAD-VERSION or LOCK-LOAD-VERSION immediately takes place if a version was found. If

---

<table>
<thead>
<tr>
<th>Operation</th>
<th>Uncreated</th>
<th>Locked by an earlier task</th>
<th>Locked by a later task</th>
<th>Unlocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD-VERSION</td>
<td>Stall</td>
<td>Stall</td>
<td>Load</td>
<td>Load</td>
</tr>
<tr>
<td>LOAD-LATEST</td>
<td>Load earlier ver. or stall</td>
<td>Stall</td>
<td>Load</td>
<td>Load</td>
</tr>
<tr>
<td>STORE-VERSION</td>
<td>Create</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>LOCK-LOAD-VERSION</td>
<td>Lock and load earlier or stall</td>
<td>Stall</td>
<td>Error</td>
<td>Lock and load</td>
</tr>
<tr>
<td>LOCK-LOAD-LATEST</td>
<td>Stall</td>
<td>Stall</td>
<td>Error</td>
<td>Lock and load</td>
</tr>
</tbody>
</table>

Table 2.1: The outcomes of performing operations on a version in different states.

---

1Since UNLOCK-VERSION does not store a new value, an implementation can avoid the creation of a new version $v_n$ by extending version $v_l$.
not, the operation stalls until a matching version is created. Importantly, the version lookup ends as soon as a smaller or equal version is found, regardless of its state – locked versions are not skipped in hope to find an unlocked version.

Versions that are locked by later tasks (namely, executed later on a sequential run) can be immediately loaded, without waiting for them to be unlocked; versions locked by earlier tasks, however, must be waited for. The purpose of those asymmetric semantics is to differentiate dependencies between read and write operations. A writer is required to use locking operations. If it is earlier than a reader that attempts to read a version the writer is locking, the reader should stall in order to maintain read-after-write dependencies. On the other hand, if the reader is earlier than the writer, it should not observe any state generated by that writer, hence need not wait for the writer (if the writer does perform a store before the earlier reader reaches the target O-structure, the store will create a version that will be skipped by the reader anyway). Lifting the need for earlier reader to stall later writers eliminates the write-after-read anti-dependency.

Certain operations result in an error when performed on versions in an inappropriate state. First, a version cannot be created twice or updated, hence STORE-VERSION is only legal if the specified version does not exist yet. Second, locking loads must not encounter a version that is locked by a later task. The purpose of locking is to allow write operations to stall following operations (both reads and writes) until the write generates the new state. Stalling ensures the new state will be available to later operations, as required in order to maintain the statically sequential semantics. A task that attempts to stall following tasks yet encounters a version locked by a later task is effectively behind the later task. Being ahead of the earlier task, the later task is not guaranteed to observe the state generated by the earlier task, violating read-after-write dependencies. Such an outcome is not the result of the current operations, but rather on some previous data race that allowed the later task to bypass the earlier one. Given the program has reached an undefined state, the actual course of action on such errors is up to the implementation.

2.4 Task-Based Execution and Versioned Memory

Task-based execution parallelizes sequential programs by hierarchically decomposing them into short, sequential tasks that can execute in parallel. In this section we discuss the ordering semantics of tasks and how O-structures can provide synchronized and deterministic task memory accesses while enabling task schedulers to both spawn and execute tasks out-of-order.

2.4.1 Background: task-based execution models

Task-based models rely on programmer annotations of procedure calls that can execute asynchronously from the calling context. This decomposition breaks a sequential program into a task tree. The backend runtime system then resolves the inter-task dependencies and extracts task parallelism whenever possible. The promise of task decomposition has prompted numerous task-based programming runtime environments [ASS08, ACD+09, BPBL06, BFJ+96, FLR98,
Parallel execution of conventional task models requires that inter-task dependencies be conveyed to the task scheduler, so data dependencies will not be violated. Different tasking models employ different synchronization methods. These can largely be categorized as *execution-based synchronization*, which rely on execution barriers [ACD+09, BFJ+96, FLR98, Rei07, SK10], and *data-based synchronization*, in which programmer annotations of task inputs and outputs are used to construct a data dependency graph [ASS08, BPBL06, RSL93, Ope13].

A key property of task-based decomposition is that tasks preserve the ordering of the original sequential code: An in-order traversal of the task tree recreates the program’s original, sequential ordering of events, and in particular the original order of memory accesses. Tasks are therefore considered to be *statically sequential* [GSS12].

Spawning a task splits the parent task into a prologue and epilogue parts. Everything that occurs in the parent before the task spawn should be ordered before the spawned task, and the parent code that precedes the spawn becomes the prologue task. Conversely, parent code that succeeds the spawn must be ordered after the task, and is regarded as the epilogue of the parent task. A tasks thus cannot spawn more than once before its current lifetime ends. The runtime can, however, choose to immediately continue execution of either the spawned task or the epilogue; in such cases, only the task identifier of the executing code must be updated.

2.4.2 How task-based models can use O-structures

In essence, O-structures provide communication channels between tasks, forming an implicit data-based synchronization mechanism. Each version is a channel that has a single producer, which is the task that created it, and any number of possible consumers, which are the tasks that read the version. Using task identifiers as O-structure versions is the natural way to define such a communication channel (static versioning, which is used in Figure 2.1 for simplicity, is usually impractical. However, other schemes are available, and discussed in Chapter 3). Synchronizing tasks at the granularity of a single location allows more parallelism, as demonstrated in Figure 2.2.

Explicit data versioning enables tasks to *spawn* out-of-order rather than only *execute* out-of-order. This is the result of using task identifiers as data versions, and having task identifiers rely on the program’s sequential order rather than parallel execution. Note that the parallelism exposed in a task-based program that uses O-structures is a direct function of the ability of the compiler to track and express dependencies between tasks. If the compiler (or programmer) does not / cannot express the dependencies then extra synchronization between tasks is required.

\footnote{For brevity, we assume that tasks are composed of complete procedures. Nevertheless, some models allow arbitrary code sequences to be executed as asynchronous tasks [ACD+09].}
2.5 Managing data structures

In this section we discuss how programs that manipulate various data structures would use O-structures and additional synchronization to achieve parallel execution.

2.5.1 Static data structures

Operations on static data structures, e.g. arrays, do not alter their structure but only their contents. If the accesses are statically analyzable then the dependencies between tasks can be encoded in the versions to unlock significant parallelism; e.g., a load done by a task x can be made directly to the value stored by task y.

On cases in which dependencies are not statically analyzable, the compiler must introduce some additional ordering. Consider the creation of a histogram: the compiler cannot predict the order of accesses to each array element, but all accesses to array elements go through a pointer to the array. That pointer could be used as a serialization point, always locked using the current version and unlocked such that the following task can lock it. The latest version of the elements can then be locked and loaded, and the new value can be stored and unlocked using the current task’s identifier. If the pointer is released after the array element is locked, overall sequential semantics are guaranteed, while accesses to different elements can be done in parallel.

2.5.2 Dynamic data structures

Operations on dynamic data structures such as lists and trees impose a greater challenge. The connectivity and memory layout of these data structures are determined by dynamic pointers. Mutating operations such as insert and delete not only change the content stored in the nodes, but also modify the node pointers, altering the layout of the data structure itself. Further, elements in these structures are not accessed directly but rather by first traversing a series of other nodes.

The unipath property is an algorithmic characteristic of operations, which always follow the same path in order to reach a certain element of a data structure. Traversal of a singly-
linked list, for example, has the unipath property: in order to reach a node, all previous nodes starting from the head must always be followed. On the other hand, mutations of balanced trees do not have the unipath property, because rebalancing can cause a revisit of nodes that have already been visited. Operations on data structures like singly-linked list, singly-linked trees and arrays reachable from a single pointer are inherently unipath; such data structures are therefore considered unipath data structures. Operations and data structures that do not have the unipath property are considered to be multipath; we discuss those properties in depth on Section 3.

## 2.6 Evaluation

Our evaluation measures the amount of memory-level parallelism (MLP) that could be obtained when operating on several common data structures. The MLP criteria distills the effect of using O-structures, which effectively provide both versioned memory and synchronization. Computational operations between memory operations, which can freely be executed in parallel, are not considered in the evaluation.

### 2.6.1 Simulation framework

Our infrastructure is trace-based. We first implemented sequential versions of all the algorithms we consider: insertion to the end of a singly linked list, insertion to a binary tree, multiplication of 3 dense matrices and multiplication of 3 sparse matrices. We hand-annotated these codes with task spawn and O-structure operations. Our approach was a “best a compiler could achieve”. The simulator takes these execution traces, and models the tasks executing on a fixed number of processing units. The simulation follows the semantics of task based execution on a system that uses O-structures: a read must stall if the target version is not available; writes are performed immediately and create a new version of the data; tasks are created out-of-order but all dependencies between tasks are faithfully modeled. Our evaluation considers a large number (128) of processing units, although more parallel execution resources and larger data set sizes improve performance.

The amount of MLP is obtained by dividing the number of O-structure operations on a given trace by the number of cycles required to complete the parallel execution of the trace. In our simulation, each unblocked operation takes one execution cycle. This includes all O-structure and task management operations. Stalls are introduced by loads of uncreated or locked versions and task dequeues when there are no ready tasks. Such stalls last until the conditions for them to continue are fulfilled. Albeit single cycle operations are hard to achieve in a real system, our goal was to measure the amount of parallelism, which is limited by the stalls described above. Lastly, since measured algorithms were implemented sequentially, it is sufficient to assume the “baseline” MLP for a conventional von Neumann execution system is 1.
Figure 2.3: MLP of insertions to the end of a linked list, running on 128 cores. Longer lists provide longer pipelines, allowing more parallelism.

2.6.2 Experimental results

Linked lists insertion and matrix multiplications scale nicely, while binary tree insertion scales with the height of the tree.

On the linked list benchmark (Figure 2.3), insertions are pipelined. This bounds the number of active insertions at any given moment to the number of elements that have previously been inserted. As linked lists grow in size more parallelism is available.

The binary tree insertion (Figure 2.4) enforces no serialization of tasks that traverse different paths, but the root still serializes all insertions. This bounds the maximal number of tasks actively traversing the tree to the height of the tree, which ranges between $N$ and $\log_2(N)$. Consequently, the inherent amount of MLP grows slowly with respect to the input size. Namely, insertions to trees obtain lower MLP than insertion to lists because although the insertions become active at an identical rate, trees insertions are completed much faster, leaving less time for multiple tasks to run in parallel.

On the dense matrix benchmark (Figure 2.5), each element of the temporary matrix can be calculated independently. Dependency exists between the calculation of the temporary matrix and the second multiplication, but given the total amount of work, its overhead is negligible.

Sparse matrices (Figure 2.6) are stored as arrays of linked lists, each list holding only non-zero elements in the row it represents. Multiplying 3 matrices could suffer from (1) the overheads introduced by having to wait for the temporary matrix to be calculated and (2) inserting to short linked lists, which bound the number of active insertion tasks. However, sparse matrices are commonly used for very low percentages of non-zero elements. If the percent of actual insertions is low enough, most of the work is done by tasks that can run in parallel. Consequently, the amount of MLP not only depends on the size of the matrices, but also on the percent of non-zero elements and their locations.

In our experiments, sparse matrices obtained higher MLP than dense matrices. This is only
Figure 2.4: MLP of insertions to a binary tree, running on 128 cores. The depth of a tree is usually much lower than the number of elements, providing a short pipeline. While insertions do not synchronized once their paths diverge, entrance into the tree restricts parallelism.

Figure 2.5: MLP of 3-way dense matrix multiplication, running on 128 cores. The large amount of work with low rate of dependent operations allow a lot of parallelism.
Figure 2.6: MLP of 3-way sparse matrix multiplication, running on 128 cores. The dynamic sparse representation introduces dependencies, but the lower number of calculations allow larger matrices to be handled.

an artifact of using larger input sizes. When larger dense matrices are multiplied the obtained MLP is higher. It is worth noting that matrix multiplication is a highly parallelizable operation, and in our experiments large enough workloads were only bound by the number of simulated processors.

2.7 Conclusions

This chapter introduced O-structures, a novel memory element designed to facilitate parallelism in task-based execution models. We began by describing the complete semantics of O-structures. We then showed how O-structures expose parallelism by removing false output name dependencies in various data structures software developers commonly employ. Finally, we presented results from our execution model simulator that demonstrated significant MLP (over 100X) from sequentially coded implementations of algorithms when O-structures are employed. On Chapter 3 we deepen the discussion on the dedicated task-based execution model that relies on O-structures, and on Chapter 4 we present hardware support for memory versioning and renaming.
Chapter 3

Deterministic Fine-grained Task Ordering using Multi-Versioned Memory

3.1 Introduction

Task-based programming models present the developer with an abstract parallel machine. Typically this abstract machine has useful guarantees for the developer, such as an order in which the tasks will be completed, or when parallel execution will be synchronized. Furthermore, the abstract machine is designed to hide many low-level details such as which cores execute which tasks and in which order.

Many task-based programming models rely on dependency and output lists, specified by the programmer, to provide non-speculative and thread-safe parallel execution (e.g., [GS11], CellSs [BPBL06], TaskSs [ECR+10], OpenMP 4.0 [Ope13] and OpenStream [PC13]). Input and output specifications are conveyed to the runtime and stored as part of each task’s metadata. Task execution is then driven by the dataflow firing rule [KM66], which dictates that a task is considered ready for execution only once all of its inputs have been produced by other tasks. Relying on the firing rule ensures that once a task is ready, it can be executed from start to end without worrying about dependencies.

A major shortcoming of the firing rule is the expressiveness of the dependency specification. In order to mark a particular datum as a dependency, that datum must be named and the task-based runtime notified. Naming can rely on a scalar variable, single pointer dereference or array offset. Some models support the naming of high-level data-structures with associated semantics, such as streams in OpenStream. Notably, all of these naming schemes exclude internal nodes of pointer-based data structures from being specified as a dependency — such nodes are anonymous even if the node within the data structure is pointed to by some variable. Consequently, the use of pointer-based data structures in task-based programming models is restricted: typically the entire data-structure is treated as a single object, leading to excessive
synchronization between tasks.

Dependency specification is not a problem for an alternative task-based execution strategy: speculation. In the speculative execution model, tasks can be executed even if their dependencies are unknown in advance. Once tasks complete, they must be committed. During the commit phase, dependencies are checked; if a violation is identified, the task is re-executed.

Since speculative task execution does not rely on dependency specification and the firing rule, it can cope with anonymous dependencies. Consequently, dynamic data structures can be used for shared state and be accessed by multiple tasks. However, speculative execution introduces another limitation: read and write sets. Those are used when checking dependencies on the commit phase. Managing and checking the sets in software allows unlimited size but is slow. Hardware-based solutions are faster but are typically limited in set sizes.

This chapter proposes the Multi-Versioned Memory (MVM) model, a task-based programming and execution model that relies on O-structures [GMOE14], an extended memory system providing data versioning and renaming. Versioning is used to embed dependencies in the memory system, eliminating the need for a predefined dependency list. In the absence of dependency specification as part of the tasks metadata, the model cannot rely on the firing rule for scheduling. Instead, tasks are considered ready as soon as they are created, and can be launched according to any scheduling policy. During execution, if a task tries to access a version that does not exist, the memory system stalls it until the version is created. The result is a fine-grain, deterministic, dataflow execution model. In addition, data renaming is obtained by storing multiple versions of the same datum. This eliminates false dependencies, further increasing parallelism and reducing the need to track accesses to shared locations.

Execution under the MVM model is designed to have the statically sequential property [GS11]. Namely, the parallel execution of a program should yield identical results to a sequential execution of the same program. This is done by mapping task identifiers to their sequential ordering. Task identifiers are then used for versioning, effectively reflecting the sequential ordering of the program in the memory system.

Importantly, MVM targets imperative code. Traditional dataflow execution models targeted functional code; such code prohibits side effects, forcing the input and output of every task to be explicit as function parameters and return values. While making dependencies clear eases scheduling, the functional model relies on value semantics, causing data structures to be copied rather than shared. I-structure [ANP89] and M-structure [BNA91] were proposed to mitigate the cost of the copies, but neither resulted in the flexibility and efficiency of imperative code.

MVM is not bulletproof: mistreated side-effects can cause dependencies to be violated, resulting in undefined behavior. Unlike a model that relies on speculation and verification to ensure correct execution, programmers (and/or compilers) have to use MVM correctly. In this chapter we’ll also explore how this is done.

This chapter focuses on the interaction between a task-based execution model and multi-versioned memory. Heavily researched aspects such as high-level programming model and optimized scheduling were minimally implemented and are left for future work.
Figure 3.1: Mapping of a hierarchical, task-based decomposed program onto a monotonic virtual timeline (horizontal axis). The decomposition enforces the sequential order of inter-task memory dependencies. This is known as the statically sequential property of task-based programming models [GS11].

3.2 Task-based programming and memory versioning

Non-speculative task-based programming models are effective in expressing parallelism, but are not backed by efficient hardware primitives that support the statically sequential property. These models must thus employ software-level primitives for synchronizing and communicating inter-task data dependencies, and thereby incur prohibitive runtime costs. In turn, programmers are forced to use coarse-grain data dependencies in order to amortize the runtime costs.

The emerging class of task-based dataflow programming models (e.g., [BPBL06, PBL08, ECR'10, GS11, Ope13]) demonstrates this tradeoff. These models extract parallelism by scheduling tasks using the dataflow firing rule. These models employ programmer annotations of task inputs and outputs in order to identify in-memory task dependencies. Task inputs and outputs are used to construct the task dataflow graph that guides the scheduler. Given task inputs and outputs, the scheduler can expose parallelism by scheduling tasks according to their explicit data dependencies. The dataflow model thus removes the programming burden of dealing with explicit synchronization primitives (e.g., mutexes, semaphores, transactional memory [HM93]).

A major limitation of programmer annotation of input and output parameters is that annotations are restricted to contiguous or strided [PBL10] memory regions. This limitation is a culmination of two constraints. First and foremost, annotations need to be expressed at coding time, and must therefore be static. Furthermore, the expressed dependencies must be compact, since those that involve large memory segments or multiple tasks might curb parallelism. As a result, programmers cannot express dependencies that involve irregular, dynamic data structures (e.g., lists and trees), which greatly limits the problem domain that such programming models can address. This limitation gives programmers who wish to process irregular data structures no choice but to use fine-grain, explicit, synchronization primitives, such as mutexes or transactional memory. The nature of these primitives makes them more difficult to reason about.
3.2.1 The need for memory versioning

Task-based programming can provide a hierarchical code decomposition that is *statically sequential* [GS11] if it implicitly preserves the original, sequential ordering of operations. We argue that the models’ inherent ordering can be leveraged to order memory operations.

Hierarchical task decomposition and memory order

Figure 3.1 demonstrates a hierarchical task decomposition and its mapping to a virtual timeline that represents the original program order. The figure depicts a decomposition of task $T_0$ (i.e., the original program) into three tasks, $T_1$, $T_2$ and $T_3$, which are further decomposed. In a sequential execution, each of the tasks views the memory in its state just prior to the task’s execution. For example, task $T_{2,0}$ expects memory elements to be at virtual time $vt_4$. However, if a datum consumed by task $T_{2,0}$ is produced by task $T_{1,1}$, then a virtual time-cognizant memory system can identify that task $T_{2,0}$ consumes an item that is ready at virtual time $vt_2$ and can, therefore, execute it in parallel to $T_1$’s remaining sub-tasks. Alternatively, if the compiler/programmer identifies that a datum produced by $T_{1,1}$ is not used by any of the remaining sub-tasks of $T_1$, it can notify the memory system to advance the virtual time of the said datum to task $T_2$ at $vt_4$ (horizontal arrow in Figure 3.1).

How versions remove these limitations

A memory system that allows data to be tagged with a version provides a fine-grained synchronization primitive, which needs not be explicitly specified as a task dependency. Since versions correlate to the sequential ordering of the program, the combination of data and version represents a producer-consumer relation. In a sequential run, the same memory location can serve multiple producer-consumer relations without causing data races. To achieve the equivalent ability in a parallel execution, each memory location can be renamed, namely store multiple versions of the same location. Importantly, only memory locations that form a dependency among tasks need be versioned and possibly renamed; task-local operations can use the conventional memory interface.

To illustrate the benefit of the proposed model, let us consider a series of operations involving sparse matrices. Figure 3.2 presents a partially created sparse matrix in the List-of-Lists (LIL) format. While it is being constructed, two tasks are trying to read the matrix: $T_{i,0}$ reads row #0, while $T_{j,1}$ reads column #1. Assuming each row was computed by a different task, the data’s version corresponds to the row number. The two tasks operate as follows:

- Task $T_{j,0}$ uses version 0, and successfully reads the first row, which was fully calculated.
- Task $T_{i,1}$ uses a different version number for each row, and traverses the list until it finds the required column (or goes past it). Due to the fine-grained synchronization provided by versioned memory, the task successfully reads column #1 of row #1, even though the row itself has not been fully calculated yet. However, when reaching row #3, the task is stalled until the value is available.
Figure 3.2: A partially created 4*4 sparse matrix (in the List of Lists format). Task $T_{i0}$ successfully reads row 0, while task $T_{j1}$ stalls when trying to get the unavailable value of column 1 in row 4.

Figure 3.2 demonstrates operations on a data structure that cannot be efficiently handled by traditional non-speculative task-based programming models. Since each row is represented by a dynamically created list in which nodes are anonymous, a dependency on a particular column could not be expressed at compile time. The complete row could be treated as a whole using coarse-grained synchronization, but then readers such as $T_{j1}$ would have been stalled even when their data is already available. Speculative models can handle the unknown dependencies, but if the matrix is large, their read-set storage (whether implemented in L1 or dedicated structure) might overflow, causing a stall or squash.

It should be noted that the versioning in Figure 3.2 could be reduced to a single bit indicating whether the data is available or not. In section 3.5 we describe more complex use cases, which use both versioning, renaming and per-version locking.

As previously noted, task-based execution models have been explored in depth (e.g., [BJK+95, FLR98, BPBL06, ACD+09, ECR+10, JSY+15]). All task-based execution models aim to execute tasks concurrently, and in some cases out of the original program order. The challenge, of course, is that an earlier (in program order) task produces values that a later (in program order) task needs. These producer-consumer dependencies can sometimes be reasoned about at compile time, but on other cases the algorithm disallows deterministic detection before execution. Statically known dependencies, revealed either via the register dependencies in the control-dataflow graph, or through alias analysis of memory accesses, can be dealt with in one of two ways: (a) imposing a scheduling constraint that delays the later (consuming) task until the earlier (producing) task has created the necessary value; or (b) speculating that the
producer-consumer relationship will not arise dynamically and/or speculating on the value the producer will write. Our goal is to provide a third option that:

1. Aids parallel execution of known dependencies, by providing fine-grained inter-task synchronization via memory.

2. Allow unknown dependencies to be maintained by providing concurrent storage of multiple values for the same memory location, along with semantics for how these values relate to each other and control inter-task synchronization.

### 3.3 O-structures

O-structures are designed to support the parallel execution of sequential code. A variety of execution models can make use of O-structures, but we will ground our work in a fairly straightforward task-based model. A compiler (or developer) carves sequential code, written in a traditional imperative language such as C or Java, into tasks. Instructions within a task are logically expected to execute sequentially. Correct execution is achieved if tasks execute sequentially, or at least appear to do so, in the order specified by the original program. Conceptually, tasks can be as small as one instruction, in which case task-based execution reverts to conventional fine-grained instruction level parallelism. In practice, task-based execution models group small sequences of sequential code together, and thereby aggregate the overhead associated with task execution and management.

The interface and semantics of O-structures were defined in Chapter 2. As a reminder, we briefly specify the interface below:

- **LOAD-VERSION**: Load a given version; stall if unavailable.
- **LOAD-LATEST**: Load the latest version smaller or equal to the given one; stall if unavailable.
- **STORE-VERSION**: Store the given value in a newly created version.
- **LOCK-LOAD-VERSION**: Lock the specified version, then LOAD-VERSION that version.
- **LOCK-LOAD-LATEST**: Lock the latest version smaller or equal to the given one; then LOAD-VERSION that version.
- **UNLOCK-VERSION**: Unlock a given version.

Task-based execution models should use O-structures selectively, as versioned operations do bear some space and time overhead comparing to conventional memory. Task-private memory accesses (e.g., to the stack) and fully analyzable data-parallel operations do not benefit from using O-structures. Moreover, memory dedicated to I/O devices need not be in O-structures, as the semantics of I/O usually involves adhering to program order and being non-speculative. Both conventional (unversioned) and O-structure (versioned) functionality should be architecturally
visible. Conventional memory is accessed using traditional LOAD and STORE operations while versioned memory uses the dedicated operations described above.

The most straightforward way to use O-structures in a task-based execution model is to ensure that all tasks have a labeling that matches the original sequential program order. For example, suppose the programmer (or compiler) decomposed a loop such that each loop iteration was in a separate task. A straightforward task labeling would assign loop iteration \( n \) with task identifier \( n \), iteration \( n + 1 \) with identifier \( n + 1 \), and so on. Nested spawning can also be supported by using multi-level identifiers (our approach), ranges [SA12] and other schemes [CMP97]. A task could then use its task identifier as a version identifier when accessing an O-structure.

Combining task-based execution with O-structures yields an execution model in which each task’s view of the memory is as if execution were sequential, but multiple such views can exist in parallel, reflecting different stages of a sequential run. O-structures guarantee task-local view by prohibiting premature reads, effectively enforcing read-after-write dependencies. In addition, they also allow parallel views by storing multiple versions, eliminating write-after-write and write-after-read dependencies.

### 3.3.1 High-level interface

The low-level O-structures API is not intended to be used directly by programmers. In this work, we have considered two possible O-structures APIs, allowing parallelization of the algorithm:

1. An extended compiler (e.g., [May17]) applies versioning and renaming based on programmer annotations. Using this API, the programmer is required to specify the skeleton of the code and key algorithmic entities that can be used to parallelize operations by pipelining them (the complete technique is detailed in Section 3.5). Listing 3.1 demonstrates the use of such annotations alongside the baseline sequential code.

2. A library intended for both programmers and compilers, wrapping the low-level API. Using this API, the programmer is required to manually replace the types of versioned locations to a wrapper template class. The programmer also needs to manage versions and task creation. However, the low-level O-structure API is replaced with a higher level abstraction, which eliminates some complexity, provides type safety and so on. Listing 3.2 demonstrates the use of a such library to parallelize the same baseline code.

The relation between the APIs resembles OpenMP, which relies on annotations, and Posix threads, which are exposed via a library interface. In our view, the higher-level annotation-based API is preferable, yet the compiler can certainly leverage library wrappers to simplify transformations.
Listing 3.1 Sequential unversioned code vs. annotated parallel code

```c
struct node t {
    node t* next; ...
};

// assume non-empty list
node t* root = init();

void insert_end(node t* n) {
    node t* prev = root;
    node t* cur = root->next;
    while (cur != nullptr) {
        prev = cur;
        cur = cur->next;
    }
    prev->next = n;
}

int main() {
    // ...
    for (int i = 0; i < N; ++i)
        insert_end(new node t {i});
    // ...
}
```

Listing 3.2 Library versioning support

```c
struct node t {
    versioned<node t*> next; ...
};

versioned<node t*> root = init();

void insert_end(node t* n) {
    versioned<node t*>* prev = &root;
    // get a specific version of the head
    versioned<node t*>* cur = root->next.lock_load(ver(tid));
    // get the latest version and block following task
    node t* next = cur->next.lock_load_last(tid);
    // unlock previous node and increment its version
    prev->next.unlock_ver(tid, tid + 1);
    prev = cur;
    cur = cur->next;
    prev->next.store_ver(n, tid);
}

int main() {
    // ...
    for (int i = 0; i < N; ++i)
        create_task(i, insert_end, new node t{i});
    // ...
}
```
3.4 Execution Model

Conventional task-based models combine data and control flow in different ways. Most notably:

- Architectures such as Edge [BKM+04] and Dyser [GHN+12] utilize dataflow at the instruction level, but base higher level scheduling on control flow. While fine-grained dataflow can be quite efficient, reconfigurable hardware works best when the same code is run many times, mitigating the overhead of the initial hardware configuration.

- Models such as Cilk [BJK+95] and statically sequential [GS11] and processors such as Task Superscalar [ECR+10] schedule tasks or blocks based on dataflow relations among tasks inputs and outputs, but rely on control flow at the instruction level. While this scheme is natural for imperative code, it is restricted by the abilities of the firing rule.

In the MVM model, O-structures embed the dataflow execution semantics into a control-flow based run. Instead of being driven by the firing rule, the scheduler executes tasks based on some heuristics, akin to non-dataflow task-based models. Dataflow semantics across tasks are only enforced among dependent memory operations, via O-structures; the rest of the instructions are run based on control flow.

Figure 3.3 demonstrates possible schedules of conventional dataflow and MVM’s fine-grained dataflow execution. On the conventional model, a task can only be run after the tasks on which it depends have completed their execution. On the MVM model, a task can be launched at any given moment — before, after or during the execution of the tasks that produce required state. The progress of a task will be determined by the availability of the particular data it needs, synchronizing at the instruction level and not the task level.

MVM’s scheduling offers two advantages over conventional models. First, as demonstrated in Figure 3.3, more parallelism is available. Second, the scheduler needs not track dependencies among tasks by matching inputs and outputs — the dependencies are embedded in the memory system using O-structures. On the other hand, premature scheduling of a task whose input state has not been generated yet will waste resources, as the task will have to stall. A common way
3.4.1 Scheduling strategies

Task scheduling has been heavily researched, and is beyond the scope of this dissertation. However, we now discuss a few aspects that are unique to the MVM model.

A spawn effectively creates two tasks: the spawned task and the continuation of the spawning task. On a sequential execution, the spawned task is ordered before the continuation, hence will execute first — like a conventional function call or loop body. However, on a parallel task-based execution, the two tasks can be scheduled in various orders. Figure 3.4 demonstrates two possible scheduling strategies, described using Cilk \cite{BJK+95} terminology:

1. **Work first**: The spawn places the continuation on the spawning task in the tasks queue and immediately proceeds to execute the spawned task.

2. **Help first**: The spawn places the new task in the task queue and immediately proceeds to execute the spawning task’s continuation, namely the epilogue.

The order of execution in the work-first strategy is closer to a sequential run, as the spawned task (which is earlier in the statically sequential order) is executed before the continuation. If the continuation depends on the spawned task, it is more likely to be fulfilled when the continuation executes (this is not guaranteed, of course, as the continuation could be executed by another processor, concurrent to the spawned task). Further, if the spawning and spawned tasks access the same memory locations, the spawned task will experience fewer cache misses. Work-first’s advantage is therefore reduced stalls due to data dependencies.

The benefit of the help-first strategy is faster spawning when tasks are created in a loop. The continuation is then the loop header, which includes a condition (possibly after some counter update) and the next spawn action. Help-first also preserves the locality of the spawning code.

---

**Figure 3.4**: A spawn creates two tasks: the spawned one and the current task’s continuation (epilogue). The work-first strategy immediately executes the spawned tasks and enqueues the epilogue in the task queue; the help-first does the opposite.

to mitigate the overhead of stalled loads is preemption: stalled tasks will be scheduled out, allowing others to execute. The current O-structures interface does not support preemption as loads are expected to stall until required versions are available; we leave interface extension to future work.
allowing its memory context to be cached. Rapidly inserting spawned tasks (namely, loop bodies) in the task queue ensures processors will not become idle due to a lack of available work.

The work-first and help-first scheduling strategies are not unique to the MVM model, and can be used in any task-based execution model. However, MVM’s unique dependency management imposes some restrictions on the scheduler. The first and foremost requirement is that execution does not deadlock. A deadlock can occur if all processors are executing tasks, which are stalled due to data dependencies. The missing data can be created by tasks that are currently in the task queue, yet due to a lack of execution resources, those tasks cannot be scheduled.

One way to avoid deadlocks is by preempting stalled tasks, allowing others to execute. Preemption can ensure that a queued task that should provide the data for a stalled task will eventually execute, allowing the stalled task to complete when it is rescheduled. However, rescheduling preempted tasks when their dependencies are available requires a dependency management akin to dataflow tag store.

Another way to avoid deadlocks is by sorting the task queue and ensuring earlier tasks are executed first. Ensuring at least one processing unit is executing the earliest task at any given moment guarantees that a deadlock will be avoided, as the earliest task is bound to complete without any stalls. However, keeping the task queue sorted bares an overhead and makes hardware support much harder.

3.4.2 Task queue ordering

Each task has an identifier, which is used for versioning the state the task creates, as well as bounding loads of the latest values. However, tasks also load previous versions in order to consume state produced by earlier tasks. Each task should have at least one load of a specific (non-latest) version, otherwise it is not fully ordered with respect to other such tasks and violates the statically sequential property. A task is thus associated with a range of identifiers: the lower one (namely, earliest in program order) is the earliest version it loads, and the higher one is the task’s own identifier. The latter is used by the task when creating new versions. That range is known when the task is spawned.

If tasks are scheduled based on the beginning of their identifiers range, the state they consume is more likely to be available, as tasks that load earlier data will be scheduled first. However, a non-preemptive execution relying on this ordering is prone to deadlocks. Consider a program in which the statically sequential order of some tasks is $T_1$, $T_2$, $T_3$ and $T_4$. Task $T_3$ depends on $T_2$ and $T_4$ depends on both $T_1$ and $T_3$. Task $T_4$ will then be scheduled before $T_3$. If no processor is available to execute $T_3$, execution will hang. Therefore, a scheduling that relies on the beginning of the identifiers range can only be used if preemption is enabled.

It should noted that nested identifiers ranges (e.g. $T_3$’s, which is nested within $T_4$’s range) is only possible if different memory locations are involved. Namely, $T_4$ cannot depend on the value of $V$ generated by $T_1$ if $T_3$ depends on the value generated by $T_2$ for the same $V$. Such a nesting cannot exist in any possible sequential execution, hence violates the statically sequential
property. It is possible, however, that $T_4$ depends on $V$ generated by $T_1$ and $U$ generated by $T_3$, while $T_3$ depends on $W$ generated by $T_2$.

On a non-preemptive execution, the conservative strategy must be used: only considering the end of the identifiers range, namely each task’s own identifier. As task identifiers reflect the statically sequential order of the tasks, relying on them ensures no deadlocks.

### 3.4.3 Task identifiers

To make tasks *statically sequential*, task identifiers must reflect the sequential order of execution. Namely, tasks should have an ordering criteria that is not affected by the actual parallel execution. When tasks are spawned in-order, achieving such ordering is trivial, by using a global counter. However, in-order spawning is neither easy to enforce nor efficient to execute. It requires that a task spawns only after all previous tasks have completed spawning.

Timestamping tasks that are spawned out-of-order has not been a widely studied problem. The easiest solution is to use an unbound timestamp representation (stored in some dynamic data structure), which allows each task to have an unbound number of children. While certain optimizations may allow programs that use the scheme to complete without exceeding a bound amount of memory [BT95], a fixed amount of memory cannot be guaranteed in the general case. Alternatives that use bound representations have also been suggested [CMP97]. Those rely on intervals or nesting levels, using fixed or variable sizes. When no new timestamps can be generated, all active timestamps can be rescaled in order to make room for new ones. However, rescaling is an expensive procedure.

Previous works [SA12, RTL+05] have used an ordering scheme that relies on intervals: if a task that was given the interval $[0,X)$ spawns, it will break into the prologue, whose interval is set to $[0,s)$; the epilogue, whose interval is set to $[t,X)$; and the spawned task, whose interval is set to $[s,t)$. A task that has an interval that is larger than 2 can spawn, and spawning needs not be in order.

The ordering scheme we have implemented relies on nesting levels. Tasks are identified by 64-bit long integers. Each byte of the integer acts as a different sub-counter, representing a different nesting level. The nesting level resembles call stack depth, but is not limited to function calls – a loop iteration could reside in a deeper level than the surrounding loop code, and calling a recursive function could keep the same nesting level. The identifier of every task contains a prefix, which is the identifier of its parent task, and a sub-counter representing its own position among the parent’s child-tasks in that nesting level. This is a compiler-friendly scheme, which allows out-of-order spawning.

To illustrate the nesting levels ordering scheme, let us consider the code in Figure 3.3. This code shows 4 nesting levels:

1. `main`, the caller of `func`, which is the very first task.
2. The execution of `func`.
3. The execution of `func1`, called by `func`.


Listing 3.3 Nested spawning

```c
1: void func()
2: {
3:   for (int i = 0; i < 10; i++)
4:     func1(i); // spawn level 3
5: }
6: }
7: void func1(int i)
8: {
9:   for (int j = 0; j < 10; j++)
10:  func2(i, j); // spawn level 4
11: }
12: }
13: void func2(int i, int j) { ... }
14: }
15: int main()
16: {
17:   // nesting level 1
18:   func(); // spawn level 2
19: }
```

Figure 3.5: Task ordering is established by task identifiers, which are determined by and within the tasks’ nesting levels.

4. The execution of `func2`, called by `func1`.

The graph of tasks created by the code is presented in Figure 3.5. Let us assume that the identifier of the first task running this code is 1000, which means it is the first task in the first nesting level\(^1\). When the first task spawns a call to `func`, the spawned task is assigned the identifier 1100. When `func` spawns a call to `func1`, the spawning task ends, and a continuation task with the identifier 1200 begins. This is crucial to provide ordering between what happened before spawning `func1` and what happens after it. Consequently, every iteration in `func` is run in a different task, all on the same nesting level. Further, each task identifier is the prefix of all its descendants. The nesting scheme provides a full order throughout the execution of the program. Moreover, the order will be kept even if task `func1` with identifier 1110 spawns its calls to `func2` after, say, task `func1` with identifier 1220 has spawned its calls to `func2`.

Relying on nesting levels eases the compiler’s work in matching writers and readers. Let us consider a case in which a write done in `func2` (from the code example in Figure 3.3) should be

\(^1\)In the task identifiers, each pair of characters holds an 8 bit value, in hexadecimal representation. Our implementation used 64 bit variables that can manage 8 nesting level; here only 4 bytes are presented for brevity.
read by the following iteration of \texttt{func}. On a serial ordering scheme, the compiler must know the exact number of tasks between the write and the read. This might be impossible, as the number of tasks might be determined at run time. Using the nesting scheme, the compiler can easily determine the task of the writer relative to the reader: if the reader is done at \texttt{XYZ0}, the writer was done at \texttt{[X-1]000}.

The nesting scheme is limited in two ways. First, the number of nesting levels is limited. This is most significant on recursive code. The solution is to avoid creating new levels on every function call. Namely, consider every recursive call a following task rather than child task. The second limitation of the scheme is the restricted number of tasks in each level that can have the same parent (namely, prefix). If each level is represented by a byte, only $2^8$ tasks can have the same prefix. This is most significant on long loops. One possible solution is to turn long loops into two nested loops, each working in a different nesting level, allowing up to $2^{16}$ tasks to be created. Breaking a loop requires a bit more control logic, but the transformation is fairly straightforward. Alternatively, the loop can spawn only $2^8$ tasks, each serially running a few iterations. If the number of tasks is still much bigger than the number of cores, the potential lost of parallelism will be negligible. Another option is to reduce the number of levels (say from 8 to 4), allowing more identifiers per level. Lastly, if all solutions fail and no task identifier could be generated, the code must serialized for a while.

### 3.4.4 Spawning order

Given task identifiers represent the sequential program order and deadlock-free scheduling, MVM supports nested spawning. Namely, a task which is the work part (e.g., a loop body rather than loop header) can spawn another work task (e.g., a nested loop). However, correlating producers and consumers on different nesting levels might be difficult, and such dependencies should be avoided when possible.

Nested spawning also enables out-of-order spawning. As a result, a scheduler that picks the earliest task from the task queue might later encounter an earlier task in the queue. Consider a scenario in which one core runs $T_1$ and another core picking the earliest task $T_2$ from the task queue. If $T_1$ then spawns, either the work $T_{1,1}$ or continuation $T_{1,2}$ will be placed in the task queue. However, as long as the scheduler picks the earliest task currently in the queue, deadlock will be avoided.

In-order spawning makes task identifiers assignment trivial and eases deadlock avoidance. On the other hand, it forbids nested spawning. Consequently, the point of parallelization (most likely a loop) must be chosen carefully, otherwise the amount of parallelism might be restricted.

### 3.5 Handling irregular data structures

One of the unique features of the MVM model is its ability to parallelize operations on a major class of pointer-based data structures, e.g., trees. Such irregular data structures are notoriously hard to parallelize in conventional task-based models because (a) dependencies are unknown in
advance and are anonymous, preventing them from being explicitly specified, and (b) breaking operations on large data structures into small speculative tasks is disadvantageous, yet large tasks might read more than a speculative runtime can handle. In this section we describe how O-structures can be used to allow large, non-speculative tasks to concurrently operate on a shared data structure.

3.5.1 Unipath, Semi-unipath and Multipath code

Data dependencies determine the amount of parallelism in an algorithm. Data parallel algorithms such as matrix multiplication, in which processing of data elements is mostly independent, are easy to accelerate on parallel execution environments. But in many ubiquitous data structures — lists, trees, graphs, and even sparse matrix representations — data elements are linked and are thus dependent. Those data structures have an irregular memory layout, which allows efficient lookup, dynamic resizing, and sometimes rebalancing. But once the layout becomes part of the data, the operation of reaching a certain element depends on previous operations that have constructed the linked path to the said element. Irregular data structures are thus harder to parallelize.

One of the main strengths of O-structures is their support for parallelizing irregular data structures. We found it useful to classify algorithms and the data structures they manipulate into three categories: unipath, semi-unipath, and multipath.

**Unipath** algorithms are series of operations in which a task accesses data in the following way: (a) entering the data structures is done via a single entry point; (b) each datum is reached by traversing a single (per datum) sequence of pointers; and (c) a task never revisits a datum once it has been passed through. Unipath algorithms are most common on irregular data structures with a single entry point such as trees or lists. Insertions, removals and lookups all start from the entry point and work their way to the required element by traversing the single path to it. Parallelizing unipath algorithms with O-structures can therefore rely on pipelining: a task can lock and unlock the memory locations that contain pointers, and so long as tasks enter the root of the data structure in program order, they will traverse the data structure in program order. This is demonstrated in the example below.

**Multipath** algorithms include a traversal that can either reach a node from two different pointer chains or revisit a node after the first access. An example of the former traversal is a breadth-first search executed on a graph. If the search is parallelized, traversals cannot be synchronized, and the order in which they will reach nodes is arbitrary. O-structures do not facilitate parallelizing multipath algorithms.

**Semi-unipath** algorithms have a part that can be considered unipath and can be parallelized accordingly, but also a part that cannot be parallelized. The amount of parallelism thus depends on the ratio between the two parts. Turning a multipath algorithm into a semi-unipath one requires algorithmic modifications, and the result might not have the exact same semantics as the original algorithm. Still, when the new semantics are acceptable (in the example we describe soon, allowing a balanced tree to be slightly unbalanced for a short period), parallelization of
semi-unipath extends the applicability of O-structures.

It should be noted that the above classification relies mostly on the algorithm. While the data structure must provide a single entry point to allow unipath operations, the traversal dynamics determine if the unipath property holds. For instance, a singly linked list seems natural for unipath operations, but if the algorithm holds pointers to different nodes at the same time, it might revisit a node it had already passed. On the other hand, a forward-only traversal on a doubly linked list will be considered a unipath operation even though the data structure allows easy reversing.

3.5.2 Parallelizing using versioning, renaming and locking

Parallelizing unipath operations requires the complete set of O-structures features. The versioning of the target data structure depends on the role of the versioned location: the root’s version is updated with each task that goes through it (some relaxation described later). This is crucial to enforce the statically sequential order of operations, and possible since all tasks access the root. Internal nodes, on the other hand, cannot be uniformly versioned, as they are not always accessed (consider a tree, in which every task traverses a different path). Since the versions of internal nodes are not known, tasks load the latest version.

Enforcing true dependencies and eliminating false dependencies can only be achieved if a task can observe the state generated all earlier tasks (in static ordering). Further, a task must never be able to observe state generated by later tasks (in static ordering). Both requirements hold even if tasks are spawned out of order. Tasks are granted to enter the data structure in the right order (by head ordering), but must maintain this ordering to avoid bypassing earlier tasks and missing their mutations. Task thus utilize O-structures locks, blocking later tasks. Locking does not create new versions, matching the semantics of the traversal stage.

Read-only tasks such as lookups do not generate any state, hence need not block later tasks. However, they must also not observe mutations done by later tasks. This isolation is obtained using renaming (a.k.a. multi-versioning), which allows new state to coexist with old state.

3.5.3 Mutating unipath operations

Listing 3.4 and Figure 3.6 demonstrate how O-structures can be used to allow parallel execution of pointer-heavy tasks. The code uses manually inserted task spawning and O-structure access primitives for clarity; higher-level interfaces were described in Section 3.3. The code implements insertions to the end of a singly linked list; those insertions are unipath operations.

1. Tasks enter the insertion routine and LOCK-LOAD-VERSION the root node of the data structure (line 5; the specific version of the root is known because of the ordered way in which tasks enter the routine). This provides an initial ordering, which is maintained by the following steps.

2. Traversal over internal pointers is done using hand-over-hand locking [BS77] (lines 9, 11). Due to the ordered entry and the unipath properties, read-after-write dependencies
Listing 3.4 List insertion using hand-over-hand locking

```
void insert_end(node_t* root, node_t* newNode, task_t curTask)
{
    node_t* prev = root; // assume non-null
    // reserve a specific version of the head (stalls if does not exist)
    node_t* cur = LOCK-LOAD-VERSION(root->next, curTask);
    while (cur != nullptr)
    {
        // reserve latest version of next pointer
        node_t* next = LOCK-LOAD-LATEST(cur.next, curTask);
        // make prev. ptr available to next task
        UNLOCK-VERSION(prev.next, curTask);
        prev = cur;
        cur = next;
    }
    // create a new version for the new node
    STORE-VERSION(prev.next, newNode, curTask);
}
```

Listing 3.5 List lookup

```
node_t* lookup(node_t* root, int val, task_t curTask)
{
    cur = LOAD-VERSION(root->next, curTask);
    while (cur != nullptr && cur->val != val)
    {
        cur = LOAD-LATEST(cur.next, curTask);
    }
    return cur;
}
```

are thus enforced in internal nodes as well. Importantly, the latest versions of each pointer are accessed, since the specific versions are unknown.

3. The use of STORE-VERSION (line 16) eliminates write-after-read dependencies, namely inserts need not wait for preceding read-only tasks. Readers can thus avoid using locks during traversal. Omitting versioning and relying only on locking would require readers to lock as well, adding overhead and reducing parallelism.

4. While many versioned pointers might be locked during an insert, only a single version is created. Locking does not produce new data, hence need not create new versions.

3.5.4 Read-only unipath operations

Read-only operations need not block later tasks and thus require less synchronization. Using versioned loads ensures read-only operations are stalled by earlier mutating operations until the load is safe, yet do not stall later tasks. The load flavor matches that of mutating operations: a specific version of the root is loaded to ensure proper ordering, yet internal nodes are read by loading the latest version (up to the current task’s identifier).
When mutating and read-only operations are interleaved, loading the root’s value requires slightly more attention for two reasons: read-only operations need not stall other operations’ entrance into the data structure, and LOAD\-VERSION does not affect the version of the root. If the ratio between operations is known at compile time (say, 1 mutation per 1 read-only), when a mutator unlocks the root it can make it available to the next mutator, whose identifier is past that of the following read-only task. This will allow both the following reader and next mutator to enter at the same time, which is both safe and efficient.

When the mix between mutating and read-only operations is unknown at compile time, readers must participate in updating the root’s version. Namely, instead of merely loading the root, readers should lock and unlock it, effectively increasing the root’s version. While traversing the rest of the data structure, however, readers can still use the lighter LOAD\-LATEST.

3.5.5 Parallelizing Semi-unipath operations

A red-black tree is a balanced binary tree. As long as the tree is sufficiently balanced, operations are similar to unbalanced binary tree operations, which have the unipath property. However, occasionally the red-black tree must be rebalanced by rotating some branches. The balancing procedure starts at the bottom of the tree and makes its way toward the root, stopping when no further rebalancing is needed but possibly reaching the root. Since nodes that were passed through are revisited during rebalance, the unipath property does not hold. Further, it is not known in advance whether a rebalance will be needed and the height it will reach on the way up, hence the rebalanced branches cannot be isolated from above.

While mutating operations on red-black trees might violate the unipath property, separating the initial mutation from the rebalance and adding read-only operations to the mix unveils some degree of parallelism. Assuming the number of read-only operations between mutations in fixed and known, let us restructure sequential execution as follows:

1. Perform the initial mutation (insertion or deletion, without rebalance)
2. Perform following read-only operations

3. Perform the complete rebalance while locking the root

4. Perform the initial mutation of the next mutating task, and so on

Under the new order, all read-only operations can execute in parallel, immediately following the initial mutation. The rebalance will be considered a separate task, hence create a state with a version later than the identifiers of the readers. The rebalance can therefore be executed immediately after the initial mutation, without being stalled by the readers. Lastly, the next mutation will only start after the rebalance of the previous mutation completed (on most cases, very little work is done on a rebalance).

Since rebalancing traverses upwards and can possibly reach the root, the following mutating operation must be blocked until the rebalance completes. Consequently, parallelization of semi-unipath operations allows a single writer. Potential speedup is thus determined by the ratio between mutating and read-only operations.

Separating the rebalance from the initial mutation and allowing readers to execute in between provides parallelism but breaks the original semantics of the algorithm, which do not allow an unbalanced data structure to be observed. The technique might therefore not be suitable in all cases, and when parallelization is inappropriate, execution must be serialized. However, given readers can only observe the mutations of a single task, even if a rebalance is needed, the degree of imbalance will be very low, and the time complexity of the traversal will remain $O(\log n)$.

Parallelization of semi-unipath operations is more complex, requires the semantics of the original operations to be slightly modified, and offers lower scalability than the parallelization of unipath operations. However, the alternative is fully serialized execution, which does not scale at all. Lastly, some data structures are both balanced and unipath, e.g., skip list, relaxed red-black tree [HST15] and relaxed AVL [STK16].

### 3.6 Evaluation

In this section, we evaluate the MVM model’s ability to parallelize operations on common pointer-based data structures: linked list, binary tree, hash table and red-black tree, and on two non-unipath algorithms: matrix multiplication and Levenshtein distance. We have not yet completed our compiler implementation effort, so our evaluation is confined to these microbenchmarks. Nevertheless they reveal both the potential and limitations of O-structures.

#### 3.6.1 Methodology

We implemented an O-structure memory system in the gem5 [BBB+11] simulator executing in system emulation (SE) mode. The implementation microarchitecture is detailed in Chapter 4. The system configuration is presented in Table 3.1. The task scheduler was implemented in software and used a static assignment of tasks to cores. This policy imposes a minimal runtime overhead, but does not support load balancing.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>2-way in-order (ARM ISA), 2GHz</td>
</tr>
<tr>
<td>L1 I/D Cache</td>
<td>32KB, 8-way associative, 64B block, 4 cycles hit latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1.5MB \times #cores, shared, 16-way associative, 64B block, 35 cycles hit latency</td>
</tr>
<tr>
<td>Memory</td>
<td>64GB, 60ns latency</td>
</tr>
</tbody>
</table>

Table 3.1: The experimental platform

Our evaluation of the unipath and semi-unipath data structures (sorted linked list, binary tree, hash table, and red-black tree) interleaves lookup, insert and delete operations in different ratios (the data structures were pre-populated). The number of insertions and deletions was set to be equal, so the effective memory footprint does not change significantly during the run. The per-task amount of work was also held steady due to the stable size of the data structures.

The evaluation considered three aspects of the implementation:

1. **Memory footprint**: the initial number of elements was small (1000) or large (10000), yielding different effective memory footprints and accordingly different caching efficiency.

2. **Operations ratio**: read-only operations (lookups) are more common than mutating operations (inserts and deletes). We evaluated two read-write ratios — read-intensive (4 reads per write) and write-intensive (1 read per write).

3. **Versioned vs. unversioned**: self-speedup of versioned code indicates scalability.

Figures 3.8, 3.9, 3.10, 3.11 and 3.12 present the scalability (self-speedup) of the measured benchmarks given a varying number of cores. However, versioning adds overhead; the real benefit is obtained by comparing parallel versioned to sequential unversioned code. Figure 3.7 thus presents the maximal speedup of versioned vs. unversioned code running all sizes and ratios.

### 3.6.2 Speedup analysis

**Linked lists**

The linked lists benchmarks (Figure 3.8) demonstrate near-linear self-speedup. A major reason is the inherent poor locality of the algorithm and a memory footprint that exceeds the L1 cache. Since the Last-level cache (LLC) must be accessed often (over 50% L1 miss rate on small 1- and 32-core runs), coherence overhead is relatively smaller with O-structures. In addition, the list is long enough to accommodate a large number of tasks operating concurrently. Interestingly, on 32-core runs, slow-moving mutating tasks rarely stall (~15% of loads) due to ordering, while fast-moving read-only tasks are frequently stalled trying to bypass earlier mutators. Finally, even though unversioned runs do not have versioning overhead, for all settings the unversioned execution is under 1.6 \times faster on a single core. This effect is mostly due to high L1 miss rate.

**Takeaways:**
The inherently poor locality masks the overhead of cross-core communication.

For sufficiently large lists, true memory dependencies do not limit parallelism. Namely, O-structures enable almost arbitrarily high speedup for task-based execution.

**Binary trees**

Binary trees (Figure 3.9) have a larger memory footprint than a linked list but retain much better temporal locality. Scalability of small runs suffers from a reduction in L1 hit rate (94% on 1 core; 88% on 32 cores), also evident in the increase of versioned load latency ($\sim 4 \times$; such loads comprise $\sim 20\%$ of versioned memory accesses). Large runs show better scalability due to a much lower rate of ordering synchronization, which occurs mostly at the top of the tree. For example, the average rate of loads that stall due to ordering on a large, 32-core run is $\sim 25\%$, while $\sim 40\%$ stall on a small run. As opposed to ordering stalls, which imply a partial serialization of the execution, conventional capacity cache misses are handled in parallel, and do not effect scalability.

**Takeaways:**

- Given reasonable cache sizes, versioning overhead does not impact performance due to the increased locality in a tree compared to a list.

- Larger trees result in less synchronization stalls.

- For really large trees, the increasing rate of capacity misses reduces performance.
Figure 3.8: Scalability of linked list operations (comparing to a single-core versioned run). Workloads differ by size and operations mix (Read-Intensive is 3 reads per 1 write; Write-Intensive is 1 read per 1 write)

Hash tables

Hash tables (Figure 3.10) store buckets in an array, which has a regular memory layout. Each bucket holds a linked list, which is an irregular data structure, shown to be effectively parallelizable using O-structures (above). To obtain the unipath property, the pointer to the table is used as the ordering point, and each list is then synchronized independently. Properly used hash tables have a load factor (avg. nodes per bucket) of about 1, which leaves little room for pipeline-style parallelization. To keep our benchmarks realistic, a load factor of 1 was used.

Hash table benchmarks with a high ratio of mutating tasks do not scale well (Figure 3.7). This is expected, as every other task locks the head (i.e., table pointer), forming a bottleneck. The versioned load miss rate rises up to 85%. Yet even operations on empty buckets are non-trivial: inserting a node requires locking the bucket, allocating a node, setting the node’s value and next fields, and creating a new bucket version to point to that node. This produces 6 memory writes; adding unversioned operations such as hash calculation and memory allocation yields a sufficiently deep pipeline. That pipeline provides reasonable speedup.

Takeaways:

- Operations on data structures that intuitively seem inappropriate for pipelining may be long enough to scale well after all.

Red-black trees

Red-black trees (Figure 3.11) are semi-unipath data structures. Mutating tasks are serialized, and read-only tasks can follow the steps of the previous mutator, but only view the unbalanced state. Parallelism is therefore strictly bound to a single writer plus the readers that come between
Figure 3.9: Scalability of binary tree operations (comparing to a single-core versioned run). Workloads differ by size and operations mix (Read-Intensive is 3 reads per 1 write; Write-Intensive is 1 read per 1 write)

it and the next writer: 2 on a 1:1 ratio, 5 on a 4:1 ratio and so on. Moreover, our remove algorithm sometimes removes the node after effectively rebalancing the tree, allowing almost no readers-writer parallelism. Running only inserts and lookups increases the speedup by 3-10%, depending on the initial rate of remove operations. Read-oriented workloads yield a much better speedup – 6× for a 9:1 ratio and 8× for a 14:1 ratio.

**Takeaways:**

- O-structures can be used to parallelize and speed up balanced tree operations, but scaling is limited.
- Some operations that can be parallelized (such as remove) require manual modifications that would be challenging for a compiler to infer.

**Matrix multiplication and Levenshtein distance**

Regular algorithms are mostly data parallel algorithms and scale well. While the unipath pattern could not be applied, hand-parallelization was simple. Neither algorithm requires renaming, and both use O-structures as I-structures [ANP89]. It should be noted that a single core matrix multiplication runs 2.5× slower than the unversioned implementation due to the large number of O-structures and versioned operations.

**Takeaways:**

- Although their full set of features might add noticeable overhead and is not fully utilized, O-structures can be effectively used to parallelize ubiquitous non-unipath algorithms.
3.6.3 TLP and MLP

The evaluation on Chapter 2 has shown that O-structures provide memory-level parallelism (MLP) by eliminating false dependencies. Given sufficient MLP, multiple threads can execute non-dependent instructions in parallel, and thereby obtain thread-level parallelism (TLP). While the initial evaluation merely estimated potential parallelism considering memory dependencies, the current evaluation adds all other types of instructions into the mix (arithmetics, control flow and so on), as well as realistic memory latencies.

Our benchmarks obtain speedup mostly due to thread level parallelism. Although the workloads are memory intensive, the rate of versioned loads in the instruction stream is 2–8%. The unipath parallelization technique never locks an O-structure for a long period of time, leaving plenty of independent code to execute in parallel. Namely, eliminating memory dependencies does in fact yield thread-level parallelism.

3.7 Related Work

3.7.1 Non-speculative task-based programming models

A plethora of non-speculative task-based programming models and runtimes have been proposed. Most of those models belong to one of three broad classes: (1) those with no support for inter-task dependencies; (2) those with rigid implicit dependence relationships; and (3) those that support programmer annotations that explicitly express inter-task dependence relationships. OpenMP 3.0 [Ope08, ACD+09] and Grappa [NHM+15] are examples of runtimes in the first category, as they do not support inter-task dependencies in the runtime. Programmers must
Figure 3.11: Scalability of red-black tree operations (comparing to a single-core versioned run). Workloads differ by size and operations mix (Read-Intensive is 3 reads per 1 write; Write-Intensive is 1 read per 1 write)

explicitly use synchronization primitives (e.g. mutex, barrier) and be mindful of the limitations and scheduling behavior of the runtime. Cilk [BJK95, FLR98] is an example of a runtime in the second category; child tasks are forced to complete before the parent task. Finally, CellSs [BPBL06], StarSs [PBAL09], TaskSs [ECR10], OmpSs [FBM14] and OpenMP 4.0 [Ope13] are examples of runtimes in the third class; programmers annotate task inputs and outputs, and tasks are restricted to executing only when their inputs are ready. Less conventional models such as ADF [GSU12] combine partial dependence specification with transactional memory, handling unspecified shared state.

While most task-based programming models rely on simple data types (scalars and arrays) for input and output specifications, some opt for a richer interface based on streams. Gordon et al [GTA06] combined the StreamIt language [TKA02] with a task-based model in which streams provide communication channels between tasks; the streams support push, pop and peek operations. OpenStream [PC13] extended streams with sliding windows, broadcasts and more. Streams simplify communication of large amounts of data among cores and allow unnamed data to be shared. Still, streams themselves must be explicitly specified, and their expected use known at compile time.

3.7.2 Thread-level speculation (TLS)

Thread-level speculation is a technique in which multiple threads operate in parallel, and the mutations of each thread are committed to the shared state only if they are consistent with respect to the previous state. While the use of coarse-grained speculation resembles that of Transactional Memory [HM93], TLS targets sequential code. As opposed to TM, in which transaction boundaries are specified by the programmer, TLS synchronizes tasks, which are
TLS has yielded many implementations (e.g., [SBV95, OHL+97, AD98, KT99, SCZM00, CMT00, RTL+05, MGQS+08, SCZM05, JSY+15]). MultiScalar [SBV95], which arguably spearheaded research on TLS architectures, is a task-based parallel architecture designed to execute imperative language programs. MultiScalar relies on hardware prediction techniques [MBVS97] for aliases, and an innovative structure called Speculative Versioning Cache (SVC) [GVSS98] in order to scale past the limits of centralized store-queue designs.

TLS and MVM both target sequential programs and use versioning as part of the implementation, but the programming models and execution strategies differ significantly. First, TLS is not exposed to the programmer or the program. This makes TLS easier to adopt, but leaves no way for the programmer to express her knowledge of the program and the available parallelism. By leveraging programmer knowledge, task-based programming models using MVM can benefit from larger amounts of non-speculative parallelism. Second, relying on speculations allows TLS to be used with a wide range of code patterns, while the use of MVM relies on known dependencies or patterns such as unipath algorithms. However, TLS implementations are limited in the number of the data dependencies they can track, whether using dedicated structures (e.g., [SBV95, AD98, JSY+15]) or L1 (e.g., [GVSS98, KT99, SCZM00, CMT00, SCZM05, MGQS+08]) to store speculative state and track access to it. O-structures, on the other hand, can track versioning at memory-scale, eliminating task size limit.
3.7.3 Fine-grained pipelining

Pipelining has been suggested as a way to handle irregular algorithms and data structures by systems such as DOACROSS [Cyt86], DSWP [RVVA04], HELIX [CJH+12] and recently BDX [LA15]. Those works excel at parallelizing tight loops, intensively operating on the data structure (namely, each and every entry is not only accessed but processed). MVM allows a wide range of irregular data structures to be shared; pipelining merely ensures read after write dependencies. Moreover, tasks that become non-dependent (e.g., traversing different tree branches) are not ordered once they diverge.

3.7.4 Memory versioning

The concept of implicitly storing multiple versions of a memory element for synchronization and checkpointing is common in *transactional memory* [HM93] (TM), *thread-level speculation* [SBV95, HWO98, KT99, MGQS+08, JSY+15] (TLS) and some task-based systems (e.g., Swam [VTN11]). While they demonstrate the feasibility of extending memory with versions, none expose versioning to the program, and none provide the set of features used by MVM.

3.8 Conclusions

This chapter discusses the MVM task-based execution model. Building on prior work on memory versioning and renaming [GMOE14] this runtime efficiently parallelizes pointer-heavy data-structures *non-speculatively*. Using the proposed O-structure semantics we were able to achieve up to $19 \times$ performance improvement on a simulated 32-core system. This speedup is encouraging and suggests that, while writing parallel software remains a challenge for pointer-rich code, architecture support for novel memory concepts such as versioning and renaming can unlock performance hidden inside such software.
Chapter 4

Architectural support for unlimited memory versioning and renaming

4.1 Introduction

Data versioning and renaming has proven to be extremely useful in multiple scenarios. At the micro-architectural level, registers are renamed to eliminate false dependencies, and mechanisms such as ARB [FS96] and Task Superscalar [ECR+10] have achieved the same goal with memory addresses. Speculative execution mechanisms such as TLS and transactional memory use versioning and renaming to order concurrent operations and provide snapshot isolation. Similar needs have been identified and handled in software, whether frameworks (e.g., STM [RFF06]) or data structures (e.g., MVBT [BGO+96]).

While versioning and renaming are common techniques, they are mostly used in tailored solutions, optimized for specific needs. The efficiency of those solutions usually comes from limiting the use of resources, which restricts the number of either the versions per address, the addresses renamed, or both. Such limitations prohibit the reuse of already proposed mechanisms. Moreover, hardware mechanisms have not made versioning and renaming architecturally visible, excluding their use by software.

This chapter describes architectural support for memory versioning and renaming using O-structures [GMOE14] and proposes a micro-architectural implementation. Our implementation transparently manages the physical storage of both data and metadata (versions) and provides various access semantics, including versioned stores, loads and locks. Further, the number of memory locations that can be renamed and versions per location is unbound, limited only by the amount of RAM dedicated by the system at run time. Our O-structures implementation provides an efficient and fast solution to software requiring memory versioning and renaming.

The proposed implementation is integrated into the caching subsystem, without modifying the main memory. Adding versioning support to the caches allow recently used versions to be directly accessed. In case of a miss, internal version linking is used to perform a full lookup. While versioning adds overhead, our hardware implementation minimizes caching footprint by compressing metadata and selectively caching versions accessed during full lookups. Internal
versions linking is isolated from user code, keeping the security guarantees provided by virtual memory.

To demonstrate the full range of capabilities, we present 3 different use cases. We start by showing how versioning can order execution of regular operations such as matrix multiplication, where data must not be consumed before it is produced, yet only a single version of each datum is required. Then, we show how renaming eliminates false dependencies and provides snapshot isolation, yielding higher parallelism and throughput than a conventional read-write lock. Finally, we show the full extent of O-structure features by parallelizing irregular operations, which leverages fine-grained locking to maintain ordering even when dependencies are not known.

4.2 O-structures

4.2.1 Semantics and instruction set

As previously described, O-structure’s interface includes the following operations:

- LOAD-VERSION: Load a given version; stall if unavailable.
- LOAD-LATEST: Load the latest version smaller or equal to the given one; stall if unavailable.
- STORE-VERSION: Store the given value in a newly created version
- LOCK-LOAD-VERSION: Lock the specified version, then LOAD-VERSION that version.
- LOCK-LOAD-LATEST: Lock the latest version smaller or equal to the given one; then LOAD-VERSION that version.
- UNLOCK-VERSION: Unlock a given version,

Use cases for the operations above will be presented in the next sub-section and demonstrated in Section 4.5. Briefly, LOAD-VERSION and STORE-VERSION allow fine-grain synchronization where dependence are known at compile time, e.g. on matrix multiplication. LOAD-LATEST allows some uncertainty, e.g. when a part of a dynamic data structure is modified by a single writer, causing each part of the data structure to contain different versions. Lastly, locking allows multiple concurrent writers to be synchronized when operating on some dynamic data structures.

Albeit optimized, O-structures should be used selectively as they incur latency and memory footprint overheads. Accesses to non-shared memory locations such as the stack do not benefit from using O-structures. Moreover, memory dedicated to I/O devices need not be in O-structures, as the semantics of I/O usually involves adhering to strict program order. Conventional (unversioned) and O-structure (versioned) functionality should be implemented at the microarchitecture level. Conventional memory is accessed using traditional LOAD and STORE operations while versioned memory uses the added instructions described above.
4.2.2 Programming model

O-structures interface is intentionally not coupled to a particular programming model. However, O-structures do provide specialized capabilities, which are a better match for certain use cases. Table 4.1 presents a non-exhaustive list of possible programming models that can utilize O-structures: Functional programming can use O-structures as I-structures [ANP89] by reducing versioning to full/empty bits, or as M-structures [BNA91] by utilizing renaming as well. Software transactional memory and concurrent data structures can use renaming to isolate readers from writers, whether in a speculative or non-speculative execution model. Lastly, parallel task-based execution of sequential code can use versioning to enforce ordering of true dependencies, renaming to eliminate false dependencies, and locking to provide ordering when dependencies are not known at compile time; such a model was described in Chapter 3.

4.2.3 Practical considerations

O-structures interface and capabilities can be implemented purely as a software runtime abstraction; we’ve indeed started with a software prototype. However, the logic added to versioned memory operations incurred too much overhead, indicating hardware support is required. Software is still expected to provide the high-level programming model, hence the architecture provides a clean, multi-purpose interface. Further, while dedicated instructions must be wrapped in a higher-level abstraction for reasonable programmer use, such an abstraction can be provided by a dedicated compiler.

Any practical implementation of O-structures needs to manage the amount of memory used by the program, namely include a garbage collection mechanism. The unique semantics of O-structures disallows the use of conventional garbage collection techniques. A poor solution is to temporarily switch from parallel to sequential execution. Then, only the latest version of each memory location will be needed, and all other versions can be safely discarded. This stop-the-world-like approach for garbage collection is not uncommon but is far from ideal. Instead, we describe a better approach, based on concurrent garbage collector ideas, in the next section.

Table 4.1: Use-cases for versioning and renaming
Figure 4.1: O-structures are primarily implemented as cache extensions. Processors use dedicated instructions to access them, but the rest of a system remains largely unchanged.

4.3 Architectural support

The proposed execution model relies on new memory system abilities. Memory operations are inherently slower than computational operations, and a software-only solution might add an unbearable overhead. Architectural support minimizes the overhead of the versioned operations and reduces the instruction count. Our solution, therefore, focuses on the architecture level. Importantly, the architectural support does not replace the von-Neumann architecture but extends it. Unlike traditional dataflow systems, conventional (in our case, unversioned) code can still be executed, and the versioning support imposes minimal overhead on such a run.

To allow an unbounded number of versions, an O-structure is implemented as a linked list of versions. The head of the list is the original location, which is assumed to be large enough to hold a pointer (this requirement can easily be enforced at compile time). Versions are stored in dedicated blocks, which are allocated by a memory manager using buffers provided by the operating system. The list is sorted by version number, allowing safe reclamation of early versions once they are guaranteed not to be needed. This form of version storage provides three major benefits:

1. From the program’s point of view, versioned values occupy the same size as unversioned values, allowing easier transformation of code.

2. The number of memory locations that can be versioned is unbounded.

3. The number of versions per memory location is unbounded.

The last two benefits do not only allow more parallelism, but also ensure that the system will not have to cope with an overflow of versions. Eliminating such erroneous execution conditions simplifies multiple layers: the memory system needs not detect overflows; the runtime needs not implement measures for handling such cases, and the programming model need not expose system limitations and require program actions.
The O-structures microarchitecture is incorporated into a conventional cache hierarchy (Figure 4.1), and the processor is modified to dispatch the newly added instructions. Operating system involvement is also minimal, limited mainly to allocating memory regions that store versioned data. The remainder of this section discusses the following top level design considerations: storage, addressing, caching and garbage collection. For brevity, we describe a 32-bit system; extending to 64-bit is trivial.

The microarchitecture supports two forms of version lookup: direct and full. In general, direct access is obtained by storing versioned data in the L1 cache, and augmenting the cache so versions are also considered in the lookup. When direct access fails, a full lookup is performed by traversing the list of versions. This might include accessing other cache levels, other cores or the RAM; we later describe optimizations mitigating lookup overheads.

**Version Block**

The core data structure used for O-structure storage is the Version Block (Figure 4.2). A version block is a 16 byte structure with five fields: a version identifier (32 bits), a pointer to the physical address of the next version block (30 bits), a locked-by field (32 bits), a bit indicating it is the head version block in a version block list (described below) and the actual data itself (32 bits). The meta-data overhead for a single version of a program address is 12 bytes. We note that this design has overhead of $4 \times$ compared to an unversioned memory location. Below we describe how this overhead is reduced to $2 \times$ by storing version blocks in compressed form in the first level cache.

**Version Block List**

For a given address that contains an O-structure, each version of data is stored in its own version block. These blocks are linked together in a list, as depicted in Figure 4.3. The list is maintained in sorted order (newest in program order closer to the head), with version $v_g$ closer to the head of the list than version $v_l$ if $v_g > v_l$. The list is sorted in this way to simplify garbage collection of earlier versions (described later). Version Blocks are linked together into a list for a given program address. While the head of the list (namely, the pointer stored in the versioned location) stores the virtual address of the first version block, internal linking is done using physical addresses. A protection scheme preventing the exploitation of version blocks to gain access to arbitrary memory locations is described below.

**Free-list**

Unused version blocks are stored in a free-list that is managed mostly by the hardware; an operating system handler is invoked only when available memory is exhausted. When a new version is created, a version block is pulled from the free-list and used as that version’s storage. When the free-list is empty the hardware must notify the software, via a trap. version blocks are

---

1 Versioning larger datums could be more efficient if fully utilized, but wasteful otherwise. We thus limit the initial design to the size of a pointer, and leave the evaluation of other sizes to future work.
just ordinary memory structures, hence the runtime layer simply allocates more memory, carves it up into version blocks, and adds them to the free-list, modifying the page table as described below. Because of the physical pointers used to link together version blocks (even in the free list), extending the free-list must be a protected operation.

**Addressing and protection**

O-structures implementation must maintain the same level of protection provided by conventional virtual memory systems, namely limit a process to the memory resources allocated to it by the system. O-structures version block lists contain both virtual and physical addresses. Since version blocks are not allocated by conventional operating system mechanisms, special care must be taken to prevent one process from accessing version blocks of another process, as well as access to arbitrary physical memory locations.

Our implementation prevents exploiting the use of physical addresses by disallowing un-versioned memory instructions from accessing version blocks. This is achieved by marking pages used for version blocks with a special bit in the page table. Using conventional loads and stores to access version block pages generates a fault. Similarly, versioned instructions will also fault when trying to access a version block address that does not resides in a version block page. The last check is only enforced when a versioned instruction follows the virtual address of the first version block, stored in the program’s address space. That address can be modified by the program (maliciously or by accident), and translating it requires a page table access. Internal version block links use physical address, which cannot be guarded by the page table, but cannot be modified by the program.
To prevent one process from gaining access to legal version blocks owned by another process, version block pages are allocated per process, and memory management (including garbage collection described below) is done on a per-process basis. The virtual address of the first version block can therefore only lead to memory allocated to the current process, otherwise a conventional page fault is raised.

Version blocks are allocated from a physical memory buffer provided by the operating system. As the first version block of every O-structure is pointed using a virtual address, a physical-to-virtual address translation mechanism is needed. Our solution is reserving a consecutive virtual address range of the same size as the physical memory buffer. The start of both the physical and virtual ranges are stored in dedicated registers. In order to obtain the virtual address of an allocated version block, the hardware than simply adds the block’s offset from the beginning of the physical address buffer to the beginning of the virtual address range. The translation mechanism can be extended to support multiple buffers by using small tables rather than a single register. We do not anticipate a need for fine-grained memory management using single pages rather then large buffers will be required, but further investigation is beyond the scope of this work.

The last measure of protection is marking the first version block of every O-structure as a head. As versions are sorted, this version block is guaranteed to be live, namely not under any garbage collection stage. Consequently, the integrity of the version block’s information is assured. When a versioned instruction accesses the first version block, it verifies the head bit is indeed set; if not, a fault is generated. The purpose of this check is not to prevent a malicious or erroneous program from setting an O-structure pointer to a version block belonging to another process — this is handled by conventional virtual memory — but rather to prevent a version block in an unspecified state from being reached. For instance, version blocks are also used by the garbage collector to maintain internal lists. If a program could insert such a version block into an O-structure, the value stored at next field might have happened to lead to the physical address belonging to another process.

4.3.1 Caching

O-structures can be supported by a system with conventional cache system, yet it is far more efficient to implement much of the O-structure logic in the cache system itself. In particular, data indexing and tagging should be extended to make optimal use of cache space. In this section we describe these changes, as they comprise the bulk of the microarchitectural implementation of O-structures.

Version lookup

To find a specific version of data in a given O-structure, the microarchitecture first attempts a direct access, relying on versioning support added to the caches. If the wanted version is already cached, its value will be immediately returned, incurring a latency similar to unversioned access to cached data. In case of a cache miss, a full lookup is performed by walking the version block
Data compression

To reduce versioning space overhead, version block data is stored in compressed form in the cache. Version block compression relies on the observation that when more than one version of data exists for a given O-structure, the version numbers tend to be close (while this is not a requirement, versions are usually created in some order, rather than randomly generated). Close version numbers are found among lockers as well. For this reason we can compress eight version blocks into one 64 byte cache line as follows (Figure 4.4): we store an 18 bit version base, a 4 bit cache line offset and 8 compressed version block entries. Each compressed version block entry contains the data (32 bits), a 14 bit version offset, and a 14 bit locker offset. The cache line offset contains the offset within a 64 byte cache line for the head of the version block list, if cached. The version base contains the upper 18 bits of the lowest version stored within the cache-line. Version and locked-by are obtained by adding the version base field with the per-Version information. The only restriction thus imposed by the compression is on the range of versions and lockers within a single cache line.

Compressed version block information is stored in the cache with only a $2 \times$ overhead. But more importantly, compressed version block data is accessed in a single cache lookup. To perform a direct access, the cache accesses the line with the physical address of the head of the version block list. If that cache line is stored in compressed form, indicated by a bit in the meta-data of each cache line, the cache-line data is inspected to search for a match. If no match is found (or if the cache line is not stored in compressed form), then a direct access falls back to a full lookup. Caches can choose any appropriate (e.g. LRU) policy to manage versions within a compressed cache line.

Supporting the simultaneous mixing of direct and full lookups in the cache is straightforward. Caches that are at least two-way associative can store both compressed and uncompressed versions of an O-structure at the same time; direct mapped caches can use multiple hash functions to achieve a similar functionality. Compressed O-structures do require a slight change to the coherence protocol: messages for a cache line that contains a version block must also convey the physical address of the head of the version block list. Since modifications to a version

---

Figure 4.4: Memory layout of 8 compressed version block cache lines in a single cache line.

list. The list is sorted, providing a mechanism to terminate early if a given version has not been created yet.
block always begin by obtaining the physical address of the version block list, the address is always readily available prior to sending any coherence message. When caches receive a coherence message, the simplest course of action is to discard the compressed version block for that O-structure. Subsequent requests to access version data can rebuild the compressed version block. Sophisticated approaches that modify compressed version blocks in situ are left for future work.

Avoiding cache pollution

When a processor requests a specific version, the cache may end up traversing a fairly long version block list. To avoid cache pollution, only the block that holds the requested version is inserted into the cache; other blocks that were fetched during traversal are not cached. Since direct version accesses outnumber traversals, cold versions thus do not take the place of hot ones.

Adding a new version

Adding a new version of data to an O-structure requires that a version block be removed from the free-list. The version block list is traversed to locate the version block (or root pointer) that must be modified to point to this new version block. To avoid deadlock, the two cache lines that must be modified – the one containing the version block already in the list and the one containing the new version block that is to be inserted into the list – must be acquired for exclusive access in a well-defined order (e.g. lowest address first). Deciding where in the list to insert a new version block and acquiring exclusive access to the necessary cache lines might cause a race. Consequently, the version block already in the list must be re-checked to ensure that it has not been modified. If it has, the procedure must be retried. After the new version block is inserted into the list, exclusive access to both cache lines is no longer required.

Locking a version

Locking a version requires gaining exclusive access to the version block that contains that version and modifying the lock field appropriately. Once the lock field is modified, exclusive access is no longer required until the version is to be unlocked. As locked versions must not be read by any tasks other than the locker, an extended coherence protocol can minimize line movements and state changes, as described below.

Version storage example

Figure 4.5 depicts operations and structures described above. First, Figure 4.5a shows an initial state, in which the compressed L1 line holds only the head of a Version Block list, and the rest of the version blocks are stored in the uncompressed list on L2. Figure 4.5b shows the insertion of a version into the compressed line following a LOAD-VERSION. Finally, Figure 4.5c shows the memory state after the creation of a new version, which becomes the head of the list.
Figure 4.5: Access to and storage of O-structures in memory. O-structure accesses first go through two indirections in the page table, preventing user mode programs from fabricating O-structure addresses and permitting fine-grained interleaving of O-structure user addresses with conventional program locations. O-structures are stored in compressed form in the L1 cache and uncompressed form in the L2 cache.
4.3.2 Reducing communication across caches

The proposed extension imposes two types of overheads on the memory system: versioning support and cross-core communication. The former is described above; the latter comes from the parallel execution of sequential code. To mitigate those overheads, additional micro-architecture optimizations can be employed:

- **Avoid moving unreleased versions:** When one core holds a line in an exclusive or modified state and another core requests that line, the line will be moved, and its state will be changed. This conventional coherence protocol is wasteful when a core tries to acquire a version that is locked by another core: first, versioning semantics does not allow the requesting core to read the version’s content; and second, the owning core will need exclusive access to the line soon in order to unlock the version. When a task tries to access a locked version, it must stall regardless of the data in the version block. Namely, the transition from modified to shared state and moving the cache line can be avoided until the version is unlocked. Avoiding the transition also eliminates the need to regain exclusive access when the version should be unlocked. Since a compressed cache line contains multiple versions, minimizing transitions significantly reduces the effect of false sharing.

- **Avoid evacuating reserved versions:** Since locked versions are expected to be unlocked soon, they should be kept in the L1 cache as long as possible, and unlocked versions or unversioned lines sharing the same set should be evacuated earlier if room is needed for another version.

- **Pushing new and unlocked versions:** A conventional store is a stand-alone operation, providing no insights regarding the following operations. Versioned store and unlocks, on the other hand, imply that the current task is done modifying a version, and in particular the cache line holding it. Since the next access to that version can come from any core, performing a write-through upon a store or unlock can save some time on the following load or lock. Our experiments show minor improvement in a straightforward implementation. A more aggressive implementation, pro-actively pushing versions into other caches, could further reduce the latency of version loads. We leave this to future work.

4.4 Garbage collection

O-structures are most suitable for use in task-based execution models that parallelize sequential code\(^2\). When task identifiers reflect the sequential order of the program, mapping task identifiers to versions naturally orders versions and dependencies. This ordering allows unneeded versions to be identified and safely discarded. A version becomes unneeded once it is \textit{shadowed}

\(^2\)Adapting garbage collection to other models is possible but left for future work.
Figure 4.6: When task 3 creates version 3 of location $L$, it shadows version 1 from future tasks. Once task 2 ends, version 1 becomes reclaimable.

by a later version, and no active (or future) task will try to lock or load it. The following subsection describes a garbage collection mechanism, which allows automatic, efficient, on-the-fly management of version blocks. The garbage collector is implemented in hardware, and relies on progress reports provided by the program. Further, the run-time system must follow a set of simple rules to ensure safe garbage collection.

4.4.1 Program cooperation

The versioning instruction set is decoupled from the execution model. This makes the design more generic but hinders garbage collection. To provide the memory system with enough information for safe version reclamations, the garbage collector expects the runtime to fulfill a small set of rules. Those rules are described below in terms of parallel task-based execution or sequential code, but as said, can be adapted to other models:

1. Use the task identifier when accessing versions, whether for read or for write, ensuring that the ordering of versions matches the ordering of the sequential program.

2. Inform the memory system when a task begins and ends, allowing progress to be monitored. This is done using two dedicated new instruction, TASK-BEGIN and TASK-END.

3. Ensure that a task with an identifier lower than the identifier of the lowest active task in the system is not created. This merely places a lower bound on the identifiers of active and future tasks (out-of-order spawning is still permitted).

These restrictions are easy to satisfy and allow unneeded versions to be identified and reclaimed.

4.4.2 Version shadowing

The garbage collection process relies on the concept of version shadowing. Once a new version of some location is created, the previous (namely, earlier) version of that location becomes shadowed; and once no task may read the old version, it can be reclaimed. Figure 4.6 depicts the following scenario: task 1 creates version 1, which task 2 may read. Task 3 then creates version 3, which shadows version 1 for future tasks. Once task 2 ends, no task will access version 1, and it can be reclaimed.
In general, rule #3 ensures that at some point there will be no active task with an identifier smaller than the last possible reader of a shadowed version. From that point onward, rule #1 guarantees that no task will try to access that version, as all active tasks will access newer versions. From rule #2, the garbage collector can monitor the range of active tasks, and occasionally reclaim such versions.

4.4.3 Version sorting

A non-trivial consequence of this garbage collection scheme is that the version list of a certain location must be sorted. Namely, a new version is not always inserted at the front of the list, and a lookup for the appropriate location must be performed. Sorting is required to avoid a situation in which out-of-order version creation causes a shadowed version to be placed in front of a non-shadowed one (shadowing is determined by version identifiers, not the time they happen to be created). If that were possible, reclaiming and recycling the shadowed version would have broken that location’s version list, and make the live, unshadowed version inaccessible. Sorting complicates the design because existing version blocks might not be at the last level cache. To simplify the procedure, the linked list structure is maintained by the last level cache and not by lower caches. Since the lookup only considers the list structure and the version numbers, it can rely on possibly stale version blocks (holding, e.g., obsolete locking status). Lastly, the insertion modifies two blocks – the newly inserted one and the one pointing to it. A non-blocking lookup must therefore save some state during traversal, and the MSHRs be enhanced accordingly.

4.4.4 Operation

The garbage collector manages three lists:

1. The Shadowed list holds a list of shadowed version blocks. The blocks may still be accessed by the program, but will become eligible for reclamation at some future time.

2. The Pending list holds a list of shadowed version blocks, obtained from the shadowed list. Those blocks can be reclaimed at the end of the garbage collection phase.

3. The Free list holds version blocks ready for use to store new versions.

When a new version is created, if it shadows a previous version, then that shadowed version will be registered in the shadowed list. Namely, the shadowed list holds versions of various memory locations that might be inaccessible to the program. However, deleting shadowed versions is not safe at this point, as some running task might still access them.

A garbage collection phase is occasionally initiated by the system. Possible triggers are a low number of version blocks in the free list; high number of version blocks in the shadowed list; time passed since previous collection; program or operating system hint and so on. Our solution is hardware-based, and automatically invokes the collection when the number of free blocks drops below a watermark.

The garbage collection phase includes the following steps:
The lists managed by the garbage collector point to version blocks. The blocks can hold live versions when on the shadowed or pending lists, or available blocks when on the free list.

1. The shadowed list is moved to the pending list, and the latest active task is recorded. Since the list is dynamic, moving it is merely copying a pointer.

2. Monitor execution using \texttt{TASK-BEGIN} and \texttt{TASK-END} reports and wait until all active tasks have completed. Namely, the earliest active task is later than the recorded task.

3. Finalize the garbage collection phase by adding all version blocks in the pending list to the free list. This too is done by merely copying a single pointer.

Note that throughout the collection phase, newly shadowed versions can freely be added to the shadowed list, without interfering with the reclamation of the shadowed blocks that were moved to the pending list. Without an intermediate list, execution would have to be serialized and stopped, to ensure that all shadowed versions are no longer reachable by any task.

### 4.4.5 Garbage collector lists structure

The lists managed by the garbage collector are implemented as \textit{unrolled linked lists}, as depicted in Figure 4.7. Each list node is a block, which holds addresses of 3 versions blocks (assuming 32 bit pointers) and a pointer to the next node. The alignment of blocks is leveraged to store a counter at the last 4 bits of the next node pointer, indicating the number of blocks pointed by the node. The list nodes somewhat increase the memory footprint of the system, but since these nodes (as opposed to version blocks) are not placed in the lower caches, the overhead is minimal. Moreover, free list nodes are treated as regular allocatable version blocks. Once all version blocks pointed by a list node are allocated, the garbage collector will return the node itself upon the next allocation request. Consequently, list nodes impose a memory overhead only while at the shadowed or pending lists.

The garbage collector manages the version blocks but does not actually access them. Instead, it merely tracks their addresses. At run time, the shadowed list obtains blocks for its nodes from the free list, in the same way new versions are allocated. The first node of each list is stored in a dedicated register in order to minimize misses. The free list can use an additional register (or more, if needed) to prefetch the next node, and further reduce latencies.
4.4.6 Overhead

The major advantage of the garbage collection design is that it incurs very little overhead on the critical path. Shadowed versions are registered when a new version is created by storing their addresses in the shadowed list. The actual garbage collection is done by moving lists, which only involves touching their first (and sometimes last) blocks. Finalizing the garbage collection phase includes finding the earliest active task. This requires a tree of comparisons, but does not require blocking any other operation. The only noticeable overheads are a rare miss on the free list when versions are allocated quickly, and the need to sort the list of per-location versions.

4.4.7 Allocating and Freeing O-structures

The final challenge regarding memory management is the reuse of a versioned address, which holds a pointer to the version block list. An ordinary memory address is converted to an O-structure simply by ensuring that its value is null and starting to perform versioned accesses to it. Converting an address from an O-structure to a conventional memory location requires that no unfinished task access that location as an O-structure. Fortunately, the protection model ensures that violations of these rules will only cause errant behavior for the process, and cannot corrupt other processes or the operating system. Nevertheless, these restrictions do warrant caution for dynamic memory allocation. The simplest solution is for programs to delay the recycling of memory after it is freed until points of execution where no parallel tasks are executing. Another technique is to merge the garbage collection of version blocks with the runtime system’s malloc pool. This is a subject of future work.

4.5 Evaluation

In this section, we evaluate O-structures implementation and the benefits they provide. We begin with matrix multiplication and Levenshtein distance, which require ordering based on versioning, but do not require renaming and locking. We then evaluate a concurrent data structure, and show how versioning provides snapshot-isolation for readers. Finally, we evaluate a set of irregular data structures supporting concurrent readers and writers, and show how O-structure’s fine-grain locking allows deterministic results.

4.5.1 Methodology

We implemented an O-structure memory system in the gem5 [BBB+11] simulator executing in system emulation (SE) mode. The system configuration is detailed in Table 4.2. The task scheduler was implemented in software and used a static assignment of tasks to cores. This policy imposes a minimal runtime overhead, but neglects load imbalance.

The regular data structures evaluation included two workloads: multiplication of 3 dense 100*100 matrices \(^3\) and Levenshtein distance between strings of size 1000. The implementation

\(^3\)Due to the complexity of the algorithm, larger (and more realistic) workloads could not be simulated in
Table 4.2: The experimental platform

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>2-way in-order (ARM ISA), 2GHz</td>
</tr>
<tr>
<td>L1 I/D Cache</td>
<td>32KB, 8-way associative, 64B block, 4 cycles hit latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1.5MB × #cores, shared, 16-way associative, 64B block, 35 cycles hit latency</td>
</tr>
<tr>
<td>Memory</td>
<td>64GB, 60ns latency</td>
</tr>
</tbody>
</table>

**Figure 4.8: Speedup of parallel versioned (32 cores) over sequential unversioned.** Small benchmarks start with 1000 elements and large with 10000; Read-intensive (4R-1W) is 4 reads per write and Write-intensive (1R-1W) is 1 read per write.

is a direct translation of the sequential code, augmented with versioning to allow parallel execution.

The snapshot isolation evaluation compared an unversioned binary tree using a read-write lock and a similar tree using versioning and renaming. Workloads were read intensive: 3 reads per 1 write; initial tree size was 10000. Scan ranges of 1, 8 and 64 were evaluated.

The task-based execution evaluation considered three aspects of the implementation:

1. **Memory footprint:** the initial number of elements was small (1000) or large (10000), yielding different effective memory footprints and accordingly different caching efficiency.

2. **Operations ratio:** read-only operations (lookups) are more common than mutating operations (inserts and deletes). We evaluated two read-write ratios — read-intensive (4 reads per write) and write-intensive (1 read per write).

3. **versioned vs. unversioned:** self-speedup of versioned code indicates scalability, but __reasonable time.____
versioning adds overhead. The real benefit is obtained by comparing parallel versioned to sequential unversioned code.

4.5.2 Versioning only — regular data structures

**Matrix multiplication and Levenshtein distance** are mostly data parallel algorithms and scale very well. Neither algorithm requires renaming since each memory location is written to only once, and they both use O-structures as I-structures \[^{ANP89}\]. Figure 4.8 shows versioning can add a non-trivial overhead on a single-threaded execution, e.g., matrix multiplication runs 2.5× slower than a conventional unversioned implementation due to the very large number of O-structures and versioned operations. However, the overhead is fixed, and Figure 4.9 shows that a multi-threaded execution scales well (comparing parallel execution to a single-threaded versioned run).

**Takeaways:**

- Heavily using O-structures can incur a high initial overhead on certain workloads.
- Albeit the initial overhead, O-structures can be effectively used to parallelize regular algorithms.

4.5.3 Versioning and renaming — snapshot isolation

**Concurrent data structures** often guarantee snapshot isolation. This ensures that scans are serializable, i.e., their interaction with concurrent updates does not result in a set of results that would not exist if the updates had been executed sequentially. A common way to obtain such isolation is by versioning and renaming, effectively eliminating write-after-read dependencies.
Another common strategy is to separate reads and writes, eliminating synchronizations but also concurrency.

Figure 4.10 presents a comparison of O-structure-based binary tree with an unversioned binary tree, protected by a read-write lock. Snapshot isolation is obtained by renaming and separating writes and reads, respectively. Both execute the same series of inserts and scans, and 3 scan ranges are considered. The unversioned code runs faster on a single-threaded execution due to the versioning overhead. However, on a parallel execution, the unversioned implementation executes reads and writes separately, yielding an average self-speedup of $7.9 \times$. On the versioned implementation, inserts and scans can overlap, yielding a self-speedup of $12.2 \times$. The higher scalability allows the versioned implementation to outperform the unversioned one by 16% on average. While this is not a huge win, snapshot isolation is just one of the use cases for the proposed architectural facility.

### 4.5.4 Versioning, renaming and locking – task-based execution

The last group of benchmarks uses a parallel task-based execution of sequential code. Namely, sequential code is divided to tasks, which are executed in parallel. The benchmarks operate on widespread irregular data structures in which dependencies are not known in advance. Ordering is maintained using the unipath pipelining technique described in Chapter 3. The result is fine-grained pipelining of operations, requiring the minimal amount of read-after-write

---

4In terms of programming effort, using a read-write lock is similar to using O-structure versioning. More involved algorithms (e.g. lock-free) supporting snapshot isolation do exist, but are much harder to implement. Further, our experience with implementing O-structures in software shows they perform much worse than hardware-based O-structures; therefore, we do not use software renaming as a baseline.
synchronization. The output of such parallel execution is identical to a sequential execution.

Our evaluation of irregular data structures interleaves lookups, inserts and deletes in different ratios on pre-populated data structures. The number of insertions and deletions was set to be equal, so the effective memory footprint does not change significantly during the run. The per-task amount of work was also held steady due to the stable size of the data structures. The speedups comparing to sequential unversioned runs are presented in Figure 4.8, and scalability of versioned executions in Figure 4.9. The key findings from the evaluation are as follows:

**Inherently poor locality masks the overhead of cross-core communication.** On the linked list benchmark, all runs experience very low L1 hit rate (below 50%). Other data structures have a higher hit rate, but pointer-based data structures have inherently poor locality. However, as opposed to stalls due to ordering, capacity misses can be handled in parallel, and do not extend the critical path. Further, obtaining cache lines from the LLC or another core have comparable latencies, hence scaling out does not incur high communication penalty.

**Root ordering can form a bottleneck.** The dependency-safe traversal relies on ordering done at the root. This ordering forms a bottleneck, whose effect depends on the rate in which tasks enter the data structure. On write-intensive hash tables, up to 85% of versioned root loads are stalled. However, readers do not lock the root, hence read-oriented hash table workloads have a much lower stall rate. Binary trees have a low depth to size ratio, hence also experience stalls (40% on small trees, 25% on large ones).

**For sufficiently large traversals, true memory dependencies do not limit parallelism.** Linked lists experience very few collisions that force versioned loads to stall while the target version is locked, since traversals are long. Trees and hash tables provide shorter pipelines, but diverge fast once the root is passed. The use of fine-grained locking thus ensures that tasks accessing different locations will never synchronize. Interestingly, on 32-core linked list runs, mutating tasks moving slowly rarely stall (15% of versioned loads), yet read-only tasks moving fast are frequently stalled trying to bypass earlier mutators.

**Algorithmic optimizations are sometimes required to increase speedup.** The red-black tree benchmark is an attempt to handle balanced data structures, which are harder to parallelize due to the rebalancing procedure. Our implementation allows a single writer, and readers might see a slightly unbalanced tree. This severely limits parallelism, forcing the root to heavily throttle traversals. While task pipelining is limited in red-black trees, particular parts can be optimized. For instance, in our baseline the delete operation was locking a deleted pointer longer than necessary; algorithmic modifications shortened the locking.

### 4.5.5 Size and latency sensitivity

We complement our gem5 model by examining the performance impact of various L1 sizes and O-structure operation latencies. The analysis shows that increased memory footprint and higher cache latency have little effect on speedups.
Figure 4.11: The performance impact of changing the L1 cache size between 8kB–128kB compared to a 32kB baseline; unversioned (U), versioned single core (1T) and versioned 32 cores (32T) runs are presented. The figure demonstrates that increasing the L1 cache size beyond 32kB has limited impact — up to $1.23 \times$ and usually much less.

### L1 size

Even if the proposed compression scheme is fully utilized, O-structure storage does increase the memory footprint. To better understand the effect of L1 size, we ran all large, read-intensive benchmarks using a range of simulated L1 sizes. The results, presented in Figure 4.11, indicate that L1 size does not significantly affect unversioned, sequential versioned, or parallel versioned runs. For instance, let us examine the binary tree numbers: on a 8kB L1, the read miss rate of unversioned run is 22%. Increasing L1 capacity to 128kB lowers the miss rate to 9%, which translates to a speedup of $1.3 \times$. The sequential versioned run has slightly higher miss rates (25% and 11%) and a similar speedup. The miss rates observed when running on 32 cores are higher due to data sharing (28% and 21%), which reduces performance by 10%. This goes to show that while L1 size has a noticeable effect on sequential runs, the effect is much smaller than that of parallelizing the code; on parallel runs, the L1 size matters less.

Admittedly, data structures that fully fit in L1 might see little gain from parallelizing. However, they are usually not a performance concern anyway. Further, loads make about 25% of instructions on average; the rest will benefit from parallelization regardless of L1 size.

### L1 latency

Support for O-structures mostly relies on extending the L1 cache logic. While most of the versioning logic can execute in parallel with other operations (e.g., comparing versions and tags), determining the exact latencies of versioned operations requires a detailed RTL model. Instead, we estimate the influence of L1 latencies by injecting a fixed latency to every versioned operation. Figure 4.12 shows that adding 10 cycles to each versioned access reduces performance by up to 16%. The impact is much milder when using smaller (and more realistic) latencies. These findings correlate with the L1 size evaluation, which indicates a fairly high miss rate: frequently accessing the LLC reduces the effect of L1 latency.
Figure 4.12: The negative speedup (slowdown) caused by injecting 2–10 cycle latencies into versioned ops vs. no versioning overhead; versioned single core (1T) and 32 cores (32T) are shown.

Relative O-structures overhead

Irregular data structures generally make suboptimal use of some central micro-architectural mechanisms: the large memory footprint overflows lower caches; irregular layout forbids prefetching; and data-dependent conditional branches yield a high rate of misprediction [GDV15]. These issues are orthogonal to O-structures – any mechanism designed to mitigate them would also work with O-structures. For the results presented in this chapter, however, the inherently low IPC of irregular codes explains why the results are relatively insensitive to cache size and latency.

4.5.6 Garbage collection overhead

Our Memory Version Manager is designed to perform garbage collection in the background, adding very little latency to the execution’s critical path. To evaluate its overhead, we executed a sequential workload of 1000 operations on a sorted linked list with 10 elements. The small list size magnifies the effect of version allocation. A tight configuration that triggered 135 garbage collection phases was only 0.1% slower than a configuration with enough free versions to avoid ever executing the garbage collection process. Further, the latter configuration was also 0.1% slower than a configuration with no version sorting. While this test only considered versions that were created in-order (hence sorted), that is the common case in real programs.

4.6 Related Work

4.6.1 Dataflow memory

Dataflow machines have a unique set of memory challenges created by the lack of a total store load order. This has been addressed using a variety of creations. Arvind et al. introduced I-structures [ANP89], which are write-once memory locations. Barth et al. [BNA91] later extended I-structures for mutable computation with M-structures, which are memory locations with full/empty bits that can be written to (“filled”) and read from (“emptied”). I-structures and
M-structures, however, do not provide total ordering between an arbitrary number of producers and consumers. Specifically, since I-structures and M-structures do not provide versioning, they can only serve as rendezvous points between exactly one producer and its consumers. Consequently, the ordering of producers and consumers is lost when executing multiple instances of a task. Indeed, the Cray MTA/XMT series of machines [ACC+ 90] used an M-structure-like concept of full/empty bits [Smi81] for thread synchronization. This model enables programs to track trivial data dependencies in memory. Other studies extended this model to data versioning without renaming [SA12] or limited dependency tracking with data renaming using a coarse-grain memory buffer level, eliminating a subset of false dependencies [ECR+ 10].

Other dataflow machines such as WaveScalar [SMSO03] statically encoded the program control flow graph on the memory operations themselves so that total store order can be reconstructed from dataflow execution. In contrast, O-structures dynamically enforce a total store order, which facilitates out-of-order memory parallelism.

4.6.2 Implicit memory versioning

The concept of implicitly storing multiple versions of a memory element for synchronization and checkpointing is common in transactional memory (TM) and thread-level speculation (TLS). This chapter demonstrates that a hardware-based memory versioning can be scaled to efficiently support various programming models.

Transactional memory [HM93] implementations use versioning for two purposes: speculation and snapshot isolation. TM enables multiple threads to speculatively produce a new versions for a collection of data elements and, ultimately, commit at most one thread’s version. TM proposals use either the cache system to store speculative versions (e.g., [HM93, HWC+ 04, RHL05]) or the memory itself (e.g., [LCF+ 14, MBM+ 06]). Storing multiple non-speculative versions provides snapshot isolation, allowing read-only transactions to always complete without aborting. This has been used in both software [RFF06, PBLK11] and hardware [LCF+ 14] implementations.

Thread-level speculation is a technique in which multiple threads operate in parallel, and mutations of each thread are committed to the shared state only if consistent with respect to the previous state. While the use of coarse-grained speculation resembles that of TM, TLS targets sequential code. As opposed to TM, in which transaction boundaries are specified by the programmer, TLS synchronizes tasks, which are obtained from the program by the compiler or the hardware, and the sequential ordering drives commit order.

TLS has yielded many implementations (e.g., [SBV95, OHL+ 97, AD98, KT99, SCZM00, CMT00, RTL+ 05, MGQS+ 08, SCZM05, JSY+ 15]). MultiScalar [SBV95], which arguably spearheaded research on TLS architectures, is a task-based parallel architecture designed to execute imperative language programs. MultiScalar relies on hardware prediction techniques [MBVS97] for aliases, and an innovative structure called Speculative Versioning Cache (SVC) [GVSS98] to scale past the limits of centralized store-queue designs.

TLS and O-structures-based tasks both target sequential programs and use versioning
as part of the implementation, but both the programming model and execution strategy differ significantly. First, TLS is not exposed to the programmer or the program. This makes TLS easier to adopt, but leaves no way for the programmer or compiler to express their knowledge of the program and the available parallelism. By leveraging that knowledge, task-based programming models using O-structures can benefit from larger amounts of non-speculative parallelism. Second, relying on speculations allows TLS to be used with a wide range of code patterns, while the use of O-structures relies on known dependencies or patterns such as pipelining. However, while TLS implementations such as SVC are limited in the number of the data dependencies they can track, O-structures can track versioning at memory-scale, eliminating task size limit.

4.6.3 Loosely related mechanisms

The need to track fine-grain data dependencies has motivated research on hardware support for explicit parallel programming. One notable direction targets low-latency communication across threads and tasks. These systems include mechanisms such as scalar operand networks [TLAA03], direct register-level communication [CBK+14, SIT+14], or a shared register file [KMN+04]. Nevertheless, these studies primarily address register-level dependencies, whereas O-structures target memory-level dependencies at scale.

4.7 Conclusions

This chapter introduced architectural support for unlimited memory versioning, renaming and fine-grained locking, using O-structures. The extended memory interface can be leveraged by various programming models, frameworks and libraries, using a subset or all of O-structures features. To demonstrate the feasibility of the proposed mechanism, the chapter also describes a supporting instruction set architecture and a microarchitectural implementation that maintains desirable system properties such as process isolation and bounded memory consumption.

While O-structures can be implemented entirely in software, microarchitectural support is necessary to see significant performance gains compared to sequential execution. Fortunately, microarchitectural support is confined mostly to the cache system and a few additional instructions. Our simulation-based study of this microarchitecture suggests that significant speedup of up to $19 \times$ is achievable on pointer-heavy benchmarks, which utilize the complete set of O-structure features.
Chapter 5

Snapshot-based Synchronization: A Fast Replacement for Hand-over-Hand Locking

5.1 Introduction

Hand-over-hand locking\(^1\) is a fine-grained synchronization technique that prevent data races among concurrent operations. Commonly applied to pointer-based data structures, operations lock nodes as they traverse the data structure. In order to prevent bypassing, a node’s lock is released by the owning operation only after it acquires the next node’s lock. Generally, operations that traverse the same path are *pipelined*. As the pattern guarantees a node will not be concurrently accessed by two threads, data races are avoided.

The fine nature of hand-over-hand locking exposes more parallelism. Given each thread locks at most two nodes at once, multiple threads can operate on a data structure concurrently. Threads are ordered, namely one is forced to wait for another only when trying to access the same node. In a tree, ordering always applies to the root, as locks are associated with nodes. However, threads operating on different branches need not be ordered once their paths diverge.

The concept of hand-over-hand locking is appealing: fine-grained locking exposes large amounts of parallelism, and ordering provides thread safety. Ordering also makes hand-over-hand locking easy to apply to sequential data structures (that have properties discussed later), providing a quick way to parallelize existing sequential code. Indeed, the popular textbook *The Art of Multiprocessor Programming* [HS08] uses hand-over-hand locking to demonstrate fine-grained locking. However, na"ive hand-over-hand locking suffers from a few inherent limitations, causing it to be rarely used in the real world.

**Poor cache utilization**: Memory latencies are the most significant shortcoming of hand-over-hand locking. Acquiring and releasing per-node locks cause memory state modifications. As a thread makes its way to a certain node, it modifies the state of each node it passes. The

\(^1\)Also known as *lock coupling*, *chain locking*, *latch coupling*, *crabbing* etc.
modification is not performed on the data that the data structure is designed to hold (keys, values and pointers) but rather to the state of each node’s lock. Consequently, even read-only accesses still require changes to memory for each node accessed. In the memory system, writes to a node that are performed on one core invalidate any cached copies of that node on other cores. Accessing nodes that are not in the cache can be two orders of magnitude slower than accessing cached nodes. Given a large enough number of threads operating on the same data structure, the overhead incurred by poor cache utilization can exceed the potential benefits of parallelism.

**Entrance bottleneck:** Locking each node during traversal provides thread safety, but also turns the entrance to the data structure into a bottleneck. Consider operations on a tree: as every thread must go through the root, the root’s lock effectively serializes all accesses. While parallelism increases as threads diverge in the tree, the serialized entrance caps potential speedup on parallel execution. The effect of the bottleneck is determined by the number of threads and the depth of the tree, which yield a ratio between threads actively traversing the tree and threads stalled at the entrance.

**Extra locking:** As each node is associated with a different lock, moving from one node to the next requires both to be locked at the beginning of the transition. Albeit for a short while, the extra locking delays the divergence of threads that share an initial prefix of their paths. This initial prefix always includes the entrance of the data structure, which should be evacuated quickly.

### 5.1.1 Snapshot-based Synchronization

Snapshot-based synchronization is designed to address the shortcomings of basic hand-over-hand locking while maintaining the same ease of use. The fundamental insights driving snapshot-based synchronization are: (1) the number of locations that must be locked at any given moment is bound by the number of threads and not the number of nodes; and (2) as long as nodes are locked in the correct order, a thread cannot overtake (namely, race with) the thread in front of it, even if it somehow gets a delayed view of the first thread’s traversal.

Building on those insights, snapshot-based synchronization decouples locks from nodes and associates them with threads. Each lock is then dynamically assigned to a single memory location, which represents the location of the node currently accessed by the thread. At any given moment, the set of locked locations can be considered to be a snapshot of all threads’ locations. As depicted in Figure 5.1, a thread that obtains such a snapshot when entering the data structure can query it throughout the traversal; as long as a node it wishes to access is not in the snapshot, the thread can freely access that node. If the node’s location happens to exist in the snapshot, the current thread must wait until the thread at that location moves on.

Snapshot-based synchronization’s main component is therefore the snapshot, which marks the locations of all other threads when taken. As threads move on, the snapshot quickly becomes outdated. However, observing outdated location can merely cause unnecessary waits; necessary waits to threads traversing the same path will never be missed. Crucially, since threads that complete an operation can reenter the data structure, a snapshot cannot be used indefinitely, and
Figure 5.1: (a) Thread $T_3$ creates a snapshot when entering tree; (b) uses it to detect potential collision; and (c) moves on after ensuring $T_1$ is no longer at $N_2$.

a thread must obtain a fresh snapshot at the beginning of each operation.

To facilitate location-based synchronization, threads must report their whereabouts in a place that is visible to other threads. Reporting should take place often to reduce unnecessary stalls caused by false synchronization. However, the use of snapshots allows location reports to be seldom read – only when a snapshot indicates possible contention must a thread reload the locations of the others.

5.2 Snapshot-based Synchronization Design

In this section we describe the basic design of snapshot-based synchronization and its core components. While the basic design overcomes most of the limitations of hand-over-hand locking, some are rooted deep in the pipelining pattern. Optimizations that address those limitations are discussed on the next section.

Hand-over-hand locking pipelines threads that traverse the same path. In other words, a thread can access a node that was locked by the thread in front of it only once the leading thread moved on and unlocked the node. Bypassing within such a pipeline is impossible, so data races are avoided. Threads whose paths diverge are no longer synchronized, consequently hand-
over-hand locking is only applicable to data structures that have no cycles (and algorithms that introduce no such cycles by, say, revisiting a node during a rebalancing phase). Snapshot-based synchronization is designed as a substitute for hand-over-hand locking, and its correctness is guaranteed only when the latter is safe.

The central component of snapshot-based synchronization is the snapshot. As depicted in Figure 5.1, when a thread enters the data structure, it records the location of all other threads. Before the thread moves to another location, it checks if the snapshot recorded any other thread at that location. If so, it must not access the location until it verifies the other thread has moved. This verification is done by obtaining the latest location of the other thread (and possibly additional ones, as discussed later). Consequently, each thread must report its current location once it moves.

Snapshot-based synchronization manages two kinds of data: private (per thread) and public (shared). Snapshot-based synchronization reduces cross-thread communication by serving most reads from private data, falling back to reading public data only when encountering possible contention. Each thread stores the snapshot in private memory. The current location of each thread, on the other hand, is stored publicly and is available to all other threads. However, public data is read only when a snapshot must be created or updated.

Snapshot-based synchronization leverages modern hardware features to reduce overheads: loads from local caches are much faster than loads from main memory, and stores do not stall subsequent operations. Table 5.1 sums the access patterns to public and private data: the snapshot is read often. Being private, it can be efficiently cached, minimizing access time. Threads frequently report their location to public storage. Due to micro-architecture features such as out-of-order execution and a store buffer, location reports do not stall the following instructions even if they incur a cache miss.

### 5.2.1 Interface and algorithms

Snapshot-based synchronization’s interface is similar to hand-over-hand locking’s, and converting code using the latter to the former is straightforward. However, the underlying operations differ significantly, and the interface naming represents the actual semantics. Briefly, when using snapshot-based synchronization, operations must start with a call to moveToHead. Before accessing a location, waitForLoc must be called to make sure no other thread is present at that location. Lastly, moveToLoc is used to publish the new location of the thread, preventing others from accessing it. Table 5.2 compares the two interfaces, and Listing 5.1 lists codes inserting

<table>
<thead>
<tr>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snapshot (private)</td>
<td>Often - before accessing node</td>
</tr>
<tr>
<td>Location (public)</td>
<td>Rarely - when creating snapshot</td>
</tr>
</tbody>
</table>

Table 5.1: Frequency of accessing synchronization data. Common operations are reads from private data and writes to public data.
### Table 5.2: API for hand-over-hand vs. snapshot-based synchronization

<table>
<thead>
<tr>
<th>Operation</th>
<th>Hand-over-hand</th>
<th>Snapshot-based synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock head</td>
<td>head-&gt;lock()</td>
<td>moveToHead(head)</td>
</tr>
<tr>
<td>Before accessing node</td>
<td>node-&gt;lock()</td>
<td>waitForLoc(node)</td>
</tr>
<tr>
<td>After access granted</td>
<td>prev-&gt;unlock()</td>
<td>moveToLoc(node)</td>
</tr>
</tbody>
</table>

### Listing 5.1 Snapshot-based Synchronization vs. Hand-over-Hand. Main differences are explicit locking of the head and locking granularity (pointer vs. node). Code simplified for brevity.

```c
1: void insert(Node* head, int key)
2: {
3:   Node** last = &head;
4:   moveToHead(&head);
5:   Node* cur = head;
6:   while (cur != nullptr)
7:   {
8:     if (cur->key == key)
9:       moveToLoc(nullptr);
10:      return;
11:   }
12:   if (cur->key > key)
13:     last = &cur->left;
14:   else
15:     last = &cur->right;
16:   waitForLoc(last);
17:   moveToLoc(last);
18:   cur = *last;
19:   *last = new Node{key};
20:   moveToLoc(nullptr);
21:   cur->unlock();
22: }
```

into a binary tree using both mechanisms.

**moveToHead**

Since most synchronization is done using the private snapshot, it is crucial that the snapshot is sufficiently up-to-date. In particular, a snapshot must include the location of each thread that entered the data structure before the current thread and has not completed its operation yet. Using a snapshot that does not include all threads ahead might yield a race.

The pipelining pattern must be maintained by snapshot creation as well. A snapshot is used to ensure a thread does not bypass (race with) threads in front of it. Given all threads enter the data structure via a single entry point, a snapshot must be created right before attempting to enter and must record all threads ahead.

Snapshots need not include threads that are behind in the pipeline. It is up to those threads behind to make sure they stay behind. However, snapshotting threads that are behind is harmless: given no cycles, a thread ahead will never try to move to a location held by a thread behind. In practice, the overhead of snapshotting threads all threads (including those behind) is most likely to be lower than the overhead of determining whether each of the other threads is ahead or behind, and snapshotting only the ones ahead.

79
The moveToHead operation consists of the following steps:

1. Establish ordering among threads competing at the entrance
2. Once the leading thread allows, create a snapshot by gathering the locations of all threads ahead
3. Wait for the entrance to become available
4. Move to the entrance and update current location
5. Allow following thread to create a snapshot

Two threads must not create a snapshot at the same time. Doing so will cause both to miss each other, and since one will eventually enter ahead of the other, the missing location will cause a race.

A significant part of entering the data structure requires serialization. Measures must therefore be taken to mitigate the bottleneck. Those measures are detailed in Section 5.3. moveToHead has no equivalent operation in hand-over-hand. Instead, in hand-over-hand the order in which threads lock the root of the data structure determines the order in which they will lock (and access) all other nodes, until their paths diverge.

**waitForLoc**

Before a thread can access a location, it must make sure no other thread will concurrently modify that location. To do so, the thread must:

1. Check if the snapshot contains any other thread at that location
2. If no thread was observed at that location, waitForLoc can safely return
3. Else, the current thread must wait until the thread ahead moves
4. Update its snapshot

The minimal update of the snapshot depends on the modifications done by the data structure algorithms. If the layout of the data structure is not modified (e.g., a value is updated in a binary search tree), only the last location of the colliding thread ahead must be updated in the snapshot. However, since synchronization relies on snapshotted locations within the data structure, operations that modify the memory layout of the data structure require a complete recreation of the snapshot. Figure 5.2 demonstrates a series of actions in which reloading only the location of a colliding thread eventually causes a race:

1. A thread $T_3$ snapshots two threads that are ahead, $T_1$ and $T_2$. $T_1$’s location is $N_3$.
2. $T_1$ moves on, $T_2$ removes node $N_3$ and exits.
3. $T_3$ proceeds and detects a collision with $T_2$, but only updates $T_2$’s location.
Figure 5.2: If only one thread’s location is reloaded upon collision, a snapshot might contain deleted locations, allowing a race. Solution: recreate complete snapshot.

4. $T_3$’s snapshot now contains $T_1$ at $N_3$, but since $T_2$ removed $N_3$, $T_3$ will never try to visit that node. Consequently, $T_3$ will never detect a collision with $T_1$. If they continue in the same path, they will race.

The simplest countermeasure against such race conditions is to fully recreate the snapshot on each potential collision. Creating a snapshot involves communication with all threads and is therefore expensive. Optimizations are possible but complicate the implementation; the actual benefit depends on the rate of collisions. Lastly, the recreated snapshot might show a new potential collision. However, since pipelining allows a single thread per node, only one new potential collision might be observed. Once that collision is resolved, the thread can safely proceed.

**moveToLoc**

Moving to the next location is simple: a thread just updates its publicly visible location. This move is equivalent to locking the next node and unlocking the previous one in hand-over-hand. The overhead, however, is noticeably lower: the state of involved nodes is not changed, and only one location is locked at any given moment. Hand-over-hand’s excessive locking is due to
the lack of support for a single atomic modification of multiple memory locations in current hardware\textsuperscript{2}, which does not allow two locks to be modified at once.

5.2.2 Locking granularity

Hand-over-hand relies on locks, and must therefore bind a lock to every object it wishes to protect. The most natural locking granularity is one lock per node\textsuperscript{3}. Locking a node prevents all its fields from being accessed by other threads. Consider a tree in which node $N_1$ points to $N_2$ and $N_3$, depicted in Figure 5.3a. Thread $T_1$ locks $N_1$, and is now considering whether it needs to delete $N_2$ (which will also involve modifying the pointer on $N_1$). Thread $T_2$ is heading towards $N_3$, but must pass through $N_1$. While neither $N_2$ nor the pointer to $N_2$ will be accessed by $T_2$, per-node locking will force $T_2$ to wait until $T_1$ unlocks $N_1$.

Figure 5.3: Locking nodes vs. locking pointers. The latter allows more parallelism.

Snapshot-based synchronization does not use lock objects, and instead (semantically) locks memory locations. Consequently, locking can be done at any desired granularity. The one we had found most useful is per pointer. Consider the previous example; as depicted in Figure 5.3b, on a per-pointer synchronization scheme, $T_1$ would have locked the pointer to $N_2$. $T_2$ could have then check $N_1$’s key, determine it needs to go to $N_3$, and freely move on without being stalled by $T_1$. On lower parts of the tree, threads usually diverge and locking granularity has little effect. However, contention is a major problem at the top of the tree, and locking pointers eliminates unneeded synchronizations.

5.3 Optimized Implementation

The basic snapshot-based synchronization scheme eliminates hand-over-hand’s poor cache utilization and excessive locking overheads. However, the root of the data structure remains a bottleneck. Creating a snapshot involves reading the current locations of all threads. Since the locations are constantly being updated by the reporting threads, creating a snapshot incurs

\textsuperscript{2}Hardware transactional memory does allow multiple modifications to happen effectively atomically, but is not ubiquitous. We discuss software TM in Section 5.4.

\textsuperscript{3}A lock array can service any number of nodes using some hash function but might cause deadlocks, and in our experiments, not faster than storing locks as node fields.
multiple cache misses. Given snapshots cannot be created in parallel, taking a snapshot before entering the data structure serializes execution for a large portion of the run. In this section, we discuss major optimizations that improve snapshot-based synchronization’s efficiency and mitigate the entrance bottleneck.

5.3.1 Using SIMD capabilities

Many modern processors support SIMD (Single Instruction Multiple Data) operations, allowing a single instruction to operate on a vector of values in parallel. Those instructions can be used to accelerate snapshot operations:

- **Snapshot creation** can use dedicated *gather* instructions to fetch locations from all threads using a single operation, without looping.

- **Snapshot lookup** can be shortened by comparing the desired location with the snapshotted locations of all other threads using a single operation, without looping.

Our implementation uses Intel’s AVX2 instructions, which support vectors of up to 256 bits. Storing complete 64 bit addresses would allow only 4 locations to be stored in each vector. To minimize the number of operations, we store only the lowest 16 bits of every address, allowing each vector to hold up to 16 locations\(^4\). Storing only a part of the address might yield false positive, but resolving a collision involves examining full addresses, hence deadlocks are impossible. Lastly, the 256-bit vectors hold the complete snapshot, comfortably fitting into a 64-byte cache line and enabling the following optimization.

5.3.2 Copying snapshots

Creating a snapshot involves accessing data constantly updated, incurring multiple cache misses. To avoid creating a snapshot from scratch, a thread can copy the snapshot used by the immediate leading thread. If the complete snapshot resides on a single cache line, copying incurs a single cache miss.

Snapshots can only be copied from the thread that entered immediately before the thread that needs the snapshot. Consider threads \(T_1, T_2\) and \(T_3\) entering a data structure, in this order. \(T_1\)’s snapshot is created first and thus does not include \(T_2\)’s location. If \(T_3\) copies from \(T_1\), it might race with \(T_2\). On the other hand, if \(T_3\) copies \(T_2\)’s snapshot it might obtain a somewhat stale view of \(T_1\)’s location. However, the worst outcome would be the detection of false collisions, which will lead to an unneeded stall but not a race.

Care must be taken to avoid using snapshots after re-entrance into the data structure. Consider a twist in the previous scenario: \(T_2\) completes its operation, enters the data structure again and tries to copy \(T_1\)’s snapshot before \(T_3\) gets to copy \(T_2\)’s. As a result, neither will have a valid snapshot. This is a variant of the ABA problem, which we solve using the conventional tool

\(^4\)AVX2’s Gather instruction can load up to 8 32-bit values. We therefore store two 16-bit locations next to each other, and load them as a 32-bit datum.
Figure 5.4: Local gates order threads coming from the same NUMA node, creating chains. The global gate orders the entrance of chains into the data structure. While trailing, prev.’s position is examined directly without using a snapshot.

– timestamps. When a thread moves to the head, it gets the timestamp of the previous thread. After copying a snapshot from the thread ahead, the current timestamp of the thread ahead is checked. If the latest timestamp indicates the thread ahead has reentered the data structure, the consistency of the snapshot is not guaranteed. Once a thread detects it copied a possibly invalid snapshot, it simply falls back to creating a new one from scratch.

Lastly, overusing a snapshot provides no gain if it causes an immediate false collision detection whose resolution requires the snapshot to be recreated. To avoid this issue, a reuse counter can be associated with the snapshot. Once above some threshold, a new snapshot will be created instead of copying the outdated one.

5.3.3 Deferring snapshot creation by trailing

A thread that immediately follows a previous thread does not need a snapshot; we call this state trailing. Due to the nature of pipelining, no thread can appear between two consecutive threads. As illustrated in Figure 5.4, while $T_2$ trails $T_1$, it can rely on $T_1$ to resolve any collision with threads in front of them, allowing $T_2$ to merely ensure it does not bypass $T_1$. $T_2$ can thus defer obtaining a snapshot until trailing breaks. Trailing thus eliminates the need to create a snapshot before entering, significantly shortening the bottleneck. Further, trailing eliminates most contention points involving more than two threads, akin to MCS locks [MCS91].

While $T_2$ trails $T_1$, $T_2$ examines the location of $T_1$ instead of checking the snapshot. As long as $T_1$ is still at the location $T_2$ wishes to move to, $T_2$ will spin; once $T_1$ moves, $T_2$ can immediately follow. While this cross-thread communication is more expensive than checking a private snapshot, it is cheaper than creating one. In the heavily-contended entrance, quickly evacuating the entrance reduces stalls. Trailing stops as soon as $T_2$ cannot be sure $T_1$ passed through the memory location it tries to access, whether because $T_1$ moved too fast to the next location or because $T_1$ turned another way. Once trailing stops, $T_2$ cannot rely on $T_1$ and must create (or copy) a snapshot before moving on.

5.3.4 NUMA awareness

On NUMA systems, accessing remote memory (associated with another NUMA node) is significantly slower than accessing local memory. Keeping as much cross-thread communication
within the same NUMA node can therefore reduce memory latencies. While snapshot-based synchronization is agnostic to the memory management of the hosting data structure, adding NUMA-awareness to the synchronization mechanism reduces its overhead.

Snapshot-based synchronization employs a technique that groups threads of the same NUMA node, orders them internally, and lets them enter the data structure in this exact order. The mechanism, depicted in Figure 5.4, resembles the one used in cohort locks [DMS12b]: a per NUMA node gate is first used to create chains of threads belonging to that NUMA node. The head of each chain (namely the first thread) competes over the global gate only with other heads. Once acquired, the head closes its following chain and announces the last thread in the chain via the global gate. The head of the next chain (probably coming from another NUMA node) will trail the last thread in the chain in front of it.

The skeleton of the implementation is presented in Listing 5.2. The code is simplified for brevity. The first part that was left out is the structure used by each thread to store private state such the identifier of the thread ahead, as well as information that should be available to the following thread such as the current thread’s identifier. Since the structure of one thread is read by the following thread, corruption due to re-entrance must be avoided. This is achieved by attaching counters to a few fields. For instance, the counter attached to the thread identifier on the gates is also stored alongside the thread identifier field available to the following thread. That counter is then used to detect whether the thread ahead is still waiting to move to the head (in which case the current thread must wait) or has it re-entered after completing its previous operation (in which case the current thread can move on).

Threads within the same chain all run on the same NUMA node. Trailing and snapshot copying among those threads are noticeably faster than across NUMA nodes. The ratio between local and remote communication is determined by the length of the chains. Interestingly, if entering the data structure becomes slow (e.g., due to some external interference) and threads accumulate at the entrance, longer chains will be created. This in turn will provide more local communication, allow threads to leave the head quicker and reduce entrance time.

5.3.5 Reader synchronization

Read-only operations such as lookups are usually easier to parallelize, as they need not synchronize with other readers (synchronization with write operations is required, of course). In a hand-over-hand algorithm, readers can thus safely bypass each other. This freedom could be of great use when threads enter the data structure. Unfortunately, the straightforward readers optimization breaks other optimizations. For instance, if writer $W_1$ trails reader $R_1$, and $R_1$ bypasses $R_2$, then $W_1$ will race with $R_2$. Similarly, writers cannot copy snapshots from readers as they might include stale locations of other readers. Our implementation includes a restricted set of reader optimizations. We do not elaborate on them due to lack of space and leave further reader optimizations for future work.
5.3.6 Putting it all together

The optimized snapshot-based synchronization overcomes inherent limitations of hand-over-hand:

- **Poor cache locality** is minimized by decoupling synchronization state from the data structure and using a snapshot to further reduce cross-thread communication.

- **The entrance bottleneck** is mitigated by using NUMA-aware algorithms, deferring snapshot creation and reusing snapshots.

- **Extra locking** is avoided by allowing an atomic move from one location to another and by locking pointers rather than nodes.

- **Reader synchronization** is reduced by allowing readers to bypass each other.

As demonstrated on the following section, snapshot-based synchronization is indeed faster than hand-over-hand locking.

5.4 Evaluation

In this section we compare the actual performance of snapshot-based synchronization (SBS) to alternative synchronization mechanisms, revealing both strengths and weaknesses. The alternative mechanisms are:

1. Traditional hand-over-hand (HOH)
2. Software transactional memory (STM)

Like SBS, STM is a synchronization mechanism external to the data structure, which can be used to parallelize sequential data structures. State-of-the-art concurrent data structures can be much faster, but synchronization is deeply integrated in the structures and associated algorithms. We therefore do not consider them comparable.

5.4.1 Experimental setup

We perform a series of micro-benchmarks, running a mix of operations on binary trees. We consider both integers (INT) and strings (STR) as key types – while the former is more common in the literature, the latter is very common in real programs and sometimes exhibits a different behavior. All evaluations execute a similar number of inserts and deletes and keep the data structure size stable; we also study the effect of the initial size. Lastly, read-write ratio on all benchmarks is 50-50. We do not analyze other ratios due to space limitations; in short, our evaluation finds snapshot-based synchronization favors write-heavy workloads.

The server used has 2 NUMA nodes and Intel Xeon E5-2630 processors running at 2.4Ghz. Hyperthreading, Turbo Boost and adjacent cache line prefetching were disabled. Each core has 32KB L1 and 236KB L2 caches; each processor has a 20MB L3 cache; and the system has
Figure 5.5: Scalability using 2 NUMA nodes (init: $10^6$). SBS in par with STM on the integer benchmark but much better than all other runs.

62GB of RAM. Code was written in C++ and compiled with GCC 7.2, which also provided the STM support.

5.4.2 Scalability

Figure 5.5 presents the throughput of all workloads running on a varying number of threads, evenly distributed between the 2 NUMA nodes. Evidently, HOH does not scale past 10 threads, and synchronization overhead overwhelms performance as the number of threads increases. On the integer workloads, SBS is slightly slower than STM. However, while STM’s scalability is consistent, SBS reaches its peak at 14 threads. The string workloads demonstrate different trends, as more work is performed during traversal (mostly string comparisons, involving multiple memory accesses in a loop). Extending traversals reduces contention at the entrance, allowing SBS to continue scaling past 16 threads. STM, however, suffers from enlarged read and write sets, causing throughputs to drop.

Figure 5.6 presents scalability when running on a single NUMA node. The results emphasize the effect of NUMA: as cross-core communication is much faster when running on the same NUMA node, HOH and SBS scale much better. Most of the gain comes from entering the tree faster due to reduced cross-thread communication latencies. STM, which does not require communication at the entrance, sees little gain in this configuration.

**Summary:** HOH and SBS are more NUMA sensitive than STM. SBS performs best on most scenarios, but in the combination of short traversal time (integer tree) and long cross-thread
5.4.3 Effect of data structure size

The size of the data structure affects the duration of the traversal. As indicated by the difference between INT and STR workloads, traversal time correlates to entrance contention, which in turn determines scalability. Figure 5.7 presents the throughput of the 6 benchmarks when running on trees of different sizes - \(10^5\), \(10^6\), \(10^7\) and \(10^8\); all using 16 threads. Accessing more memory locations as the tree grows causes STM throughput to decrease. SBS, however, has about the same throughput on the smaller 3 sizes. This somewhat unexpected behavior is clearer when examining the results in the opposite direction: SBS throughput does not increase as the tree size becomes smaller, indicating the size is not the dominant factor. For SBS, 16 threads is the scalability limit on \(10^7\) trees; on smaller trees, entrance is even more contended, canceling the benefit of shorter traversals.

Summary: SBS is more appropriate for trees of size \(10^7\) and higher when using simple INT keys. When using STR keys that increase SBS traversal times and STM’s read sets, SBS consistently performs best.
Figure 5.7: Effect of initial size (16 threads). SBS is more effective on larger trees, where traversal time is longer.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Overhead</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evacuate global gate</td>
<td>Cache misses on local read and remote write</td>
<td>Once per chain</td>
</tr>
<tr>
<td>Create a snapshot</td>
<td>Varies</td>
<td>Rare, due to trailing</td>
</tr>
<tr>
<td>Await node after head</td>
<td>Varies</td>
<td>Always</td>
</tr>
<tr>
<td>Move to node after head</td>
<td>Sometimes cache miss on local write</td>
<td>Always</td>
</tr>
<tr>
<td>Arrival of updated location to next thread</td>
<td>Cache miss on read</td>
<td>Once per chain remote miss, otherwise local miss</td>
</tr>
</tbody>
</table>

Table 5.3: Breakdown of overhead between accessing the head and allowing the following thread to access the head.

5.4.4 Effect of read-write ratio

Since in our snapshot-based synchronization implementation readers enter the data structure one-by-one, entrance bottleneck has a similar effect on scalability regardless of reader-write ratio. Figure 5.8 shows that the write-only throughput of SBS is slightly higher than STM, but STM becomes faster as the percent of read operations is increased.

Summary: Further optimizing readers could make SBS scale better, but the implementation is non-trivial. Instead, snapshot-based synchronization can be integrated with mechanisms such as RCU \[\text{DMS}^+\text{12a}\], combining multiple concurrent writers with wait-free readers.

5.4.5 Entrance bottleneck analysis

Serialization at the entrance limits parallelism; we now dive deeper into this part of execution. In our implementation, execution can be divided into 3 parts: (1) initial ordering, (2) accessing
the head, and (3) traversing the tree. The first and last parts are mostly parallel. Accessing the head, however, can be done by a single thread at a time. A thread can not access the head until it detects the previous thread has moved to another node. Single-threaded execution thus takes place between the time one thread detects it can access the head to the time the following thread detects it can move on.

Before a thread can allow the following one to access the head, it needs to move to another node. If the thread is the first in a chain, it must first make the global gate available for the next chain once it accessed the head. It must then move to the other node and report its new location. Lastly, the following thread must read that report. The overheads of this sequence are detailed in Table 5.3. In our experiments a 16-thread write-only SBS execution the sequence took an average of 700 cycles. Multiplying this sequence latency by the throughput of 3M ops/s yields 2.1G cycles. The latency incurred by the traversal of the head is the execution’s critical path, and roughly matches our processors 2.4GHz frequency.

Summary: Scalability is limited by the rate in which threads access the head. Our implementation minimizes accesses to remote memory, but cache misses that involve communication with a core residing on the same NUMA node incur significant overhead. Serialized execution time can be reduced by either eliminating operations or using faster cross-core communication; x86 MONITOR and MWAIT, once available in user mode, are certainly of interest [ALI13].

5.4.6 Memory overhead of synchronization

A by-product of snapshot-based synchronization that can be beneficial is lower memory overhead comparing to conventional lock-based hand-over-hand. The amount of memory used by snapshot-based synchronization is proportional to the number of threads and is not affected by
the size of the synchronized data structure. In practice, the data structures used by snapshot-based synchronization (snapshots, tickets etc.) occupy a few hundreds of bytes per thread, which are practically negligible. Conventional hand-over-hand locking, however, rely on per-node synchronization state, which imposes an overhead proportional to the size of the synchronized data structure.

The minimal space overhead of hand-over-hand locking is one bit per node, which can be used as a spinlock. While that bit can be placed at the end of some pointer guaranteed to hold an aligned address, such a simple lock suffers from two shortcomings: it is neither recursive nor allow read-write separation. The latter is of higher importance, and as shown by Figure 5.8, can significantly reduce parallelism. Our implementation uses a read-write spinlock, which occupies 3 words per node (much smaller than pthread_rwlock_t and C++17 std::shared_mutex, which on our system occupy 56 bytes). The lock almost doubles the memory footprint of each node, and increases accordingly the amount of memory used by the data structure.

Summary: Snapshot-based synchronization can significantly reduce the memory footprint of the concurrent data structure it protects. Minimizing the memory overhead of traditional hand-over-hand locking will either disallow the use of read-write locks or introduce potential deadlocks (if locks are placed in a fixed-size array). Reducing memory footprint can also be useful in concurrent data structures where hand-over-hand is only used in a limited part of the algorithm, as described in Section 5.5. While not used often, the memory overhead of the locks is always incurred on those data structures.

5.5 Related Work

The hand-over-hand locking scheme (also known as lock coupling, latch coupling, crabbing etc.) was first described by Bayer and Schkolnick [BS77] as a way to construct concurrent B-trees. It has since been used to parallelize various data structures. As the major synchronization mechanism, it was used in linked lists [HS08], B-trees [Rod08], skip lists [SS11], relaxed red-black trees [OKON13] and a Treap [GGBA+11]. As a utility for a certain part of the algorithm, it was also used in priority queues [TMR16], B+-trees [SC93, MKM12], B\textsuperscript{link}-trees [ELS97, JSSS05] and hash tables [Ell85].

Data structures with properties allowing hand-over-hand synchronization have been defined as Unipath [GMR+17] and Dominance Locking [GGBA+11]. Those properties allow serializability verification [ARR10, VHHS06] and even automatic parallelization [GGBA+11].

Locking individual memory locations has been supported in various forms. Lock-box [TLEL99] provided architectural support for SMT threads to lock particular addresses without using conventional synchronization mechanisms. The Synchronization State Buffer [ZSHG07] extended this idea to a many-core system, while vLock [YTZ+13] offered a software solution. TL2 [DSS06] incorporated an array of locks in an STM library and thus enabled a fixed (yet large) set of locks to protect any number of locations. ROKO [SA12] synchronized accesses using versioning memory locations, and O-structures [GMR+18] added renaming to eliminate false dependencies, as well as per-version locking.
5.6 Conclusions

Hand-over-hand locking is a widespread fine-grained synchronization technique. The simple interface makes hand-over-hand attractive, and it has been used to parallelize multiple data structures. Furthermore, the method is simple to reason about, allowing verification and automatic parallelization. However, fine-grained locking comes at a price: locking causes cache misses on every node access. As all threads enter at the same place, the top of the data structure becomes a bottleneck that disallows scaling past a small number of threads.

Snapshot-based synchronization is a drop-in replacement for hand-over-hand locking, but uses a very different synchronization mechanism under the hood. Leveraging the data structure layout, private snapshots allow threads to avoid data races without communicating with other threads. Leveraging modern hardware, communication minimally interferes with the surrounding algorithm. In our evaluation, on large data structures snapshot-based synchronization is on average 2.6× faster than hand-over-hand locking and 1.6× faster than STM.

While its interface is simple and easy to use, Snapshot-based synchronization’s implementation is considerably more complex than simple per-node locks. Albeit undesired in general, complexity brings about many optimization opportunities. We consider the implementation described in this chapter a baseline: other implementations, possibly using newer hardware features, can make snapshot-based synchronization scale even better. In particular, reducing data structure entrance time and relaxing reader-to-reader synchronization are of interest.
Listing 5.2 NUMA-aware head entrance implementation (simplified)

```cpp
// --- public interface ---
void moveToHead(ThreadID tid, Snapshot& snapshot, void* headPos) {
    ThreadID tidAhead = enterLocalLine(tid);
    bool isChainHead = noThread(tidAhead);
    if (isChainHead)
        tidAhead = enterGlobalLine(); // last thread in previous chain

    // report new location (before actual move to avoid races)
    snapshot.setCurPos(headPos);

    // wait till the thread ahead moves away from the head
    bool isTrailing = initTrailing(tid, tidAhead);
    if (isChainHead)
        leaveGlobalLine(tid);
    if (!isTrailing)
        createSnapshot(tid); // if not trailing a snapshot must be created
}

// --- internal implementation ---
ThreadID enterLocalLine(ThreadID tid) {
    std::atomic<ThreadID>* numaNodeGate = getNodeGate(getNumaNode(tid));
    // add thread to the current node's chain
    tidAheadInChain = numaNodeGate->exchange(tid);
    return tidAheadInChain; // NoThread if first in chain
}

ThreadID enterGlobalLine() {
    ThreadID tidAhead;
    // lock global gate by setting a special value (when possible)
    while ((tidAhead = globalGate.exchange(RESERVED)) == RESERVED)
        pause();
    return tidAhead;
}

void leaveGlobalLine(ThreadID tid) {
    std::atomic<ThreadID>* numaNodeGate = getNodeGate(getNumaNode(tid));
    // close chain by setting NoThread as the last thread
    ThreadID lastInChain = numaNodeGate->exchange(NoThread);
    // release global gate by placing the ID of last thread in current chain
    globalGate.store(lastInChain);
}
```

Technion - Computer Science Department - Ph.D. Thesis PHD-2018-05 - 2018
Chapter 6

Conclusion and future work

This dissertation describes a novel memory interface that offers versioning, renaming and location-based locking. Using the extended interface, task-based programming models can exploit potential parallelism that exists in widespread workloads but is unavailable to other programming models. While the memory extensions can be implemented in software, the proposed hardware support is by far more efficient, completing the high-level programming model with a performant low-level implementation.

The memory element providing the extended interface is called an O-structure. The basic new instructions include a versioned store, which creates a new version of the target location; a version load, which loads a specific version; and another versioned load, which loads the latest version. Since loading the latest version is scheduling-dependent on its own, version locking and unlocking instructions are also provided. O-structures allow dependencies to be encoded in memory, along with the shared state. Such dedicated support for parallel code is unavailable in conventional memory systems, which are based on the von Neumann architecture, and merely provide a unified view via the cache coherence protocol.

O-structures are designed to facilitate a statically sequential task-based programming model. As opposed to advance task-based models in which dependencies are specified with tasks definitions, O-structures allow dependencies to be maintained with the state of the program, in the memory system. This model has a few advantages:

1. Dependence specification is not syntactically limited. A dependent location need not be named at compile time or have a known address at schedule time, namely before task execution. Instead, dependencies are only enforced upon actual access to each memory location. Consequently, unnamed state such as internal nodes of some pointer-based data structures can also be synchronized.

2. Dependencies need not be tracked at run time. Given large numbers of cores and tasks, the overhead of dynamically constructing and tracking the dependence graph is non-trivial, and was a major challenge for traditional dataflow architectures.

3. Tasks can be arbitrarily large. Since synchronization is done in a fine-grained manner, somewhat dependent tasks can execute mostly in parallel. On models where scheduling is
driven by the firing rule, tasks cannot be launched until all their dependencies are fully available, making shorter tasks preferable. Further, on speculative models, long tasks might slow down commit verifications and even overflow the buffers used for speculative states.

O-structures allow task-based models to parallelize operations on data structures that have the unipath property. Such data structures have a single entry point, and every internal node is accessible via a single pointer. The entry point can be used to fully order tasks operating on the data structure according to their sequential order of execution. Using a hand-over-hand locking scheme, tasks can maintain that ordering with respect to other tasks. Lastly, renaming allows readers to proceed without locking, as they will observe the correct version of the state even if writers bypass them. Since ubiquitous data structures such as lists, trees and hash tables have the unipath property, O-structure and the task-based model allow programs using them to be parallelized.

O-structures can also be used by other programming models and to parallelize non-unipath data structures. For instance, versions can be used as full-empty bits, indicating whether some immutable state has been created or not. Such synchronization can be useful on a series of dense matrix multiplications, for example. The snapshotting provided by renaming can also be used in concurrent data structures, providing snapshot isolation. Both example neither require the task-based programming model nor provide the statically sequential property.

Hardware support is implemented mostly in the caches. Dedicated buffers in main memory, which are allocated by the operating system, are carved into version blocks. Each block can hold a single datum and the matching version. Each O-structure is then comprised of a list of version blocks. The head of that list is a single pointer in the process address space. An access protection scheme ensures that versioned data is only available using the dedicated instructions, and conventional loads and stores cannot corrupt version block lists, or allow one process to access versioned data that belongs to another process. In order to reduce the memory consumption overhead of version blocks, the L1 cache supports a compressed representation, which reduces the overhead by half.

The dynamic storage scheme solves three notable problems:

1. The number of versions per O-structure is unlimited
2. The number of O-structures per program is unlimited
3. Almost automatic, on-the-fly, garbage collection ensures inaccessible version blocks are reclaimed and reused

Obviously, the actual benefit of each solution is workload dependent. However, eliminating those problems simplifies the high-level programming model: no fallback is needed in case the limits of versions number or versioned location are exceeded, and the programmer needs not worry about memory management.

Lastly, the parallelization technique used on the task-based model has also yielded a useful insight: the number of locations locked at any given moment depends on the number of threads,
not the number of locations. This insight was leveraged by the snapshot-based synchronization mechanism, which provided a fast alternative to hand-over-hand locking in software. Snapshot-based synchronization decouples locking state from the target data structure and uses minimal cross-thread communication instead.

6.1 Future work

The work described in this dissertation spans a range of layers, starting at the high-level programming model, through the software/hardware interface, and down to the micro-architectural implementation. Once the initial system and models have been established, each layer can be further developed and optimized. The following challenges are of particular interest:

6.1.1 Automatic unipath detection

Both the multi-versioned memory task-based model and snapshot-based synchronization leverage the unipath property in the algorithms they parallelize. Automatic detection of the property within a given algorithm can significantly ease the programmer’s work. While compilers cannot decide whether any given algorithm either maintains or violates the unipath property, dedicated analyses such as shape analysis [WSR00] can indicate the property possibly exists and highlight the relevant parts of the algorithm. Further, unipath detection can be performed at compile time using static analysis, at run time using instrumentation, or post-mortem based on execution traces.

6.1.2 Task scheduling

The multi-versioned memory task-based model raises unique scheduling challenges. As inputs and outputs are not included in task specifications, the scheduler cannot construct a dependency graph at run time and ensure tasks are not prematurely executed, forcing execution to stall. Such lack of dependency specification also exists at the Cilk model, which requires the user to order dependent tasks, and speculative models, which implement complicated re-execution mechanisms. Useful scheduling hints can be found in the task identifier and the versions a task will load.

Another important aspect of scheduling is the action taken when a task is stalled by the memory system. The initial implementation does not support preemption, hence an attempt to load an unavailable version will leave the processor idle until the load can be completed. Allowing other tasks to use the processor, effectively preempting the stalled task, can improve processor utilization but also introduces preempted tasks into the scheduler’s considerations. Further, the current memory interface neither informs the scheduler about a stall nor allows a versioned operation to be resumed after preemption. A more involved scheduler should therefore be architected.
6.1.3 Hardware support for explicit cross-thread communication

On modern multi-core systems, shared memory is a leaky abstraction [Spo04]. Namely, while the memory interface provides a unified view of the program’s state, access latencies give away implementation details such as caching and cross-core communication. Implementing software cross-thread communication mechanisms on top of the conventional interface must rely on reading and writing shared locations. Communication among cores is managed by the cache coherence protocol, which is optimized for state unification and not direct, architecturally visible, communication. For instance, sending a single bit from one core to the other via the cache coherence protocol might include multiple cache line invalidations, false sharing, tight spinning and so on.

One feature that turned out to be missing during this work was the ability to control the location of the state. For instance, an attempt to lock an already locked version should not involve moving the target line out of the locking core’s cache. Doing so will be useless for the failing locker, and will incur an additional cache miss when the locker tries to eventually unlock the version. On the other hand, O-structures hold state, which is shared by design. Once a new version is created, proactively pushing it to other cores could save a cache miss on their part. Whether a line should or should not be pushed somewhat depends on the program, hence state location management may greatly benefit from dedicated instructions.

Another feature that could have been useful is lightweight message passing. On the snapshot-based synchronization mechanism, trailing is maintained by having the thread in front signal the following thread it can proceed. When this is done via a shared memory location, the follower will poll by constantly reading the location, until the thread ahead writes to it. But before the write can be performed, the coherence protocol must first invalidate the polled line, then allow the write, and then send the line back. The overall communication, involving both coherence messages and line movements, is expensive — especially on NUMA system. If the small amount of data the program actually intended to move across threads could be explicitly sent and received without interfering with state management, communication overhead would have been much lower.
Bibliography


[CJH+12] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic parallelization of


101


[HWC+04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and


Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb’s journal, 30(3), 2005.


לעברית. אם כן, יש להתחדש כדי לשוחות עם זה שייש WindowManager. אם לא, יникח להתקדם ב巴西יתת.

תורוטות של התיעוד מה הם הפחתות שמוספרות בתכונות התיקון של התחום, ונכון הפרד הוא מנותון
הנעלת ממצב התחום. חלפאתו מותרת, אך יקרה יקרה לא דרוש כדי לשני מבצעי את האיבר
(ה Beetle וה Beetle רכובים erbiciות הדימויים הרובים ומעוטים את מצלמי הא.ALIGN). נבדק, מתאפק
יאחרון על החרב יותר של מבנה התינוקים במטמון של מערכות היחידה.

לסיום, העלו את החרה שה킬יט במטמון מערכות ירח מתורגמן. ראו ויימיט, מוניות המומלע
שאותו מערכות ירח והוхотות لتحقيقו האמסטרים. ערים, מוציאים של תכונות מבוסס
 tableView, אשר משמשים על תכונות מחלק התיקון של מערכות ירח המורחבת כדי למתקפל פועלות על מבני
תונות ד뉜ים. שילוש, מציגים ממחושב חורף של הרחבת מערכות היורון. ריביון, מיצוגים שית
מקבלי תאר מבוסס על תכונות מצוותת על מערכות ירח המורחבת. השיטה הא筌ורנה וממקס
לא משמשת בתיעוד מערכות ירח המתורגמן, אך היא מחוות דוגמה לתכונות המקבילות של ירח
לשרפ
באמפיית הרחבת על מערכות ירח, מעבר למסימק הקומפקטית.
more, because the calculation is parallel. Its design allows for easier handling of numerical sequences, particularly in situations where calculations are performed sequentially. It can be used to validate and ensure the correctness of the results, and to verify the logical correctness of the data, even though the methods used are not guaranteed to be correct or sound.

One of the unique features of the parallelization model is that it is designed to work on the data in its sequential form, which can significantly improve the efficiency of the process.

Parallelization allows for the efficient handling of complex data structures and algorithms, which can significantly improve the efficiency of the process.

This method is particularly useful in situations where the data is processed sequentially, such as in the case of stream processing. It can be used to handle large datasets efficiently and can significantly improve the performance of the system.

In conclusion, the parallelization model offers a powerful tool for handling complex data and algorithms, which can significantly improve the efficiency of the process.
The thesis explores the benefits of using data flow models in the context of parallel computing. The author discusses the limitations of traditional models, such as the garden variety sequential model, and the challenges they pose in maintaining the synchronization of shared data.

The thesis introduces a new model that addresses these limitations by providing a more flexible and efficient way of managing data flow. This model is based on a more sophisticated understanding of data dependencies and the need for efficient memory management.

The thesis provides a detailed analysis of the model's components, including the use of data caches and the management of shared memory. It also discusses the implications of this model for parallel processing and the potential for improving performance and efficiency.

Overall, the thesis presents a compelling case for the use of data flow models in parallel computing, and provides a valuable contribution to the field.

Technion - Computer Science Department - Ph.D. Thesis PHD-2018-05 - 2018
תקציר

majors 提出了使用 mild 特性的微处理器来提高程序的执行效率。这包括两个方面：

1. 通过降低每个指令的执行时间来提高指令的执行效率。
2. 通过减少程序中指令的执行时间来提高程序的执行效率。

这些方面的提高可以有助于提高程序的执行效率。然而，这些技术的发展仍然需要更多的研究和发展。


Eran Gilad, Tehila Mayzels, Elazar Raab, Mark Oskin, and Yoav Etsion. Architectural support for unlimited memory versioning and renaming. In International Parallel & Distributed Processing Symposium (IPDPS), 2018.

ריצה מקלילית באמעעות יכור מروب

גייסאות

гибור על מחקר

לשם مليוי חלקי של הדרישה לגבי התואר
דוקטור לפילוסופיה

ערן גלעד

הנהלת הטכניון – מכון טכנולוגי לישראל
שם ההערכה: חיפה
יולי 2018

Technion - Computer Science Department - Ph.D. Thesis PHD-2018-05 - 2018
ריצה מקבילה באימוץ זיכרון מורחב

גירסאות

ערן גלעד