Provably Efficient Coscheduling of Computation and Memory through Disentanglement

Jatin Arora

CMU-CS-24-141

August 2024

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

Thesis Committee:
Umut A. Acar, Chair
Guy E. Blelloch
Robert Harper
K. Rustan M. Leino (Amazon)

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

This research was sponsored by the National Science Foundation under award numbers CCF-1314590, CCF-1408940, CCF-1629444, CCF-1901381, CCF-2028921, CCF-2107241, CCF-2115104, and CCF-2119352. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: parallel programming, functional programming, parallel algorithms, automatic memory management, garbage collection, disentanglement, hierarchical memory management, race conditions

For my godfather

Abstract

Because of its many desirable properties, such as its ability to control effects and thus potentially disastrous race conditions, functional programming offers a viable approach to programming modern multicore computers. This has led to the development of several parallel functional languages, typically based on dialects of ML and Haskell. However, these languages have traditionally underperformed compared to procedural languages such as C and Java. The primary reason for this underperformance has been the lack of scalable memory management techniques that can match the increased demand of memory in parallel execution.

Building on a long line of work on parallel memory management, this thesis proposes provably efficient techniques for managing memory in parallel functional programs. The key idea behind our techniques is to coschedule the parallel computation with its data, enabling the memory manager to exploit the disentanglement hypothesis—the idea that parallel tasks of a program largely execute independently and avoid side-effecting data that may be accessed by others–for efficiency. We implement these techniques in the MPL compiler for parallel ML, and our experimental results show that the techniques can marry the safety benefits of functional programming with performance.

Acknowledgments

First, I want to thank my advisor Umut Acar, for making my PhD journey possible. Umut has been the most understanding, appreciative, and larger than PhD advisor I could have hoped for. He taught me more things than I can list in both life and research. Perhaps my favorite lesson from Umut was learning to recognize, appreciate, and expect quality work. Thank you, Umut!

I would like to thank members of my thesis committee: Robert Harper, Guy Blelloch, and Rustan Leino. Bob's (Robert) ideas and convictions have always been an inspiration to me and I really appreciate his support throughout the thesis process. Guy has been a supportive mentor and teacher, from whom I have learnt a lot. Rustan: I want to thank you for hosting me at Amazon and for being such a great mentor from Day 1. Thank you all for your time and cooperation with my thesis defense.

I have been fortunate to work with a lot of great people during my time at CMU. I want to especially thank Sam Westrick for essentially being my second advisor, and also tolerating all the stupid ideas I have had throughout the years. I learnt a lot from you Sam, and the sheer joy with which you approach your work is a reference for me. I also want to thank Stefan Muller, Pengyu Liu, Dantong Li, Mingkuan Xu, and Yongshan Ding, who taught me a lot. It was a pleasure to work with you all.

I am very grateful to my mentors who have guided me to where I am today. Thank you Frank Pfenning for introducing me to Programming Languages, which was my favorite class. Thank you Jan Hoffmann, Marijn Heule, Stephen Brookes, David Kahn. I am very grateful to Krishna Subramaniam, Parosh Abdulla, Faouzi Atig, Supratik Chakraborty, Akshay S, Rupak Majumdar, Anthony Lin, Sumith Kulal, Shubham Goel for introducing me to research in my undergrad.

I feel most fortunate to have my friends, who provide me with a warm, comfortable space. I want to thank my roommates, old and new, Praneeth Kacham, Vishnu Raghuraman, Saket Dingliwal, Divyansh Pareek, for putting up with me. My experience would not have been the same without Peter Manohar, Magdalen Dobson, Hugo Sadok, Long Pham, Luiz Sa, Rachel Lee, Alex Chen, Beatrice Lee, David Kahn, and Yue Niu. Special thanks to Akshat Gupta, Braham Chawla, Kshitij Jain, and Shaan Vadiya for always being there, even without being here. Thank you all for so many memories.

During my PhD at CMU, I met many people who made being on campus fun. Thanks Nirav Atre, Ziv Scully, Jay Bosamiya, Siva Somayyajula, Daniel Anderson, Abhiram Kothapalli, Yue Yao, Lucio Dery, Justin Whitehouse, Matthew Weidner, Bernardo Subercaseaux, Shir Maimon, Mikhail Khodak, Siddharth Prasad, Dorian Chan, Suhas Jayaram, Ananya Joshi, Aditi Kabra,

and Dravyansh Sharma. Thank you all for creating an enjoyable campus atmosphere.

Thank you, Urvee, for your love and unwavering support. Your child-like lightheartedness will always remind me to enjoy the simple things in life.

To my parents, Bharat and Seema, thank you for always prioritizing my life and giving me an amazing, comfortable platform. Thank you Nana, for all the memories and card games. Thank you Massis, Renu and Puja, for spoiling me with your food, gifts, and warmth. A special note for two people who played a formative role in my life and who I miss dearly: Ajay Saluja, for always pushing me to pursue my dreams, and Jatinder Mehra, my godfather, for making me believe in myself.

Contents

List of Figures

1 Introduction

Every computing device today, ranging from smartphones with 10 cores and workstations with dozens of cores [161], to servers with hundreds [65], and even thousands of cores [141], is a parallel computer. Motivated by these hardware advances, researchers have designed high-level programming languages to utilize modern multicore chips. These languages abstract away the low-level hardware and memory details, providing programmers with high-level constructs such as parallel tuples, parallel-for, fork-join, and async-finish to express parallelism. This abstraction relieves programmers from the burden of managing the parallel execution manually.

Instead, runtime systems manage the parallel execution. These runtime systems are responsible for scheduling parallel programs on processors and managing memory. A key goal of such runtime systems is to provide an abstract cost specification that allows the programmer to reason about the performance of their code. To achieve this goal, nearly all parallel languages strive to abide by Brent's theorem, which states that a program with work W (total number of instructions) and span S (longest chain of dependencies) can be executed in time T_P on P cores such that

$$T_P \leq \frac{W}{P} + S$$

Indeed, many effective scheduling algorithms that match the Brent bound have been designed (e.g., [1, 6, 22, 43, 48]), and implemented in a wide variety of parallel programming systems [38, 42, 59, 83, 109, 111, 112, 136, 163]. However, all of this work either ignores the space performance and cost of memory management, or assumes perfect (manual) memory management. This shortcoming invalidates the performance guarantees provided by the scheduling algorithms, because they do not account for the cost of memory management—a core component of almost all high-level programming systems. The absence of bounds for parallel memory management is not just a theoretical curiosity, unless carefully accounted for, the cost of

garbage collection can easily undo the benefits of parallelism.

In this thesis, we develop provably work- and space- efficient techniques for automatically managing the memory of parallel programs. Specifically, we bound the work and space for P-processor runs in terms of work and space of a sequential run. The bounds account for the cost of scheduling and the cost of garbage collection. We implement our memory management techniques in a parallel functional language called MPL and establish that these techniques are not only theoretically sound, but also practically efficient, as they scale to dozens of cores.

1.1 Fork-Join Parallelism and Programming Languages

Fork-Join parallelism is a powerful technique for utilizing modern multicore processors. It abstracts away the complexities of managing parallelism manually by providing programmers with high-level constructs such as parallel tuples, parallel-for, fork-join, and async-finish. These constructs are then managed by the runtime system, which automatically creates and schedules parallel tasks using a task scheduler. As a result, fork-join parallelism has been adopted by many programming languages and libraries including procedural languages such as Intel Thread Building Blocks (a C++ library) [98], Cilk (an extension of C) [50, 84], OpenMP [134], Task Parallel Library (a .NET library) [111], Rust [144], Java Fork/Join Framework [109], Habanero Java [97], and X10 [59]. While these languages deliver great performance, they often make parallel programming challenging because of their lax control over effects or mutation. With little or no control over effects, it is easy for programmers to create race conditions, which can have disastrous consequences [8, 10, 51, 52, 54, 75, 117, 130, 166].

To address these challenges, researchers have proposed functional languages that make parallel programming much simpler and safer, such as multiLisp [90], Id [23], NESL [38, 42], various forms of parallel Haskell [92, 112, 116, 136], Multicore OCaml [159], and several forms of Parallel ML [83, 89, 133, 138, 157, 163, 174, 183]. Some of these languages only support pure or mutation-free functional programs, thereby eliminating data races by construction. Others such as Parallel ML [89, 174, 178] and Multicore OCaml [159] allow side effects, but use powerful type systems to simplify reasoning about data races, for example, by separating mutable and immutable data. Functional languages also support higher-order functions, such as map, filter, reduce over collections of data, which enable expressing parallel algorithms elegantly and succinctly.

By reducing the emphasis on in-place updates/mutation and supporting the use of higher-order bulk operators, functional languages promote a style of parallel programs that is simpler and safer. However, achieving practical efficiency and scalability with this approach has been challenging. The primary reason for this is memory: such programs often rely on immutable data structures and require frequent allocations [14, 16, 26, 71, 72, 86, 87, 116]. This allocation rate further escalates with parallelism, because multiple cores can allocate at the same time, overwhelming traditional memory management techniques.

Thus, writing correct, efficient, and scalable parallel code remains challenging for programmers today. In this thesis, we address this challenge by improving the efficiency of parallel

functional programs. Specifically, our goal is to design automatic memory management techniques that scale to meet the allocation demands of these programs.

1.2 The Performance Challenge

Automatic memory management, essential for safety of parallel programs, presents significant challenges for their performance. Our goal is to support fast and parallel memory management because parallelism increases allocation rates as multiple processors demand memory simultaneously. In parallel functional languages, nearly all memory managers facilitate this by partitioning the memory into "processor-local" regions or ***heaps***, so that each processor can allocate in its own private heap [26, 108, 116, 158].

While heaps enable fast and parallel allocation, they increase the underlying complexity and costs of memory management due to inter-heap pointers. ***Inter-heap pointers*** are references from objects in one heap to objects in another heap. If the memory manager does not account for these pointers, it may reclaim objects that are still in use, leading to program errors or crashes. Therefore, the memory manager must account for these pointers to ensure that all "live objects" are preserved. Managing these pointers can require expensive synchronization, with costs that increase with scale. Researchers have broadly proposed two techniques to handle such pointers:

1. *Stop-the-world synchronization.* This technique halts all processors during garbage collection and determines the "roots" from each processor. While efficient for low core counts, it becomes impractical as the core count increases. The synchronization overhead inherent in the stop-the-world significantly increases with the number of cores and harms scalability. This is particularly problematic for parallel functional programs that require frequent garbage collections [15, 19].
2. *Promotion.* This technique preemptively copies objects from processor-local heaps to a shared global heap to manage inter-heap pointers. Promotions require copying reachable objects from an inter-heap pointer and can be triggered by scheduler actions, making it difficult to prove bounds on their runtime and space impact. Promotions have proved to be expensive in practice [89, 159].

Thus, even though heaps enable fast and synchronization-free allocation, they introduce a **performance challenge due to inter-heap pointers**. These pointers are a fundamental and unavoidable consequence of using heaps, as they exist even in purely functional programs that perform no mutable effects and are deterministic. When the scheduler migrates tasks between processors, it causes multiple processors to access the same memory locations, resulting in pointers between their heaps. Managing these pointers results in overhead and harms scalability. This creates a tug of war between synchronization-free allocation, which is essential for the performance of functional programs, and scheduling, which is essential for scalability.

Because of these fundamental challenges, current techniques for the processor-local-heap architecture do not guarantee provable space and work bounds [26, 71, 72, 116, 159]. Perhaps surprisingly, this limitation applies even to purely functional programs, which are easier to analyze due to their deterministic behavior and freedom from side effects. This is in stark contrast

to techniques for sequential memory management, where nearly all programming models are provably space and work (time) efficient [101].

To begin addressing these challenges, we propose the disentanglement hypothesis, which motivates a novel organization of heaps and enables efficient tracking of inter-heap pointers. This sets the foundation for *coscheduling* (Section 1.5) of computation and memory, which exploits program parallelism to deliver provable efficiency.

1.3 Disentanglement Hypothesis

Consider a parallel program written using the fork-join primitive, which allows any task to fork new child tasks that run in parallel, and then wait until the children join. These computations form a tree of tasks, where parent-child relationships represent sequential dependencies and sibling tasks run in parallel. For such fork-join programs, we propose the disentanglement hypothesis, which categorizes each memory object as disentangled or entangled, based on how tasks access (read/write) objects relative to the tasks that allocated them:

- **Entangled objects**: Allocated by one task and accessed by other parallel tasks
- **Disentangled objects**: Accessed only by tasks that are sequentially dependent on the task that allocated them, not by parallel tasks.

The **disentanglement hypothesis** states that most objects in a parallel fork-join program are disentangled, meaning they are never accessed by tasks executing in parallel. This is based on our observation that objects become entangled through races, where parallel tasks perform mutable operations on the same memory objects. Such races only involve a small portion of the memory, as most parallel programs prioritize independence among tasks to achieve good parallelism and avoid side-effecting memory objects that are allocated by others, avoiding entangled objects.

For example, in a computation where two tasks A and B are executing in parallel, task A can only access an object allocated by B through a race condition, where B writes its object in a mutable cell and A reads it; our hypothesis is that such objects are rare.

We validate the disentanglement hypothesis both theoretically and empirically for many fork-join programs. The hypothesis holds universally for all deterministic programs, including purely functional and race free programs, because they don't have any entangled objects [3, 20, 178]. Nondeterministic programs, which use mutable effects in ways that create nondeterministic behavior, typically have only a small portion of their memory entangled, thereby satisfying the disentanglement hypothesis. We empirically validate this for a wide range of parallel algorithms including those from C++ benchmark suites such as PBBS, Ligra, and ParlayLib [12, 46, 150, 168].

1.4 Independent Memory Management

In this section, we present an overview of our memory manager that supports all fork-join programs with arbitrary mutable effects. It achieves the following three scalability properties:

- Each task can allocate memory without synchronizing with other tasks.
- Each task can reclaim memory independently, by only tracing its own portion of the memory.
- Tasks may share objects without any restriction, and without copy operations (a.k.a promotions) or object movement

To achieve this goal, our memory manager exploits the disentanglement hypothesis by organizing memory as a dynamic hierarchy of task-local heaps that mirrors the parallel structure of computation. For each task fork/join operation, the memory manager creates and joins heaps. Within this heap hierarchy, the memory manager distinguishes between disentangled and entangled objects using our entanglement tracking algorithm. This algorithm is optimized for disentangled objects, ensuring near-zero overhead for their allocations and accesses, while only incurring some overhead for entangled objects.

By differentiating between disentangled and entangled objects, the memory manager can efficiently manage inter-heap pointers to support independent garbage collection. Pointers into disentangled objects are relatively straightforward to track because disentangled objects are only accessed by sequentially dependent tasks. Pointers into entangled objects do not need to be tracked, because our entanglement tracking algorithm maintains all of them and the garbage collector keeps them live. By tracking entangled objects, our memory manager can efficiently account for inter-heap pointers. This allows the memory manager to independently garbage collect any heap by only tracing memory within that heap. The memory manager also simultaneously enables fast allocation, since each task can allocate within its own heap. Crucially, the memory manager never stops the world nor promotes any data.

Our use of the disentanglement hypothesis is similar to how generation-based memory managers exploit the (weak) generational hypothesis, which asserts that most objects die young and can be quickly reclaimed. Generation-based memory managers exploit this hypothesis by organizing the memory into "generations", grouping objects of similar "age", and optimizing the garbage collection for common case of the young objects [101]. Similarly, our approach organizes the memory into "task-local heaps" by grouping objects allocated by the same task and optimizing performance for disentangled objects.

1.5 Coscheduling

Partitioning memory into a dynamic hierarchy of task-local heaps that mirror the task tree enables independent garbage collection of each heap (see Section 1.4). However, this hierarchy has numerous heaps, making it challenging to ensure that the cumulative work and space cost of garbage collection is bounded. Additionally, independent garbage collection assumes that the inter-heap pointers are live, which can lead to unbounded space usage. For example, if heap A

has pointers to objects in heap B and B is garbage collected independently, then the objects in B cannot be reclaimed because they are referenced by objects in A, potentially accumulating garbage in heap B. This design raises a critical question: how can we coordinate the garbage collection of these heaps to achieve both provable and practical efficiency?

We address this challenge by **coscheduling computation with memory**, where our scheduler simultaneously assigns tasks and heaps to processors. Our scheduler consists of a task scheduler and a heap scheduler that distribute tasks and heaps on processors. The heap scheduler dynamically partitions all the heaps into **heap clusters** and assigns each cluster to a processor. Each processor then independently garbage collects the heaps within its cluster.

The heap scheduler coordinates the garbage collection of different heaps by clustering them based on the parent-child relationships in the heap tree. By keeping closely related parent-child heaps in the same cluster, it ensures that objects in these heaps are garbage collected together by the same processor. This clustering is provably space-efficient because each cluster roughly corresponds to the memory in a sequential execution. Simultaneously, different processors can garbage collect their own clusters independently, because the clusters are disjoint.

To create the clusters, the heap scheduler closely follows the decisions of the task scheduler, guaranteeing that each active task and its corresponding heap are coscheduled on the same processor. This allows the same processor to efficiently manage both allocation and garbage collection for the task. Together the task scheduler and the heap scheduler migrate tasks and heaps between processors.

But we must take care with heap scheduling: As the heap scheduler migrates heaps between processors, it can increase the work cost of garbage collection. In sequential garbage collectors, the collection work can be easily amortized against the work of allocations. However, when objects allocated on one processor are garbage collected by another, it complicates the justification of the work cost. We address this via a collection policy. The collection policy determines when a processor must garbage collect to meet the desired work and space efficiency bounds. The policy is fully distributed: each processor makes its decisions independently of all other processors, without any synchronization. The key aspect of this policy is that it ensures that each processor roughly allocates as much as it collects by charging the collection work against future allocations. However, because the policy relies on future allocations to justify the collection work, it lets each processor perform one extra garbage collection. This additional work shows up as an extra term in the work bound below.

1.6 Work and Space Bounds

With our collection policy and coscheduling, we prove space and work bounds on P-processor computations. We start by considering deterministic parallel programs and then extend our results to programs with nondeterministic effects.

Deterministic Programs. Deterministic programs, including those that use mutable effects in a deterministic fashion, guarantee that their executions perform the same instructions irrespective of the number of cores. This allows us to use a cost metric R, the high watermark of memory used in sequential runs, as the baseline for P-processor runs. We formally define the metric R using a cost semantics in Chapter 5. We show the following key results for executing a deterministic program with work W and sequential space R on P processors:

- total parallel work, including the cost of garbage collection, is $O(W + R \cdot P)$, and
- total parallel space used for execution is $O(R \cdot P)$.

The total parallel space, $O(R \cdot P)$, is not just the live/reachable space; it is the amount of maximum space the memory manager allocates during execution, including all its overheads. We observe that this bound is tight for our setting. Specifically, for a work-stealing task scheduler, like the one we use, it is expected that parallel executions can require at least P times as much space, even assuming perfect/manual memory management. Our space bound, after performing fully automatic memory management, show the same asymptotic scaling, and thus is tight. The work bound is almost tight, except it has an additive term $R \cdot P$, which is a consequence of the heap migrations performed by the heap scheduler. These migrations trigger P extra garbage collections, each requiring $O(R)$ work in the worst case.

Nondeterministic Programs. Now we consider programs that are nondeterministic due to (determinacy) races. To account for the work and space cost of such programs, we develop a cost semantics that tracks the memory associated with races. The semantics assigns each computation a *race factor* that bounds the size of the memory that may be accessible via nondeterministic races. We use this race factor to bound the space consumption of parallel runs, as our memory manager keeps the objects affected by races (including entangled objects) live until tasks involved in the races join.

Another key challenge in accounting for the cost of racy programs is the lack of an obvious sequential baseline to compare their performance. A particular parallel run of a racy program may not have a corresponding sequential run, because it is not always possible for a racy execution to repeat on a single core. To account for this, we define a cost metric called sequentialized space, R^*, that approximates a sequential cost for a given parallel execution. For a parallel execution with work W and sequentialized space R^*, and race factor r, we show the following work and space bounds:

- total parallel work including the cost of garbage collection is $W + (R^* + r) \cdot P$ work, and
- total parallel space is $(R^* + r) \cdot P$.

These results show that our memory manager's cost is well-behaved for racy programs: its overhead is linear to race factor r, which represents the amount of memory associated with races. If this amount is small then the overheads are correspondingly small. Note that these results are similar to the bounds for deterministic programs, where the race factor r is zero.

1.7 Implementation and Evaluation

Our techniques are unique in how they integrate the scheduling of tasks and garbage collection within the same scheduler. Given their reliance on concurrency for efficiency and scalability, a natural concern is practicality: can these techniques be made to work well in practice? This thesis includes a major implementation effort that develops the MPL compiler for the Parallel ML language. MPL extends the MLton for Standard ML by implementing support for nested fork-join parallelism, supporting all Parallel ML programs including those with unrestricted use of references and mutation.

Using this compiler, we have implemented a substantial benchmark suite, containing parallel algorithms from various problem domains such as graphs, computational geometry, numerical analysis, and quantum computing. Many of these benchmarks were ported from C++ benchmark suites such as PAM, PBBS, Ligra, and ParlayLib [12, 46, 150, 168]. We implemented these state-of-the-art non-blocking concurrent data structures as an open-source library for MPL, which these parallel algorithms utilize. Some of these benchmarks have entangled objects, for example those using hash tables, but all of them validate the disentanglement hypothesis—only a minority of the objects are entangled.

Our experiments show that MPL performs well, incurring relatively small overheads compared to sequential runs, and scaling well to dozens of cores. Notably, MPL delivers both performance and compactness: parallel runs usually consume less memory than sequential runs and deliver significant speedups. We also perform a comparison with several other languages including Go, Java, Multicore OCaml, and C++. Our results show that MPL is competitive with these languages, roughly within 2x of the fastest C++ implementations on 72 cores.

1.8 Disentanglement Hypothesis beyond Fork-Join

The promising results for memory management of fork-join programs raise the question: Are the memory management techniques applicable beyond the fork-join model? While a comprehensive answer to this question is beyond the scope of this thesis, we present some theoretical evidence that the techniques could be extended to support more general forms of parallelism. Specifically, we show that the guiding principle of our memory management techniques—the disentanglement hypothesis—also holds for programs with futures.

Invented in the 1970s [29], futures allow you to create a parallel task and demand the result from the task at a later time when needed (hence the name "future"). Unlike fork-join which is a control-flow construct, futures are first-class values, making parallelism a "first-class citizen" of the programming language. This feature contributes to their expressive power: futures can express data-dependent parallelism, pipelining, asynchrony, and interaction.

Surprisingly, we find that a broad class of programs written with futures, even in the presence of asynchronous interaction and data dependencies between tasks, satisfy the disentanglement hypothesis. That is, most objects are accessed by sequentially dependent tasks, rather than by concurrent tasks. We consider a calculus that combines futures with state and I/O, and

prove that determinacy-race-free programs written using this calculus do not have any entangled objects, and thus satisfy the hypothesis. This result suggests that our memory management techniques could be extended, potentially in a provably efficient fashion, to a wider range of parallel applications based on futures.

Outline

The main chapters of this thesis are as follows:

- A language semantics for describing the heap hierarchy for fork-join parallel programs and an algorithm for coscheduling computation and memory (Chapter 2)
- An entanglement semantics that distinguishes between disentangled and entangled objects, and quantifies the amount of entanglement. We use the semantics to state and provide evidence for the disentanglement hypothesis (Chapter 3).
- Memory management techniques for supporting independent allocation and independent garbage collection of each heap in the memory hierarchy. This includes the algorithm for tracking entangled objects and proofs that the tracking only incurs small work/time and space overheads, proportional to the amount of entanglement (Chapter 4).
- Space and work bounds for memory-managed fork-join programs (Chapter 5).
- Implementation of the MPL compiler for Parallel ML (Chapter 6). We implement all our techniques in MPL.
- Parallel benchmarks from a variety of problem domains and their evaluation (Chapter 7)
- A semantics for defining disentanglement for futures and proofs that determinacy-race-free programs with futures do not have entangled objects. (Chapter 8)

Peer-Reviewed Publications

The thesis contains work that appeared in the following publications.

- Provably Space-Efficient Parallel Functional Programming [17], at POPL'21. This paper presented the coscheduling algorithm (Chapter 2) and bounds for work and space for deterministic programs (Chapter 5).
- Efficient Parallel Functional Programming with Effects [19], at PLDI'23. This paper presented memory management techniques for entangled objects (Chapter 4), by exploiting the disentanglement hypothesis.
- Disentanglement with Futures, State, and Interaction [20], at POPL'24. This paper generalized the theory of disentanglement to futures (Chapter 8).

Thesis Statement

Our theoretical and practical results support the following thesis statement:

By coscheduling the computation with its memory, we can exploit the disentanglement hypothesis to perform provably and practically efficient memory management.

2

Coscheduling of Computation and Memory

In this chapter, we propose a coscheduling algorithm that integrates task scheduling with memory management of parallel programs. The key idea behind our approach is to organize memory into a tree of heaps and schedule heaps by actively mapping them to processors, just like a task scheduler that assigns threads (or tasks) to processors. Each processor in turn allocates memory only in the heaps that are assigned to it and is responsible for collecting garbage in those heaps. The decisions to garbage collect are mediated by a fully distributed collection policy which we formulate.

Note that it is possible to garbage collect each heap independently without coscheduling. While this would be efficient in isolation for each heap, this approach falls short, because it offers no guarantees on the overall work and space costs of garbage collection. This is also challenging because the heap tree has numerous heaps, each potentially containing pointers to objects in other heaps. These inter-heap pointers can lead to unbounded space usage because for independent garbage collection of each heap, we must assume they are live.

To overcome these limitations of per-heap garbage collection, our coscheduling approach carefully coordinates the garbage collection of heaps. It partitions the heap tree into clusters and schedules garbage collections of each cluster. Each cluster is designed to roughly mimic a sequential execution, enabling us to bound the space of every cluster. As we show in subsequent chapters, by carefully coordinating the garbage collection of heaps, coscheduling achieves both provable and practical efficiency.

To present our coscheduling technique, we first define a call-by-value functional language that supports fork-join parallelism. We define a language semantics that organizes the computational tasks as a tree and also creates a corresponding heap tree that mirrors the task tree. We then describe the coscheduling algorithm that schedules tasks and heaps on processors and also implements our collection policy.

$$
\begin{array}{rlll}
\textit{Variables} & x, f & & \\
\textit{Numbers} & m & \in & \mathbb{N} \\
\textit{Types} & \tau & ::= & \texttt{nat} \mid \tau \times \tau \mid \tau \to \tau \mid \tau\ \texttt{ref} \\
\textit{Memory Locations} & \ell & & \\
\textit{Storables} & s & ::= & m \mid \texttt{fun}\ f\ x\ \texttt{is}\ e \mid \langle \ell, \ell \rangle \mid \texttt{ref}\ \ell \\
\textit{Expressions} & e & ::= & \ell \mid s \mid x \mid e\ e \mid \langle e, e \rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{ref}\ e \mid\ !\, e \mid e := e \mid \langle e \parallel e \rangle \\
\textit{Memory} & \mu & \in & \textit{Locations} \rightharpoonup \textit{Storables} \\
\textit{Task Identifiers} & u, v & & \\
\textit{Heaps} & h & ::= & \emptyset \mid h, \ell \\
\textit{Task \& Heap Tree} & T & ::= & \mathsf{ALeaf}(u, h) \mid \mathsf{PLeaf}(u, h) \mid \mathsf{Par}(u, h, T, T)
\end{array}
$$

Figure 2.1: Syntax

2.1 Language

For a formal analysis of our techniques, we consider a simple call-by-value functional language extended with fork-join (nested) parallelism. Richer constructs like arithmetic operators and arrays could be added, but we omit them for brevity. The language supports unrestricted mutable effects. We give an operational semantics that evaluates the expressions of the language and also defines a task tree and a heap tree for each step. The trees encode the structure of the parallelism of the program and our heap scheduling algorithm operates on the trees.

2.1.1 Syntax

Figure 2.1 presents the syntax for the language. We cover some aspects of the syntax here and discuss others as needed.

Types. The types include a base type of natural numbers, function types and product types for expressing parallel pairs. The type system also supports mutable references.

Expressions. Expressions in our language include variables, locations, storables, and introduction and elimination forms for the standard types. The language includes the parallel pair ($\langle e \| e \rangle$) for expressing parallel computations, where the expressions within the pair can be evaluated in parallel. For an expression e, we use $\mathsf{locs}(e)$ to denote the set of locations referenced by it.

Memory Locations and Storables. The language tracks memory operations by distinguishing between ***storables*** s and ***memory locations*** ℓ. Storables are allocated in a memory store μ and include natural numbers, named recursive functions, pairs of memory locations, and mutable references. Storables step to locations, which are the only irreducible form of the language. We say that a location ℓ ***points to*** location ℓ' if the storable at ℓ mentions ℓ'.

Memory Store. The ***memory store*** μ maps locations to storables. We use $\mathrm{dom}(\mu)$ for the set of locations mapped by μ, $\mu(\ell)$ to look up the storable mapped to ℓ, and $\mu[\ell \hookrightarrow s]$ to extend μ with a new location.

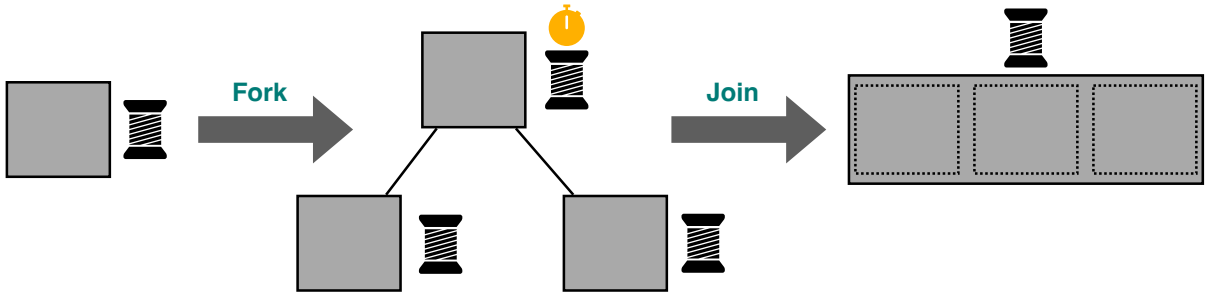

Figure 2.2: Forks and joins. Tasks are spools and their heaps are gray rectangles. The "stopwatch" denotes suspended tasks, waiting for its children to finish.

2.1.2 Task Trees

The semantics maintains a *dynamic* task tree to encode the structure of the parallel computation. Each node in the tree is a task and the parent-child relationships represent the sequential dependencies between tasks. Each task evaluates some expression and terminates when the expression is fully evaluated.

The computation starts with a root task. When a task evaluates a parallel pair, it ***forks*** two child tasks and suspends its own execution, as it waits for the children to terminate. When both children of a task have terminated, they ***join*** into the parent and are removed from the tree.

The ***internal tasks*** in the tree are ***suspended*** because they await the completion of their children tasks. Leaf tasks are either ***active*** or ***passive***. Active leaf tasks are currently executing their computations, while passive leaf tasks have finished their computations but are waiting for their sibling tasks to complete before they can join.

2.1.3 Heap Trees

We organize all locations in the memory store μ as a heap tree that mirrors the structure of the task tree, where each task is assigned its own heap for memory allocation. As we will demonstrate, this design facilitates efficient memory management for tasks.

Each task is given its own ***heap***, which is simply a set of memory locations allocated by that task. We use the variable h to represent heaps and use notation h, ℓ to extend the heap with location ℓ. New tasks are initialized with fresh empty heaps, and when sibling tasks join with their parent, we merge their heaps into the parent heap. This ensures that all locations allocated by a task are transferred to its parent upon completion. We use similar terminology for heaps as for tasks: internal heaps are ***suspended***, and leaf heaps are either ***active*** or ***passive***

(determined by the status of their corresponding tasks).

To maintain a clear association between tasks and their heaps, our semantics superposes the task tree and heap tree into a unified tree. Each node of the tree contains both a task and its corresponding heap. We use identifiers like u, v to denote tasks and identifiers h_u, h_v to denote their respective heaps. An internal node of the tree is of the form $\mathsf{Par}(u, h, T_1, T_2)$, representing an internal task u with heap h, and child trees T_1 and T_2. An active leaf node, corresponding to unfinished tasks, is denoted $\mathsf{ALeaf}(u, h)$, and passive leaf nodes, corresponding to terminated tasks, are denoted $\mathsf{PLeaf}(u, h)$.

Figure 2.2 illustrates how tasks and heaps are created/merged at forks and joins in a one-to-one fashion. The larger heap (gray box) resulting from the join signifies that it contains all the locations from the children heaps and the parent heap (see Join below for more details).

2.1.4 Semantics

Our operational semantics steps a program state consisting of three components: (i) the memory store μ, (ii) a task tree T, and (iii) an expression e. We can write the semantics relation as: $\mu \,;\, T \,;\, e \rightarrow \mu' \,;\, T' \,;\, e'$. Figure 2.3 shows the rules for the semantics.

Allocation. The allocation rule Alloc extends the memory store μ with location ℓ mapped to storable s. The rule also checks that the location ℓ is "fresh", i.e., $\ell \notin \mathrm{dom}(\mu)$. It records this location in the heap h of task v, thus tracking all allocations in the heaps of respective tasks.

Parallel Evaluation. The rule Fork creates two child tasks, v and w, to evaluate the components of the parallel pair $\langle e_1 \parallel e_2 \rangle$. It creates the subtree $\mathsf{Par}(u, h_u, \mathsf{ALeaf}(v, \emptyset), \mathsf{ALeaf}(w, \emptyset))$, adding the new nodes v and w as children of node u. Initially, both the new tasks are active leaves in the tree, and their heaps are as empty sets. After the fork, rules ParL and ParR step the left and right sides of the parallel pair. The rules ParL and ParR can be interleaved non-deterministically. This non-determinism is a fundamental aspect of parallelism: when executing the program on a real system, the order in which the left and right expression of a parallel pair are evaluated depends on task scheduling decisions and other runtime artifacts.

Passive Leaves. After a leaf task finishes evaluation, the rule ActPass makes the active leaf task passive. Note that the initial expression for this step is a location ℓ, denoting that the task has finished evaluation (locations are the only irreducible form of the language). After the step, the tree node is $\mathsf{PLeaf}(u, h)$ which denotes that the task u is passive.

Join. Once both children of a task terminate, the rule Join removes the children tasks from the tree and turns the parent task into an active leaf. The rule also merges the heaps h_v and h_w of the children tasks with the parent heap h_u. We denote heap merge with the operator $+\!\!+$, which simply takes a union of the locations in the heaps. After the step, all locations from these heaps are in the extended heap h' belonging to the parent u.

$$\frac{\ell \notin \mathrm{dom}(\mu) \qquad \mu' = \mu[\ell \hookrightarrow s] \qquad h' = h, \ell}{\mu \,;\, \mathsf{ALeaf}(v, h) \,;\, s \to \mu' \,;\, \mathsf{ALeaf}(v, h') \,;\, \ell}\ \textsc{Alloc}$$

$$\frac{\mu \,;\, T \,;\, e_1 \to \mu' \,;\, T' \,;\, {e_1}'}{\mu \,;\, T \,;\, (e_1\ e_2) \to \mu' \,;\, T' \,;\, (e_1'\ e_2)}\ \textsc{ASL} \qquad \frac{\mu \,;\, T \,;\, e_2 \to \mu' \,;\, T' \,;\, {e_2}'}{\mu \,;\, T \,;\, (\ell_1\ e_2) \to \mu' \,;\, T' \,;\, (\ell_1\ e_2')}\ \textsc{ASR}$$

$$\frac{\mu(\ell_1) = \texttt{fun}\ f\ x\ \texttt{is}\ e_b}{\mu \,;\, \mathsf{ALeaf}(v, h) \,;\, (\ell_1\ \ell_2) \to \mu \,;\, \mathsf{ALeaf}(v, h) \,;\, [\ell_1, \ell_2\ /\ f, x]e_b}\ \textsc{App}$$

$$\frac{\mu \,;\, T \,;\, e \to \mu' \,;\, T' \,;\, e'}{\mu \,;\, T \,;\, (\texttt{fst}\ e) \to \mu' \,;\, T' \,;\, (\texttt{fst}\ e')}\ \textsc{FstS} \qquad \frac{\mu(\ell) = \langle \ell_1, \ell_2 \rangle}{\mu \,;\, \mathsf{ALeaf}(v, h) \,;\, (\texttt{fst}\ \ell) \to \mu \,;\, \mathsf{ALeaf}(v, h) \,;\, \ell_1}\ \textsc{Fst}$$

$$\frac{\mu \,;\, T \,;\, e \to \mu' \,;\, T' \,;\, e'}{\mu \,;\, T \,;\, (\texttt{ref}\ e) \to \mu' \,;\, T' \,;\, (\texttt{ref}\ e')}\ \textsc{RefS}$$

$$\frac{\mu \,;\, T \,;\, e \to \mu' \,;\, T' \,;\, e'}{\mu \,;\, T \,;\, (!\, e) \to \mu' \,;\, T' \,;\, (!\, e')}\ \textsc{BangS} \qquad \frac{\mu(\ell_1) = \texttt{ref}\ \ell_2}{\mu \,;\, \mathsf{ALeaf}(v, h) \,;\, (!\, \ell_1) \to \mu \,;\, \mathsf{ALeaf}(v, h) \,;\, \ell_2}\ \textsc{Bang}$$

$$\frac{\mu \,;\, T \,;\, e_1 \to \mu' \,;\, T' \,;\, {e_1}'}{\mu \,;\, T \,;\, (e_1 := e_2) \to \mu' \,;\, T' \,;\, (e_1' := e_2)}\ \textsc{USL} \qquad \frac{\mu \,;\, T \,;\, e_2 \to \mu' \,;\, T' \,;\, {e_2}'}{\mu \,;\, T \,;\, (\ell_1 := e_2) \to \mu' \,;\, T' \,;\, (\ell_1 := e_2')}\ \textsc{USR}$$

$$\frac{}{\mu_0[\ell_1 \hookrightarrow s] \,;\, \mathsf{ALeaf}(v, h) \,;\, (\ell_1 := \ell_2) \to \mu_0[\ell_1 \hookrightarrow \texttt{ref}\ \ell_2] \,;\, \mathsf{ALeaf}(v, h) \,;\, \ell_2}\ \textsc{Upd}$$

$$\frac{}{\mu \,;\, \mathsf{ALeaf}(v, h) \,;\, \ell \to \mu \,;\, \mathsf{PLeaf}(v, h) \,;\, \ell}\ \textsc{ActPass}$$

$$\frac{}{\mu \,;\, \mathsf{ALeaf}(u, h) \,;\, \langle e_1 \parallel e_2 \rangle \to \mu \,;\, \mathsf{Par}(u, h, \mathsf{ALeaf}(v, \emptyset), \mathsf{ALeaf}(w, \emptyset)) \,;\, \langle e_1 \parallel e_2 \rangle}\ \textsc{Fork}$$

$$\frac{h' = h_u +\!\!+ h_v +\!\!+ h_w}{\mu \,;\, \mathsf{Par}(u, h_u, \mathsf{PLeaf}(v, h_v), \mathsf{PLeaf}(w, h_w)) \,;\, \langle \ell_1 \parallel \ell_2 \rangle \to \mu \,;\, \mathsf{ALeaf}(u, h') \,;\, \langle \ell_1, \ell_2 \rangle}\ \textsc{Join}$$

$$\frac{\mu \,;\, T_1 \,;\, e_1 \to \mu' \,;\, T_1' \,;\, e_1'}{\mu \,;\, \mathsf{Par}(u, h, T_1, T_2) \,;\, \langle e_1 \parallel e_2 \rangle \to \mu' \,;\, \mathsf{Par}(u, h, T_1', T_2) \,;\, \langle e_1' \parallel e_2 \rangle}\ \textsc{ParL}$$

$$\frac{\mu \,;\, T_2 \,;\, e_2 \to \mu' \,;\, T_2' \,;\, e_2'}{\mu \,;\, \mathsf{Par}(u, h, T_1, T_2) \,;\, \langle e_1 \parallel e_2 \rangle \to \mu' \,;\, \mathsf{Par}(u, h, T_1, T_2') \,;\, \langle e_1 \parallel e_2' \rangle}\ \textsc{ParR}$$

Figure 2.3: Language Dynamics defining task trees and heap trees.

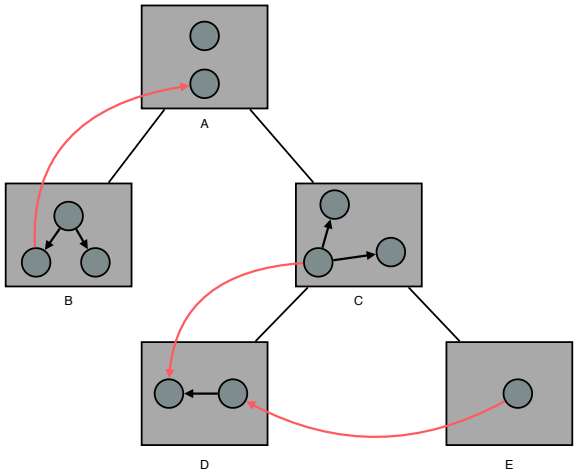

Figure 2.4: The figure shows a heap tree where the gray boxes denote heaps. The circles denote locations allocated in each heap. The arrows denote pointers. The black arrows denote internal pointers, those between locations within the same heap and the red arrows denote inter-heap pointers, those between locations across heaps.

Other rules and aspects of the language (sequential pairs, functions, and mutation) are sequential and do not alter the structure of the tree. We skip their description for sake of brevity. The evaluation of a program e starts with the state $(\emptyset\,;\,\mathsf{ALeaf}(u)\,;\,e)$, where the memory store μ is empty and the root task u is an active leaf evaluating the expression e. And the program terminates with a state $(\mu\,;\mathsf{PLeaf}(u)\,;\ell)$, where the initial expression e has been fully evaluated to location ℓ in the memory store μ and the root task u has become a passive leaf.

2.2 Heap Tree and Pointer Directions

Within a heap tree, every pointer can be classified as either *up*, *down*, *internal*, or *cross*, depending on the relative positions of locations within the heap hierarchy. In particular, consider two locations ℓ and ℓ' and their corresponding heaps $H(\ell)$ and $H(\ell')$, and suppose ℓ points to ℓ' (i.e. ℓ has a field which is a pointer to ℓ'). We classify this pointer as follows:

1. if $H(\ell)$ is a descendant of $H(\ell')$ then the pointer is an ***up-pointer***;
2. if $H(\ell)$ is an ancestor of $H(\ell')$ then it is a ***down-pointer***;
3. if $H(\ell) = H(\ell')$ then it is an ***internal pointer***;
4. otherwise, it is a ***cross-pointer***.

Figure 2.4 shows all the pointers in the heap tree with heaps labelled A, B, C, D, E. The circles denote memory locations and arrows denote pointers between them. The black arrows denote internal pointers and the red arrows denote inter-heap pointers, including an up pointer, a down pointer, and a cross pointer.

The heap tree tracks the sequential order in which locations are allocated. Locations in de-

scendant heaps are allocated "sequentially after" those in ancestor heaps, because descendant heaps are created after forks and locations in ancestor heaps are allocated before forks. Consequently, up pointers always reference older locations, while down pointers always reference newer locations. Cross pointers reference objects allocated by parallel tasks, where the relative allocation times are incomparable. Up pointers are ubiquitous in (eager) functional programs because they largely rely on immutable data structures, where new locations reference older, existing locations. Down/Cross pointers are typically created with mutable effects by modifying objects.

We analyze these inter-heap pointers more closely in Chapter 4, where we give an example of how they are created (Section 4.2.1) and also develop techniques for tracking them. For the purposes of coscheduling and this chapter, we assume that the pointers can somehow be tracked efficiently and that each heap can be independently garbage collected.

2.3 Coscheduling Tasks and Heaps

We consider executing a fork-join program on P processors, with identities $0 \leq p < P$. Each processor executes tasks and may perform garbage collection. As is typical with fork-join programs, a scheduling algorithm assigns active tasks to processors dynamically in an online fashion; each processor then executes the task that they are assigned. Thus, each active task runs on a processor and there are at most P active tasks at any moment.

To enable efficient and scalable garbage collection of the heaps in our dynamic tree, we introduce *coscheduling*. Coscheduling integrates task scheduling and memory management, using the task scheduling decisions to distribute the work of garbage collection among processors.

The crux of our coscheduling technique is a ***heap scheduler*** that dynamically partitions the heap tree into ***heap clusters*** and assigns each cluster to a processor. Each processor in turn only allocates memory in its heaps and is responsible for garbage collecting their objects. The heap scheduler assigns a ***heap cluster*** M_p to each processor p such that

- each and every heap is assigned to a processor,
- for different processors p and q, $M_p \cap M_q = \emptyset$.

By dynamically partitioning the heap tree into clusters and assigning each cluster to a processor, the heap scheduler enables each processor to independently manage the memory within its assigned cluster. This method distributes the work of garbage collection and also avoids the need for costly synchronization across processors.

The most important difference between our heap scheduler and standard task schedulers is that our scheduler must also assign suspended and passive heaps, which are not actively used for allocation but still consume memory. To ensure these heaps are garbage collected, the heap scheduler clusters them with active heaps, ensuring that each heap is assigned to some processor. However, arbitrary heap clusters do not guarantee efficiency. Our clustering strategy is carefully designed to minimize pointers between clusters by keeping parent-child heaps together and ensuring that each cluster internally represents the behavior of a sequential execution. This ensures provable efficiency with independent garbage collection, as we show

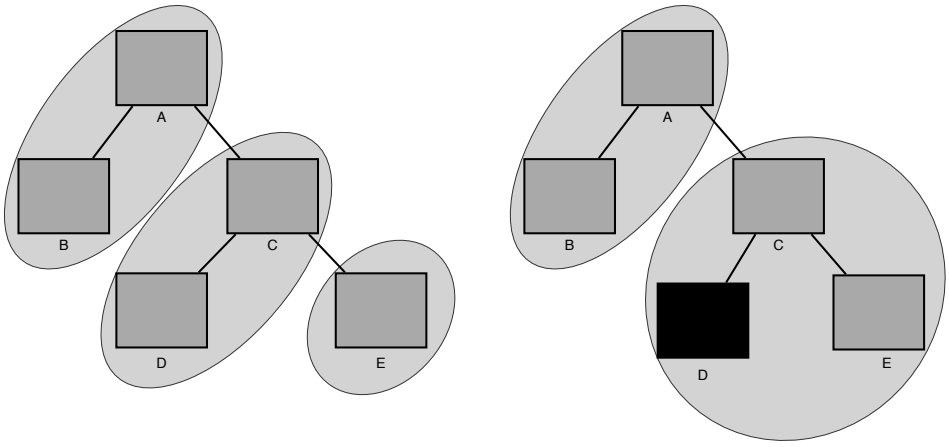

Figure 2.5: Two heap trees representing a parallel evaluation with five tasks: A, B, C, D, and E. The gray boxes denote heaps and the ellipses/bubbles denote heap clusters. The left and right trees show the heap clustering before and after task D terminates and becomes passive. When task D is active, the heap scheduler keeps heap D and its sibling E in separate clusters. After task D becomes passive, the heap scheduler puts them together, and the clustering appears as if D executed sequentially before E.

in Chapter 5. We provide a detailed overview of the clustering strategy here and dive into the specifics of the algorithm in the next subsection.

2.3.1 Overview and Examples of Heap Scheduling

Our coscheduling approach is unique in how it dynamically clusters heaps based on the parent-child relationships in the heap tree. It achieves three key things: (i) active tasks and heaps are always assigned to the same processor, ensuring that the same processor can allocate and manage the task's memory, (ii) parent heaps are always clustered with at least one of their children, ensuring that they are garbage collected together by the same processor, and (iii) each cluster internally mimics the memory behavior of a sequential execution.

Coscheduling Active Tasks and Heaps. The heap scheduler ensures that every active task and its heap are always "coscheduled" on the same processor. To do this, it follows the decisions of the task scheduler, migrating heaps between processors as the task scheduler migrates active tasks. This is crucial for efficiency because it ensures that the processor responsible for executing a task also manages its allocation and garbage collection.

Keeping Parent-Child Heaps Together. Whenever possible, the scheduler keeps closely related ancestor-descendant heaps in the same cluster, so that they are garbage collected by the

same processor. It guarantees that a parent always has (at least) one child in the same cluster, i.e., if a suspended/parent heap is in a heap cluster M_p, then one of its children is also in M_p.

By keeping parent-child heaps together, the heap scheduler directly addresses the challenges of inter-heap pointers for independent garbage collection. Most inter-heap pointers exist between a parent and its children (as explained by the disentanglement hypothesis in Chapter 3). Ensuring that a parent heap is never isolated from its children minimizes the number of pointers that cross cluster boundaries. This is important because independent garbage collection of the clusters assumes that the pointers between them are live, which can lead to accumulation of garbage objects, if not managed carefully. By clustering parent-child heaps together, we ensure that they are garbage collected together by the same processor, enabling for more effective reclamation of garbage objects.

Each cluster mimics a sequential execution. The heap scheduler ensures that the memory in each cluster mimics the memory of a sequential run of the program—each cluster resembles a segment of a sequential execution. This is important because, by comparing to a sequential execution, we can bound the space usage of each cluster.

For example, consider the two heap trees in Figure 2.5, which shows the clustering for a heap tree with five tasks/heaps A, B, C, D, and E. The figure shows two trees but we return to the right tree later in the section. In the left tree, each gray box denotes a heap and the bubbles around heaps denote heap clusters. Imagine a sequential run of the program, which executes tasks A, B, C, D, E in that order and these tasks allocate in their heaps in the same order. In a parallel execution, as shown in the example, each cluster contains heaps from a segment of our imagined sequential execution. The first cluster contains heaps A and B; the second contains C and D; the third one contains E. All of these are contiguous segments of our sequential execution, and loosely speaking the space of each cluster can be upper bounded by the space of the sequential execution.

Challenges of Passive Heaps. Passive heaps present a challenge for the heap scheduler. They correspond to tasks that have finished executing but whose siblings are still active. When active heaps become passive, they must be reassigned, because if left as is, a passive heap could get clustered with the heap of an unrelated active task which is scheduled on that processor. This would disrupt the sequential order that exists within each cluster.

To prevent this, the heap scheduler rearranges the clusters when a task terminates and becomes passive. It ensures that the passive task's heap is relocated to the same cluster as its sibling. After this rearrangement, the heap cluster appears as if the passive task executed sequentially, since both sibling's memory is at the same processor.

To illustrate this rearrangement, let's consider left and the right heap tree in Figure 2.5, which represents clusters before and after heap D task becomes passive. The black box, corresponding to heap D, represents that it is passive. After heap D becomes passive, the heap scheduler rearranges the clusters, ensuring that heaps C and D are in the same cluster as heap E. By this process it creates the heap cluster containing C, D, and E together, which a segment

of the sequential execution (A, B, C, D, E). This new cluster appears as if the tasks C and D executed sequentially before E, thus "sequentializing" the passive heap.

In summary, the heap scheduler ensures that each cluster roughly corresponds to a segment of a sequential execution. This is crucial for the proof of space bounds in Chapter 5, as it allows us to bound the space of each cluster.

2.3.2 Heap Scheduling Algorithm

In this section, we present our heap scheduling algorithm, which is a distributed algorithm implemented by each processor. The heap scheduling algorithm is integrated with a work-stealing task scheduler that assigns tasks to processors. Scheduling algorithms like work-stealing assign one active task to each processor and migrate tasks between processors by assigning each processor a double ended queue or ***deque***, which contains the tasks that the processor may execute. Figure 2.6 shows the pseudocode of our scheduling algorithm. For simplicity, the pseudocode ignores concurrency issues unlike our implementation; the concurrency details are similar to those in the work-stealing literature.

Assumed Functions. We assume the implementation of following helper modules:

1. Module `Deque` provides the type `Deque.deque` for deques and functions like `Deque.empty`, `Deque.popBottom`, `Deque.pushBottom` that are used to modify and query the deque,
2. Module `Task` provides functions like `parent`, `sibling` and `Instructions` that implement task trees
3. Module `Heap` provides similar functions for heap trees, gives a function `Merge` for merging leaf heaps with their parent, and a function `HeapOf` that returns the heap of a corresponding task. If a task does not have a heap, the function creates a new heap for it.

Additionally, the code also leaves abstract the function `stealWork`, which a processor executes to *steal* tasks from the deques of other processors. The function only returns after the processor successfully steals a task. Various stealing strategies can be used to implement this function. The `collect` function executes a collection algorithm that reclaims the unreachable locations in the input heap cluster. The collection algorithm is described later in the section.

Cluster Invariants. The task scheduler and the heap scheduler guarantee the following invariants for every processor p and its cluster M_p:

1. if a processor p is executing a task, then the heap of the task is assigned to p
2. if a suspended heap is in the cluster M_p then at least one of its children is also in M_p
3. every passive heap belongs to the same processor as its sibling.

We can establish the following useful lemma from the cluster invariants.

Lemma 1. *Every suspended heap has an active descendant heap assigned to the same processor.*

This is because every suspended heap has at least one child on the same processor (Invariant 2). The child itself is either suspended (and thus has its own child), or active (and is a leaf). If a child on the same processor is passive, then by Invariant 3, we can consider its sibling heap, which must be active or suspended on the same processor. This parent-child relationship continues until we reach a leaf node, which must be active (Invariant 1).

These invariants are integral to how the scheduler works and we use them while describing the pseudocode of the algorithm. We prove them for the algorithm in the next subsection.

Coscheduler. Figure 2.6 shows the functions each processor implements to run the heap scheduling algorithm. Each processor p maintains three pieces of state: a counter λ_p, a deque $\mathcal{R}(p)$, and a heap cluster M_p. Initially, the root task is placed into the deque of the processor 0, while other deques are empty. All the heap clusters are empty, and the counter of all the processors is zero. Each processor begins by running the function `findWork` on their own state. We describe the steps for a given processor p. The processor p first checks its deque $\mathcal{R}(p)$ for an available task. If the deque is empty, p executes the `stealWork` function, which returns only after it successfully steals a task from another processor (line 34). The processor then pops a task t from the bottom of its deque, and calls the function `coschedule` with the task t as the input.

The function `coschedule` takes an active task t as input and manages its heaps, execution, and forks/joins. It first gets the heap h_t of the task and adds the heap to the heap cluster M_p. It then calls the function `executeTask` on task t. The function `executeTask`(t) iterates through the computational instructions of the task t and executes them. Before executing each instruction, the function checks the space usage of the heap cluster on processor p and ensures that it is below a certain threshold. If not, the processor garbage collects by calling the function `collect`. This check implements our collection policy, which we describe in Section 2.4.

When the task forks or terminates, the function returns back to `coschedule` function. The function `coschedule` handles the forks and termination as follows.

Fork. In the case of `fork`, the processor p adds the right child t_2 to the bottom of the deque and then recursively calls the function `coschedule` on the left child t_1, starting its execution. The current task t becomes suspended.

Task Termination. When a task t terminates, it returns the instruction `terminate` to the function `coschedule`, which then takes steps based on whether task t is the root task (see line 13). If task t is the root task, it marks the end of the computation, and we call `exitProgram` to stop all processors. Otherwise, the task t has a sibling task t', which may be either active, suspended, passive, or still in the deque.

First, if the task t' is passive, then both the siblings have terminated and can be joined. As part of the join, the processor joins the children heaps $h_t, h_{t'}$ with their parent heap `parent(`h_t`)`. Observe that it is guaranteed that all three heaps are already assigned to the processor p: heap h_t is assigned to processor p because it corresponds to task t, which was active before this step

```
1  λ_p: int // Size of live set
2  R(p): Deque.deque // Work deque
3  M_p : heap cluster
4
5  procedure coschedule (t):
6    h_t ← HeapOf(t)
7    M_p ← M_p ∪ h_t
8    I = executeTask (t)
9    case I of
10     fork (t_1, t_2) →
11       Deque.pushBottom(t_2)
12       coschedule (t_1)
13     terminate →
14       if isRootTask (t):
15         exitProgram ()
16       t' = sibling(t)
17       if t' is passive:
18         // join heaps of tasks t and t' with their parent
19         join (h_t, h_t', parent(h_t))
20         coschedule (parent(t))
21       else if t' is suspended or active:
22         // processor q has the heap of sibling t'
23         var q : h_t' ∈ M_q
24         surrenderHeaps (q)
25         findWork ()
26       else:
27         assert (t' = Deque.getBottom (R(p)))
28         coschedule (Deque.popBottom (R(p)))
32 procedure findWork ():
33   if Deque.empty(R(p)) then
34     R(p) ← stealWork()
35   t ← Deque.popBottom(R(p))
36   coschedule(t)
37
38 procedure surrenderHeaps (q):
39   M_q ← M_q ∪ M_p
40   M_p ← ∅
41
42 procedure executeTask (t):
43   for I in Instructions(t):
44     if ( |M_p| ≥ κ · λ_p ) then
45       collect(M_p)
46       λ_p ← |M_p|
47     case I of
48       fork(t_1, t_2) →
49         return I
50       otherwise →
51           execute I
52   return terminate
```

Figure 2.6: The Coscheduling Algorithm.

(Invariant 1), heap $h_{t'}$ is assigned to processor p because it is passive (t' is passive) and our heap scheduler ensures that passive heaps and their siblings are assigned to the same processor (Invariant 2), and lastly `parent(`h_t`)` is assigned to processor p because both its children on processor p, and the heap scheduler never assigns a parent to a heap without assigning at least one of its children (Invariant 3). Thus, the `join` operation of these heaps does not interact with other processors, and the heap clusters of other processors remain unchanged. After joining the heaps, the parent task becomes active and the processor recursively calls `coschedule` to execute it.

Second, if the task t' is active or suspended, its heap must belong to a processor other than p. This is because if task t' is active, its heap can not belong to processor p, because every active task and its heap are coscheduled on the same processor (Invariant 1), and processor p is not executing task t'. If task t' were suspended, Lemma 1 requires that one of its descendants would also be active on the same processor. Since task t was the active task on processor p in the previous step, Lemma 1 implies that the heap of its (suspended) sibling t' can not be on processor p.

Given that heap of task t' is on a different processor, the processor p finds the processor q which has the heap of task t' and ***surrenders*** all its heaps to that processor. This is implemented by the function `surrenderHeaps`. The function takes the processor q as argument, which represents the processor to which the heaps are surrendered. This function transfers all the heaps of heap cluster M_p to the heap cluster M_q and reassigns the heap cluster M_p as empty (denoted $\emptyset$). This surrender operation ensures that heap h_t, which is now passive because task t has finished, is on the same processor as its sibling $h_{t'}$. After surrendering the heaps, the processor p calls function `getWork` to execute other tasks.

Third, when the sibling task t' is neither active nor suspended nor passive, the work-stealing scheduler's deque structure guarantees that task t' is at the bottom of the deque of processor p. This is because each processor only manipulates the bottom of its own deque, effectively using it as a stack (for processor-local operations), and adds the spawned tasks in order. When a task finishes, its sibling is either stolen or is the next one to be popped from the deque for execution. We assert this property, which is established for work-stealing, on line 27. The function pops the sibling from the bottom of the deque and calls itself recursively to execute it.

Then it proceeds as follows: at the start of each step, the processor checks if it needs to collect M_p. If the size of M_p is more than κ times the counter, the processor executes the `collect` function and updates its local counter.

Otherwise, $|M_p|$ is within limits, and the processor p executes an instruction of the task that it is working on.

Instructions other than `fork` and `join` match with the `otherwise` case of the pseudo code. These are simply executed without any updates to the state of the processor. Thus, if the processor p does not change the task it is working on, no changes are made to M_p except if some other processor surrenders and synchronizes with it.

2.3.3 Proof of the Cluster Invariants

In this section, we prove that our coscheduling algorithm maintains the cluster invariants we stated in Section 2.3.2. The invariants state the following for each processor p with heap cluster M_p:

1. if a processor p is executing an active task, then the heap of the task is assigned to p
2. if a suspended heap is in the cluster M_p then at least one of its children is also in M_p
3. every passive heap belongs to the same processor as its sibling.

Like our algorithm, we ignore the concurrency of these steps to simplify the analysis. Additionally, we prove that the invariants before each call to the function executeTask. They may not hold in transition, when the processors are reclustering their heaps We prove Invariant 1 directly and then use induction to prove Invariants 2 and 3.

Proof of Invariant 1.

Proof. This invariant holds by construction. Before a processor p calls the function `executeTask` to execute an active task t, it adds the task's heap to its heap cluster. For all subsequent steps. the heap stays on the processor p, until the task terminates, at which point the task is no longer active. This is because (i) the actions of the task itself do not change the heap cluster, and (ii) other processors can only add to the heap cluster of processor p by surrendering to it but they never steal any heap from it. Thus, the only way a heap leaves a processor is when the processor itself surrenders, or merges heaps, which only happens when the task terminates. □

Proofs of Invariants 2 and 3.

Proof. We prove the invariants by induction on the number of steps. We assume that only one processor performs actions at each step. The invariants hold trivially at the beginning, when no steps have been taken, because no processor is executing a task and there are no heaps.

Next, we consider the nth step that a given processor p takes. If the processor executes a program instruction (in function `executeTask`), then no clusters change, and the invariants continue to hold. If the processor executes a `fork` instruction in function `coscheduler`, then the task t becomes suspended, the processor adds the right child task t_2 to its deque, and then schedules the left task t_1 on itself. Thus, Invariant 2 holds after a fork for processor p, because the new suspended task t, has a child task t_1, whose heap is assigned to the same processor by the recursive call to `coschedule`. The invariant continues to hold for other processors, because their clusters are unchanged. Invariant 3 also continues to hold because no passive heaps are created or reassigned.

Now, let's consider the case where the active task t executes the `terminate` instruction. If the task t is a root task, then the program has finished and the invariants hold. Otherwise, if the terminating task t has a sibling task t', then there are three cases. The cases consider the status the sibling task t'.

Task t' is passive. If the task t' is passive, the function `coscheduler` joins the heaps of the siblings with their parent, and recursively coschedules the parent task. Observe that it is guaranteed that all the three joined heaps are already assigned to the processor p: heap h_t is assigned to processor p because it corresponds to task t, which was active before this step (Invariant 1), heap $h_{t'}$ is assigned to processor p because it is passive (t' is passive) and our heap scheduler ensures that passive heaps and their siblings are assigned to the same processor (Invariant 2), and lastly `parent`(h_t) is assigned to processor p because both its children on processor p, and the heap scheduler never assigns a parent to a heap without assigning at least one of its children (Invariant 3). We have we established that all these three heaps are on the same processor p and the merge does not affect any heap cluster other than processor p. Thus, this satisfies the Invariants 2 and 3, because in this step, a suspended heap (corresponding to the parent) has become active, and a passive heap has been removed (corresponding to the sibling), both of which do not affect the invariants, since they are on the same processor.

Task t' is active or suspended. If the task t' is suspended or active, then the processor p surrenders all its heap to a processor q which has the heap of task t'. The surrendering is implemented by function `surrenderHeaps` in Figure 2.6. Let M_p and M_q be the heap clusters on processors p and q before this step and let M'_p and M'_q be the corresponding clusters after the step. The invariants trivially hold for cluster M'_p because it is empty after the step. Consider a heap $h \in M'_q$; note that M'_q is the union of clusters M_p and M_q, If the heap h is in $M'_q \cap M_q$, then both invariants hold for it, because all the heaps in M_q are in M'_q, and thus the invariants continue to hold.

For a heap $h \in M'_q \cap M_p$, we take three cases. First, if the heap is suspended, then it has a descendant in M_p by Invariant 2 on the previous step. Because all heaps in M_p are now in M'_q, the Invariant 2 continues to hold. Second, if the heap is corresponds to the newly passive task t, then its sibling is in M'_q by construction, because the processor q was chosen because it had the sibling heap. Third, if the heap is passive, and does not correspond to the task t, then it was passive before this step, and by Invariant 3, its sibling must also be in M_p and M'_q.

Task t' is still in the deque If the task t' is neither active, nor passive, nor suspended, then the task is in the deque. Given that our task scheduler is work-stealing, we assume in this case that the task is at the bottom of the deque. This case is straightforward, because the processor p executes the sibling task t' and adds its heap to the heap cluster M_p. This addition ensures that Invariant 3 holds after the step, because the sibling of now passive heap of task t is on the processor. □

2.4 Collection Policy

Each processor manages its assigned heaps and decides when to perform garbage collection independently. This decision is governed by a fully distributed collection policy, implemented on all processors.

Processor p maintains a local counter λ_p, which tracks the amount of memory that survived its last collection. This counter serves as an estimate of the maximum live data within the processor's heap cluster, M_p. The processor ensures that the space remains within a constant factor, $\kappa > 1$ of this estimate, i.e., $|M_p| < \kappa \cdot \lambda_p$. When $|M_p|$ exceeds this threshold, the processor triggers a garbage collection of its heaps, identifying and reclaiming unreachable locations. After the collection is completed, the counter λ_p is reset to the new size of M_p, and the processor resumes its assigned task.

To identify the reachable locations, processor p assembles the roots for its heaps, including both the standard program roots (e.g., registers, program stack) and inter-heap pointers from other processors' heaps. These inter-heap pointers introduce dependencies that must be considered for safe, independent garbage collection. Our memory manager achieves this by accounting for all inter-heap pointers, allowing any heap to be garbage-collected independently, without tracing other heaps. We describe the details of our memory manager in Chapter 4.

3

Disentanglement Hypothesis

In Chapter 2, we described our memory manager partitions memory into a tree of task-local heaps, and then uses the coscheduling algorithm to schedule the heaps on processors. The coscheduling algorithm assumes, however, that each heap of the tree can be garbage collected independently. Given the many decades of research on garbage collection, there are numerous possible designs and techniques that could be used to achieve this. In our memory manager, the guiding principle for all the design decisions is the *disentanglement hypothesis*.

In this section, we propose and give evidence for the disentanglement hypothesis. The hypothesis distinguishes between two kinds of objects, *disentangled* and *entangled*, based on relative relationships of tasks that allocate them and tasks that access them. We say that two tasks of the task tree are *concurrent* if they are not in an ancestor-descendant relationship; if they are, then we call them *sequentially dependent*. We classify objects into the following two categories:

- **Entangled objects**: Allocated by one task and accessed by concurrently executing tasks.
- **Disentangled objects**: Accessed only by tasks that are sequentially dependent on the task that allocated them, not by concurrent tasks.

The ***disentanglement hypothesis*** states that most objects in a parallel fork-join program are disentangled. The hypothesis is supported by a key theoretical result proving that all objects in a race-free program are disentangled [174, 178]. As we show, an object gets entangled only if it participates in determinacy races [19]. Because races typically lead to correctness bugs [8, 10, 51, 52, 54, 75, 117, 130, 166], they are rare in parallel programs, leading to the hypothesis that most objects in a fork-join parallel programs are disentangled.

In this chapter, we define our notion of object-level disentanglement and then present a semantics that classifies each object to be entangled and disentangled. The semantics produces

two cost metrics ***entanglement factor*** and ***entanglement ceiling***, which quantify the amount of entanglement for an execution of the program. We then formalize the disentanglement hypothesis, based on the metrics, and analyze a wide class of parallel algorithms, illustrating why the hypothesis holds for them. In Chapter 7, we give evidence for the hypothesis empirically, by measuring the number/size of entangled objects for a wide variety of benchmarks.

3.1 Disentanglement

Westrick et al. [178] defined disentanglement as a property of programs where concurrent tasks **never** access each other's allocations. The intuition for disentanglement is that in many parallel programs, concurrent tasks execute independently from each other and avoid side-effecting memory objects that may be accessed by others. We generalize this intuition by introducing the concept of object-level disentanglement. At a high level, an object is disentangled if it is not accessed by any concurrent task. The key difference here is that we treat disentanglement as a continuous property of memory objects as opposed to prior treatment of disentanglement as an all-or-none property of programs.

One of the benefits of defining disentanglement on a per-object basis is that it enables us to reason about programs that have entangled objects. Many interesting parallel programs create entangled objects by using various forms of concurrency to communicate between tasks. For example, consider a shared hash table that is accessed by many tasks concurrently. As tasks insert and retrieve elements of the hash table, they may read objects allocated by concurrent tasks. Such objects, accessed by concurrent tasks, become entangled.

Figure 3.1 illustrates the hash table example with a heap tree containing heaps A, B, and C. Recall that each heap contains the allocations performed by its corresponding task. In this tree, the parent task A is suspended and the children task B and C are executing concurrently. The figure shows a hashtable `ht` in heap A containing the key `k` allocated in heap B, as shown by the pointer from `ht` to `k`. If task C accesses the hash table and its contents, including the key `k`, then the key becomes entangled (denoted by the orange highlight), because tasks B and C are concurrent, and task C is accessing an allocation performed by task B.

Note, however, an entangled object does not remain entangled throughout the execution. When tasks join, objects that are entangled between them become disentangled. Thus, the classification of objects as disentangled/entangled changes throughout the execution. Figure 3.1 shows the heap tree after tasks B and C join. After the join, the heaps of A, B and C are merged together to create the heap $A +\!\!+ B +\!\!+ C$. As a result of the join, the key `k` becomes disentangled.

In the next section, we formalize the notion of object-level disentanglement with a formal semantics that tracks and quantifies entanglement in parallel programs.

3.2 Entanglement Semantics

We introduce a language semantics to classify each memory object as either entangled or disentangled. The semantics identifies entangled objects by tracking the threads forked by the

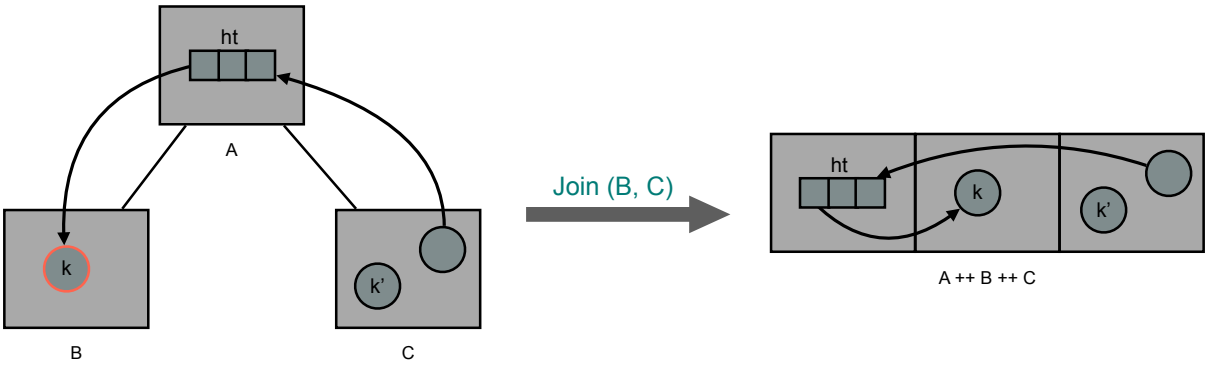

Figure 3.1: The example shows a heap tree along with allocations performed by the corresponding tasks A, B, and C. The arrows denote pointers between the allocations. The hash table `ht` in heap A contains key `k` allocated in heap B. If task C reads the hash table and its contents, it would access the key `k`, making the key entangled, as denoted by the orange highlighting around k. But, this is temporary. After tasks B and C join, we merge their heaps with the parent A to create the heap $A +\!\!+ B +\!\!+ C$. Since tasks B and C have joined, we don't consider them concurrent, and the key `k` becomes disentangled. We denote this by removing the orange highlight from key `k`. Thus, whether an object is disentangled/entangled may change as the execution proceeds.

program and observing their memory actions. When threads join, the semantics resolves any entanglement that they created and updates the corresponding objects to be disentangled. The semantics also quantifies the amount of entanglement by computing two cost metrics: an ***entanglement factor***, ϵ, which quantifies the total amount (size) of entangled objects, and an ***entanglement ceiling***, δ, which tracks the peak amount (size) of entangled objects at any step.

We use the cost metrics to formalize the disentanglement hypothesis and reason about the overhead of our memory manager (Chapter 4). We note that our semantics only observes accesses on mutable objects and does not track accesses on immutable objects. This is motivated by a practical concern: we want to implement the semantics and identify entangled objects. Not tracking accesses to immutable objects allows us to skip read barriers on immutable objects, which is crucial for practical performance. This is because a majority of reads in a typical functional program are for immutable objects and we want to minimize that overhead.

3.2.1 Syntax and Task Trees

Much of the language syntax and semantics is similar to our language in Chapter 2. We revisit the important details below.

Our language contains parallel pairs $\langle e_1 \parallel e_2 \rangle$ where e_1 and e_2 may execute in parallel. The language tracks memory operations by defining ***storables*** s, which are allocated in memory, and ***memory locations*** ℓ, which are indices into the memory. Locations are mapped to storables

$$
\begin{array}{rlll}
\textit{Variables} & x, f & & \\
\textit{Numbers} & m & \in & \mathbb{N} \\
\textit{Types} & \tau & ::= & \texttt{nat} \mid \tau \times \tau \mid \tau \to \tau \mid \tau\ \texttt{ref} \\
\textit{Memory Locations} & \ell & & \\
\textit{Storables} & s & ::= & m \mid \texttt{fun}\ f\ x\ \texttt{is}\ e \mid \langle \ell, \ell \rangle \mid \texttt{ref}\ \ell \\
\textit{Expressions} & e & ::= & \ell \mid s \mid x \mid e\ e \mid \langle e, e \rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{ref}\ e \mid\ !e \mid e := e \mid \langle e \parallel e \rangle \\
\textit{Memory} & \mu & \in & \textit{Locations} \rightharpoonup \textit{Storables} \\
\textit{Task Identifiers} & u, v & & \\
\textit{Task Tree} & T & ::= & \mathsf{ALeaf}(u) \mid \mathsf{PLeaf}(u) \mid \mathsf{Par}(u, T, T)
\end{array}
$$

Figure 3.2: Syntax

in the memory store μ. Locations are the only irreducible form of the language.

Locations that are mapped to mutable references are called ***mutable locations*** and those that are mapped to other types of storables are called ***immutable locations***. We say that a location ℓ has a pointer to another location ℓ' if the storable at location ℓ mentions ℓ'. The pointer is mutable if the location ℓ is mutable and otherwise, if the location ℓ is immutable, its pointers are immutable.

The semantics tracks the structure of parallelism using a ***task tree***. The task tree arranges the tasks of the program with vertices that represent tasks and edges that express parent/child relationships between tasks. When a task forks, we add two children to the task tree, and when it joins we remove the children from the task tree. We use identifiers u, v to represent tasks. Two tasks u and v are ***concurrent*** if neither is an ancestor of the other, denoted $\mathsf{concurrent}(u, v)$.

3.2.2 Entanglement Sources, Regions, and Cost Metrics

Tasks create entanglement when they read locations allocated by other concurrent tasks. To track entanglement, the semantics intercepts the reads of mutable references and identifies *entanglement sources*. Each entanglement source causes a region of memory to become entangled.

Entanglement sources. A memory location ℓ becomes an ***entanglement source*** when:

- A task u performs a mutable read (dereference) and obtains ℓ as the result, and
- The task u is concurrent with the task that allocated ℓ.

To track entanglement sources, the semantics maintains an ***allocator map*** α, mapping each location to the task that allocates it, and a ***reader history*** H, mapping each location to a set of tasks that performed a dereference operation resulting in that location. Given a task tree T, reader history H, and allocation map α, the ***set of entanglement sources***, $\mathcal{E}(H)$, is:

$$\mathcal{E}(H) = \{\ell \mid \exists\ u \in H(\ell) : \mathsf{concurrent}(u, \alpha(\ell))\},$$

where the relation $\mathsf{concurrent}(u, v)$ holds if neither u nor v is an ancestor of the other in tree T. This definition states that for an entanglement source ℓ, there is a task u, which is in the

reader history of location ℓ. This means that task u has performed a mutable dereference on some location to obtain location ℓ. Furthermore, the location ℓ was allocated by task $\alpha(\ell)$, and that task is concurrent to task u.

Entanglement region. An entanglement source affects its vicinity: a task can use the entanglement source to read locations that are reachable from it and create more entanglement. If the source is mutable, such reads will be accounted by the reader history. However, the reader history does not track reads of immutable locations.

To account for this, we associate with each entanglement source an ***entanglement region*** which contains all locations reachable from the source using immutable pointers only. We can formally define it as follows: let $\mathsf{out}(\ell)$ be the set of locations to which location ℓ points to. Then the entanglement region of location ℓ, written $\mathsf{er}(\ell)$, is defined by the recurrence:

$$\mathsf{er}(\ell) = \begin{cases} \{\ell\} \cup \bigcup_{\ell' \in \mathsf{out}(\ell)} \mathsf{er}(\ell'), & \text{if } \ell \text{ is immutable} \\ \{\ell\} & \text{if } \ell \text{ is mutable} \end{cases}$$

If a location ℓ belongs to an entanglement region, we say that ℓ is ***entangled***.

Cost metrics. To quantify the amount of entanglement, the semantics calculates entanglement factor and entanglement ceiling. The ***entanglement factor***, ϵ, accumulates the amount of memory that becomes entangled throughout an execution. The ***entanglement ceiling***, δ, tracks the maximum amount of entangled objects at any point of the execution. The ceiling provides a reference for the peak amount of entanglement by accounting for the fact that this amount decreases when tasks join. As an example, consider two parallel subcomputations whose executions have entanglement factors ϵ_1, ϵ_2 and ceilings δ_1, δ_2. Then for their sequential composition, the entanglement factor is $\epsilon_1 + \epsilon_2$ and the ceiling is $\max(\delta_1, \delta_2)$. Because the first computation finishes before the second begins, its entanglement is resolved and does not affect the second. The entanglement ceiling accounts for this resolution.

3.2.3 Semantics

Our semantics, presented in Figure 3.3, is a transition relation that steps a program state consisting of six components: allocator map α, reader history H, memory μ, entanglement factor ϵ, task tree T, and expression e. The transition relation has the form: $(\alpha\ ;H\ ;\mu\ ;\epsilon\ ;T\ ;e) \rightarrow (\alpha'\ ;H'\ ;\mu'\ ;\epsilon'\ ;T'\ ;e')$.

Fork and parallel evaluation. The rule Fork (see Figure 3.3) forks two children, v and w to evaluate the components of the parallel pair $\langle e_1 \parallel e_2 \rangle$. It creates the par-node $\mathsf{Par}(u, \mathsf{ALeaf}(v), \mathsf{ALeaf}(w))$. Then, the rule ParL steps the left side of the parallel pair e_1 with subtree T_1. The rule ParR for stepping the right side is similar. Rules ParL and ParR may interleave non-deterministically.

$$\frac{\ell \notin \mathrm{dom}(\mu) \qquad \mu' = \mu[\ell \hookrightarrow s] \qquad \alpha' = \alpha[\ell \hookrightarrow v]}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{ALeaf}(v) \,;\, s \to \alpha' \,;\, \epsilon \,;\, H \,;\, \mu' \,;\, \mathsf{ALeaf}(v) \,;\, \ell} \textsc{Alloc}$$

$$\frac{\mu(\ell_1) = \texttt{fun}\ f\ x\ \texttt{is}\ e_b}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{ALeaf}(v) \,;\, (\ell_1\ \ell_2) \to \alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{ALeaf}(v) \,;\, [\ell_1, \ell_2\ /\ f, x]e_b} \textsc{App}$$

$$\frac{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T \,;\, e \to \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T' \,;\, e'}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T \,;\, (\texttt{ref}\ e) \to \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T' \,;\, (\texttt{ref}\ e')} \textsc{RefS}$$

$$\frac{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T \,;\, e \to \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T' \,;\, e'}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T \,;\, (!\, e) \to \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T' \,;\, (!\, e')} \textsc{BangS}$$

$$\frac{\mu(\ell) = \texttt{ref}\ \ell' \qquad H' = H[\ell' \hookrightarrow H(\ell') \cup \{v\}] \qquad \epsilon' = \epsilon + |\mathsf{er}(\mathcal{E}(H')) \setminus \mathsf{er}(\mathcal{E}(H))|}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{ALeaf}(v) \,;\, (!\, \ell) \to \alpha \,;\, \epsilon \,;\, H' \,;\, \mu \,;\, \mathsf{ALeaf}(v) \,;\, \ell'} \textsc{Bang}$$

$$\frac{}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu_0[\ell \hookrightarrow \texttt{ref}\ \ell''] \,;\, \mathsf{ALeaf}(v) \,;\, (\ell := \ell') \to \alpha \,;\, \epsilon \,;\, H' \,;\, \mu_0[\ell \hookrightarrow \texttt{ref}\ \ell'] \,;\, \mathsf{ALeaf}(v) \,;\, \ell'} \textsc{Upd}$$

$$\frac{}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{ALeaf}(u) \,;\, \langle e_1 \parallel e_2 \rangle \to \alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Par}(u, \mathsf{ALeaf}(v), \mathsf{ALeaf}(w)) \,;\, \langle e_1 \parallel e_2 \rangle} \textsc{Fork}$$

$$\frac{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T_1 \,;\, e_1 \to \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T_1' \,;\, e_1'}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Par}(v, T_1, T_2) \,;\, \langle e_1 \parallel e_2 \rangle \to \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, \mathsf{Par}(v, T_1', T_2) \,;\, \langle e_1' \parallel e_2 \rangle} \textsc{ParL}$$

$$\frac{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T_2 \,;\, e_2 \to \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T_2' \,;\, e_2'}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Par}(v, T_1, T_2) \,;\, \langle e_1 \parallel e_2 \rangle \to \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, \mathsf{Par}(v, T_1, T_2') \,;\, \langle e_1 \parallel e_2' \rangle} \textsc{ParR}$$

$$\frac{\alpha' = \alpha\{u/v\}\{u/w\} \qquad H' = H\{u/v\}\{u/w\}}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Par}(u, \mathsf{ALeaf}(v), \mathsf{ALeaf}(w)) \,;\, \langle \ell_1 \parallel \ell_2 \rangle \to \alpha' \,;\, \epsilon \,;\, H' \,;\, \mu \,;\, \mathsf{ALeaf}(u) \,;\, \langle \ell_1, \ell_2 \rangle} \textsc{Join}$$

Figure 3.3: Language Dynamics.

Allocation. The rule Alloc (see Figure 3.3) shows how storables are allocated. The rule steps a storable s to a fresh location ℓ ($\ell \notin \mathrm{dom}(\mu)$), and extends the memory store μ with location ℓ mapped to storable s. It assigns the location its allocator task in the allocator map α.

Mutable reads and entanglement factor. The rules RefS and BangS step $\texttt{ref}\ e$ and $!\, e$ respectively by stepping the sub expression e.

Rule Bang performs the dereference operation and updates the reader history and the entanglement factor. When a task v dereferences a mutable location ℓ to read a location ℓ', the rule adds the reader v to the reader history of location ℓ'. Because this dereference may increase the amount of entanglement, the rule updates the entanglement factor. To update the

entanglement factor, the rule first identifies locations entangled because of this read. We can write the set of entangled locations as $\cup_{\ell \in \mathcal{E}(H)} \mathsf{er}(\ell)$, where er computes the region for a source, H is a reader history, and $\mathcal{E}(H)$ contains all the sources for history H. This set is the union of entanglement regions of all the entanglement sources. We write this concisely as $\mathsf{er}(\mathcal{E}(H))$.

The rule computes the set difference $\mathsf{er}(\mathcal{E}(H')) \setminus \mathsf{er}(\mathcal{E}(H))$ because the difference represents all entangled locations in the new (after the read) reader history H' that were not entangled in the old (before the read) history H. The rule increases the entanglement factor by the size of this set, which it computes with $|.|$ (see Figure 3.3).

Mutable updates. Rule Upd creates a pointer from a mutable location ℓ to a location ℓ'. Mutable updates do not affect entanglement because the task already can access both locations.

Join. Once a parallel pair is fully evaluated, we turn it into a sequential tuple. When this happens, we remove the children from the tree and as a result, the parent becomes the leaf. The rule Join replaces the par-node $\mathsf{Par}(u, \mathsf{ALeaf}(v), \mathsf{ALeaf}(w))$ with the leaf node $\mathsf{ALeaf}(u)$. Because the children have terminated, we transfer the ownership of the allocations of the children to the parent. The Join rule does this by substituting task u for the children tasks v and w in the allocator map α.

After the join, the locations entangled between children tasks v and w are no longer entangled because tasks v and w are no longer concurrent (they have joined). To reflect this in the program state, the Join rule remaps the accesses of the children tasks (v and w) to the parent task (u). In the semantics, the rule modifies the reader history and substitutes task u for tasks v and w. The modified reader history H' is equal to $H\{u/v\}\{u/w\}$.

Entanglement ceiling. Entanglement ceiling quantifies the maximum amount of memory reachable from entanglement sources at any step of an execution. Formally, suppose an execution starts with state S_0, proceeds as $S_0 \rightarrow S_1 \ldots S_n$, and finishes with state S_n. Then, the entanglement ceiling, represented as δ, for this execution is $\max_i |\mu^+(\mathcal{E}(H_i))|$, where $\mathcal{E}(H_i)$ is the set of entanglement sources for history H_i and μ^+ computes memory reachable from them. Entanglement ceiling bounds the space cost incurred by our memory manager for managing entanglement, as the memory manager preserves all memory reachable from entanglement sources.

Other rules. Other rules are standard and we skip them for brevity. The evaluation begins from the state $(\emptyset \ ; \emptyset \ ; \emptyset \ ; 0 \ ; \mathsf{ALeaf}(v) \ ; e)$ where the allocator map α, the reader history H, and the memory μ are empty, the task tree is $\mathsf{ALeaf}(v)$, the entanglement factor is 0, and e is the program. At the end of evaluation, the program state contains the result and also the entanglement factor of the execution.

3.3 Evidence for the Disentanglement Hypothesis

The ***disentanglement hypothesis*** states that most objects in a parallel fork-join program are disentangled. Specifically, for any given execution of a program, both the entanglement factor ϵ and the entanglement ceiling δ are a small fraction of the total memory used. While the hypothesis may not hold universally for all fork-join programs, theoretical results and empirical evidence show that it holds for many parallel programs.

In the rest of this section, we consider several classes of programs written in our language MPL and describe which of them have entangled objects and their extent. We include examples ranging from simple to sophisticated parallel algorithms, demonstrating that the disentanglement hypothesis holds for a wide range of programs including those that utilize concurrent data structures for parallelism. We have measured the entanglement factor ϵ of these examples and many other benchmarks, and in all of them the entanglement factor is much smaller ($< 1\%$) than the memory footprint (see Chapter 7 for numbers).

3.3.1 Deterministic Programs

First, we observe that all purely functional programs satisfy the disentanglement hypothesis because they do not have entangled objects by construction. Entangled objects/sources are only created by mutable reads/writes, which are absent in purely functional programs. Consequently, purely functional programs are naturally race-free and deterministic, making them a reliable medium for expressing parallel programs. A wide range of collections such as sets, maps, and sequences, and operations such as union, filter, intersection, reduction, and range queries can be expressed purely functionally with excellent parallel performance [68, 167, 178]. All of these satisfy the disentanglement hypothesis.

As a concrete example, Figure 3.4 shows the code for the `reduce` function that "sums" an array of elements with a given binary operator in parallel. The code first checks if the array is empty, and if so returns the identity value of the input function `f`. Otherwise, it recursively evaluates two halves of the array in parallel, as denoted by the parallel operator ||.

Beyond purely functional programs, many programs that utilize mutable state for performance benefits also satisfy the disentanglement hypothesis. Westrick et al. [178] showed that a program free from determinacy races—where two concurrent tasks access the same memory location, with at least one modifying it—satisfies program level disentanglement. Such programs do not have entangled objects and thus adhere to the disentanglement hypothesis. [1]

Determinacy race free programs are guaranteed to be deterministic [78]. Many programs use mutation in a deterministic fashion, including those with array-based sequences (lists), that allow for mutable updates, and support common operations like map, filter, and reduce

[1]Note that their formal definition of disentanglement is different from ours as they define it as a program property, whereas we define it on a per-object basis. Formally, we must show that our definitions are semantically equivalent for disentangled programs to use their result. But adapting their result for our setting is relatively straightforward.

```
fun reduce f id a =
  if length(a) = 0 then id
  else
    let mid = length(a) / 2
        (left, right) = ⟨reduce f id a[0 .. mid] || reduce f id a[mid .. ]⟩
    in
      f (left, right)
    end
```

Figure 3.4: Parallel `reduce` function takes a binary operation `f`, its identity `id`, and an array and computes the "sum", `f(a[0], f(a[1], ... f(a[n-2], a[n-1]) ... ))`.

in a deterministic fashion. This is often achieved by ensuring that parallel tasks operate on separate, non-overlapping array sections [178].

3.3.2 Nondeterministic Programs Without Entanglement

Many irregular algorithms use atomic operations, typically implemented with mutable effects, to mediate interaction between parallel tasks. One such operation is the "compare and swap" (a.k.a., "CAS") operation that performs an atomic non-blocking read-modify-write on a mutable cell. The motivation for these operations comes from the practice of parallel programming, where many algorithms and implementations use nondeterminism for efficiency reasons. CAS operations, when applied on *unboxed values* like integers or floats, which are not heap-allocated, do not lead to entangled objects (because there is no sharing of allocations between tasks). Thus, programs using such operations on unboxed values satisfy the disentanglement hypothesis.

As an example of a class of algorithms that use CAS operations for improved efficiency, consider the classic parallel breadth-first-search that takes a graph and a source vertex and traverses the graph from the source, visiting each level of the graph in parallel. The BFS algorithm uses a CAS operation on a mutable cell in an array to ensure that each vertex is visited exactly once, preventing repeated work. The signature of the CAS operation is

`CAS : (`α `Array * int) →` $(\alpha * \alpha)$ `→ bool`.

It takes an array, the position of a cell, the expected value v_e, and the new value v_n, and atomically swaps the value v of the cell with the new value if v is equal to the expected value v_e and does nothing otherwise (v is different from v_e).

Figure 3.5 shows the MPL code for such a BFS. The workhorse is the recursive `search` function that takes the `graph`, an array of `visits`, and a `frontier` as an argument. Each invocation of the `search` visits all the vertices on a specific level. The `frontier` is the array of vertices to be visited and the array `visits` indicates completed visits: `visits[u] = -1` if the vertex is not visited and `visits[u] = u` if it is visited. To visit the vertices in the frontier, the algorithm parallel-maps a `visit` function over the frontier. The `visit` function takes each outgoing edge of a vertex and attempts to "claim" the target of the edge with compare-and-swap,

```
fun search(graph, visits, frontier) =
  if length(frontier) = 0 then
    visits
  else let
    fun visit(v) =
      filter (fn u ⇒
              CAS (visits, u, (-1, v)))
              (neighbors(graph, v))
    frontier =
      flatten(map(visit, frontier))
  in
    search (graph, visits, frontier)
  end

fun bfs(graph, source) =
  let
    fun init(i) =
      if i = source then source
      else -1
    n = numVertices(graph)
    visits = tabulate(init, n)
    frontier = singleton(s)
  in
    search (graph, visits, frontier)
  end
```

Figure 3.5: The implementation of non-deterministic BFS.

`CAS`, operation, and returns the array of neighbors that it has successfully claimed. The search function constructs, by using the `flatten` function, the new frontier to include the vertices claimed by the visit function.

Because the BFS algorithm visits each reachable vertex and edge once, its work and space is bounded by the size of the graph. Because the algorithm uses compare-and-swap operations on shared vertices, it is non-deterministic: different executions of BFS could yield different executions. Notice, however, that the races only involve unboxed integer values, and involve no memory allocations that are shared between tasks, thus avoiding any entangled objects.

Key Takeaway. Nondeterministic programs that use concurrent mutable operations, such as compare-and-swap, on small, word-size, values of primitive types (e.g., int, and bool) do not create entangled objects. Thus, they satisfy the disentanglement hypothesis.

```
type α ht = {T: α option array, h: α → int}
fun insert ({T, h}: α ht, k: α) : unit =
  let fun loop i =
    if (T[i] = SOME k) then ()
    else if (T[i] = NONE and (CAS (T, i) (NONE, SOME k))) then ()
    else loop ((i + 1) mod |T|)
  in loop (h(k)) end
```

Figure 3.6: Concurrent hash tables: `insert` function.

3.3.3 Entangled Programs

The parallel breadth-first-search example above illustrates a very simple concurrent data structure consisting an array of atomic cells, each of which allows atomic read and write operations (implemented via compare-and-swap). More broadly, concurrent data structures, such as hash tables and lists, are essential for efficiency in many parallel algorithms. Although these data structures contain heap-allocated objects that are entangled because they are shared between tasks, the total amount of entangled memory in the entire program often remains low. Let's examine this using a concurrent hash table in the context of a graph algorithm.

Concurrent Hashing. Figure 3.6 shows a hash table type α `ht`, which is a record containing an array `T` and a hash function `h`. The `insert` function uses linear probing to insert key `k` into hash table `{T, h}`: it starts at the index `h(k)` and loops until it finds an empty cell in the array `T`. When it finds an empty cell, it attempts to insert the key using a compare-and-swap (`CAS`) to atomically update the empty cell; the attempt may fail if another task concurrently performs an insertion, in which case the function probes for the next empty cell. We assume for simplicity that the hash table has sufficient space.

Since many concurrent tasks access the keys of the hash table, they make the keys entangled. For example, when a task probes for an empty slot, it reads the keys inserted by other tasks, making them entangled if they are inserted by other concurrent tasks.

The hash table is useful for many parallel algorithms, including for memoization and deduplication. Let's examine how we can use a hash table for *deduplication*, i.e., removing duplicates from an array of keys. Given an array, we can fork parallel tasks that attempt to insert the keys into the hash table. The hash table guarantees that the attempt only succeeds if the key has not been inserted previously, thereby removing duplicates. For this algorithm, the entanglement factor (amount of entanglement) is bounded by the number of unique keys—only those keys that are successfully inserted into the hash table can be read by other tasks. Thus, for a sequence of n keys with r unique keys, the entanglement factor is bounded by $O(r) \in O(n)$. While this can be theoretically high, especially in the case where all elements are unique, deduplication is typically used within a larger parallel algorithm that makes this entanglement factor much smaller in comparison, particularly when there are a lot of duplicates. We consider an example

of this below.

Graph Reachability As another parallel program that uses concurrent hash tables, we consider parallel graph reachability for undirected graphs. Figure 3.7 shows the code for the `reach` function, which labels each vertex of the graph such that two vertices have the same label if they are in the same connected component. The function works in rounds: each round decomposes the graph into clusters using a low-diameter decomposition function `ldd` and then contracts the graph using function `contract`. The `ldd` function takes a parameter $\beta < 1$ and returns a $(\beta, O(\frac{\log n}{\beta}))$-decomposition of the graph, meaning it partitions the graph into clusters, where each cluster has a diameter of $O(\frac{\log n}{\beta})$ and the number of edges between clusters is at most a fraction β of the total edges m.

After the decomposition, the `reach` function calls `contract`, which collapses each cluster into a single vertex, removing edges within each cluster, and deduplicating edges between clusters. This contraction step simplifies the graph, making it smaller and easier for computing reachability. Since each round reduces the number of edges by a fraction of β, the `reach` function terminates after a logarithmic number of rounds.

```
1  (* β is a floating-point number in (0, 1) *)
2  fun reach(V, E) =
3  let  L = ldd(G, β)
4       (V', E') = contract ((V, E), L)
5  in
6    if |E'| = 0 then
7      L
8    else
9      let L' = reach(V', E')
10     in { v ↦ L'[L[v]] |  v in V }
11     end
12 end
```

Figure 3.7: The code shows a MPL program for computing graph reachability. The program calls functions ldd and contract repeatedly, but each time with a smaller input. Theoretically, the worst-case entanglement ceiling of reach is $O(\beta \cdot m)$, which is fraction β of the total memory. In practice we observe that the amount of entangled memory is $< 1\%$.

The precise implementation details of the `ldd` function are not crucial here, but interested readers can refer to Miller et al [119] for more details. Roughly speaking, the `ldd` function, based on the algorithm of Miller et al [119], samples vertices and perform parallel breadth-first searches (BFS) to form clusters. If multiple searches reach the same vertex, it uses compare-and-swap operations to ensure that a vertex is assigned to at most one cluster. While these compare-and-swap operations introduce nondeterminism, they don't create entangled objects because they only involve unboxed integer values.

The contract function, however, does create entanglement, because it uses a shared hashtable to deduplicate inter-cluster edges. Theoretically, the worst-case amount of entangled memory is the number of distinct remaining edges which is $O(\beta \cdot m)$, a fraction β of the total memory $O(m)$.

Key Takeaway. For the `reach` algorithm, we can bound the worst case amount of entangled memory as a fraction β of the total memory, where β is a parameter to the algorithm, typically around 0.3 for large graphs. In practice, our experimental results suggest that the amount of entangled memory is significantly smaller, because the worst case is conservative. We observed that, after just one round, the number of edges dropped dramatically from roughly 200 million to 11 thousand, far exceeding the worst case reduction of $\beta = 0.3$. **Of the 200 million edges that are allocated, only 11 thousand are entangled and others are disentangled.** This program thus satisfies the disentanglement hypothesis.

The low-diameter decomposition and contraction steps are highly effective in minimizing the number of inter-cluster edges. Consequently, the amount of entangled memory is very low. Additionally, all of this entanglement is temporary. When the `contract` function finishes, the entanglement is resolved because the tasks performing the deduplication have joined.

3.4 Limitations and Extensions

Although the disentanglement hypothesis applies to many fork-join parallel programs, there are interesting classes of programs that may create higher amounts of entanglement. This is particularly the case in interactive settings. For example, a PDF viewer with user threads and application threads communicating through concurrent effects on shared memory (e.g., by reading and writing to the same data structures simultaneously) could create a large number of entangled objects, under the fork-join model.

To overcome this limitation, we can use a more powerful model of parallelism, where communication is a first-class citizen of the language. For example, many modern programming languages and frameworks support a more powerful form of parallelism called futures. Futures allow you to create a parallel task and demand the result from the task at a later time when needed (hence the name "future"). In Chapter 8, we demonstrate that even interactive applications adhere to the disentanglement hypothesis when expressed using futures. We consider a functional calculus with futures, state, and I/O, and define a semantics for disentanglement for futures. We show that determinacy race free programs with futures only have disentangled objects, thereby generalizing the disentanglement hypothesis to include interactive programs.

4

Memory Management

Our coscheduling algorithm organizes the memory into a tree of heaps and assigns each heap to a processor. Our goal is to enable efficient and scalable memory management by allowing each processor to independently manage its heaps without much synchronization. This means that each processor can allocate memory and perform garbage collection within its heaps, without tracing memory outside of them.

At a high level, the memory manager exploits the disentanglement hypothesis to enable independent memory management. It tracks entangled objects efficiently and uses this information to enable each heap to be independently garbage collected.

To track entangled objects, the memory manager observes the memory actions of parallel tasks using read/write barriers. For efficiency, the memory manager only uses read and write barriers for mutable data and does not use any barriers for immutable data. The memory manager also tracks when tasks join and resolves the entanglement between them punctually, thereby ensuring that entanglement does not "snowball" to include a large number of objects. We prove that our memory manager is precise in tracking entangled objects, by showing that the work and space for tracking them is proportional to the amount of entanglement in the program (Section 4.4). **This ensures that tracking entangled objects imposes near-zero overhead on the allocation and access of disentangled objects.** Because entangled objects are a minority in parallel programs (disentanglement hypothesis), shielding their overheads from disentangled objects ensures that tracking of entangled objects is efficient.

By efficiently tracking entangled objects, the memory manager can account for all inter-heap pointers and enable independent garbage collection of each heap. Specifically, inter-heap pointers into disentangled objects can be cheaply tracked because they are between sequentially dependent heaps. Inter-heap pointers to entangled objects, on the other hand, do not need explicit tracking because the entanglement tracking algorithm already identifies and preserves

these objects. Thus, each heap can be garbage collected independently, as the memory manager accounts for all incoming pointers.

We start this chapter by giving some relevant garbage collection background and describing independent garbage collection along with the problems posed by inter-heap pointers.

4.1 Background: Garbage Collection

Garbage Collection relieves programmers from the burden of manually managing the memory, improving the programmer's productivity by guaranteeing safety from errors like dangling pointers and memory leaks. We give an intro-level overview of the key concepts in garbage collection, relevant for our discussion in the thesis. We recommend the book by Jones et al. [101] for a comprehensive introduction.

Memory as a Graph. We can abstractly represent a program's memory as a graph, with ***memory objects*** as vertices, and ***pointers***, which are references between objects, as directed edges. The graph is dynamic because the program continuously allocates new objects and modifies (mutates) pointer edges during execution. The program accesses the memory graph through a set of designated objects called ***program roots***. These roots include global variables, task-local stacks, and other objects directly accessible to the program.

Liveness and Reachability. An object is ***live*** if it is reachable from program roots either directly or through a series of pointers. Conversely, objects that are not reachable from any program roots are ***garbage or dead*** objects. The garbage objects are guaranteed to be never accessed by the program and can be safely reclaimed.

Garbage Collection. The garbage collector is responsible for identifying live objects and reclaiming the memory occupied by dead objects, making space available for new allocations. Our focus in this work in on *tracing garbage collectors*, which traverse the memory graph to determine live and garbage objects. Such garbage collectors typically have two steps: 1) *Tracing*: Starting from the program roots, the collector follows pointers to discover and mark all reachable objects, and 2) *Reclamation*: Memory occupied by unmarked objects is reclaimed, either by directly freeing them or by relocating live objects to compact the memory. Compaction, as achieved by relocation of live objects, not only reduces fragmentation by using memory space in a contiguous fashion, but also improves memory locality, boosting the program's performance.

Heaps and Independent Garbage Collection. In parallel programs, where multiple tasks allocate memory simultaneously, memory managers partition memory into ***heaps***. A heap is an abstract data type representing a set of objects and supports operations such as creating a new heap, merging two heaps, allocating within a heap, freeing objects within a heap, and tracking whether an object is in the heap. The benefit of partitioning memory into heaps is that processors can use different heaps to allocate and garbage-collect objects independently.

By ***independent garbage collection***, we mean that the garbage collector determines the live/-garbage objects of a heap by only tracing objects within the heap and not tracing those outside it. This can be efficient both in terms of the work and parallelism of the garbage collector, because it allows a processor to garbage collect a portion of the memory without tracing the entire memory graph, and also allows multiple processors to garbage collect independently in parallel.

However, independent garbage collection is difficult to achieve, because of ***inter-heap pointers***—pointers that reference objects in different heaps—complicate this, as the collector must consider pointers from objects outside the heap. When collecting a heap independently, any object that is target of pointer from an object outside the heap, must be treated like a root. This is because, there is no way to determine if that object is live, and furthermore, even if the object may be garbage, freeing that object would create a dangling pointer, which is unsafe. Thus, the garbage collector must account for all the inter-heap pointers and keep their target objects live. Accounting for inter-heap pointers can be challenging because it typically requires synchronization between processors and can lead to high overheads and scalability problems.

Connection to Coscheduling. When garbage collecting a heap, we can assume that the target object of all pointers coming into the heap are live. This is a conservative approximation because its possible that those inter-heap pointers are themselves are garbage and could be collected. In Chapter 2, we discuss our coscheduling technique, which bounds how conservative this approximation is by "clustering" these heaps and assigning clusters them to processors All heaps in a cluster can be garbage collected together by the same processor, allowing for accurate treatment of inter-heap pointers within a cluster. Different clusters are garbage-collected independently.

4.2 Accounting for Inter-Heap Pointers

Our memory manager partitions memory into heaps, with each task assigned to its own heap for allocations, and organizes the heaps as a ***heap tree***. To enable efficient and independent memory management of heaps in the tree, the memory manager must account for inter-heap pointers, which are pointers between objects in different heaps. These pointers create dependencies between heaps, as the liveness of objects in one heap can depend on objects and pointers in another.

In this section, we present techniques to account for all the inter-heap pointers and ensure that their target objects are not prematurely reclaimed during garbage collection. By keeping the target objects of inter-heap pointers live, we account for all the inter-heap pointers and enable independent garbage collection of each heap, i.e., any heap in the tree can be garbage collected without tracing the memory in other heaps (Section 4.5). We begin by considering the different types of inter-heap pointers that can exist within the heap tree; the classification is same as the one we described in Chapter 2.

4.2.1 Types of Inter-Heap Pointers and Example

Within a heap tree, we classify pointers based on the relative positions of the objects they reference. Every pointer can be classified as one of four categories:

1. **Up Pointers:** These pointers originate from objects in a descendant heap and target/reference objects in an ancestor heap. They point "upwards" in the heap tree.
2. **Down Pointers:** These pointers originate from objects in an ancestor heap and reference objects in a descendant heap. They point "downwards" in the heap tree.
3. **Internal Pointers:** These are between objects within the same heap (they are not "inter-heap").
4. **Cross Pointers:** These pointers are between objects in heaps belonging to concurrent tasks.

This classification of pointers not only distinguishes the relative positions of objects, but also gives insights into how the program creates these pointers. Given that each task gets a new heap when it is created and that heaps are merged into parent heaps at joins, the heap tree tracks the sequential order in which objects are allocated. Objects in descendant heaps are allocated "sequentially after" those in ancestor heaps, because descendant heaps are created after forks and objects in ancestor heaps are allocated before forks. Consequently, up pointers always reference older objects, while down pointers always reference newer objects. Cross pointers reference objects allocated by concurrent tasks, where the relative allocation times are not directly comparable.

Up pointers are common in functional programs due to the frequent use of immutable data structures, where new objects reference older, existing objects. Down pointers are associated with mutable objects, as they are created by modifying an existing object to point to a newer object. Cross pointers are created because of entanglement and determinacy races, because they are between objects allocated concurrently.

Example. Figure 4.1 illustrates how down pointers and cross pointers are created by mutable effects. It shows an example program and the heap tree at a step of program execution. The code first creates a mutable reference `a` and then executes functions `add1` and `add2` in parallel. The function `add1` allocates an immutable object, corresponding to value `Some 4`, and writes it to reference `a`, creating a down pointer. The function `add2` loops until the write by function `add1` takes place, and then stores the result in a reference `b`, creating a cross pointer.

4.2.2 Barriers, Remembered Sets, and Heap Tree

To track the creation, deletion, and usage of inter-heap pointers, the key mechanisms we use are barriers and remembered sets. A read/write barrier is a snippet of code that the compiler inserts before every read/write. Our memory manager uses barriers to detect and account for inter-heap pointers. Because barriers can potentially add overhead to every program read/write, it is crucial that they are as efficient as possible. In our memory manager, we ensure that all

```
1  val a = ref None
2  fun add1() =
3    a := Some 4;
4    return a
5  fun add2() =
6    case !a of
7      None ⇒ add2()
8    | Some _ ⇒
9      b = ref (!a);
10     return b
11 val (c, d) = fork (add1(), add2 ())
```

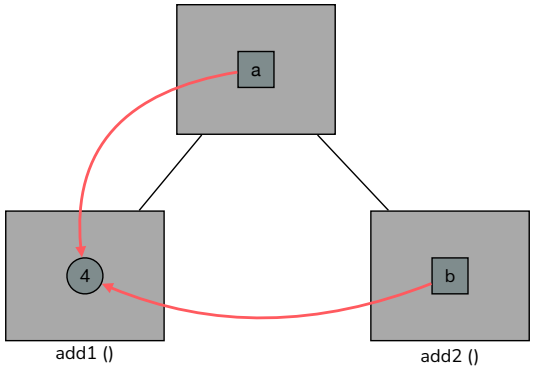

Figure 4.1: The code on the left demonstrates how down and cross pointers are created by mutable effects. The figure on the right shows a heap tree where the gray boxes denote heaps, green boxes are mutable references and circles represent immutable objects. The red arrows indicate pointers.

immutable objects are read directly without any barriers, which is important because most reads in a parallel functional language involve immutable objects. Additionally, we design several techniques to ensure that the overhead of the read barrier on mutable objects remains low (see "pin sharing" in Section 4.3 and "entanglement frontier" in Chapter 6).

When the memory manager detects new inter-heap pointers using barriers, it stores them in a ***remembered set***. Each entry in the remembered set typically contains a source object, which is the origin of the pointer, and a target object, which is the object being referenced. We maintain a remembered set for each heap to allow the creation and removal of entries in parallel across different heaps. At a fork, when a task creates child tasks, the memory manager creates two empty heaps for the children tasks and initializes their remembered set as empty sets. At a join, it merges the parent heap with the children heaps and merges the remembered sets of the children with those of the parent.

Our data structure for implementing the dynamic heap tree supports many useful operations. For example, the tree allows for efficient querying of an object's heap and also tracks information about the relationships between heaps, such as their depth within the tree and whether two heaps are concurrent (i.e., not in an ancestor-descendant relationship). We leave the implementation details of this data structure to Chapter 6.

4.2.3 Managing Up Pointers

Up pointers, which target objects in ancestor heaps from objects in descendant heaps, are the most ubiquitous inter-heap pointers in parallel functional programs. This is because up pointers are primarily created by immutable objects, which point to old objects allocated before them.

To manage up pointers, we use a ***snapshot-at-fork*** technique which efficiently captures

all potential up pointers at the time of fork. When a task forks, it passes object references to its children, which can then use these references to create up pointers. At the time of fork, we record all the objects the parent passes to its children and add them to the parent heap's remembered set. These objects and any objects reachable from them through pointers, are the complete set of objects that the children tasks—or any of the descendants—can access and use to create up pointers. Our memory manager thus treats these objects as roots and ensures that all objects targeted by up pointers are preserved and can be safely accessed.

Note, however, child tasks can delete pointers in the parent heap and change the reachability of objects from these roots, potentially causing the garbage collector to miss live objects. To address this, we use write barriers to intercept pointer deletions. The write barriers identify deleted pointers and add them to the heap-local remembered sets, ensuring that they are considered as roots for garbage collection. This technique is similar to snapshot-at-the-beginning techniques used in garbage collection that preserve the reachability in a heap w.r.t. a given point in time [101]. In our setting, we perform this snapshot at the time of fork and use it to account for up pointers.

4.2.4 Managing Down Pointers

Down pointers target objects in descendant heaps from objects in ancestor heaps. Since descendant objects are younger than ancestor objects, down pointers represent a old-to-young relationship between objects, which is only possible through mutation (immutable objects can only point to objects allocated before them). Thus, a program creates a down pointer by modifying an old mutable object to reference a newer object. For example, a parent task may pass a reference to a mutable object to a child task, which can then modify it to create a down pointer to its an object in its own heap.

We can use this observation to track down pointers using write barriers. The write barrier intercepts memory updates and checks whether a pointer is being created and whether the pointer is a down pointer. If a down pointer is created, the write barrier stores it in the remembered set of the target object. This ensures that the target of the down pointer is not reclaimed and the pointer can be accessed safely.

4.2.5 Cross Pointers

Cross pointers are pointers between objects in heaps corresponding to concurrent tasks. These pointers typically originate in two steps. First, a task accesses an object in a concurrent task's heap via a down pointer. Then, the task stores a reference to that object in its own heap.

Cross pointers are challenging to track because, once a cross pointer is created, it can lead to more cross pointers. Tasks can create new cross pointers, through an existing cross pointer, by using it to reference objects in other heaps. This proliferating effect makes tracking all cross pointers impractically expensive. For instance, even new object allocations can create cross pointers because they may reference objects in a concurrent heap. Tracking cross pointers

would thus require inspecting all allocations, even those unrelated to cross pointers, resulting in significant overhead.

Instead of tracking individual cross pointers, we can feasibly track objects that are targets of cross pointers. These target objects, accessed by concurrent tasks, correspond to the entangled objects in the heap tree. By tracking all entangled objects and ensuring they remain live, we can avoid tracking individual cross pointers. Keeping entangled objects live guarantees that all cross pointer accesses are safe and prevents dangling pointers. We discuss entanglement tracking and its overheads in the next section, Section 4.3

4.3 Tracking and Managing Entanglement

In this section, we describe our techniques to manage entangled objects in the heap tree architecture. Our memory manager tracks all entangled objects in the heap tree and keeps them live, ensuring that concurrent tasks can safely access the entangled objects in each other's heap. To do this, we roughly follow the of our entanglement semantics, described in Section 2.1. Our memory manager tracks ***entanglement sources***—the objects where entanglement begins and their ***entanglement regions***—the objects that become dependent on other concurrent tasks as a result of entanglement sources. To reclaim memory of each heap independently, we use garbage collection algorithms that keep the objects of entanglement regions in place, but move and compact other objects.

Identifying and managing entanglement regions correctly and efficiently poses several challenges. First, we must mediate between the concurrent tasks and the collector, because new objects can become entangled virtually at any point during execution, e.g., **a concurrent task may try to access an object in a heap, while the garbage collector is actively relocating objects within that heap.** Second, entangled objects are not permanently entangled and become disentangled once tasks join; we must recognize the resolution of entanglement punctually, so as not to penalize computations more than necessary. Third, we must do all of this efficiently, in work proportional to their number/size, without imposing any cost on allocations and accesses to disentangled objects. Crucially, our tracking algorithm must not use any barriers on immutable objects, which is crucial for the performance of parallel functional programs, because most memory operations involve immutable objects.

In the rest of this section, we present our techniques and describe how we overcome these challenges. Our techniques culminate in a theorem that bound the work (run-time) and space cost of tracking entanglement in terms of the cost metrics defined by the semantics. Before we dive into details we give an overview and consider a simple example.

4.3.1 Overview

The memory manager maintains, for each task, a ***source set*** for tracking entanglement sources. Its components—the read/write barriers and the collector—work together to track and manage entangled objects.

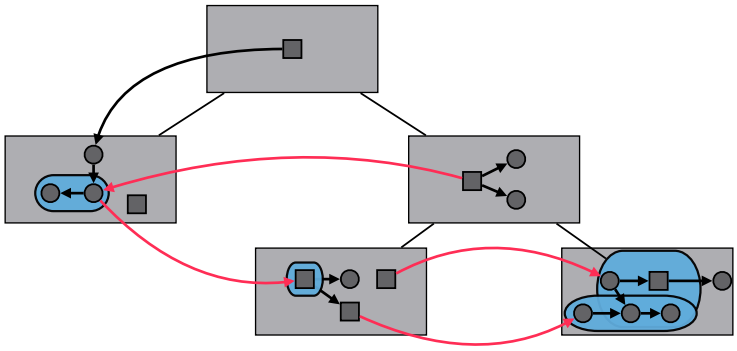

Figure 4.2: The figure shows a heap tree with mutable and immutable objects represented as squares and circles respectively. The black pointers are between ancestor-descendant objects and the red pointers are between concurrently allocated objects. The targets of red pointers are the entanglement sources. Each entanglement source has an entanglement region, which includes objects reachable from the source upto a mutable frontier. The entanglement regions are depicted using blue bubbles. The figure shows two entanglement regions that overlap and also shows an entangled region with a single mutable object.

Barriers and collector. The barriers intercept memory accesses and identify entanglement sources. When they identify a source, they add it to the source set of the task that allocated it. The collector treats the sources as roots and keeps them live.

The barriers and the collector also achieve a key performance goal: "barrier-less immutable accesses". Rather than intercepting immutable accesses to manage entanglement, the barriers ***pin*** the ***entanglement region*** of every source and the collector refrains from reclaiming or relocating pinned objects. The entanglement region spans all immutable objects that may be accessed via the source: it starts at the source (which may be mutable or immutable) and goes up to its "mutable frontier", i.e., it contains all objects reachable from the source by only considering immutable pointers. Here immutable pointers mean pointers of immutable objects. When a barrier identifies an entanglement source, it pins the source's entanglement region to ensure that the collector does not relocate or reclaim any object of the region. This way the barriers and the collector manage entangled immutable objects without intercepting immutable accesses.

Example. Consider the heap tree shown in Figure 4.2 with mutable objects depicted as squares and immutable objects as circles. The black pointers are either internal to a heap or are between objects of ancestor-descendant heaps whereas the red pointers are between objects of independent heaps. The objects at the target end of red pointers are entanglement sources. The regions of the sources are represented with blue bubbles. One of the bubbles wraps only a single square demonstrating that if the source is mutable, its region only contains the source itself. Another bubble illustrates that the entanglement region does not extend beyond mutable ob-

jects, but may contain series of immutable objects (circles). As the figure shows, entanglement regions may overlap.

Expired entanglement sources. Entanglement sources increase space usage because the collector assumes that these sources are live. However, we can remove entanglement sources when two tasks join and the entanglement between them is resolved, leading to the expiration of the sources entangled between them. By removing expired sources, we can remove the space cost of keeping them live. The challenge lies in tracking when sources expire. To do this, the barriers track an ***exp-depth*** (expiration depth) for each entanglement source. The exp-depth represents the depth at which all tasks that read the source through a mutable reference join. Once a source's heap reaches exp-depth in the heap tree, it is guaranteed that all the relevant joins have occurred and the source has expired. At this point, we can discard the source.

Invariants. To track entanglement sources, entanglement regions, and their expiration depth, We use read and write barriers on mutable objects. The barriers ensure the following invariants for each source:

1. the source is in the source set of the task that allocated it,
2. the source's entanglement region is pinned, and
3. the source's exp-depth is correct, i.e., it expires once its heap reaches the exp-depth.

Figure 4.3 shows the pseudocode for the barriers that shows how these invariants are maintained. In addition, the code considers the interaction between the collector and the barrier, e.g., it shows how a barrier may find a "forwarding pointer" on an object it is about to pin, because the collector has moved it. To ensure that a barrier is never blocked by a collection and vice versa, our code is lock-free. We discuss the code for the collector in Section 4.5.

4.3.2 Read barrier for mutable objects

Procedure read in Figure 4.3 shows the pseudocode for the read barrier. It takes as arguments the reader task t and an object ℓ and returns the result of the read, the object ℓ'. But first, it performs an entanglement check by comparing task t to the allocator task of object ℓ', task t'. The procedure allocator returns the task which allocated the object. If t' is concurrent to t (line 4), then the read is entangled, otherwise the read is disentangled. In the case of disentanglement, the barrier returns without further actions.

In case of entanglement, object ℓ' is an entanglement source and the read barrier acts to ensure the three invariants. It calls the procedure get_sinfo to check if the object is already flagged as a source. The procedure get_sinfo returns an object called sinfo, a tuple of two elements containing a boolean flag that tells if the object is a source and the exp-depth of the source. If the object is flagged as a source then the first two invariants are guaranteed and we only need to ensure the third invariant, i.e., ensure that the exp-depth is correct. The read barrier calls set_info, to update the exp-depth appropriately (discussed later in detail).

```
1 procedure read (ℓ, t):
2   ℓ′ = !ℓ
3   t′ = allocator(ℓ′)
4   if (concurrent(t, t′)) {
5     (is_source, ed) = get_sinfo(ℓ)
6     if (is_source) {
7       set_sinfo(ℓ′)
8       return ℓ′
9     }
10    else {
11      ℓ″ = pin_region(ℓ′)
12      e_t′ ← e_t′ ∪ {ℓ″}
13      set_sinfo(ℓ″)
14      return ℓ″
15    }
16  }
17  // fast path for disentangled reads
18  else return ℓ′
19
20 procedure set_sinfo (ℓ, t, t′):
21  sinfo = get_sinfo(ℓ)
22  (is_source, ed) = sinfo
23  if (!is_source)
24    ed = DCADepth (t, t′)
25  else
26    ed = min (DCADepth (t, t′), ed)
27  nsinfo = (true, ed)
28  if cas (sinfo_addr(ℓ), sinfo, nsinfo)
29    return
30  else set_source (ℓ, t, t′)
```

```
31 procedure write (ℓ, ℓ′):
32   t′ = allocator(ℓ′)
33   if (allocator(ℓ) ≠ t′)
34     r_t′ ← r_t′ ∪ {(ℓ, ℓ′)}
35   ℓ ← ℓ′
36   return
37
38 procedure pin_region (ℓ):
39   h = get_header (ℓ)
40   if (forwarded(h)) {
41     ℓ′ = get_forward_object (h)
42     return pin_region (ℓ′)
43   }
44   if (is_pinned(h)) {
45     return ℓ
46   }
47   if (mutable(h)) {
48     if cas(addr_header(ℓ), h, pinned_version(h))
49       return ℓ
50     else return pin_region (ℓ)
51   }
52   else {
53     foreach i ∈ ptr_fields(ℓ) {
54       ℓ′_i = pin_region(ℓ[i])
55       if (ℓ′_i ≠ ℓ[i]) {
56         ℓ[i] ← ℓ′_i
57       }
58     }
59     if cas(addr_header(ℓ), h, pinned_version(h))
60       return ℓ
61     else return pin_region (ℓ)
62   }
```

Figure 4.3: Pseudocode read shows our read barrier. It uses pin_region and set_sinfo as helper functions to pin entanglement regions and maintain the exp-depth of entanglement sources respectively. The procedures account for concurrency with other tasks and also with the garbage collector. All procedures are lock-free.

If the object is flagged as a source, the barrier (i) calls procedure pin_region to pin the entanglement region of ℓ', (ii) adds the object to the source set $e_{t'}$ of task t', and (iii) calls procedure set_sinfo to mark the object ℓ' as a source and set its exp-depth (see lines 11-13).

4.3.3 Entanglement region and its pinning

We implement pinning by reserving a bit in the header of each object, and when we set it we say that we ***pin*** the object. The procedure pin_region pins the objects in the entanglement region of its argument, accounting for the facts that each object in the region may be relocated by the collector, or may simultaneously be pinned by other tasks. The procedure's pseudocode is shown by Figure 4.3.

Relocation by the collector. First, the procedure checks if its argument object has been relocated by the collector. When a collector relocates an object, it creates a copy of the object and installs a forwarding pointer to the copy in the header of the object. The procedure pin_region calls get_header to read the header of the argument object ℓ and calls forwarded to check if the header is a forwarding pointer (see Figure 4.3). If the header is a forwarding pointer, the procedure pin_region recursively calls itself on the copy of the object and returns (line 40). This recursive call pins the entanglement region of the object's copy and returns it.

When the header is not a forwarding pointer, the procedure pin_region checks if the header is pinned. If the header is pinned, it is guaranteed that the entanglement region of the object is also pinned (see "pin sharing" below). In this case, the procedure returns (line 45).

Mutable and immutable objects. In the case where the object is not pinned, the procedure pin_region pins the entanglement region of the object, a region that only includes the object itself when it is mutable, but goes up to the object's mutable frontier when it is immutable.

When the object is mutable, the procedure attempts to pin the object by calling cas (which stands for atomic compare-and-swap) on the header of the object (line 48). The procedure cas takes three arguments: the address of the header addr_header(ℓ), the expected version of the header h, and the pinned version of the header pinned_version(h). The cas updates the object's header if it finds the expected version h, returning true and otherwise performs no updates, returning false. If the cas fails, the procedure loops by calling itself.

When the object is immutable, the procedure must pin the entanglement region of the immutable object, consisting of all objects pointed by the immutable object and their respective entanglement regions. To do so, it recursively calls itself on each object pointed by the immutable object. Specifically, it calls procedure ptr_fields to obtain a sequence of pointers in the immutable object and recurses on the ith pointer $\ell[i]$, thereby pinning the entanglement region from $\ell[i]$ (line 54). However, this recursive call may return a copy of the argument object $\ell[i]$, if that object has been forwarded by the collector. In such a case, the procedure pin_region deletes the pointer from the immutable object to the relocated object and replaces it with a pointer to the copy (ℓ'_i in line 56). After it iterates through all the pointers, the procedure attempts to pin the immutable object. If it succeeds it returns the immutable object, and otherwise, it recurses.

Pin sharing. Multiple tasks may concurrently attempt to pin the same objects, raising the question if we can avoid redundant pinning operations. To do so, tasks pin objects in a specific order, determined by pointers of immutable objects: if an immutable object ℓ points to another object ℓ', then object ℓ' is pinned before object ℓ. This ordering guarantees that when an object is pinned all objects in its entanglement region are also pinned, because the region only contains immutable pointers from the object. Thus, if a task encounters an object that is already pinned, it can safely assume that the entire entanglement region is also pinned (see line 45 in Figure 4.3). This allows the task to skip the pinning process for that region, avoiding redundant work.

4.3.4 Expiration depth of entanglement sources

An object becomes an entanglement source when read by concurrent tasks via a mutable object. The entanglement source expires (becomes disentangled) when the tasks that read it and the task that allocated it have joined. This occurs at the depth of their deepest common ancestor in the heap tree

Definition 4.3.1. The expiration depth (exp-depth) of an entanglement source ℓ is the deepest depth in the heap tree at which all tasks that have read ℓ through a mutable reference have joined with the task that allocated ℓ.

The exp-depth of a source can be calculated as follows: let $v_0 \dots v_n$ be the tasks that read object ℓ from some mutable object and let $\alpha(\ell)$ be the allocator task of object ℓ. Then, the exp-depth of source ℓ is equal to $\mathsf{DCADepth}(v_0, v_1 \dots v_n, \alpha(\ell))$, where DCADepth calculates the depth of the deepest common ancestor of a set of tasks in the task tree.

The procedure set_sinfo calculates the exp-depth in an online fashion. It takes the source ℓ, its allocator t, and its reader t' and sets the sinfo object of source ℓ. Recall that sinfo is a tuple which contains a flag that tracks whether the object is a source and also contains its exp-depth. We give the pseudocode of the procedure in Figure 4.3. The procedure first reads the sinfo of the object ℓ by calling get_sinfo and then checks if the object is already a source. If not, then the exp-depth of the source is equal to $\mathsf{DCADepth}(t, t')$. Otherwise, if the object is a source, it already has exp-depth ed, and the procedure picks the minimum among that depth and the depth $\mathsf{DCADepth}(t, t')$. To update the object's depth, it creates a new sinfo object named nsinfo, containing the correct exp-depth. Then, it attempts to install the nsinfo on the object using a cas and if it fails (because another task updated the sinfo), it tries again. In our implementation we flatten the sinfo tuple to a word.

4.3.5 Write barrier for mutable updates

The write barrier tracks inter-heap pointers created by mutable updates and adds such pointers to the remembered sets. We show its pseudocode in procedure write of Figure 4.3. It takes as arguments a mutable object ℓ and an object ℓ', and compares their allocator tasks to check if they are in different heaps (line 33). If so, it adds the tuple (ℓ, ℓ') to the remset ($r_{t'}$) of task t'. After this, it performs the update. The collector treats remembered set entries as roots, ignoring those that may have become internal as a result of joins.

4.4 Bounding the Overhead of Tracking Entanglement

We analyze the performance of our entanglement tracking algorithm by giving bounds on its work cost and space cost (in terms of liveness).

Theorem 1 (Work bound)**.** *An execution with work W requires $O(W + \epsilon)$ work to execute including the cost of entanglement tracking, where ϵ is the entanglement factor of the execution.*

Our entanglement tracking algorithm has three sources of overhead: identifying entanglement sources, pinning their entanglement region, and maintaining the exp-depth. The work bound shows that these overheads are localized to entangled objects, i.e., accesses to disentangled objects are not penalized. The bound also confirms that the penalty to entangled objects is additive: it is paid once per entangled object and does not increase with the number of times an entangled object is accessed. Our proof assumes that contention per object is low, i.e., we assume that each compare-and-swap (cas) instruction succeeds after a constant number of attempts. All of them are borne at mutable reads by the read barrier. We prove the following lemma to bound the work of mutable reads.

Lemma 2. *The total work for mutable reads is $O(r + \epsilon)$, where r is the number of mutable reads and ϵ is the entanglement factor.*

Proof. We prove the lemma, assuming that contention per object is low and each cas succeeds after some constant number of tries. In this model, if we ignore constant factors, then we can analyze by directly assuming that the cas (compare-and-swap) always succeeds. The constant number of tries do not play a role in a big O bound.

For reads of disentangled mutable objects, there is a constant overhead because the read barrier checks for disentanglement and in that case, returns the object. Because this is constant time, this work can be charged against r, which is the number of mutable reads. Reading an entangled object is also constant time, if the object is an entanglement source. In this case, the read barrier only calls set_sinfo, which takes a constant number of steps and thus, this work can also be charged against the number r.

The main overhead is incurred when an entangled read makes the object a source and as a result, the barrier traverses the objects in the entanglement region to pin them. This overhead can be charged against the entanglement factor ϵ. To do so, recall that the cost semantics (Figure 3.3) bumps the entanglement factor each time a new location is added to the entanglement region. We argue that the work of the read barrier is exactly equal to the number of locations added to an entanglement region because of this read.

Let's consider three cases. First, suppose the object that becomes the source, is already in an entanglement region of another source. Then, the procedure pin_region does not perform any additional pinning and returns. Likewise, the cost semantics does not increase the entanglement factor because no new objects are added to the entanglement region.

Second, suppose the object that becomes the source has no locations pinned in its entanglement region. Then the procedure performs work equal to the size of the entanglement region.

Likewise, the cost semantics increases the entanglement factor by the size of the entanglement region.

Third, suppose the entanglement region of the source is the set of objects E_1 and it overlaps some other entanglement regions (whose objects are already pinned), and their union is E_2. Then the procedure pin_region only traverses the set difference $|E_1 \setminus E_2|$ amount of memory. When the procedure pin_region recurses on immutable objects, it stops as soon as it hits a pinned object. This pinned object must belong to an objects in E_2. Thus, the procedure never traces any pointers of objects in E_2. Thus, the work cost is $|E_1 \setminus E_2|$. Likewise, the cost semantics increases the entanglement factor by the size of new locations in an entanglement region, which is $|E_1 \setminus E_2|$. □

Liveness Bound Our entanglement tracking algorithm keeps entanglement sources live and increases the liveness of the execution for the garbage collector. The bound states that the algorithm only keeps $O(\delta)$ additional memory live, confirming that our entanglement tracking is precise: entanglement sources are kept live only for the duration they are entangled.

Theorem 2 (Liveness bound)**.** *If R is the maximum live memory of an execution of a parallel program, then maximum live memory of that execution, including the impact on liveness from entanglement tracking, is $O(R + \delta)$, where δ is the entanglement ceiling of that execution.*

Proof. To prove this theorem, we compare the set of entanglement sources, as defined by the semantics, to the entanglement sources, as computed by our implementation. Assuming that they match, the result follows from the definition of entanglement high-watermark. The entanglement high-watermark is the maximum amount of memory reachable from any entanglement source, at any step. If our tracking of entanglement sources is accurate w.r.t. the semantics, then we assume, at any step, no more than $O(\delta)$ amount of memory to be live. Thus, the total is $O(R + \delta)$.

We compare the tracking of entanglement sources at each step of the execution. For a step i, let $\mathcal{S}_s^i$ and $\mathcal{S}_a^i$ be the set of entanglement sources as tracked by our semantics and algorithm respectively. Our semantics tracks entanglement sources by maintaining a reader history. Each time a task reads an object from a mutable reference, it adds the task to the reader history of that object. If H_i is the reader history at step i, then $\mathcal{S}_s^i = \mathcal{E}(H_i) = \{\ell \mid \exists\ v :\ v \in H_i(\ell) \wedge \mathsf{concurrent}(\alpha(\ell), v)\}$. We rephrase this as $\mathcal{E}(H_i) = \{\ell \mid \mathcal{C}(H_i(\ell)) \neq \emptyset\}$, where $\mathcal{C}(H_i(\ell)) = \{v : v \in H_i(\ell) \wedge \mathsf{concurrent}(\alpha(\ell), v)\}$.

For a step i, we show the following by induction:

1. $\mathcal{S}_s^i = \mathcal{S}_a^i$
2. for an entanglement source ℓ, the exp-depth of object ℓ, as maintained by the algorithm, is equal to $\mathsf{DCADepth}(\mathcal{C}(H_i(\ell)), \alpha(\ell))$.

At the start, there are no entanglement sources and the hypotheses hold trivially. Let's assume that they hold upto step i and consider the next step. We consider the following cases based on the instruction executed in this step:

Mutable read In this step a task v reads a mutable object ℓ' from object ℓ. As a result, the semantics adds task v to the reader history $H(\ell')$ of object ℓ'.

Let u be the allocator task of object ℓ', i.e., $u = \mathsf{allocator}(\ell')$ be the allocator task of object ℓ'. If the allocator and the task v are not concurrent, the read is disentangled. In the semantics the set $\mathcal{C}(H_i(\ell'))$ does not change and thus, nothing changes. Likewise, the entanglement tracking algorithm does not do anything and the hypotheses continue to hold.

When the allocator task u and the task v are concurrent, we have that $\mathcal{C}(H_{i+1}(\ell')) = \mathcal{C}(H_i(\ell')) \cup \{v\}$. We consider two cases. If the object ℓ' was not an entanglement source, then $\mathcal{C}(H_i(\ell')) = \emptyset$ and the proof is easy: both the semantics and the algorithm make it a source after the step. Furthermore, $\mathcal{C}(H_{i+1}(\ell')) = \{v\}$ and the exp-depth is equal to $\mathsf{DCADepth}(u, v)$.

However, if the object ℓ' was an entanglement source prior to the read, then it continues to be so after the read and the first hypothesis holds. For the second hypothesis, we need to consider the exp-depth carefully. Suppose the exp-depth of object ℓ' is ed before the read. After the read, the algorithm updates the exp-depth to $\min(ed, \mathsf{DCADepth}(u, v))$. To prove the second hypothesis, we show that

$$\min(ed, \mathsf{DCADepth}(u, v)) = \mathsf{DCADepth}(\mathcal{C}(H_{i+1}(\ell)), u).$$

From the inductive hypothesis, we have that $ed = \mathsf{DCADepth}(\mathcal{C}(H_i(\ell')), u)$.

Consider the deepest common ancestor of the tasks in $(\mathcal{C}(H_i(\ell)) \cup \{u\})$ and let it be r. If task v is a descendant of task r, then the deepest common ancestor of $(\mathcal{C}(H_i(\ell)) \cup \{v\} \cup \{u\})$ does not change and thus, and $\mathsf{DCADepth}(\mathcal{C}(H_{i+1}(\ell)), u) = \mathsf{DCADepth}(\mathcal{C}(H_i(\ell)), u)$. Furthermore, $ed < \mathsf{DCADepth}(u, v)$ because $ed = \mathsf{DCADepth}(\mathcal{C}(H_i(\ell)), u)$. Thus, the exp-depth after the step remains equal to ed, because its lesser than $\mathsf{DCADepth}(u, v)$, and the second hypothesis holds. If task v is not a descendant of task r, then we observe that the deepest common ancestor of u and v, say r', must be an ancestor of r. This is because r and r' are both ancestors of task u and r' can not be a descendant of r: otherwise, because r' is an ancestor of v, r is an ancestor of v. Now, because r' is the deepest common ancestor of u and v and it is an ancestor of tasks in $\mathcal{C}(H_i)$, it follows that r' is the deepest common ancestor of $\mathcal{C}(H_i) \cup \{v\} \cup \{u\} = \mathcal{C}(H_{i+1}) \cup \{u\}$. The algorithm sets the exp-depth to the depth of r' and the second hypothesis continues to hold.

Join After a join, some entanglement sources expire. We show that the semantics and the algorithm agree on the sources that expire. We take two cases. First, we consider an object that is expired according to the semantics and show that it is also expired according to the algorithm. Second, we show vice-versa. The semantics expires sources by remapping accesses in the reader history. Specifically if tasks v and w join and their parent is u, then $H_{i+1} = \{u/v\}\{u/w\}H_i$.

First, Consider an object $\ell \in \mathcal{S}_s^i$, i.e., the object is an entanglement source at step i. But as a result of the join, it is not a source anymore, i.e., $\ell \notin \mathcal{S}_s^{i+1}$. We show that the algorithm follows this and for the algorithm, $\ell \notin \mathcal{S}_a^{i+1}$. We have that $\mathcal{C}(H_i(\ell)) \neq \emptyset$ because $\ell \in \mathcal{S}_s^i$. and that $\mathcal{C}(H_{i+1}(\ell)) = \emptyset$ because $\ell \notin \mathcal{S}_s^{i+1}$. From the definition of $\mathcal{C}(H)$, it follows that either the joining task $v = \alpha(\ell)$ and $\{w\} = \mathcal{C}(H_i(\ell))$, or $w = \alpha(\ell)$ and $\{v\} = \mathcal{C}(H_i(\ell))$. Without loss of generality, suppose $w = \alpha(\ell)$ and $\{v\} = \mathcal{C}(H_i(\ell))$. From the inductive hypothesis, the

exp-depth of object ℓ before the step, is equal to $\mathsf{DCADepth}(\mathcal{C}(H_i(\ell')), \alpha(\ell))$, which is equal to $\mathsf{DCADepth}(w, v)$, which is equal to the depth of u. The object ℓ expires, according to the algorithm, when it reaches the depth of task u and because $\alpha(\ell) = v$ before the join, $\alpha(\ell) = u$ after the join. Thus, we get that source ℓ expires after the join.

Second, consider an object $\ell \in \mathcal{S}_a^i$, i.e., the object is an entanglement source at step i. But as a result of the join, it is not a source anymore (according to the algorithm), i.e., $\ell \notin \mathcal{S}_a^{i+1}$. Then, the depth of the parent task u is equal to the exp-depth of object ℓ. Furthermore, because $\alpha(\ell) = u$ after the join and $\alpha(\ell) \neq u$ before the join, then $\alpha(\ell) = v$ or $\alpha(\ell) = w$. Without loss of generality, let $\alpha(\ell) = v$, before the join. From the inductive hypothesis, we know that $\mathsf{DCADepth}(\mathcal{C}(H_i), v) = d(u)$, where $d(u)$ represents the depth of task u. Because task u is a parent of v, the only possible set $\mathcal{C}(H_i)$ is $\{w\}$. Thus, before the join, we have $\alpha(\ell) = v$ and $\mathcal{C}(H_i) = \{w\}$. Because the semantics remaps the access history H_i and forgets the distinction between v and w, we get that $\mathcal{C}(H_{i+1}) = \emptyset$ and $\ell \notin \mathcal{S}_s^{i+1}$.

□

4.5 Independent Garbage Collection of Heaps

Our inter-heap pointer tracking guarantees that all targets of inter-heap pointers are either in the remembered set or the source set, for entangled objects, and that all the entangled objects are pinned. With this guarantee, our memory manager can perform independent garbage collection within any heap, meaning each heap can be collected without tracing memory in other heaps.

In our memory manager, we treat internal and leaf heaps differently. We collect internal heaps by adapting a concurrent mark-sweep algorithm for our heap hierarchy. The memory in internal heaps may be accessed by all its descendant tasks, and to support that without paying for synchronization overheads, we use a concurrent mark-sweep algorithm that does not move live objects and frees the garbage objects.

For leaf heaps, we pause the corresponding leaf task and use a hybrid algorithm—an algorithm that does not relocate the pinned objects because they are entangled, but copies and compacts other objects. The compaction aspect of this algorithm is crucial for our performance because it improves locality and defragments the heap. In fact, initially, we tried a mark-sweep approach for the leaf heaps and observed performance hits of upto 20% in time and 75% in space. The hybrid algorithm runs in three parts: (1) filtering roots, (2) tracing the heap graph, and (3) reclaiming memory using epoch-based reclamation. In all three parts, the algorithm accounts for concurrent readers, consisting of other tasks that concurrently attempt to read and pin the objects of the heap. To guarantee that the collector never blocks a concurrent reader, the hybrid algorithm is lock-free.

4.5.1 Identifying and Discarding Expired Entanglement Sources

To collect a heap, the collector treats the entanglement sources in the source set as roots. However, some of the sources may have expired because they reached their respective expiration

depth. The collector can safely discard such sources, as they no longer need to be treated as roots. To do so, it (i) unmarks the object as a source, (ii) unpins the entanglement region of the source, and (iii) removes the source from the source set. However, there is a subtle issue here: if the entanglement region of a stale source overlaps with the region of a real entanglement source, then unpinning the region of the stale source would be incorrect because it would unpin the overlapping portion of the other region. Worse, such a source, with an overlapping entanglement region, could be added concurrently by the read barrier while the collector is unpinning. To address this issue, the collector unpins in three steps. First, it takes a snapshot of the source set and use the snapshot, to compute a set of objects that can be unpinned. It does this as follows: it maintains two sets, P (pin) and U (unpin), and traverses the entanglement regions of sources in the snapshot. If a source is stale, it adds the objects of its entanglement region to the set U and otherwise adds them to the set P . The set difference $U \setminus P$ is the set of objects that can be unpinned. Then, it unpins all the objects in this set. These two steps guarantee that unpinning is correct, at least for the snapshot. Considering the example above, if two regions overlap and one of them is still entangled, t hen this approach will not unpin their overlap. The overlap would appear in both sets U and P and taking the set difference will ensure that it stays pinned. As the third step, the collector compares the heap's source set with its snapshot: if new sources were added to the source set during the first two steps, it traverses their entanglement regions and ensures that they are pinned.

4.5.2 Discarding Stale Remembered Set Entries

The write barrier adds to a heap's remembered set, when a task creates a pointer from another heap to an object of that heap. A remembered set entry is a tuple of objects, for example (ℓ, ℓ'), and tuple contains the origin and target of the pointer. Entries in the remembered set can become stale, for example, if the origin and target come in the same heap because of joins. The collector discards stale remembered set entries. A remembered set entry (ℓ, ℓ') is stale if either objects ℓ and ℓ' are in the same heap, or the pointer from object ℓ to ℓ' has been destroyed by a mutable update.

4.5.3 Tracing the Heap

To trace a heap, our hybrid algorithm maintains a ***mark*** bit in the object header and maintains a ***to-space*** for relocating the objects. Initially, every object is unmarked. By the end, the algorithm ensures that every pinned object is marked and every unpinned object is either garbage (unreachable) or has a forwarding pointer, pointing to the object's copy in the to-space.

The algorithm uses a stack of objects to trace the heap and begins by adding all the roots to the stack. Then, it iteratively pops an object from the stack and checks if the popped object is pinned. If so, it marks the object; otherwise, if the object is unpinned, it copies the object to the to-space and installs a forwarding pointer in the header of the object (unless the object is already in the to-space). After either marking the object or copying it, the algorithm adds the pointers within the object to the stack. This continues until the stack is empty.

```
1  procedure try_copy (ℓ):
2   h = get_header (ℓ)
3   if (is_pinned (h)) return ℓ
4   ℓ′ = copy_in_tospace(ℓ)
5   if cas (addr_header(ℓ), h,
6    fwd_ptr(ℓ′))
7    return ℓ′
8   else {
9    delete_in_tospace (ℓ′)
10   return ℓ
11   }
```

Figure 4.4: procedure try_copy

To copy an unpinned object to the to-space, the collector must account for concurrent readers that may attempt to pin the object. Figure 4.4 presents the pseudocode that accounts for such readers. The procedure try_copy takes an object ℓ and attempts to copy it to the to-space. The procedure reads the header of the object and checks if the header is pinned. If the header is pinned, it returns the object without copying it. Otherwise, it copies the object and attempts to replace the object's header with a pointer to the object's copy. This pointer is called a forwarding pointer because it "forwards" the readers of the object to the object's copy. The procedure try_copy uses a cas operation (atomic compare and-swap) to install the forwarding pointer on the header. If the cas succeeds, the procedure returns the copy of the object. The cas operation can fail if a concurrent reader changes the header by pinning the object before the procedure replaces the header with a forwarding pointer. Because the object is pinned in this case, the procedure deletes the copy and returns the original object.

4.5.4 Concurrent Reclamation of Memory

By tracing the heap, the hybrid algorithm pins or relocates live objects to the to-space, leaving the garbage unpinned objects in the from-space. But, it cannot immediately free unpinned objects of the from-space as they may be accessed by concurrent tasks that have not detected that the algorithm has relocated them. Specifically, a read barrier issued by a concurrent task may attempt to pin an unpinned object of the from-space because it has not reached the point where it discovers the forwarding pointer. The algorithm can only free objects in the from-space after all barriers have crossed this "discovery point" and discovered the relocation. The read barrier discovers the relocation when it attempts to pin the object with a cas operation (compare-and-swap), but the operation fails because the collector has forwarded the object. (See line 11 in Figure 4.3: the location ℓ' may be in the from-space because the object has been relocated to location ℓ''. In this case, the procedure pin_region discovers the forwarding pointer to location ℓ'' after a cas fails.)

To ensure that all read barriers have crossed the discovery point, the hybrid algorithm employs epoch-based reclamation [56]. [1] This technique *retires* from-space objects instead of freeing them, marking the objects with the current global *epoch*. It frees retired objects only after all the read barriers have crossed into a future epoch. Because epochs are incremented after read barriers cross the discovery point, all retired objects from the old epoch become safe to reclaim.

[1] Epoch-based reclamation is a technique developed for lock-free data structures which face a similar problem: when a thread removes a node from a lock-free data structure, it is not safe to immediately free that node, because other threads may still access the node.

To amortize the work cost of retiring, our algorithm retires whole pages instead of individual objects. That is, the algorithm retires a page only if all objects within that page are garbage. As an optimization, the algorithm avoids the cost of retiring by directly freeing pages that are not susceptible to entanglement. (For example, if a garbage page is only reachable from the "local roots" of the collection, then the algorithm frees the page directly.)

After the algorithm retires the unpinned objects of the from-space, it traces the to-space to unmark the marked objects.

Show me a completely smooth operation and I'll show you someone who's covering mistakes. Real boats rock.

Frank Herbert, Dune

5 Provable Efficiency

Taking into account all the costs of running the program (including memory management), we are interested in bounding the ***work*** (total number of instructions executed) and ***space*** (peak memory footprint) required for executing a program on P cores. Ideally, in parallel computing, these bounds would be stated in terms of the work and space required for sequential execution, eliminating the need to reason about interleaving of parallel tasks. However, not all programs, particularly those with races, admit equivalent sequential executions; their behavior can diverge arbitrarily from sequential executions.

Given the complexity of accounting for races in the work and space cost, we prove our bounds in two stages. We first consider determinacy-race-free programs which are deterministic in the sense that all runs of the program execute the same instructions. Even for such programs, the order in which parallel tasks complete can be nondeterministic due to the scheduler, potentially leading to varying space requirement. To account for this scheduling-level nondeterminism, we show that it suffices to consider a "little bit of" non-determinism by defining a metric, which we call unordered reachable space. This quantity bounds the reachable space over all sequential computations, where the two sides of a parallel pair are executed in either order (i.e., left before right, and right before left).

We then prove work and space bounds for P-processor executions of determinacy-race-free programs. For space, we prove that for such a program with unordered reachable space of R, any P-processor run with our coscheduling algorithm requires $O(R \cdot P)$ space. This bounds the maximum amount of memory required by the memory manager for executing the program on P processors. This upper bound is tight for our setting, because a work-stealing task scheduler, like the one we use, it is expected that parallel executions can require at least P times as much space [48]. We also prove that the total work for a P-processor execution is $O(W + R \cdot P)$, where W is the work of the computation, i.e., the time for a sequential run. This bound includes

$$
\begin{array}{rlll}
\textit{Variables} & x, f & & \\
\textit{Numbers} & m & \in & \mathbb{N} \\
\textit{Memory Locations} & \ell & & \\
\textit{Types} & \tau & ::= & \texttt{nat} \mid \tau \times \tau \mid \tau \to \tau \mid \tau\ \texttt{ref} \\
\textit{Storables} & s & ::= & m \mid \texttt{fun}\ f\ x\ \texttt{is}\ e \mid \langle \ell, \ell \rangle \mid \texttt{ref}\ \ell \\
\textit{Expressions} & e & ::= & \ell \mid s \mid x \mid e\ e \mid \langle e, e \rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{ref}\ e \mid\ !\, e \mid e := e \mid \langle e \parallel e \rangle \\
\textit{Memory} & \mu & \in & \textit{Locations} \rightharpoonup \textit{Storables} \\
\textit{Task Identifiers} & u, v & & \\
\textit{Task Tree} & T & ::= & \mathsf{ALeaf}(u) \mid \mathsf{PLeaf}(u) \mid \mathsf{Par}(u, T, T)
\end{array}
$$

Figure 5.1: Syntax

the cost for garbage collection. The additive term $R \cdot P$ is a consequence of the heap migrations performed by the heap scheduler, as in the worst case, these migrations may trigger P extra garbage collection, each requiring $O(R)$ work.

We then extend this result by considering programs with nondeterministic mutable effects, including those exhibiting races. To account for the impact of races, we give our language a cost semantics that assigns each computation a *race factor* that bounds the size of the memory that may be accessible via non-deterministic races. We then define a sequentialized cost R^* for such executions of racy programs, by considering the program to execute in parallel and then retrieving a sequential cost from it. For a parallel program with sequential work W and sequentialized space R^*, we show the following work and space bounds for P-core runs:

- total parallel work including the cost of garbage collection is $W + (R^* + r) \cdot P$ work, and
- total parallel space is $(R^* + r) \cdot P$.

Note that these results are similar to the bounds for race-free programs, where the race factor r is zero.

5.1 Revisiting Language Syntax

In this section, we revisit the syntax of our core language, which we use to define the various cost metrics of this section. Figure 5.1 gives the syntax. The syntax is mostly similar to our language in Chapter 2.

Types. The types include a base type of natural numbers, function types and product types for expressing parallel pairs. The type system also supports mutable references.

Memory Locations and Storables We distinguish between storables s, which are always allocated in the heap, and memory locations ℓ. Storables include natural numbers, named recursive functions, pairs of memory locations, and mutable references to other memory locations. Storables step to locations. Locations are the only irreducible form of the language.

Expressions Expressions in our language include variables, locations, storables, and introduction and elimination forms for the standard types. Parallelism is expressed using parallel pairs ($\langle e \parallel e \rangle$). For an expression e, we use $\mathcal{L}(e)$ to denote the set of locations referenced by it.

Memory In order to give an operational semantics for memory effects, we include a map μ from locations to storables. We refer to μ as the memory, $\mathrm{dom}(\mu)$ for the set of locations mapped by μ, $\mu(\ell)$ to look up the storable mapped to ℓ and $\mu[\ell \hookrightarrow s]$ to extend μ with a new mapping.

5.2 Memory Management: An Abstract View

In this section, we present an abstract view of our memory manager, to emphasize key aspects and assumptions for understanding its theoretical cost behavior. Practical implementation details for optimizing efficiency and scalability are discussed in Chapter 4 and Chapter 6.

5.2.1 Data Structures

Our memory manager organizes the memory into a hierarchy of heaps that mirrors the task tree. The heaps of this tree are partitioned among P processors by our heap scheduler that follows closely the decisions of the task scheduler—a work-stealing scheduler in which parallel execution starts when a *thief* processor ***steals*** a task from a *victim* processor. Working with the task scheduler, the heap scheduler assigns each processor a ***heap cluster***, denoted as M_p for processor p (See Section 2.3.2 for full details.)

Our memory manager supports independent garbage collection of each heap cluster by accounting for all the inter-heap pointers. To account for these pointers, it tracks memory locations/objects that are the targets of these pointers and keeps them live. A key aspect of our memory manager is that it "coarsens" the tracking of inter-heap pointers using the heap clusters. If there is an inter-heap pointer between heaps belonging to the same cluster, then that inter-heap pointers is not assumed to be live during garbage collection. Specifically, if location ℓ in heap h has an inter-heap pointer to location ℓ' in h', then the memory manager only considers ℓ' as a root, if h and h' are in different heap clusters. When the two heaps h and h' are assigned to the same processor, they are garbage collected together by that processor, eliminating the need to track pointers between them.

To do this, it maintains the following for each processor p:

1. a snapshot, $\mathsf{S}(p)$, that accounts for memory locations referenced by stolen tasks at the time of steal,
2. a remembered set, $\mathsf{R}(p)$, that tracks all locations of the processor's heap cluster that may be referenced by pointers from heaps not assigned to the processor p.

5.2.2 Maintaining Snapshots and Remembered Sets

Figure 5.2 illustrates the actions of our heap scheduler and also shows how the memory manager maintains processor-local remembered sets and snapshots. When a processor reads/writes a memory location in another processor's heap set, the memory manager adds the location to the corresponding processor's remembered set. As tasks join and their heaps are merged or reassigned to the same processor, the memory manager discards stale snapshot and remembered set entries, ensuring that only references between processors remain in the remembered sets and snapshots.

Our snapshot accounts for locations referenced by stolen tasks at the time of steal, which may later create up pointers into the heap set. By recording these locations during the steal, we eliminate the need to detect up pointers that could be created by subsequent allocations. The remembered set $\mathsf{R}(p)$ includes any locations read using existing pointers, down pointers created by mutable updates, and entanglement sources for cross pointers.

We define reachability within a heap cluster M_p as the set of locations that are reachable from the snapshot $\mathsf{S}(p)$ and remembered set $\mathsf{R}(p)$, following only the pointers within the heap cluster. Our snapshots and remembered sets together guarantee the following **invariant**: For any processor p with heap cluster M_p, any locations in M_p that are read/updated by tasks on other processors are reachable from locations either in snapshot $\mathsf{S}(p)$ or remembered set $\mathsf{R}(p)$. The invariant is maintained by the following actions (Figure 5.2).

Steal. When a thief processor p steals a task v from a victim processor q, the processor p creates a new heap h_v for task v and reinitializes its heap cluster M_p. This creates pointers from the thief's heap cluster to the victim's heap cluster because the stolen task, specifically its expression, has pointers to the victim's heap cluster. To track these inter-heap pointers, the processor p inserts them to the snapshot $\mathsf{S}(q)$ of the victim q.

The procedure StealFrom in Figure 5.2 shows how the snapshot of victim q is updated on a steal. The processor p adds to the snapshot $\mathsf{S}(q)$ entries of the form $\{$*root:* ℓ, *from-heap:* $h_v\}$, where ***root*** is the destination of some pointer from outside the heap cluster and ***from-heap*** corresponds to the heap that holds the pointer, which in this case is the heap h_v of the stolen task v. As we show, the from-heap helps us identify stale remset entries.

Surrender. As soon as a processor finishes its task, it ***surrenders*** its heaps and its remset to the sibling's processor. The procedure SurrenderTo in Figure 5.2 shows the pseudocode where a processor p surrenders its heap cluster M_p and its snapshot and remset, $\mathsf{S}(p)$ and $\mathsf{R}(p)$, to processor q. As a result of this, all pointers between the heap clusters M_p and M_q become internal to M_q and their corresponding entries in the remset become stale. We can identify stale entries using the from-heap: a remset entry of the form $\{$*root:* ℓ, *from-heap:* $h\}$ is ***stale*** when the heap $H(\ell)$ of location ℓ and the from-heap h are in the same heap cluster.

Mutable Read. The procedure Read intercepts mutable reads. When a task v (at processor p) reads location ℓ' by dereferencing location ℓ, we check whether location ℓ' is in another

```
1  // let p be a processor, S(p) is its snapshot, R(p) is its remembered set, M_p is the heap cluster
2  var p, S(p), R(p), M_p
3
4  // Processor p steals task v, evaluating expression e, from processor q
5  procedure StealFrom (q, v, e):
6    M_p = {h_v}
7    for location ℓ ∈ ℒ(e):
8      S(q) = S(q) ∪ {root: ℓ, from-heap: h_v}
9
10 // Processor p surrenders to q after finishing task v
11 procedure SurrenderTo (q, v):
12   M_q = (M_p ∪ M_q)
13   S(q) = S(p) ⊎ S(q)
14   R(q) = R(p) ⊎ R(q)
15   S(p) = ∅
16   R(p) = ∅
17
18
19 // Let u be the active task performing reads/writes on processor p
20
21 procedure write (ℓ, ℓ′):
22   // A mutable write may destroy an existing pointer.
23   var ℓ″ = μ(ℓ)
24   var q : H(ℓ) ∈ M_r
25   var r : H(ℓ″) ∈ M_q
26
27   // account for a new pointer being created.
28   var q : ℓ′ ∈ M_q
29   if (H(ℓ) ∉ M_q):
30     R(q) = R(q) ∪ {root: ℓ′, from: ℓ, from-heap: H(ℓ)}
31
32   // perform the write operation
33   μ(ℓ) := ℓ′
34
35 procedure read (ℓ):
36   var ℓ′ = !ℓ
37   if (H(ℓ′) ∉ M_p):
38     var q : H(ℓ′) ∈ M_q
39     R(q) = R(q) ∪ {root: ℓ′, from-heap: h_u}
40     return μ(ℓ)
```

Figure 5.2: Maintenance of the snapshots, remembered sets, and heap clusters.

processor's heap cluster. If ℓ' is in some other processor q's heap cluster, then to q's remset, the entry $\{$*root:* ℓ', *from-heap:* $h_v\}$, where heap h_v corresponds to the reader task v.

Mutable Write. The procedure Write accounts for destructive mutable updates. It considers a task v creating a pointer from location ℓ to location ℓ'. The write barrier accounts for a new pointer being created. We check whether ℓ and ℓ' belong to different heap clusters (i.e., check if the pointer is between two heap clusters). In that case, suppose location ℓ' is in the heap cluster M_q of processor q. Then, we add the entry $\{$*root:* ℓ', *from:* ℓ, *from-heap:* $H(\ell)\}$ to its remset; the heap $H(\ell)$ denotes the heap of location ℓ. For this case, we also track the ***from*** location which is the reference that holds the pointer. This allows us to inspect stale entries more aggressively: if this pointer is deleted by a future mutable update, we can disregard this entry at the time of collection. This can be checked by dereferencing location ℓ and confirming if it points to location ℓ'.

We note here a special case of pointers that are tracked by our remset: ***down pointers***. The entry $\{$*root:* ℓ', *from:* ℓ, *from-heap:* $H(\ell)\}$ is for a down pointer if the from-heap $H(\ell)$ (corresponding to location ℓ) is an ancestor of heap $h_{\ell'}$ of location ℓ'. Once the from-heap is in the same heap-set or the down pointer is deleted (the from location stops pointing to the root location), the entry becomes stale and we remove it.

5.2.3 Down Pointers Assumption

We make a simplifying assumption for our analysis: we assume that down pointers, as stored by our remembered sets, are not conservative, i.e., locations pointed by a down pointer in the remembered set are live. This assumption does not affect the correctness of our algorithm but it simplifies its analysis. While this assumption may not hold universally for all parallel fork-join programs, it is reasonable in practice. In the fork-join model, children tasks primarily use down pointers to return results back to their parent task. A child task can allocate a result object in its heap and then create a down pointer to store the result in memory allocated by the parent. Thus, until the child task joins, we expect the down pointer to be live. After the join, we merge the child's heap with the parent's heap, turning the down pointer into an internal pointer, at which point the assumption no longer applies.

Note that this assumption is not relevant to purely functional programs, as they do not have mutable references and therefore do not have down pointers.

5.2.4 Collection Policy and Algorithm

We use a collection policy that determines when a processor garbage-collects.

Collection policy Each processor independently triggers collections based on a processor-local counter λ. Whenever the size of a processor's heap-set M exceeds its counter λ by a constant factor i.e., when $|M| > \lambda * \kappa$, that processor performs a collection on its heap cluster. The constant factor κ is a tunable parameter in our design. The processor performs collection by

pausing its current active task and then executing a collection algorithm (described below) on its heap cluster. After the collection finishes, the processor resets the counter λ to the amount of memory that survives the collection and resumes its task.

Collection algorithm When a processor decides to collect, it runs a tracing collection algorithm. The algorithm first filters out stale remset entries and then traces all objects reachable from the remaining entries, without tracing pointers that go out of the heap cluster.

Because the collection proceeds without stopping the world, the locations in the heap cluster may be concurrently accessed by other processors. Our collection algorithm is careful to not change the addresses of objects that are accessible by other processors. We present the full details of our algorithm in the Implementation section (Chapter 4), which combines both in-place and moving collection algorithms. These details are important in practice, but not for our theoretical results: for our theory, the only requirement is that the algorithm uses $O(|M|)$ work and space for collecting a heap cluster M of size $|M|$. Any algorithm that ensures the safety of concurrent accesses and adheres to the above work and space costs would suffice. For example, an in-place collection algorithm such as mark-and-sweep could be used to meet our bounds.

5.2.5 Structural Properties of the Heap Clusters

In Chapter 2, we showed how our coscheduling algorithm partitions the heap tree into clusters and proved the following invariants. We restate the invariants here and also prove an additional lemma, that is useful for our efficiency proofs. The invariants state the following for each processor p with heap cluster M_p:

1. if a processor p is executing an active task, then the heap of the task is assigned to p
2. if a suspended heap is in the cluster M_p then at least one of its children is also in M_p
3. every passive heap belongs to the same processor as its sibling.

Intuitively, these invariants guarantee that each heap cluster is a subtree of the heap tree. These invariants imply properties that are useful for our proofs. We discuss them below.

We restate a guarantee of these invariants, proved in Chapter 2.

Lemma 3. *Every suspended heap has an active descendant heap assigned to the same processor.*

Based on these properties we prove another structural property.

Lemma 4. *If heap h is in a heap cluster M_p, any ancestor h' of h not in M_p, is an ancestor of all heaps in M_p.*

Proof. Let $\leq_H$ be a relation on heaps, which holds when a heap is an ancestor of another in the heap tree, i.e., $h \leq_H h'$ if and only if heap h is an ancestor of heap h'. Suppose there are three heaps h_u, h_v, h_w such that h_u is an ancestor of h_v and both h_v and h_w are in the heap set M_p. Then, we must show that h_u is an ancestor of h_w.

We consider four cases. First, suppose both h_v and h_w are suspended. In this case, by Lemma 3 above, at least one of their descendants must be active on processor p. Since only one heap can be active on processor p, the heaps h_v and h_w must share the same descendant. Two

heaps in a tree share a descendant only when they are in an ancestor-descendant relationship. Now, since the heaps h_v and h_w are in an ancestor-descendant relationship and heap h_u is an ancestor of h_v, then we have that h_u and h_w are also in an ancestor-descendant relationship. If h_w is an ancestor of h_u, which in turn is an ancestor of heap h_v, then Invariant 2 implies that h_u is in the heap set M_p, which is a contradiction. Thus, h_w must be a descendant of heap h_u/

Second, suppose h_v is passive and h_w is suspended. Then, in this case the sibling heap of h_v, say h'_v, is in the heap cluster M_p (Invariant 3). Since h_u is an ancestor of h_v, it is also an ancestor of its sibling h'_v. Because heap h'_v is suspended, otherwise the heaps would be joined and removed, we can apply the first case with heap h_u being an ancestor of h'_v, and heaps h'_v and h_w being suspended.

Third, suppose h_v is suspended and h_w is passive. Since h_w is passive, it must have a suspended sibling h'_w. Applying the first case with h_u as an ancestor of h_v, and h_v and h'_w suspended, we get that h_u is an ancestor of h'_w. Since h'_w and h_w are siblings, h_u is an ancestor of h_w. Fourth, suppose h_v is passive and h_w is passive. In this case, we can establish that h_u is an ancestor of h_w by applying the first case with their suspended siblings h'_v and h'_w.

□

5.3 Determinacy Race Free Programs

In this section, we assume programs are deterministic in the sense of being *determinacy-race-free* (defined below), and show work and space bounds for their execution on P-processors. The determinism guarantees that parallel and sequential executions are similar enough in order to prove our bounds. A ***determinacy race*** occurs when two concurrent tasks access the same memory location, and at least one of these accesses modifies the location [78]. This is essentially the same notion as a "data race"; however there is a subtlety which leads us to prefer the term "determinacy race". Programs which are determinacy-race-free are **deterministic in a strong sense**: not only is the final result the same in every possible execution, but also the specifics of how that result is computed are precisely the same every time.[1]

To prove the bounds, we first define a cost metric called *unordered reachable space*, which is a baseline space cost for sequential executions of deterministic programs.

5.3.1 Unordered Reachable Space: Sequential Baseline

In this section, we give a cost semantics that defines the ***unordered reachable space***, which is the space usage of sequential executions of determinacy-race-free programs. The sequential executions we consider, however, allow for some flexibility in the evaluation order. Specifically, the semantics allows for some non-determinism in the order of evaluation of parallel pairs, i.e., it can non-deterministically decide which side of the pair to evaluate first, either left before right or right before left. However, the semantics fully evaluates one side before evaluating the

[1]This is also known as *internal determinism* [45] at the level of individual memory reads and writes.

other. This ensures that the evaluation proceeds in a "depth-first" fashion, characteristic of a sequential execution, as the components of a parallel pair are never interleaved.

We include this non-determinism because, even though determinacy-race-free programs offer strong program-level guarantees about consistency of instructions for all executions, task scheduling itself inherently relies on randomized decisions and is non-deterministic. The specific order in which tasks corresponding to a parallel pair finish can vary across parallel executions. For example, if two tasks u and v correspond to components of a parallel pair, then either u might finish before v, or v might finish before u, fundamentally requiring different space usage. The unordered reachable space accounts for this non-determinism. We sometimes refer to unordered reachable space as unordered sequential space.

Cost Semantics. The cost semantics for computing unordered reachable space is a standard transition (small-step) semantics of a functional language with mutation. We write the semantics relation as:

$$\rho \vdash \mu \,;\, T \,;\, R \,;\, e \to \mu' \,;\, T' \,;\, R' \,;\, e'.$$

The cost semantics sequentially executes the given parallel program and computes its space usage. Space usage is defined as the peak memory footprint of an evaluation. To calculate space usage, we treat the memory store, μ, as a directed graph, where nodes represent locations and edges are the pointers between locations. A location ℓ is said to point to location ℓ' if the storable at ℓ has a pointer to the storable at ℓ'. Locations that are explicitly referenced by the program expression are called roots. The context ρ tracks the roots of the program and the rules of the semantics extend the context appropriately to track all roots. Only locations reachable (in the memory graph) from the context ρ count towards the space usage of the program. At some step t of the evaluation, suppose the set of roots (***root set***) as ρ^t. The set of reachable locations is denoted as $\mu^+(\rho^t)$ and the cumulative size of storables mapped by this set is represented as $|\mu^+(\rho^t)|$. Space usage is formally defined to be $\max_t |\mu^+(\rho^t)|$.

The context ρ tracks the roots of the program and the rules extend the set of roots based on the locations mentioned in the program expression. For example, the rule ASL (application step left) adds all the locations mentioned by expression e_2 to the context ρ before evaluating expression e_1. This ensures that the locations mentioned in expression e_2 are considered as roots. The semantics maintains the space usage of the evaluation in R. Since the reachable memory can increase only after an allocation, R is updated only after the program allocates (see rule ALLOC). This rule computes the reachable memory from the root set $(\rho \cup \mathsf{locs}(s))$, adds the size of the newly allocated storable and updates R if the space usage has increased.

Sequentially executing parallel pairs. The key rules that execute the program sequentially are rules ExL and ExR. The rule ExL fully evaluates the left side, expression e_1 of the parallel pair, before evaluating the right side, expression e_2. The rule ExR is symmetric, as it instead full evaluates the right side of the parallel pair, expression e_2, and then evaluates the left side. Note that the relation used in these rules is labeled with the side picked first, for example, in ExL we use the relation $\to_L$.

$$\frac{R' = \mathsf{max}(\ |\mu^{+}(\rho \cup \mathsf{locs}(s))| + |s|, R\ ) \qquad \ell \notin \mathrm{dom}(\mu) \qquad \mu' = \mu[\ell \hookrightarrow s]}{\rho \vdash \mu\ ;\ \mathsf{ALeaf}(u)\ ;\ R\ ;\ s \to \mu'\ ;\ \mathsf{ALeaf}(u)\ ;\ R'\ ;\ \ell}\ \textsc{Alloc}$$

$$\frac{\rho \cup \{\mathsf{locs}(e_2)\} \vdash \mu\ ;\ T\ ;\ R\ ;\ e_1 \to \mu'\ ;\ T'\ ;\ R'\ ;\ {e_1}'}{\rho \vdash \mu\ ;\ T\ ;\ R\ ;\ (e_1\ e_2) \to \mu'\ ;\ T'\ ;\ R'\ ;\ (e_1'\ e_2)}\ \textsc{ASL}$$

$$\frac{\rho \cup \{\ell_1\} \vdash \mu\ ;\ T\ ;\ R\ ;\ e_2 \to \mu'\ ;\ T'\ ;\ R'\ ;\ {e_2}'}{\rho \vdash \mu\ ;\ T\ ;\ R\ ;\ (\ell_1\ e_2) \to \mu'\ ;\ T'\ ;\ R'\ ;\ (\ell_1\ e_2')}\ \textsc{ASR}$$

$$\frac{\mu(\ell_1) = \texttt{fun}\ f\ x\ \texttt{is}\ e_b}{\rho \vdash \mu\ ;\ \mathsf{ALeaf}(u)\ ;\ R\ ;\ (\ell_1\ \ell_2) \to \mu\ ;\ \mathsf{ALeaf}(g \oplus n)\ ;\ R\ ;\ [\ell_1, \ell_2\ /\ f, x]e_b}\ \textsc{App}$$

$$\frac{\mu(\ell_1) = \texttt{ref}\ \ell_2}{\rho \vdash \mu\ ;\ \mathsf{ALeaf}(u)\ ;\ R\ ;\ (!\,\ell_1) \to \mu\ ;\ \mathsf{ALeaf}(g \oplus n)\ ;\ R\ ;\ \ell_2}\ \textsc{Bang}$$

$$\frac{}{\rho \vdash \mu_0[\ell_1 \hookrightarrow s]\ ;\ \mathsf{ALeaf}(u)\ ;\ R\ ;\ (\ell_1 := \ell_2) \to \mu_0[\ell_1 \hookrightarrow \texttt{ref}\ \ell_2]\ ;\ \mathsf{ALeaf}(g \oplus n)\ ;\ R\ ;\ \ell_2}\ \textsc{Upd}$$

$$\frac{}{\rho \vdash \mu\ ;\ \mathsf{ALeaf}(u)\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to \mu\ ;\ \mathsf{Par}(u, \mathsf{ALeaf}(v), \mathsf{ALeaf}(w))\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle}\ \textsc{Fork}$$

$$\frac{}{\rho \vdash \mu\ ;\ \mathsf{Par}(u, \mathsf{ALeaf}(v), \mathsf{ALeaf}(w))\ ;\ R\ ;\ \langle \ell_1 \parallel \ell_2\rangle \to \mu\ ;\ \mathsf{ALeaf}(u)\ ;\ R\ ;\ \langle \ell_1, \ell_2\rangle}\ \textsc{Join}$$

$$\frac{\begin{array}{c}\rho \cup \{\mathsf{locs}(e_2)\} \vdash \mu\ ;\ T_1\ ;\ R\ ;\ e_1 \to^{*} \mu'\ ;\ T_1'\ ;\ R'\ ;\ \ell_1 \\ \rho \cup \{\ell_1\} \vdash \mu'\ ;\ T_2\ ;\ R'\ ;\ e_2 \to^{*} \mu''\ ;\ T_2'\ ;\ R''\ ;\ \ell_2\end{array}}{\rho \vdash \mu\ ;\ \mathsf{Par}(u, T_1, T_2)\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to_L \mu''\ ;\ \mathsf{Par}(u, T_1', T_2')\ ;\ R''\ ;\ \langle \ell_1 \parallel \ell_2\rangle}\ \textsc{ExL}$$

$$\frac{\begin{array}{c}\rho \cup \{\mathsf{locs}(e_1)\} \vdash \mu\ ;\ T_2\ ;\ R\ ;\ e_2 \to^{*} \mu'\ ;\ T_2'\ ;\ R'\ ;\ \ell_2 \\ \rho \cup \{\ell_2\} \vdash \mu'\ ;\ T_1\ ;\ R'\ ;\ e_1 \to^{*} \mu''\ ;\ T_1'\ ;\ R''\ ;\ \ell_1\end{array}}{\rho \vdash \mu\ ;\ \mathsf{Par}(u, T_1, T_2)\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to_R \mu''\ ;\ \mathsf{Par}(u, T_1', T_2')\ ;\ R''\ ;\ \langle \ell_1 \parallel \ell_2\rangle}\ \textsc{ExR}$$

$$\frac{\begin{array}{c}\rho \vdash \mu\ ;\ \mathsf{Par}(u, T_1, T_2)\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to_L \mu'\ ;\ \mathsf{Par}(u, T_1', T_2')\ ;\ R'\ ;\ \langle \ell_1' \parallel \ell_2'\rangle \\ \rho \vdash \mu\ ;\ \mathsf{Par}(u, T_1, T_2)\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to_R \mu''\ ;\ \mathsf{Par}(u, T_1'', T_2'')\ ;\ R''\ ;\ \langle \ell_1'' \parallel \ell_2''\rangle \qquad R' \geq R''\end{array}}{\rho \vdash \mu\ ;\ \mathsf{Par}(u, T_1', T_2')\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to \mu'\ ;\ \mathsf{Par}(u, T_1', T_2')\ ;\ R'\ ;\ \langle \ell_1' \parallel \ell_2'\rangle}\ \textsc{PickL}$$

$$\frac{\begin{array}{c}\rho \vdash \mu\ ;\ \mathsf{Par}(u, T_1, T_2)\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to_L \mu'\ ;\ \mathsf{Par}(u, T_1', T_2')\ ;\ R'\ ;\ \langle \ell_1' \parallel \ell_2'\rangle \\ \rho \vdash \mu\ ;\ \mathsf{Par}(u, T_1, T_2)\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to_R \mu''\ ;\ \mathsf{Par}(u, T_1'', T_2'')\ ;\ R''\ ;\ \langle \ell_1'' \parallel \ell_2''\rangle \qquad R'' > R'\end{array}}{\rho \vdash \mu\ ;\ \mathsf{Par}(u, T_1', T_2')\ ;\ R\ ;\ \langle e_1 \parallel e_2\rangle \to \mu''\ ;\ \mathsf{Par}(u, T_1'', T_2'')\ ;\ R''\ ;\ \langle \ell_1'' \parallel \ell_2''\rangle}\ \textsc{PickR}$$

Figure 5.3: Sequential Cost Semantics

The rules PickL and PickR then select the order in which the space usage is higher. In this way, the cost semantics "closes over" all possible executions and returns the maximum space across all such executions. It thus assigns a unique cost to the program.

Even though the semantics evaluates the program sequentially, it includes the rules Fork and Join for managing task creations and deletions; all tasks execute sequentially. Other rules in the semantics are standard and we skip their description for sake of brevity. The semantics for a program e starts with the state $(\emptyset\,;\,\mathsf{ALeaf}(u)\,;\,0\,;\,e)$, where u is the name of the root task. Suppose it terminates with the state $(\mu\,;\,\mathsf{ALeaf}(u)\,;\,R\,;\,\ell)$. Then the unordered reachable space of the program is R.

5.3.2 Space Bound

Theorem 3 (Space Bound)**.** *Given a determinacy-race-free program with unordered reachable space R, its parallel execution on P processors with our memory manager uses at most $O(R{\cdot}P)$ memory.*

Proof. In our collection policy, each processor maintains a counter and keeps the size of its heap set within some constant κ times this counter. If the size exceeds this limit, then the processor collects. For processor p, $|M_p| < \kappa{\cdot}\lambda_p$, where λ_p denotes the counter and $|M_p|$ denotes the size of its heap cluster. By Lemma 5 (stated and proved below) the counter $\lambda_p \leq R$. Thus, the size of M_p is less than $O(\kappa{\cdot}R)$ when the processor is not collecting. The collection algorithm requires at most $O(|M_p|)$ memory when it collects. Thus, the maximum space used by processor p is $O(R)$. □

Bounding the counter. After every collection, the processor updates its counter to the size of memory that survived the collection. Because the counter is not updated otherwise, its value is bounded by this size. We show that whenever a processor garbage collects, the size of memory it preserves is bounded by the unordered reachable space R. Thus, the counter at every processor is bounded by R.

Lemma 5. *At any step during a parallel execution of a deterministic program, if a processor garbage collects, the size of objects it preserves from its the root set and its remembered set is bounded by the unordered reachable space.*

Proof. The crux of the proof is to show for a garbage collection at any processor, the amount of memory preserved by the processor is equivalent to the reachable memory in some sequential execution. Consider an arbitrary step t taken by processor p before it garbage collects. We construct a sequential execution, whose reachable memory at some step t', matches the memory preserved in the parallel evaluation after garbage collection at step t. We construct such a sequential execution by carefully defining an order of evaluation for parallel pairs.

Defining a sequential execution. Suppose processor p is executing task u at step t. Let the root to leaf path in the task tree be $\mathsf{RootLeaf}(u) = u_0, u_1 \ldots u_n$, where $u_n = u$. Also, let

$v_1 \ldots v_n$ be the siblings of the tasks $u_1 \ldots u_n$; note that the root task u_0 has no siblings. We define a precedence relation $<_p$ for these tasks as follows:

1. If task v_i has terminated by step t, then $v_i <_p u_i$.
2. Otherwise, $u_i <_p v_i$.

For the other tasks and their siblings, let the relation $<_p$ be arbitrary. This precedence relation defines a sequential execution $\mathcal{S}$, where siblings tasks are evaluated in the order defined by $<_p$. The sequential execution starts with the root task u_0. Then, the order of execution for any two siblings u_i and v_i, is given by the precedence relation. If $u_i <_p v_i$ then u_i is evaluated completely before v_i is executed. Otherwise v_i is evaluated first. This sequential execution corresponds to an execution in the semantics for unordered reachable space. Thus, the reachable space at any step of this sequential execution is bounded by R^*.

Reachability Equivalence. Since the program is deterministic, there exists a step (t') in the sequential execution (S) where it executes the same instruction that processor p executes at step t in the parallel execution. We will show that the amount of memory preserved in heap set M_p after a garbage collection at step t, is upper bounded by the amount of reachable memory in the sequential execution at step t'.

To do this, we first observe that only the actions of a specific subset of tasks affects the garbage collection of heaps in cluster M_p:

1. Tasks within M_p: these are the tasks that directly allocated and manipulated the memory in the heap cluster M_p, affecting the garbage collection in M_p
2. Descendant tasks of stolen tasks: These tasks have been executed (and are potentially running) on other processors, and potentially access the objects in heap cluster M_p, requiring the memory manager to update the remembered sets and snapshots appropriately.

The actions of other tasks are irrelevant for garbage collection at processor p. This is because of race freedom and the absence of cross-pointers. Race freedom guarantees that the order in which memory locations are accessed and modified is consistent across all executions. It also guarantees the absence of cross-pointers (see Chapter 3), meaning actions of concurrent tasks (those not in ancestor-descendant relationships) are irrelevant for garbage collection. Since there are no cross pointers, the collection of objects within a heap cluster M_p is solely determined by actions of ancestor/descendant tasks.

We formally define the relevant tasks as follows. Let M_p be the heap cluster assigned to processor p and let $\mathsf{T_p}$ be the tasks whose heaps are in M_p, i.e., $\mathsf{T_p} = \{u \mid h_u \in M_p\}$. Let $\mathsf{S_p}$ be the set of successor tasks that were stolen from processor p and, let $\mathsf{D_p} \supseteq \mathsf{S_p}$ be the set of all the tasks in $\mathsf{S_p}$ and their descendants. Let $\mathsf{A_p}$ be the set of ancestor tasks of those in $\mathsf{T_p}$. The liveness of the memory in M_p, is only influenced by tasks in $\mathsf{T_p}$, $\mathsf{A_p}$ and $\mathsf{D_p}$.

Comparing task execution. First, let's analyze the impact of tasks in $\mathsf{T_p}$. For tasks in $\mathsf{T_p}$ that are also on the root-to-leaf path $\mathsf{RootLeaf}(u)$, the same instructions have been executed up to steps t and t' because they are ancestors of the current task u. For tasks in $\mathsf{T_p}$ that are

not on the root-to-leaf path $\mathsf{RootLeaf}(u)$, they must be passive leaves and siblings of tasks in $\mathsf{RootLeaf}(u)$; these tasks have been fully evaluated in the parallel execution (they are passive). Due to the precedence relation, these tasks must also have been fully evaluated in the sequential execution by step t'.

Second, consider the tasks in $\mathsf{A_p}$. These task are guaranteed to be on the root-to-leaf path $\mathsf{RootLeaf}(u)$. We can show by contradiction. Suppose there is a task $u_A \in \mathsf{A_p}$ which is an ancestor of some task in T_p, but is not on the root-to-leaf path. Then, by Lemma 4, the task u_A is an ancestor of the leaf task u. This is a contradiction because the root to leaf path contains all ancestors of the task u. Thus, the tasks in $\mathsf{A_p}$ are guaranteed to be on the root-to-leaf path $\mathsf{RootLeaf}(u)$. Given this, it is guaranteed that for these tasks, the same instructions have been executed up to steps t and t' of the parallel and sequential execution respectively, because at the steps they are executing the same instruction.

Third, consider tasks in D_p (stolen tasks and their descendants). These tasks potentially affect the memory preserved by the memory manager, because they may read locations in the heap set M_p, causing our memory manager to add any such locations to the remembered set (see Section 5.2), and keep them live. However, due to race freedom, a task in D_p can only access locations that were reachable from it at the time of its steal. This is because gaining access to any other memory locations in heap set M_p would require a race, as a concurrent task would have to create a pointer, exposing a new location to the tasks in D_p; this can not happen in a race free program. Therefore, the actions of a task in D_p only add those locations to the remembered set that were already reachable at the time of its steal. We show next that all such locations are reachable in the sequential execution at step t'.

In our sequential execution upto step t', first assume that tasks in D_p have not executed at all (we prove this below). Under this assumption, the reachable space at step t' in the sequential execution is computed with tasks in D_p not having executed, thus including all the locations that are reachable from a task at time of steal in the parallel execution. Thus, the memory preserved by remembered set entries due to stolen tasks is upper bounded by the reachable space in the sequential execution.

To prove the assumption above, consider any task $v \in \mathsf{D_p}$. We define task v' as the the ancestor of v such that $v' \in \mathsf{T_p}$. In the parallel execution, task v has not terminated by step t and thus, v' has not terminated as well. By definition of the set $\mathsf{S_p}$, task v' is a sibling of some task in $\mathsf{RootLeaf}(u)$. In the sequential execution, task v' is thus ordered later than its sibling in the precedence relation and because the sibling, say u', has not terminated v' has not executed. Because v' has not executed, its descendant v has not executed either in the sequential execution.

□

5.3.3 Work Bound

The collection algorithm performs $c \cdot |M|$ units of work to collect $|M|$ amount of memory. We prove the following bound on the work done in all the collections:

Theorem 4 (Work Bound). *If a determinacy-race-free program performs W units of program work in a sequential execution, then the total work on P processors, including the cost of garbage collection, is upper bounded by $O(W + P \cdot R^*)$.*

Proof. Let $|M_p^i|$ be the size of heap cluster before the i*th* collection on worker p. If λ_p^{i+1} is the size of heap cluster after the collection, then the memory reclaimed is $(|M_p^i| - \lambda_p^{i+1})$. The memory reclaimed by all the collections can not be greater than the memory allocated by the program. Thus, if worker p performs n_p collections and α is the total memory allocated then:

$$\sum_p \sum_{i=1}^{i=n_p} (|M_p^i| - \lambda_p^{i+1}) \leq \alpha \tag{5.1}$$

In our collection policy, a worker starts a collection only when the size of its heap cluster grows a constant κ times its counter, i.e, the worker p starts the i*th* collection because $|M_p^i| \geq \kappa * \lambda_p^i$. Moreover, by Lemma 5, the value of the counter λ_p does not exceed R^*. Thus, it follows that:

$$\sum_p \sum_{i=1}^{i=n_p} \lambda_p^{i+1} \leq \sum_p \sum_{i=1}^{i=n_p} \lambda_p^i + \sum_p \lambda_p^{(n_p+1)} \leq \sum_p \sum_{i=1}^{i=n_p} \frac{|M_p^i|}{\kappa} + P \cdot R^*$$

After substituting this in Equation 5.1, it follows that:

$$\sum_p \sum_{i=1}^{i=n_p} |M_p^i| \leq (\alpha + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1} \leq (W + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1}$$

We assume that allocation of one unit of memory requires one unit of work and thus, $\alpha \leq W$. The total memory traced in all collections is $(\sum_p \sum_{i=1}^{i=n_p} |M_p^i|)$. Suppose c_1 is the work efficiency of the collector, i.e., the collection of $|M|$ amount of memory requires $c * |M|$ amount of collection work. Then, the total work done in collections is upper-bounded by $c \cdot (\sum_p \sum_{i=1}^{i=n_p} |M_p^i|) \leq c \cdot (W + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1}$. Thus, total work, including program work and collection work, is $O(W + P \cdot R^*)$. □

5.4 Nondeterministic Programs

In this section, we analyze the work and space costs for nondeterministic parallel programs with arbitrary races. This is challenging because of the unpredictable cost behavior of executions of such programs. Unlike deterministic programs, where all executions perform the same instructions, nondeterministic programs can exhibit arbitrarily different behaviors due to races. To account for this inherent non-determinism, we reason about the work and space costs on a per-execution basis. This means that instead of accounting for all behaviors beforehand, we analyze the costs for a specific execution that has occurred.

We introduce two key metrics to analyze the cost of an execution:

$$
\begin{array}{rrcl}
\textit{Allocation Sets} & \alpha & \subseteq & \textit{Locations} \\
\textit{Task Tree} & T & ::= & \mathsf{PLeaf}(v) \mid \mathsf{ALeaf}(v, \alpha) \mid \mathsf{Par}(v, \alpha, T, T) \\
\textit{Reader, Writer Stores} & \mathsf{R}, \mathsf{W} & \in & \textit{Locations} \rightharpoonup \mathcal{P}(\textit{Tasks} \times \textit{Locations}) \\
\textit{Program State} & S & ::= & (\mathsf{R}\,;\mathsf{W}\,;\mu\,;T\,;e)
\end{array}
$$

Figure 5.4: Additional Syntax for defining race factor

1. **Sequentialized Space $\mathbf{R}^*$** : This metric analyzes generalizes the unordered reachable space for deterministic programs, estimating a sequential space cost even for parallel executions that might not have a sequential equivalent. It does this by simulating sequential execution along each path in the task tree, capturing the maximum memory usage under different task orderings. Sequentialized space addresses the issue that some executions of a racy program may not even be possible on a single processor because they rely on intricate interleaving of concurrent tasks.
2. **Race Factor r**: This metric measures the high watermark of memory impacted with races. We present a cost semantics that tracks the race factor as it executes a program by observing the memory actions of each task and identifying memory "exposed" to concurrently executing tasks via effects. This metric quantifies the additional cost our memory manager incurs for racy programs.

After defining the metrics, we show that, given an execution on P processors with program work W, sequentialized space R^*, and race factor r, the work cost of our memory manager is bounded by $O(W + (R^* + r) \cdot P)$ and the space cost of the execution is $O((R^* + r) \cdot P)$. These bounds show that our memory manager incurs linear work and space overhead, proportional to the amount of memory impacted by races.

5.4.1 Race Factor for Nondeterministic Programs

To account for the impact of races on our memory manager's performance, we define a metric called *race factor*, that measures the amount of memory affected by races. We present a cost semantics that defines the race factor throughout the execution by observing the memory actions of each task and identifying memory "exposed" to concurrently executing tasks via effects. The final race factor for the execution is the maximum value across all steps. This is important because the semantics resolves races when threads join, potentially leading to a reduction in the race factor as the computation proceeds The core language and syntax are similar to our language in Section 2.1. Figure 5.4 shows the new/modified pieces of syntax for the language.

The cost semantics for computing the race factor is a transition relation that steps a program state containing five components: reader store R, writer store W, memory μ, task tree T, and expression e. We write the relation as:

$$(\mathsf{R}\,;\mathsf{W}\,;\mu\,;T\,;e) \rightarrow (\mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;T'\,;e')$$

Figure 5.5 shows the key rules of the semantics. The task tree T is the standard dynamic task

tree, which contains suspended tasks waiting for the children to finish, active leaves that can take more steps, and passive leaves that are waiting for their sibling to finish.

Reader/writer stores. To calculate the race factor, the semantics maintains a reader and a writer store. The ***reader store*** R remembers for each memory location the tasks that reads it along with the (mutable) location it was read from. The reader store is primarily updated by rule Bang (see Figure 5.5), which dereferences a mutable location ℓ to obtain location ℓ'. The rule adds the tuple (v, ℓ) to the reader store of ℓ'. Observe that the mutable dereference is performed on location ℓ, but the reader store is updated for location ℓ'. This is because the reader store tracks the locations obtained from mutable dereferences, not the references themselves.

The ***writer store*** W remembers for each location the reference it is written into along with the task that performed the write. The writer store is updated at the step Dupd, which updates the mutable reference ℓ to point to location ℓ' The step adds the tuple (v, ℓ) to the writer store at location ℓ', where task v is performing the write. Observe that the mutable effect is performed on mutable reference ℓ, but the store is updated for the location ℓ'. This is because the writer store tracks the locations written into mutable references, not the references themselves.

The rule RefAlloc, which allocates mutable references also updates the writer store. It steps the storable `ref` ℓ' to a fresh location ℓ and extends the memory store μ with location ℓ mapped to `ref` ℓ'. It also adds the tuple (v, ℓ) to the set $\mathsf{W}(\ell')$. This can be seen as recording an "initialization write" where the task v writes location ℓ' to initialize the reference ℓ.

Exposed locations. At a given program state $(\mathsf{R}; \mathsf{W}; \mu; T; e)$, we determine the set of locations exposed due to a race, where a task writes to a mutable reference and another concurrent task reads from the reference. Specifically, a location ℓ' is ***exposed***, if a race condition exists such that one task writes ℓ' to a mutable reference ℓ and a different, concurrent task reads ℓ' by dereferencing ℓ. Essentially, the race condition on reference ℓ *exposes* the location ℓ' to a concurrent reader. We define the set of exposed locations, denoted as $\nabla(\mathsf{R}, \mathsf{W})$, as follows.

$$\nabla(\mathsf{R}, \mathsf{W}) = \{\ell' \mid \exists\, u, v, \ell : \mathsf{concurrent}(u, v) \wedge (u, \ell) \in \mathsf{W}(\ell') \wedge (v, \ell) \in \mathsf{R}(\ell')\}$$

Here, $\mathsf{concurrent}(u, v)$ denotes that tasks u and v are not in an ancestor-descendant relationship within the task tree.

Race factor. The ***race factor*** of a given program state $(\mathsf{R}; \mathsf{W}; \mu; T; e)$ is the maximum amount of memory that is reachable from exposed locations along any root-to-leaf path of the task tree. Figure 5.6 calculates the race factor r of the state using a judgement $\mathsf{R}; \mathsf{W}; \mu; A \vdash T \downarrow r$. The context A tracks the allocations along the root-to-leaf path.

The rules of the judgement consider all root-to-leaf paths in the task tree and take the maximum race factor along any path. To do this, consider the rule Par which computes the race factor of a task tree $\mathsf{Par}(v, \alpha, T_1, T_2)$, with suspended task v and its allocations α. The rule computes the race factors r_1 and r_2 along subtrees T_1 and T_2 and takes the maximum. Note that

$$\frac{\mathsf{pure}(s) \qquad \ell \notin \mathrm{dom}(\mu) \qquad \mu' = \mu[\ell \hookrightarrow s]}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{ALeaf}(v,\alpha)\,;s \to \mathsf{R}\,;\mathsf{W}\,;\mu'\,;\mathsf{ALeaf}(v,\alpha \cup \{\ell\})\,;\ell}\ \textsc{PureAlloc}$$

$$\frac{\ell \notin \mathrm{dom}(\mu) \qquad \mu' = \mu[\ell \hookrightarrow \mathtt{ref}\ \ell'] \qquad \mathsf{W}' = \mathsf{W}[\ell' \hookrightarrow \mathsf{W}(\ell') \cup \{(v,\ell)\}]}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{ALeaf}(v,\alpha)\,;\mathtt{ref}\ \ell' \to \mathsf{R}\,;\mathsf{W}'\,;\mu'\,;\mathsf{ALeaf}(v,\alpha \cup \{\ell\})\,;\ell}\ \textsc{RefAlloc}$$

$$\frac{\mathsf{R}\,;\mathsf{W}\,;\mu\,;T\,;r\,;e_1 \to \mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;T'\,;r'\,;e_1{}'}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;T\,;r\,;(e_1\ e_2) \to \mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;T'\,;r'\,;(e_1'\ e_2)}\ \textsc{ASL}$$

$$\frac{\mu(\ell_1) = \mathtt{fun}\ f\ x\ \mathtt{is}\ e_b}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{ALeaf}(v,\alpha)\,;(\ell_1\ \ell_2) \to \mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{ALeaf}(v,\alpha)\,;[\ell_1,\ell_2\ /\ f,x]e_b}\ \textsc{App}$$

$$\frac{\mathsf{R}\,;\mathsf{W}\,;\mu\,;T\,;e \to \mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;T'\,;e'}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;T\,;(!\,e) \to \mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;T'\,;(!\,e')}\ \textsc{BangS}$$

$$\frac{\mu(\ell) = \mathtt{ref}\ \ell' \qquad \mathsf{R}' = \mathsf{R}[\ell' \hookrightarrow \mathsf{R}(\ell') \cup \{(v,\ell)\}]}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{ALeaf}(v,\alpha)\,;(!\,\ell) \to \mathsf{R}'\,;\mathsf{W}\,;\mu\,;\mathsf{ALeaf}(v,\alpha)\,;\ell'}\ \textsc{Bang}$$

$$\frac{\mathsf{W}' = \mathsf{W}[\ell' \hookrightarrow \mathsf{W}(\ell') \cup \{(v,\ell)\}]}{\mathsf{R}\,;\mathsf{W}\,;\mu_0[\ell \hookrightarrow \mathtt{ref}\ \ell'']\,;\mathsf{ALeaf}(v,\alpha)\,;(\ell := \ell') \to \mathsf{R}\,;\mathsf{W}'\,;\mu_0[\ell \hookrightarrow \mathtt{ref}\ \ell']\,;\mathsf{ALeaf}(v,\alpha)\,;\ell'}\ \textsc{Dupd}$$

$$\frac{}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{ALeaf}(u,\alpha)\,;\langle e_1 \parallel e_2\rangle \to \mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{Par}(u,\alpha,\mathsf{ALeaf}(v,\emptyset),\mathsf{ALeaf}(w,\emptyset))\,;\langle e_1 \parallel e_2\rangle}\ \textsc{Fork}$$

$$\frac{\mathsf{R}\,;\mathsf{W}\,;\mu\,;T_1\,;e_1 \to \mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;T_1'\,;e_1'}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{Par}(v,\alpha,T_1,T_2)\,;\langle e_1 \parallel e_2\rangle \to \mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;\mathsf{Par}(v,\alpha,T_1',T_2)\,;\langle e_1' \parallel e_2\rangle}\ \textsc{ParL}$$

$$\frac{\mathsf{R}\,;\mathsf{W}\,;\mu\,;T_2\,;e_2 \to \mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;T_2'\,;e_2'}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{Par}(v,\alpha,T_1,T_2)\,;\langle e_1 \parallel e_2\rangle \to \mathsf{R}'\,;\mathsf{W}'\,;\mu'\,;\mathsf{Par}(v,\alpha,T_1,T_2')\,;\langle e_1 \parallel e_2'\rangle}\ \textsc{ParR}$$

$$\frac{\alpha' = \alpha \cup \alpha_1}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{Par}(u,\alpha,\mathsf{ALeaf}(v,\alpha_1),T_2)\,;\langle \ell_1 \parallel e_2\rangle \to \mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{Par}(u,\alpha',\mathsf{PLeaf}(v),T_2)\,;\langle \ell_1 \parallel e_2\rangle}\ \textsc{SurrL}$$

$$\frac{\alpha' = \alpha \cup \alpha_2}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{Par}(u,\alpha,T_1,\mathsf{ALeaf}(w,\alpha_2))\,;\langle e_1 \parallel \ell_2\rangle \to \mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{Par}(u,\alpha',T_1,\mathsf{PLeaf}(w))\,;\langle e_1 \parallel \ell_2\rangle}\ \textsc{SurrR}$$

$$\frac{\mathsf{R}' = \mathsf{R}\{u/v\}\{u/w\} \qquad \mathsf{W}' = \mathsf{W}\{u/v\}\{u/w\}}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;\mathsf{Par}(u,\alpha,\mathsf{PLeaf}(v),\mathsf{PLeaf}(w))\,;\langle e_1 \parallel \ell_2\rangle \to \mathsf{R}'\,;\mathsf{W}'\,;\mu\,;\mathsf{ALeaf}(u,\alpha)\,;\langle \ell_1,\ell_2\rangle}\ \textsc{Join}$$

Figure 5.5: Cost Semantics for calculating the race factor by tracking the readers and writers of mutable references.

$$\frac{r = |\nabla^+_\mu(\mathsf{R},\mathsf{W}) \cap (A \cup \alpha)|}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;A \vdash \mathsf{ALeaf}(v,\alpha) \downarrow r}\ \textsc{aLeaf} \qquad \frac{}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;A \vdash \mathsf{PLeaf}(v) \downarrow 0}\ \textsc{pLeaf}$$

$$\frac{\mathsf{R}\,;\mathsf{W}\,;\mu\,;A \cup \alpha \vdash T_1 \downarrow r_1 \qquad \mathsf{R}\,;\mathsf{W}\,;\mu\,;A \cup \alpha \vdash T_2 \downarrow r_2}{\mathsf{R}\,;\mathsf{W}\,;\mu\,;A \vdash \mathsf{Par}(v,\alpha,T_1,T_2) \downarrow \max(r_1,r_2)}\ \textsc{Par}$$

Figure 5.6: Rules for computing the race factor of a given program state $(\mathsf{R};\mathsf{W};\mu;T;e)$. The context tracks the set of allocations A along the root-to-leaf path, and the rules inductively add allocations of each task from the task tree to this set. The reader store R, writer store W, and the memory store μ remain unchanged by the rules.

for both cases, the rule Par extends the context A with locations in α, because the node v is on every root-to-leaf path that contain the nodes in the subtrees T_1 and T_2.

The rule ALeaf corresponds to the case of the active leaf $\mathsf{ALeaf}(v,\alpha)$. It computes the race factor as $|\nabla^+_\mu(\mathsf{R},\mathsf{W}) \cap (A \cup \alpha)|$. To parse this,

1. **Reachability from Exposed Locations.** First, we consider $\nabla^+_\mu(\mathsf{R},\mathsf{W})$, which is the set of all locations reachable by following pointers from the exposed locations $\nabla(\mathsf{R},\mathsf{W})$.
2. **Intersection with the Root-to-Leaf Path**. Next, we take the intersection of $\nabla^+_\mu(\mathsf{R},\mathsf{W})$ with the set $(A \cup \alpha)$. The set $(A \cup \alpha)$ contains all locations along the root-to-leaf path (to leaf v), and these only locations that count towards the race factor along this path.
3. **Sum of Sizes.** We compute the size of all locations in this intersection. which is the sum of sizes of all storables mapped to the locations.

For passive leaves, we define the race factor to be zero, and the rule PLeaf corresponds to this case. This is because our semantics performs a surrender step when an active leaf becomes passive. This step transfers the ownership of all allocations of the leaf to its parent. For example, the rule SurrL takes an active task v that has finished and marks its leaf as passive. After the leaf becomes passive, the rule merge its allocations with the parent; we call this a ***surrender*** because the leaf "surrenders" its allocations to the parent. This way, the parent becomes responsible for allocations of the passive leaf. The rule SurrR is symmetric. This surrender is similar to the surrender step in our coscheduling algorithm (Chapter 2).

We can use this definition of race factor at a state to define race factor for an execution. An execution proceeds as $S_0 \to S_1 \cdots S_n$, where the initial state $S_0 = (\emptyset\,;\emptyset\,;\emptyset\,;\mathsf{ALeaf}(r,\emptyset)\,;e)$ has stores R, W, and μ as empty, the task tree as $\mathsf{ALeaf}(v,\emptyset)$, and the expression e as the program. The race factor for this execution is the maximum race factor observed at any state S_i. Note that taking the maximum is important because the race factor may decrease at joins, because joins "resolve" the races between the joining tasks.

Join. The rule Join executes the join step and also resolves all the races and exposed locations among the joining children (see Figure 5.5). To do this, the rule remaps all the reads and writes performed by the child tasks v and w to the parent task u. We denote this with $\mathsf{R}\{u/v\}\{u/w\}$

and $\mathsf{W}\{u/v\}\{u/w\}$, which can be read as, "substitute u for v and w in the reader and writer set of every location." The net effect of this substitution is that any race conditions between tasks v and w that were tracked in reader and writer stores, have now been removed, thereby also removing the exposed locations between them. However, races involving v and w with other tasks are now represented as races with u ensuring that these race conditions are still tracked and considered towards the race factor.

Other rules. Rules ASL, ASR, and App are standard rules for evaluating functions, their arguments, and the application respectively. Rules FstS and Fst show how the sequential tuple $\langle e_1, e_2 \rangle$ evaluates. Note that none of these rules extend the reader and writer stores because the stores only monitor mutable accesses.

Relationship between exposed and entangled locations. In Section 3.2, we defined that an object is entangled when it is allocated by a task and accessed by a concurrently executing task. There is a subtle difference between entangled locations and exposed locations. First, all entangled locations are exposed. A location becomes entangled when it is allocated by a task and then later exposed to a concurrently executing task through mutable effects. Thus, by accounting for the impact of exposed locations on the cost of memory management, we also account for the impact of entangled locations.

However, an exposed location may not be entangled, because locations are considered exposed regardless of the task that performs the allocation. For example, suppose a task A allocates a mutable location ℓ and two other locations ℓ' and ℓ'', such that ℓ points to ℓ'. Task A forks children tasks B and C. Task B updates the mutable location ℓ to point to location ℓ''. If task C, which is concurrent to B, reads this update, then the location ℓ'' becomes exposed. However, since location ℓ'' was allocated by the parent A of task C, it is not considered entangled.

5.4.2 Sequentialized Space

Sequential executions are often used as a baseline for reasoning about the costs of parallelism. We showed that for a deterministic (race-free) programs establishing a sequential baseline is straightforward: we simply sequentialize execution of parallel tuples. (Section 5.3.1). However, this approach does not work at all for nondeterministic programs, because they can have vastly different executions due to race conditions. Even if we fix the instructions executed in a particular run, it may not be possible or meaningful to create a corresponding sequential run, because some races rely on a very specific interleaving of parallel tasks, which can only be reproduced when tasks are executed concurrently.

To address this challenge, we define the sequentialized space R^*. This metric approximates a sequential space cost even for parallel executions that might not have a true sequential equivalent. We achieve this by simulating the space behavior of a sequential execution along each path in the task tree and computing the maximum memory usage along all the paths. This approach is necessary because racy executions are not guaranteed to be feasible on a single processor.

$$\frac{\ell \notin \mathrm{dom}(\mu) \qquad \mu' = \mu[\ell \hookrightarrow s]}{\mu \,;\, \mathsf{ALeaf}(v, \alpha) \,;\, s \implies \mu' \,;\, \mathsf{ALeaf}(v, \alpha \cup \{\ell\}) \,;\, \ell}\ \textsc{Alloc}$$

$$\frac{\mu \,;\, T \,;\, e_1 \implies \mu' \,;\, T' \,;\, {e_1}'}{\mu \,;\, T \,;\, (e_1\ e_2) \implies \mu' \,;\, T' \,;\, (e_1'\ e_2)}\ \textsc{ASL}$$

$$\frac{\mu(\ell_1) = \texttt{fun}\ f\ x\ \texttt{is}\ e_b}{\mu \,;\, \mathsf{ALeaf}(v, \alpha) \,;\, (\ell_1\ \ell_2) \implies \mu \,;\, \mathsf{ALeaf}(v, \alpha) \,;\, [\ell_1, \ell_2\ /\ f, x]e_b}\ \textsc{App}$$

$$\frac{\mu \,;\, T \,;\, e \implies \mu' \,;\, T' \,;\, e'}{\mu \,;\, T \,;\, (\texttt{ref}\ e) \implies \mu' \,;\, T' \,;\, (\texttt{ref}\ e')}\ \textsc{RefS}$$

$$\frac{\mu \,;\, T \,;\, e \implies \mu' \,;\, T' \,;\, e'}{\mu \,;\, T \,;\, (!\, e) \implies \mu' \,;\, T' \,;\, (!\, e')}\ \textsc{BangS}$$

$$\frac{\mu(\ell_1) = \texttt{ref}\ \ell_2}{\mu \,;\, \mathsf{ALeaf}(v, \alpha) \,;\, (!\, \ell_1) \implies \mu \,;\, \mathsf{ALeaf}(v, \alpha) \,;\, \ell_2}\ \textsc{Bang}$$

$$\frac{}{\mu_0[\ell_1 \hookrightarrow \texttt{ref}\ \ell] \,;\, \mathsf{ALeaf}(v, \alpha) \,;\, (\ell_1 := \ell_2) \implies \mu_0[\ell_1 \hookrightarrow \texttt{ref}\ \ell_2] \,;\, \mathsf{ALeaf}(v, \alpha) \,;\, \ell_2}\ \textsc{Dupd}$$

$$\frac{S_1 = \mu^+(\mathcal{L}(e_1)) \quad S_2 = \mu^+(\mathcal{L}(e_2))}{\mu \,;\, \mathsf{ALeaf}(u, \alpha) \,;\, \langle e_1 \parallel e_2 \rangle \implies \mu \,;\, \mathsf{Par}(u, \alpha, S_1 \otimes S_2, \mathsf{ALeaf}(v, \emptyset), \mathsf{ALeaf}(w, \emptyset)) \,;\, \langle e_1 \parallel e_2 \rangle}\ \textsc{Fork}$$

$$\frac{\mu \,;\, T_1 \,;\, e_1 \implies \mu' \,;\, T_1' \,;\, e_1'}{\mu \,;\, \mathsf{Par}(u, \alpha, S_1 \otimes S_2, T_1, T_2) \,;\, \langle e_1 \parallel e_2 \rangle \implies \mu' \,;\, \mathsf{Par}(u, \alpha, S_1 \otimes S_2, T_1', T_2) \,;\, \langle e_1' \parallel e_2 \rangle}\ \textsc{ParL}$$

$$\frac{\mu \,;\, T_2 \,;\, e_2 \implies \mu' \,;\, T_2' \,;\, e_2'}{\mu \,;\, \mathsf{Par}(u, \alpha, S_1 \otimes S_2, T_1, T_2) \,;\, \langle e_1 \parallel e_2 \rangle \implies \mu' \,;\, \mathsf{Par}(u, \alpha, S_1 \otimes S_2, T_1, T_2') \,;\, \langle e_1 \parallel e_2' \rangle}\ \textsc{ParR}$$

$$\frac{\alpha' = \alpha \cup \alpha_1}{\mu \,;\, \mathsf{Par}(u, \alpha, S_1 \otimes S_2, \mathsf{ALeaf}(v, \alpha_1), T_2) \,;\, \langle \ell_1 \parallel e_2 \rangle \implies \mu \,;\, \mathsf{Par}(u, \alpha', \emptyset \otimes \emptyset, \mathsf{PLeaf}(v), T_2) \,;\, \langle \ell_1 \parallel e_2 \rangle}\ \textsc{SurrL}$$

$$\frac{\alpha' = \alpha \cup \alpha_2}{\mu \,;\, \mathsf{Par}(u, \alpha, S_1 \otimes S_2, T_1, \mathsf{ALeaf}(w, \alpha_2)) \,;\, \langle e_1 \parallel \ell_2 \rangle \implies \mu \,;\, \mathsf{Par}(u, \alpha, \emptyset \otimes \emptyset, T_1, \mathsf{PLeaf}(w)) \,;\, \langle e_1 \parallel \ell_2 \rangle}\ \textsc{SurrR}$$

$$\frac{}{\mu \,;\, \mathsf{Par}(u, \alpha, \emptyset \otimes \emptyset, \mathsf{PLeaf}(v), \mathsf{PLeaf}(w)) \,;\, \langle \ell_1 \parallel \ell_2 \rangle \implies \mu \,;\, \mathsf{ALeaf}(u, \alpha) \,;\, \langle \ell_1, \ell_2 \rangle}\ \textsc{Join}$$

Figure 5.7: Cost semantics for sequentialized space in parallel executions with races

$$\frac{R = (|\mu^+(P \cup \mathcal{L}(e)) \cap (A \cup \alpha)|)}{\mu\,;A;P \vdash \mathsf{ALeaf}(v,\alpha);e \downarrow R_v}\ \textsc{ALeaf} \qquad \frac{R = (|\mu^+(P \cup \{\ell\}) \cap A|)}{\mu\,;A;P \vdash \mathsf{PLeaf}(v);\ell \downarrow R}\ \textsc{PLeaf}$$

$$\frac{\neg(e_2\ \mathsf{loc}) \qquad \mu\,;A \cup \alpha;P \cup \{\ell_1\} \vdash T_2;e_2 \downarrow R}{\mu\,;A;P \vdash \mathsf{Par}(u,\alpha,\emptyset \otimes \emptyset,\mathsf{PLeaf}(v),T_2)\,;\langle \ell_1 \parallel e_2\rangle \downarrow R}\ \textsc{PassL}$$

$$\frac{\neg(e_1\ \mathsf{loc}) \qquad \mu\,;A \cup \alpha;P \cup \{\ell_2\} \vdash T_1;e_1 \downarrow R}{\mu\,;A;P \vdash \mathsf{Par}(u,\alpha,\emptyset \otimes \emptyset,T_1,\mathsf{PLeaf}(v))\,;\langle e_1 \parallel \ell_2\rangle \downarrow R}\ \textsc{PassR}$$

$$\frac{\neg(e_1\ \mathsf{loc}) \qquad \neg(e_2\ \mathsf{loc}) \qquad \mu\,;A \cup \alpha;P \cup S_2 \vdash T_1;e_1 \downarrow R_1 \qquad \mu\,;A \cup \alpha;P \cup S_1 \vdash T_2;e_2 \downarrow R_2}{\mu\,;A;P \vdash \mathsf{Par}(u,\alpha,S_1 \otimes S_2,T_1,T_2)\,;\langle e_1 \parallel e_2\rangle \downarrow \max(R_1,R_2)}\ \textsc{Par}$$

Figure 5.8: Computing Sequentialized Space of a State

We formalize this approach with a cost semantics. The semantics is a transition relation that steps a program state $\hat{S}$ consisting of three components: (i) a memory store μ, (ii) a task tree T, and (iii) an expression e. We write the relation as :

$$\mu\,;T\,;e \implies \mu'\,;T'\,;e'$$

Figure 5.7 shows the rules for the semantics. The semantics tracks information in task trees to determine the *sequentialized space* for each program state $\hat{S}$, which is the maximum size of reachable locations within the allocations along any root-to-leaf path of the task tree. We formalize the definition for each program state in Figure 5.8. The sequentialized space R^* for an execution $\hat{S}_0 \implies \hat{S}_1 \implies \cdots \hat{S}_n$ is the high watermark of the sequentialized space of every state $\hat{S}_i$.

Task Trees and Snapshots. To calculate the sequentialized space, the task tree stores the allocations performed by each task. An active leaf task v is of the form $\mathsf{ALeaf}(v,\alpha)$, where α is the set of allocations performed by task v. The allocations are tracked at rule Alloc, which adds newly allocated locations to the allocation set of the leaf task (Figure 5.7).

In addition to storing allocations, an internal task of the form $\mathsf{Par}(u,\alpha,S_1 \otimes S_2,T_1,T_2)$ also stores snapshots S_1 and S_2. When parallel tasks execute in a nondeterministic program, their memory effects can interfere with each other, making it difficult to measure memory usage as if the tasks were executing sequentially. To negate this effect and simulate a sequential cost, we use snapshots. Snapshots record the memory state at forks where a parent task spawns child tasks. They allow us to isolate the effects of one child's execution on the reachability cost of another child.

A ***snapshot*** of a task is the set of locations that are reachable from it before it starts executing. For a task, evaluating expression e_1, its snapshot is $\mu^+(\mathcal{L}(e_1))$ (here $\mathcal{L}(e_1)$ is the set of locations mentioned in e_1). Snapshots are created at forks. For instance, the rule Fork in the

semantics (Figure 5.7) creates the snapshot S_1 for expression e_1 and S_2 for expression e_2, creating the internal node $\mathsf{Par}(u, \alpha, S_1 \otimes S_2, \mathsf{ALeaf}(v, \emptyset), \mathsf{ALeaf}(w, \emptyset))$. These snapshots are not updated while the children execute. We use them to define the sequentialized space as follows.

Sequentialized Space. Given a program state $(\mu; T; e)$, we define the ***sequentialized space*** for that state as the maximum size of reachable locations within the allocations along any root-to-leaf path of the task tree. Figure 5.8 shows the judgement $\mu; A; P; \vdash T; e \downarrow R$ that calculates the sequentialized space R. The context A tracks the allocations along the root-to-leaf path and the variable P tracks snapshots along the path. The rules of the judgement compute the sequentialized space along every root to leaf path and then take the maximum across all paths.

For instance, the rule Par considers an internal task whose children are still executing, denoted by $\neg(e_1\ \mathsf{loc})$ and $\neg(e_2\ \mathsf{loc})$ for expression e_1 and e_2; this asserts that both expressions are not yet locations, meaning that they haven't finished evaluating. The rule considers a node of the form $\mathsf{Par}(u, \alpha, S_1 \otimes S_2, T_1, T_2)$, and computes the sequentialized space for both children. The rule extends the context A with the allocations α of the task u. When computing the sequentialized space along the left child T_1, it adds the snapshot S_2 to the root set P; the snapshot S_2 contains all the locations referenced by the sibling at the time of fork. This ensures that reachability calculation along the paths in T_1 remains unaffected by actions of tasks in T_2. The rule computes the reachability along the paths in T_2 similarly. After computing the sequential space R_1 and R_2 along subtrees T_1 and T_2 respectively, it takes the maximum among them.

The rule ALeaf considers the case for an active leaf v. It computes the sequentialized space as $(|\mu^+(P \cup \mathcal{L}(e)) \cap (A \cup \alpha)|)$, which uses the set $(P \cup \mathcal{L}(e))$ containing snapshots and locations of expression e as "program roots". It calculates all locations reachable from those roots using μ^+, and then takes the intersection with the set $A \cup \alpha$ containing the locations along the root to leaf path to the task v. The rule PLeaf is similar except passive leaves don't have an allocated set of locations α. This is because when a task finishes, all its locations are transferred to the parent by the rules for surrendering.

Surrendering and Sequentialization. As soon as a sibling task finishes evaluation, it ***surrenders*** its allocations to the parent and the semantics deletes the snapshots of both the children. This design simulates a sequential execution, where one child has finished before the other. For example, consider rule SurrL of the semantics (Figure 5.7) in which the left child finished first. We surrender the allocations of the left child, making its leaf passive, and delete both snapshots. By deleting snapshots, we commit to a sequential execution where the left child finished before the right child. The rule SurrR is symmetric. Once both the children have finished, the corresponding passive leaves are removed (see rule Join) and their parent resumes.

The rules PassL and PassR calculate the sequentialized space for cases where one of the children is passive (see Figure 5.8). When one child is passive, no snapshots are used. Instead, the location computed by the passive child is added to the root set P. Note that the allocations of the passive child also count towards the root-to-leaf path, but they have already been

transferred to the parent.

Relation to the Race Factor Semantics. Our semantics for sequential space and the semantics for race factor (Section 5.4.1) are morally similar for expression evaluation. Their task tree stores different data at each node, for computing the cost metrics, but otherwise the rules are similar. In fact, we could merge them, but we chose to keep them separate for presentation.

5.4.3 Work and Space Bounds

We start with a quick recap of our collection policy: our memory manager uses a heap scheduler to divide the memory among the processors and each processor manages it own memory. Each processor p maintains a counter λ_p and starts a collection when the size of its heap cluster exceeds $\kappa * \lambda_p$. After the collection is done, we reset the counter λ_p to the amount of memory that survives collection.

Given an execution, we observe its steps and use our cost semantics to define the race factor and the sequential space. We show that with our memory management techniques the execution satisfies the following space bound, due to the memory management.

Theorem 5 (Space bound)**.** *Given an execution on P processors with sequential space R^* and race factor r, our memory manager requires no more than $O((R^* + r) \cdot P)$ space for that execution.*

To establish the space bound, we show that the counter λ_p never exceeds $(R^* + r)$ (in Lemma 6). Assuming this, we have that for any processor p its heap set M_p is never larger than $\kappa \cdot \lambda_p$ (otherwise the processor collects and resets the counter). Because there are P processors, the total memory is

$$\sum_p |M_p| \leq \sum_p \kappa \cdot \lambda_p \leq \kappa \cdot (R^* + r) \cdot P$$

To bound the counter, we show that the space after garbage collection from the roots and remembered sets at any processor is bounded by $R^* + r$. Intuitively speaking, our memory manager tracks pointers between heap sets and each such pointer targets either locations of the snapshots or the locations exposed by mutable references. This way, each pointer is accounted by either the sequential space R^* or the race factor r respectively. Both these cost metrics only consider root-to-leaf paths (and passive leaves) in the semantics and this corresponds to how our heap sets are designed. Because our memory manager punctually discards stale entries at surrenders, it is able to maintain its precision w.r.t. these cost metrics. We prove this in the next section (Lemma 6).

We also show a bound on the work of our memory manager, w.r.t. a collection algorithm that collects a heap set M in $c \cdot |M|$ work.

Theorem 6 (Work bound)**.** *Given an execution on P processors with program work W, our memory manager does not do more than $\kappa' \cdot (W + (R^* + r) \cdot P)$ work, where $\kappa' = c \cdot \frac{\kappa}{\kappa - 1}$, c is the work efficiency of our collection algorithm, and κ is a tunable parameter of the collection policy.*

Proof. At each collection, the collector does no more than $c \cdot |M|$ amount of work, which as shown above is bounded by $c \cdot \kappa \cdot \lambda$. After the collection finishes, the counter λ is reset to the

amount of memory that survives the collection. Roughly speaking, before the next collection, the program must allocate at least $(\kappa - 1) \cdot \lambda$ amount of memory, so these allocations essentially pay for the collection. The first collection at each processor is extra work and that shows up in the bound above as the term $\kappa' \cdot (R^* + r)$ and is paid by each processor.

Let $|M_p^i|$ be the size of heap cluster before the i*th* collection on worker p. If λ_p^{i+1} is the size of heap cluster after the collection, then the memory reclaimed is $(|M_p^i| - \lambda_p^{i+1})$. The memory reclaimed by all the collections can not be greater than the memory allocated by the program. Thus, if worker p performs n_p collections and α is the total memory allocated then:

$$\sum_p \sum_{i=1}^{i=n_p} (|M_p^i| - \lambda_p^{i+1}) \leq \alpha \tag{5.2}$$

In our collection policy, a worker starts a collection only when the size of its heap cluster grows beyond κ times its counter, i.e, the worker p starts the i*th* collection because $|M_p^i| \geq \kappa * \lambda_p^i$. Moreover, by Lemma 6, the value of the counter λ_p does not exceed R^*. Thus, it follows that:

$$\sum_p \sum_{i=1}^{i=n_p} \lambda_p^{i+1} \leq \sum_p \sum_{i=1}^{i=n_p} \lambda_p^i + \sum_p \lambda_p^{(n_p+1)} \leq \sum_p \sum_{i=1}^{i=n_p} \frac{|M_p^i|}{\kappa} + P \cdot R^*$$

After substituting this in Equation 5.2, it follows that:

$$\sum_p \sum_{i=1}^{i=n_p} |M_p^i| \leq (\alpha + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1} \leq (W + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1}$$

We assume that allocation of one unit of memory requires one unit of work and thus, $\alpha \leq W$. The total memory traced in all collections is $(\sum_p \sum_{i=1}^{i=n_p} |M_p^i|)$. Thus, the total work done in collections is upper-bounded by $c_1 \cdot (\sum_p \sum_{i=1}^{i=n_p} |M_p^i|) \leq c_1 \cdot (W + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1}$. □

5.4.4 Bounding the counter

In the following lemma, we show that the processor-local counter λ_p for processor p, which tracks the amount of memory surviving the last garbage collection, is always bounded by the sum of the sequentialized space R^* and the race factor r. This bound is crucial because it ensures that the memory used by each processor remains bounded, allowing us prove a space bound on the entire execution (Section 5.4.3).

Lemma 6. *The processor-local counter λ_p for processor p is bounded by $R^* + r$.*

Proof. A processor updates its counter after every collection and the value is set to the amount of memory that survives, i.e., the amount of memory reachable from the program roots, the remembered set, and the snapshot stored the processor. We show that the amount of memory surviving is always less than $R^* + r$ for any processor p.

Proof Strategy. To prove this, we must show that the memory reachable from the remembered set and snapshot tracking algorithm used our memory manager (Section 5.2) is bounded by $R^* + r$. We achieve this by establishing a connection between this algorithm and a theoretical "frame remset" that implements the cost semantics. The frame remset, although not practically implementable, serves as reference point for analyzing the efficiency of our practical algorithm. By demonstrating that the practical algorithm is as space-efficient as the frame remset, we can leverage the bounds established by the cost semantics to bound the processor-local counter.

Heap Cluster, Semantics, and Frame Remset. The cost semantics for the race factor and sequentialized space include the step surrender, which happens when a leaf becomes passive. In this operation, the passive leaf transfers all of its allocations (tracked in variable α) to the active sibling. Thus, a root to leaf path in the semantics also contains the allocations from any passive siblings. This structure is identical to the way the heap scheduling algorithm maintains the heap clusters. For any heap cluster M_p and active task v, the memory in the cluster M_p is a subset of the memory allocations in the root-to-leaf path of the semantics. Since the reachability along the root-to-leaf path to v using the frame set is bounded by $R^* + r$, the reachability in heap M_p from same frame remset is also bounded by $R^* + r$.

Since each heap cluster created by the heap scheduler contains a subset of the allocations along some root-to-leaf path in the semantics, we can specialize the semantics for heap clusters. We define a frame remset that implements the cost semantics for both sequentialized space and race factor precisely, for each heap cluster. The frame remset maintains the reader and writer stores as defined in the cost semantics for the race factor. in Section 5.4.1. Furthermore, the frame remset creates and stores snapshots, as defined the semantics for sequentialized space Section 5.4.2, for each heap cluster M_p when a task is stolen from the processor p. Then at a surrender, the frame remset deletes the snapshot that was stored at the corresponding steal, staying consistent with the semantics for sequentialized space. Since the definitions of sequentialized space R^* and race factor r bound reachable memory from the frame remset along all root-to-leaf paths, the memory reachable from the frame remset within a heap cluster is bounded by $R^* + r$.

Adequacy Property of the Frame Remset. We assume an adequacy property of the frame remset that ensures that garbage collecting any heap cluster using the frame remset is "safe", i.e., any location that may be accessed by tasks on other processors is not reclaimed. Specifically, if any task v on processor p accesses location ℓ that is in another processor's heap cluster then: (i) either location ℓ is an exposed location, i.e., $\ell \in \nabla(\mathsf{R}, \mathsf{W})$ and it stays exposed until task v joins with the task whose heap is $H(\ell)$ or (ii) there location ℓ is allocated by an ancestor v' of task v and is present in the snapshot stored at the processor which contains the location. We formalize this by defining two relations for a given step i of the execution. $\mathsf{ExposedUntil}(i, \ell, u, v)$:

$$\mathsf{ExposedUntil}(i, \ell, u, v) = \exists \ell', u', v' : (u', \ell') \in \mathsf{R}_i(\ell) \wedge (v', \ell') \in \mathsf{W}_i(\ell) \wedge \mathsf{LCA}(u', v') \leq_T \mathsf{LCA}(u, v)$$

Here, $t \leq_T t'$ denotes that task t is an ancestor of task t'. The relation ExposedUntil formally states that (i) location ℓ is exposed at step i and (ii) for tasks u' and v' whose actions expose the

location, their least common ancestor is an ancestor of the least common ancestor of tasks u and v. The second condition ($\mathsf{LCA}(u', v') \leq_T \mathsf{LCA}(u, v)$). ensures that location ℓ stays exposed until u' and v' join, which only happens after u and v join because of the positions of their least common ancestors.

The relation $\mathsf{InSnapshot}(i, \ell, u)$ is defined follows:

$$\mathsf{InSnapshot}(i, \ell, u) = \overline{H(\ell)} \leq_T u \wedge \exists u' : u' \leq_T u \wedge \ell \in \mathsf{S}_i(u')$$

, where $\mathsf{S}_i(u')$ denotes the snapshot for task u' (according to the cost semantics), and $\overline{H(\ell)}$ is the task (represented by the overline) of the heap of location ℓ.

Formally, for a task v with expression e, that is executing on processor p at step i, we assume (as adequacy property) the following:

$$\forall \ell \in \mathcal{L}(e) : \ell \notin M_p \Rightarrow \mathsf{InSnapshot}(i, \ell, v) \vee \mathsf{ExposedUntil}(i, \ell, v, \overline{H(\ell)})$$

Practical Remset and Frame Remset. For each processor, our memory manager maintains a snapshot $\mathsf{S}(p)$ and a remembered set $\mathsf{R}(p)$ (Section 5.2).

We show that every entry in these sets corresponds to an entry in the frame remset. For the proofs here, we consider the remembered set and the snapshot together as one, i.e., our ***practical remset*** $\mathsf{SR}(p)$ contains locations in both the snapshot $\mathsf{S}(p)$ and a remembered set $\mathsf{R}(p)$.

We consider an execution $S_0 \rightarrow^* S_n$ and prove the following invariants before ith step with state S_i on processor p:

1. for every entry $\{$*root:* ℓ, *from-heap:* $h_u\}$ that is in $\mathsf{SR}(p)$, either $\mathsf{InSnapshot}(i, \ell, u)$ or $\mathsf{ExposedUntil}(i, \ell, u, \overline{H(\ell)})$
2. for every entry $\{$*root:* ℓ, *from:* ℓ', *from-heap:* $h_u\}$ in $\mathsf{SR}(p)$, if the entry was created by a task v, either location ℓ is the target of a down pointer or $\mathsf{ExposedUntil}(i, \ell, \overline{H(\ell)}, \overline{H(\ell')})$, or $\mathsf{ExposedUntil}(i, \ell', \overline{H(\ell')}, v)$ and ℓ and h_v are in the same heap cluster, or $\mathsf{InSnapshot}(i, \ell, \overline{H(\ell')})$.

Intuitively, the above properties imply that every location in our practical remset is either exposed or in some snapshot. Every entry corresponds to an entry in the frame remset. Since, the memory reachable in any heap cluster due to the frame remset is upper bounded by $R^* + r$, the memory from the practical remset is also upper bounded by $R^* + r$.

We establish the invariants by induction. The base case with $i = 0$ holds trivially because the remset is empty. From state S_i suppose we take the ith step on processor p and reach S_{i+1}. We consider all the possible steps at that processor:

Case Steal When processor p steals a task u with expression e from processor q, its adds all the locations mentioned in e (that are in the heap cluster M_q) to the remset of q. For each such location, it creates the entry $\{$*root:* ℓ, *from-heap:* $h_u\}$. Because u executes the expression e at step $i + 1$, we get from the adequacy condition that $\forall \ell \in \mathcal{L}(e) : \ell \notin M_p \Rightarrow \mathsf{InSnapshot}(i + 1, \ell, v) \vee \mathsf{ExposedUntil}(i + 1, \ell, v, \overline{H(\ell)})$. So, every such entry satisfies hypothesis (1) after the step.

Case SURRENDER In this step, suppose task v at processor p finishes and the processor p surrenders all its heaps to another processor q, which has the sibling's heap. In this case, processor p adds all the entries in the remset of p to remset of q. Then we clear stale entries: an entry is stale if either the from-heap is in the same heap cluster as the root, or it has a from location and it does not point to the root. Let's consider the remaining ***valid*** (not stale) entries in the remset of q one-by-one and prove the hypotheses for them.

First consider an entry $\{$*root:* ℓ, *from-heap:* $h_u\}$ such that $\mathsf{InSnapshot}(i, \ell, u)$, i.e., location $\ell \in \mathsf{S}_i(u')$ for some $u' \leq_T u$ (from the inductive hypothesis). Because the entry is valid, it means that $h_u \notin M_q$ and thus $u \neq v'$ and $u \neq v$ from $h_u \notin M_q$ and $h_v \in M_q$ and $h_{v'} \in M_q$. Thus, task u is not involved in this surrender and u' is not the parent of v and v'. Because our frame remset algorithm only deletes the snapshots stored at the parent of v and v', $\ell \in \mathsf{S}_{i+1}(u')$. Thus, $\mathsf{InSnapshot}(i+1, \ell, u)$. Second, let's consider an entry either of the form $\{$*root:* ℓ, *from-heap:* $h_u\}$ such that $\mathsf{ExposedUntil}(i, \ell, u, \overline{H(\ell)})$. Because no exposed locations are cleared and the reader and writer sets are unchanged at surrender, we have $\mathsf{ExposedUntil}(i+1, \ell, u, \overline{H(\ell)})$.

Now, let's consider the valid entries of the form $\{$*root:* ℓ, *from:* ℓ', *from-heap:* $H(\ell')\}$. By validity, we get that $H(\ell') \notin M_p$ and location ℓ' points to ℓ. First, if location ℓ is the target of a down pointer, it stays the target of a down pointer because the heap $H(\ell')$ is not involved in the surrender ($H(\ell') \notin M_p$) and its position w.r.t. location ℓ remains unaffected. Second, if $\mathsf{InSnapshot}(i, \ell, \overline{H(\ell')})$, then the reasoning is similar to the case above for entry $\{$*root:* ℓ, *from-heap:* $h_u\}$, with $\overline{H(\ell')} = u$. Third, in cases where $\mathsf{ExposedUntil}(i, \ell', \overline{H(\ell')}, v)$ and ℓ and h_v are in the same heap cluster, or $\mathsf{ExposedUntil}(i, \ell, \overline{H(\ell)}, \overline{H(\ell')})$, the hypotheses continue to hold because no exposed locations are cleared and the reader and writer sets are unchanged at surrender

Case JOIN Suppose task v joins with task v'. At the join, the heaps h_v and $h_{v'}$ are merged together with their parent and the reader and writer sets are relabelled, potentially turning some exposed locations into "non-exposed". Note that the heaps h_v and $h_{v'}$ of the joining tasks are on the same processor prior to this step. This is because one of the tasks v and v' must have been passive, and passive heaps are on the same processor as their sibling.

For our hypotheses, those cases which hold because of $\mathsf{InSnapshot}$ trivially hold because no snapshots are deleted in the frame remset at joins. We consider the cases with relation $\mathsf{ExposedUntil}$. First consider an entry $\{$*root:* ℓ, *from-heap:* $h_u\}$ such that $\mathsf{ExposedUntil}(i, \ell, u, \overline{H(\ell)})$. Because the entry is still valid (not stale), we get that $h_u \notin M_p$. Thus, it follows that u did not join in this step. Thus, $\mathsf{ExposedUntil}(i, \ell, u, \overline{H(\ell)})$ continues to be satisfied because task u has not joined in this step.

Similarly, consider an entry $\{$*root:* ℓ, *from:* ℓ', *from-heap:* $h_u\}$ in $\mathsf{SR}(p)$, such that $\mathsf{ExposedUntil}(i, \ell, \overline{H(\ell)}, \overline{H(\ell')})$. Because the entry is still valid (not stale), we get that $H(\ell') \notin M_p$. Thus, we get that $\mathsf{ExposedUntil}(i, \ell, \overline{H(\ell)}, \overline{H(\ell')})$ continues to be satisfied because task $\overline{H(\ell')}$ has not joined in this step ($H(\ell') \notin M_p$). Lastly, suppose $\mathsf{ExposedUntil}(i, \ell', \overline{H(\ell')}, w)$ and ℓ and h_w are in the same heap cluster. Because the entry is valid, $H(\ell') \notin M_p$. Thus, in this step, task w and $\overline{H(\ell')}$ could not have joined, because the heaps of both joining tasks are

in M_p. Thus, the condition $\mathsf{ExposedUntil}(i, \ell', \overline{H(\ell')}, w)$ continues to hold.

Case BANG In this step, suppose processor p dereferences location ℓ to read location ℓ'. If location ℓ' is in p's heap cluster M_p, then the remset is unchanged and all hypotheses continue to hold.

Suppose, ℓ' is not in p's heap cluster and instead is at some other processor q, i.e., $\ell' \in M_q$. Then processor p will add the entry $\{$*root:* ℓ', *from-heap:* $h_v\}$ to q's remset. We know from the adequacy conditions that either $\mathsf{InSnapshot}(i, \ell', u)$ or $\mathsf{ExposedUntil}(i+1, \ell', u, \overline{H(\ell')})$. Because no snapshots are deleted in this step, we get either $\mathsf{InSnapshot}(i+1, \ell', u)$ or $\mathsf{ExposedUntil}(i+1, \ell', u, \overline{H(\ell')})$. From this we get hypothesis (1).

Case DUPD In this step, a processor, say p executes a mutable update of task v, installing a pointer from location ℓ to ℓ'. If locations ℓ and ℓ' are in the same heap cluster, the remset does not change and all the hypotheses continue to hold. Otherwise, locations ℓ and ℓ' are in different heap clusters, say $\ell' \in M_q$ and $\ell \in M_r$ for processors r and q such that $q \neq r$. In this case, we add the entry $\{$*root:* ℓ', *from:* ℓ, *from-heap:* $H(\ell)\}$ to the remset of q. We show the hypothesis (2) for this entry (hypothesis (1) is inapplicable).

First suppose $q \neq p$ and $r \neq p$. By applying the adequacy conditions with locations ℓ and ℓ', we get that either $\mathsf{InSnapshot}(i, \ell', v)$ or $\mathsf{ExposedUntil}(i+1, \ell', v, \overline{H(\ell')})$ and either $\mathsf{InSnapshot}(i, \ell, v)$ or $\mathsf{ExposedUntil}(i+1, \ell, v, \overline{H(\ell)})$. If $\mathsf{ExposedUntil}(i+1, \ell', v, \overline{H(\ell')})$ and $\mathsf{ExposedUntil}(i+1, \ell, v, \overline{H(\ell)})$, then $\mathsf{ExposedUntil}(i+1, \ell', \overline{H(\ell')}, \overline{H(\ell)})$. If $\mathsf{ExposedUntil}(i+1, \ell', v, \overline{H(\ell')})$ and $\mathsf{InSnapshot}(i, \ell, v)$, then either the pointer from location ℓ to ℓ' is a down pointer or $\mathsf{ExposedUntil}(i+1, \ell', \overline{H(\ell')}, \overline{H(\ell)})$. If $\mathsf{InSnapshot}(i, \ell', v)$ and $\mathsf{ExposedUntil}(i+1, \ell, v, \overline{H(\ell)})$, the either $\mathsf{InSnapshot}(i, \ell', \overline{H(\ell)})$ or $\mathsf{ExposedUntil}(i+1, \ell, \overline{H(\ell')}, \overline{H(\ell)})$. In the case where $\mathsf{InSnapshot}(i, \ell', v)$ and $\mathsf{InSnapshot}(i, \ell, v)$, then either the pointer from location ℓ to ℓ' is a down pointer or $\mathsf{InSnapshot}(i, \ell', \overline{H(\ell)})$.

Second, suppose $q = p$ and $r \neq p$. By applying the adequacy conditions with location ℓ, we get that either $\mathsf{InSnapshot}(i, \ell, v)$ or $\mathsf{ExposedUntil}(i+1, \ell, v, \overline{H(\ell)})$. Consider $\mathsf{InSnapshot}(i, \ell, v)$ and the heap h_v of task v. We have that the heap $\overline{H(\ell)}$ is an ancestor of h_v, from $\mathsf{InSnapshot}(i, \ell, v)$. If ℓ' is in heap h_v, then the pointer ℓ to ℓ' is a down pointer and we get the hypothesis. Otherwise, if ℓ' is in some other $h_w \in M_p$, then we can use Lemma 4 with $\overline{H(\ell)}$, h_v and h_w to get that $\overline{H(\ell)}$ is an ancestor of h_w . Thus, the pointer ℓ to ℓ' is a down pointer and we get the hypothesis. Otherwise, if $\mathsf{ExposedUntil}(i+1, \ell, v, \overline{H(\ell)})$, then we have $\mathsf{ExposedUntil}(i+1, \ell, v, \overline{H(\ell)})$ and that v and ℓ' are in the same heap cluster.

Third, suppose $q \neq p$ and $r = p$. By applying the adequacy conditions with location ℓ', we get that either $\mathsf{InSnapshot}(i, \ell', v)$ or $\mathsf{ExposedUntil}(i+1, \ell', v, \overline{H(\ell')})$. If $\mathsf{InSnapshot}(i, \ell', v)$ then $H(\ell')$ is an ancestor of both v and $H(\ell)$ and thus, we have $\mathsf{InSnapshot}(i, \ell', \overline{H(\ell)})$. Otherwise, if $\mathsf{ExposedUntil}(i+1, \ell', v, \overline{H(\ell')})$, then either the pointer from location ℓ to ℓ' is a down pointer or $\mathsf{ExposedUntil}(i+1, \ell', \overline{H(\ell)}, \overline{H(\ell')})$

□

6 Implementation

To evaluate the efficiency of our parallel memory management techniques, we developed a language called MPL (MaPLe), which is a high-performance compiler for Parallel ML (SML). MPL is built as an extension of the MLton compiler for (sequential) Standard ML, and supports the full SML language, including including unrestricted mutable state. MPL extends SML with parallel tuples for fork-join parallelism. MPL is open-source[1] and has been used in several publications [17–19, 89, 138, 174, 175, 178].

The implementation of MPL is crucial for evaluating the real-world performance and scalability of our techniques. Our implementation follows the coscheduling and memory management techniques described in Chapter 2 and Chapter 4 respectively. In this chapter, we cover some remaining aspects of the MPL implementation, including important efficiency optimizations.

Acknowledgements

MPL has been under-development for over a decade by multiple collaborators, including Sam Westrick, Ram Raghunathan, Stefan Muller, Adrien Guatto, Rohan Yadav, Larry Wang, Guy Blelloch, Umut Acar, Matthew Fluet, and myself.

6.1 Coscheduling and Heaps

We implemented MPL as an extension of the MLton compiler, which is one of the fastest compilers for sequential Standard ML. The compilation steps of MLton and MPL are

[1] https://github.com/MPLLang/mpl.git

roughly the same, except MPL supports fork-join parallelism by a language primitive called `par : (unit → α) * (unit → α) → (α * β)`, which takes two functions to be evaluated in parallel, and returns a tuple of their result. In the implementation, the `par` primitive creates new tasks for computing the input functions.

For load-balancing tasks across processors, MPL uses a work-stealing task scheduler with concurrent deques [21]. Tasks are implemented as one-shot continuations with heap allocated call stacks. We adapt the work-stealing scheduler to implement our coscheduling algorithm, as described in Section 2.3.2. At steals, when a new task begins execution, the coscheduling algorithm creates a heap for the task and all allocations of the task are performed in that heap. When a task finishes, we implement a lazy form of surrendering which occurs only after both siblings are ready to join. When this happens, we merge the heaps of the children with the heap of the parent. We discuss the implementation of heaps as follows.

Heaps. The coscheduling algorithm and our memory management techniques use the following heap operations:

```
type heap
type addr
create : unit → heap
allocate : heap * size → addr
heapOf : addr → heap
depth : heap → int
merge : heap * heap → heap
dca : heap * heap → heap
concurrent : heap * heap → bool
```

Representation. We implement the heap datatype, primarily in C, because it allows tight control over memory allocation and manipulation. A heap is a linked list of memory chunks, where each chunk is a contiguous array of bytes. A new heap is initialized an empty linked list.

Allocation. Within a heap, memory is allocated by **bump-allocation**. Each chunk of the heap maintains pointers *frontier* and *limit*. The compiled program includes code for checking the frontier and limit pointers. For an allocation of size `s`, the code checks if `s + frontier <= limit`. If the check succeeds, the allocation is performed directly by bumping the frontier, and the program continues. Otherwise, the compiled code makes a call into the runtime system, and the heap allocates a new memory chunk large enough to perform the allocation. Allocations at the level of memory chunks are handled by a Hoard-style block allocator [34].

Each allocated object comprises of a *header* and a *payload*. The header stores meta information about the object and is used exclusively by the runtime system. The payload corresponds to the value of the object, and is used by the program instructions.

Object Query. Given an allocated object, we can determine its heap using the `heapOf` operation, which takes the address of the object and returns the corresponding heap. To support this operation, we include a ***chunk decsriptor*** at the beginning of each memory chunk. The descriptor stores information about its corresponding heap. Given an object at a memory address, we can determine its chunk in constant time by ensuring that all memory chunks are appropriately aligned at power-of-two addresses. To find the chunk of an object from its address, we only need to zero the appropriate lower order bits. Once the chunk found, we can refer to the chunk descriptor to determine the heap. The `heapOf` operation is used by the memory manager for tracking inter-heap pointers and also for restricted the scope of garbage collection. For example, when a processor garbage collects a cluster of heaps, it does not trace objects outside those heaps; it determines that an object is outside by using the `heapOf` operation.

Merge. Representing heaps as a linked list facilitates constant time `merge` operations of heaps. To merge two heaps, we only need to append the corresponding linked lists and update the chunk descriptors. Since updating the descriptor of each chunk could result in overhead, we use a thread-safe union-find data structure that merges heap names at heap merges. This eliminates the need to update each chunk eagerly, and instead, the chunk descriptors are updated as part of path-compression of the union-find data structure.

Entanglement Checking and Management The operations `concurrent` and `dca` are used for tracking and managing entangled objects. The `concurrent` takes two heaps and returns a boolean. It returns `true` when the heaps are concurrent in the heap tree, i.e., when they are not in an ancestor-descendant relationship. The operation is used for detecting entangled objects: when a task v reads an object ℓ, the location ℓ is an entanglement source if `concurrent` $(h_v, \mathsf{heapOf}(\ell))$, where h_v denotes the heap of task v (implemented by storing the heap in each task). This is used within our read barrier (Section 4.3).

The operation `concurrent` can be implemented using the operation `dca`, which takes two heaps and computes their deepest common ancestor in the heap tree. For example, the deepest common ancestor of two siblings is their parent. When two heaps are not concurrent, one of them is their `dca`, because they are in an ancestor-descendant relationship. The `dca` operation is used by the entanglement tracking algorithm for computing the expiration depth of entanglement sources (Section 4.3). To implement this, we use the `DePa` algorithm, a series-parallel order maintenance data structure [177].

6.2 Tracking of Inter Heap Pointers

We use a combination of barriers and remembered sets to track all targets of inter heap pointers. At a high level, we use three forms of remembered sets: (i) *forgotten set* for pointer deletions, (ii) *down set* for tracking down pointers, and (iii) *source set* for tracking entanglement sources. Our goal is to maintain these sets, with as little overheads as possible in the form of read/write

barriers. Specifically, we want zero overhead for reads of immutable objects, near-zero overhead for reads of disentangled, mutable objects, with some overhead acceptable for mutable writes.

Up Pointers. To account for up pointers, we snapshot every internal heap at the time of fork and collect internal heaps w.r.t. the snapshots. The snapshotting is performed by storing the continuations of both the children task and the parent task at the time of fork, and also using a Yuasa-style barrier for pointer deletions [182], that negates the effects of a mutator deleting pointers by adding the target objects to a deleted set of pointers. At the write barrier, we check if the task is deleting a pointer within objects in an internal heap; if so, we add the target object to the ***forgotten set*** of the corresponding heap. Since many tasks may delete pointers simultaneously, we implement the forgotten set as a list that supports concurrent additions. Overall, these techniques are similar to "snapshot-at-the-beginning" used by concurrent collectors, except our snapshots are taken at the time of fork [101].

Down Pointers. The write barrier detects down pointers and adds them to the ***down set***. Before a pointer is created from object x to y, the write barrier considers the heaps $H(x)$ of object x and $H(y)$ of object y. If the pointer is a down pointer, i.e., $H(x) \neq H(y)$ and $H(x) = \texttt{dca}\,(H(x), H(y))$, the write barrier adds the pointer to the down set of heap $H(y)$.

Cross Pointers. As we discussed in Section 4.2, we do not track cross pointers, because tracking them would be infeasible, and instead track all entangled objects. To achieve this, we use a read barrier, but carefully design it to minimize its cost. We discuss our optimizations for the read barrier in the next section.

6.3 Optimizing the Read Barrier

One of the key challenges for efficiency our approach is the use of read barriers, which can significantly impact performance if not carefully optimized. To address this, we implemented three key optimizations to minimize the overhead of read barriers: 1) No read barriers on any immutable objects, 2) Near zero overhead for reads on disentangled, mutable objects, and 3) One time cost to entangled objects. The most important among them is Optimization 1, which ensures barrier-less accesses to immutable objects—a crucial feature because parallel functional programs frequently access such objects. Chapter 4 already covered this important optimization (Section 4.3) and also covered Optimization 3 (Section 4.4). We describe Optimization 2 in this section.

Entanglement Frontier. The read barrier intercepts the reads of mutable objects and performs an entanglement check: when a task v reads a mutable location ℓ, the check performs a graph query that tests whether the read is entangled, i.e., if $\mathsf{concurrent}(v, \mathsf{allocator}(\ell))$ (see Section 4.3). This check has non-zero overhead, which can accumulate over reads of the mutable objects. We optimize this overhead by implementing a **fast path** and only performing a check

on the **slow path**. In the slow path, the read barrier proceeds as usual: perform an entanglement check to determine if the read creates an entanglement source, and if so, the read barrier adds the entangled object to the ***source set*** of the corresponding heap. Since multiple tasks may attempt to add entanglement sources to the source set, we implement the source set as a list that supports concurrent insertions. Even though such concurrent operations impose overhead, this overhead is restricted to entangled objects and is imposed only once, per entangled object.

In the fast path, the read barrier just checks a lower order bit in the header of the object being read; if the bit is not set, then no checks are needed. To achieve this, we design an algorithm to track an ***entanglement frontier***, which contains the set of all mutable objects whose reads may cause entanglement. Objects in the frontier are checked when read and reads of objects outside the frontier take the fast path. Testing whether an object in the frontier is cheap: we maintain a bit in the object header and set it for frontier objects. Since the object is already being read, testing this bit has near zero overhead.

Our barriers add two types of mutable objects to the frontier because they may create entanglement. First, when a mutable object has a down or cross pointer to another object outside its heap, its read may enable a task to create entanglement. The write barrier detects when a task creates an inter-heap pointer from an object (to some other object) and adds the object to the entanglement frontier. Second, when an object is at the boundary of an entanglement region, it is entangled, and other (concurrent) tasks may read it, creating more entanglement. The read barrier adds such objects to the frontier when it pins an entanglement region.

Frontier objects become non-frontier after joins. Because joins merge heaps, they turn inter-heap pointers into internal pointers and void the first type of frontier objects. Furthermore, because joins expire entangled objects, they void the second type of frontier objects. To identify these changes to the frontier, the barriers associate a“join point” with every frontier object. When a frontier object reaches its join point, it is removed from the frontier. We reserve a bit in the header of mutable objects and toggle it to add/remove objects from the frontier. Because frontier objects are create in parallel, it is important to clear frontier objects in parallel. We parallelize the clearing of frontier objects, by creating fork-join tasks and adding them to the scheduler.

This optimization is delivers significant benefits, upto 20% in running time on 72 cores. This is because a vast majority of reads of mutable objects can not create entangled objects. The optimization removes the overhead of checking entanglement from all these reads, resulting in near-zero overhead for most disentangled objects.

6.4 Garbage Collection Algorithms

We covered most of the details about the garbage collection algorithms in Section 4.5. Here, we cover some additional optimizations important for efficiency.

Internal Heaps. Internal heaps are snapshotted at forks and we collect them using a concurrent mark and sweep algorithm. To garbage collect an internal heap, we fork a "GC task" that is added on the work-stealing deque and is then scheduled by the task scheduler on a processor. The collection can proceed concurrently to the rest of the program, because the heap is snapshotted and the mark and sweep algorithm does not move objects, allowing for concurrent accesses to the heap's objects.

Note that it is possible for the internal heap to become a leaf heap while it is being garbage collected. This is because the program tasks can join, and the internal heap may be needed for new allocations. To support this, we create a new heap for allocations, which is used by the task for the time being, while the collection is being performed. Once the concurrent garbage collection is finished, we merge this new heap with the collected heap.

Hybrid Garbage Collection of Leaf Heaps. For leaf heaps, we pause the corresponding leaf task and use a hybrid collection algorithm. The hybrid collection algorithm uses a mark and sweep style algorithm for entangled objects, and performs semi space copying collection for the disentangled objects. The defragmentation aspect of the hybrid collector it crucial for both time and space efficiency, improving upto 20% in time and 75% in space.

For efficiency, the collection proceeds in two phases. First, the collector runs a hybrid phase taking the source set and the entanglement frontier as roots. This phase traces all objects that are entangled and also those that are disentangled, but are susceptible to entanglement, i.e., those reachable from the entanglement frontier. In this phase, the collector relocates objects while accounting for potential concurrent readers that may access them; it does this by performing compare-and-swap operations on the object header (as described in Section 4.5).

After the first phase, the objects that are remaining in the heap, are completely local to the leaf task and are guaranteed to be inaccessible to other tasks. As an optimization, we perform purely Cheney-style semi space reclamation, with no hybrid component, because there is no possibility of concurrent tasks accessing these objects. We also optimize the relocation of such objects, performing no compare-and-swap operations them.

7 Evaluation

In this chapter, we evaluate and establish the practicality of our techniques. We compare our implementation, called MPL, against multiple other language implementations (including both procedural and functional languages) and demonstrate its efficiency and scalability.

Benchmarks Our evaluation considers two different benchmarking suites, totaling 48 benchmarks across five different languages.

The first suite includes 26 different benchmarks written in MPL. Half (thirteen) of these benchmarks are highly concurrent and entangled. To implement these benchmarks, we first implemented a number of non-blocking concurrent data structures, including Harris's non-blocking list [94], the Lindén-Jonsson priority queue [113], the Michael-Scott non-blocking queue [118], Kumar et al.'s persistent arrays [107], and many others. Using these, we then implemented sophisticated parallel algorithms for

- quantum circuit synthesis,
- delaunay triangulation,
- various graph analyses, including reachability/connectivity, $O(k)$-spanner, low-diameter decomposition boundaries, etc., and
- deduplication via concurrent hashing.

In addition, we implemented synthetic benchmarks that operate on concurrent data structures by mixing parallelizable work with updates and queries on the shared data structure(s).

Some of our benchmarks—such as the quantum synthesis and delaunay triangulation—are complex and have taken multiple person-months of work (each) to implement. The benchmarks are heavily concurrent; for example, the quantum circuit synthesizer implements a parallel version of the Solovay-Kitaev algorithm [66, 105] and uses both concurrent hash tables

and concurrent lists. Other benchmarks are from various problem domains, such as graph analysis, text processing, digital audio processing, image analysis and manipulation, numerical algorithms, computational geometry, and others; some are from prior work [176, 178]. Many of these benchmarks have been have been ported from highly optimized C/C++ implementations and include state of the art algorithms that have been designed over the past ten years [12, 45, 46, 67, 150, 151, 168].

Our second benchmarking suite contains the implementations of eight benchmarks in four languages (other than MPL) : C++, Go, Java, and Multicore OCaml. The C++ benchmarks come from PBBS [12, 151] and ParlayLib [46]. We ported these to Go, Java, and OCaml, while re-using existing Java implementations of two benchmarks. We selected these benchmarks for diversity (covering both disentanglement and entanglement, as well as both memory- and compute-intensive benchmarks), and for ease of implementation, as it takes significant work to implement each benchmark in multiple languages.

Methodology To measure timings, we first do a warmup by running the benchmark back-to-back for at least 5 seconds, and then do 20 back-to-back runs. All of this happens in the same program instance. The number reported is the average of the 20 runs, and the warmup is disregarded.

To measure space usage, we measure the average of the maximum resident set size (as reported by Linux) of 20 back-to-back runs of the benchmark. Back-to-back runs are executed in the same program instance to ensure that any effects of memory management amortization thresholds are taken into account (for example, a garbage collection might run only once every five runs). We use the maximum resident set size measurement because it takes into account all potential sources of space usage, including allocation freelists, GC metadata, etc., thus allowing for comparison across systems with different memory management strategies.

We write T_P for time on P processors, and similarly R_P for the max residency on P processors. Unless stated otherwise, all times are in seconds and all space numbers (max residencies) are in GB.

Experimental Setup We run all of our experiments on a 72-core Dell PowerEdge R930 consisting of 4 $\times$ 2.4GHz Intel (18-core) E7-8867 v4 Xeon processors, 1TB of memory, and running Ubuntu version `16.04.7` with Linux version `4.10.0-40-generic x86_64`. In Section 7.1, we use MLton version `20210117`. In Section 7.2, we use MPL version `0.3`. For cross-language comparisons (Section 7.4), we use the following systems: multicore OCaml version `5.0.0+dev4-2022-06-14` with default settings and the library `domainslib` version `0.4.2`; Go version `1.18.4`; `g++` version `10.3.0` with the `jemalloc` library, and the compiler flags `-O3`, `-march=native`, `-std=c++17`, and `-mcx16`; Java OpenJDK version `11.0.14`. For Java, we used the `G1GC` collector (which we found yielded the best performance) and controlled the number of threads with the setting `-XX:ActiveProcessorCount=`N.

	Time (s)					Space (GB)					Bytes Entangled
Benchmark	T_s	T_1	OV $\frac{T_1}{T_s}$	T_{72}	SU $\frac{T_s}{T_{72}}$	R_s	R_1	BU_1 $\frac{R_1}{R_s}$	R_{72}	BU_{72} $\frac{R_{72}}{R_s}$	ϵ_{72}
centrality	14.7	20.9	**1.42**	.466	**32**	33	6.6	**.20**	5.8	**.18**	0
delaunay	8.65	17.1	**1.98**	.667	**13**	2.7	1.7	**.63**	11	**4.1**	9.5 M
find-influencers	14.5	15.5	**1.07**	.433	**33**	35	7.4	**.21**	5.8	**.17**	.23 M
grep	1.43	2.16	**1.51**	.041	**35**	4.6	.61	**.13**	.85	**.18**	0
harris-linked-list	5.84	5.97	**1.02**	.163	**36**	.034	.021	**.62**	.094	**2.8**	3.3 M
hash-dedup	2.46	3.79	**1.54**	.081	**30**	6.8	.92	**.14**	1.3	**.19**	.56 M
interval-tree	2.89	4.30	**1.49**	.067	**43**	.53	.12	**.23**	.64	**1.2**	0
ldd-boundary	22.3	30.8	**1.38**	.705	**32**	31	7.0	**.23**	17	**.55**	.12 G
linden-pq	5.84	6.98	**1.20**	.193	**30**	.14	.11	**.79**	.15	**1.1**	9.3 M
linefit	3.01	2.34	**0.78**	.149	**20**	8.5	8.2	**.96**	8.3	**.98**	.16 M
max-indep-set	13.2	16.3	**1.23**	.379	**35**	27	7.0	**.26**	5.7	**.21**	0
mcss	1.88	4.74	**2.52**	.080	**23**	4.3	4.1	**.95**	4.1	**.95**	0
ms-queue	5.95	7.94	**1.33**	.303	**20**	.27	.18	**.67**	3.3	**12.2**	80 M
msort-int64	3.30	4.45	**1.35**	.085	**39**	5.0	.66	**.13**	.93	**.19**	0
nearest-nbrs	1.32	1.69	**1.28**	.039	**34**	1.5	.98	**.65**	2.1	**1.4**	0
persistent-arr	4.61	7.60	**1.65**	.109	**42**	.65	.60	**.92**	1.5	**2.3**	3.6 K
quant-synth	12.9	17.2	**1.33**	.319	**40**	4.6	.99	**.22**	12	**2.6**	1.8 M
quickhull	2.50	3.59	**1.44**	.115	**22**	15	18	**1.2**	20	**1.3**	0
range-query	14.1	16.2	**1.15**	.287	**49**	13	4.5	**.35**	4.3	**.33**	0
reachability	11.3	14.7	**1.30**	.412	**27**	37	5.9	**.16**	15	**.41**	82 K
reverb	1.01	1.36	**1.35**	.043	**23**	6.9	1.5	**.22**	2.0	**.29**	0
seam-carve	7.58	7.95	**1.05**	.280	**27**	1.2	.13	**.11**	.33	**.28**	0
spanner	11.2	16.9	**1.51**	.421	**27**	34	8.8	**.26**	46	**1.4**	5.9 M
tokens	1.55	1.89	**1.22**	.042	**37**	13	1.1	**.08**	1.2	**.09**	0
triangle-count	4.67	5.18	**1.11**	.110	**42**	2.7	.76	**.28**	1.2	**.44**	0
wc	5.18	7.44	**1.44**	.120	**43**	1.9	3.6	**1.9**	3.6	**1.9**	.28 M
geomean			**1.34**		**31**			**0.34**		**0.68**	

Figure 7.1: Comparison with sequential baseline: times, max residencies, overheads (OV), speedups (SU), space blowups (BU), and entanglement factors (ϵ).

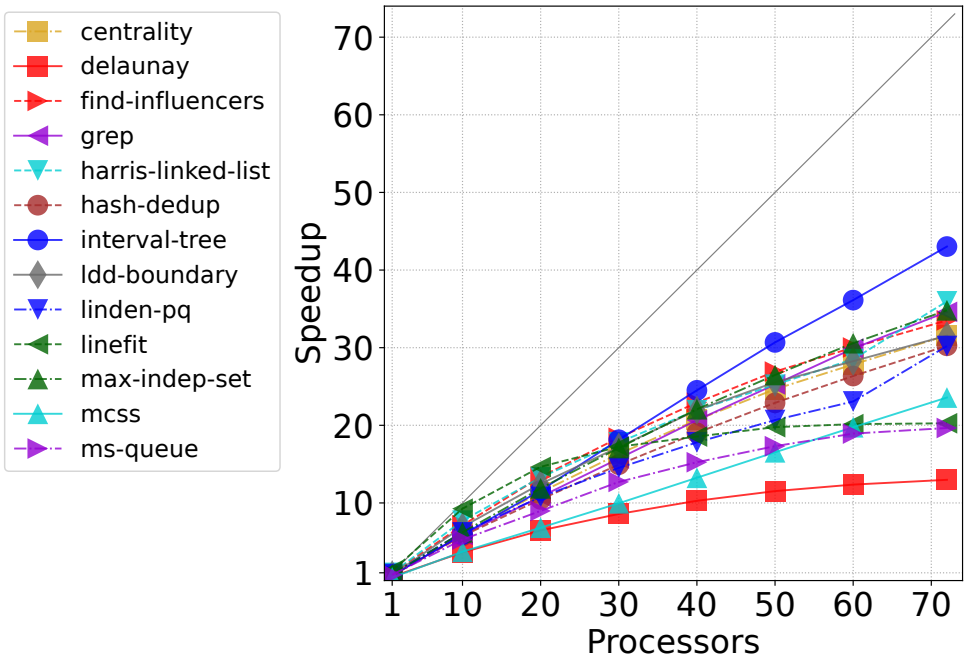

Figure 7.2: Speedups, continued in Figure 7.3

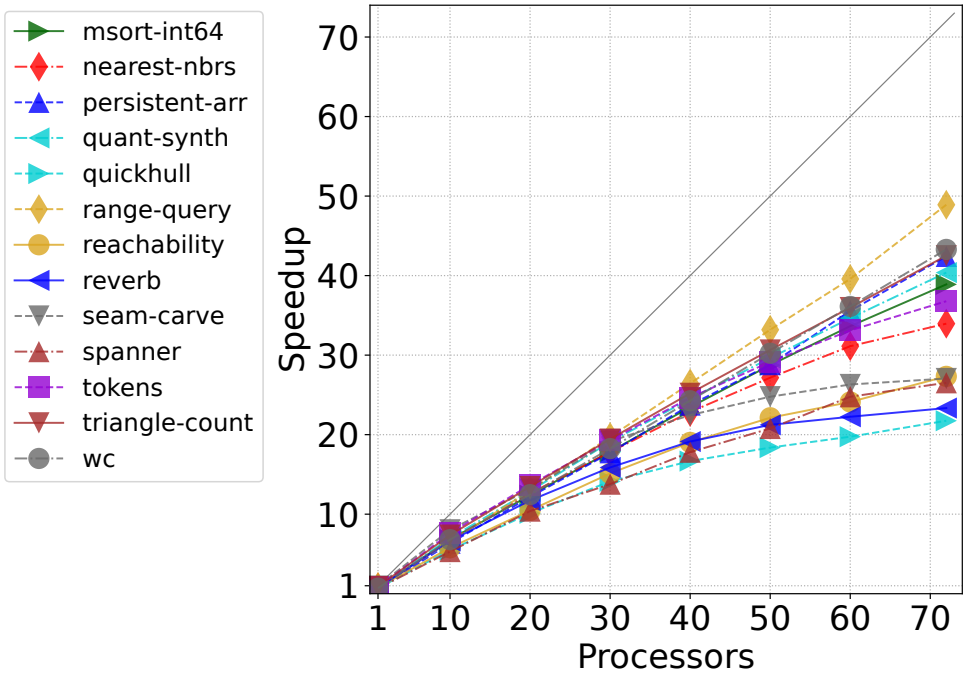

Figure 7.3: Speedups, continued

7.1 Overheads and Scalability

In this section, we compare against the MLton [120] compiler, which is a compiler for (sequential) Standard ML. Our MPL extends MLton with support for parallelism; there are minimal compilation differences between the two systems. (For MLton, we compile the sequential elision of each benchmark.) This comparison therefore allows us to determine the overheads and scalability of our memory manager by using executables produced by MLton as a sequential baseline. We also measure here the entanglement factor ϵ of each benchmark.

The results of the comparison are shown in Figure 7.1. The column T_s is the sequential baseline time, using MLton. The columns T_1 and T_{72} are the times of our MPL on 1 and 72 processors, respectively. Similarly, R_s is the maximum residency of the sequential baseline, and R_1 and R_{72} are the single-processor and 72-processor residencies of MPL. The overheads T_1/T_s indicates the performance of MPL on 1 processor relative to the sequential baseline; smaller numbers are favorable. Speedups on 72 processors in comparison to the sequential baseline are calculated as T_s/T_{72}, where larger numbers are favorable. To compare space usage, we compute the space blowup on p processors as R_p/R_s. This number indicates how much more memory MPL uses in comparison to the sequential baseline. Larger blowups indicate more space usage.

Speedups. We first observe that, on average, MPL achieves 31x speedup on 72 processors over MLton. These speedups range from 13x to 49x, with benchmarks such as `interval-tree`, `quant-synth`, and `triangle-count`—being compute bound—see higher speedups. In contrast, memory bound benchmarks such as `seam-carve`, `delaunay`, and `reverb` deliver lower speedups. The Figure 7.2, where we plot the speedup of MPL relative to MLton, also shows this trend. Some benchmarks plateau in speedup before 72 processors (e.g., `delaunay`, `seam-carve`, `linefit`). This is expected, as these benchmarks are memory-bound. All other benchmarks scale approximately linearly with the number of processors.

It's important to note that a perfect 72x speedup on 72 processors is unrealistic due to memory access bottlenecks. Memory is not 72-way parallel, and introduces fundamental scalability bottlenecks such as limited memory bandwidth and cache coherency delays. For this reason, the 31x speedup observed here is excellent, especially considering that state-of-the-art parallel programs written in C/C++ achieve similar speedups (see Section 7.4 for details).

Blowups. Across the board, we also see that MPL almost always uses less memory than MLton (Figure 7.1). On average, MPL uses approximately 30% less memory on 72 processors than MLton does sequentially. This is because MPL has a more aggressive GC policy than MLton; in anticipation of high memory usage for parallel applications, MPL collects aggressively.

Validating the Disentanglement Hypothesis. The disentanglement hypothesis states that most objects in a parallel fork-join program are disentangled. To validate the hypothesis in practice, we report the entanglement factor on 72 cores, ϵ_{72}, which represents the cumulative number of bytes that are entangled throughout the computation. The results across 26

benchmarks (Figure 7.1) show that entanglement factors are generally low in practice. Approximately half of the benchmarks considered here do not have any entangled objects, and therefore have an entanglement factor of 0. Amongst the entangled benchmarks, we can see that the amount of entangled bytes is generally small, especially in comparison to the maximum resident memory needed for 72 processor execution. For example, the graph reachability benchmark, `reachability`, has $\epsilon_{72} = 82$K, which is less than 1% of its resident memory $R_{72} = 0.41$GB. The low entanglement factor of this algorithm is analyzed and explained in detail in Section 3.3.3. At a high level, the reason for low entanglement factors is primarily that entangled objects are created only due to races, which are rare in these parallel programs.

Altogether, these results demonstrate that MPL is able to achieve significant speedups while controlling space usage effectively. The results also validate the disentanglement hypothesis, as the amount of entanglement for all benchmarks is less than 1%. Next, we analyze how well our implementation takes advantage of this hypothesis.

7.2 Disentanglement is Not Penalized

To show that our entanglement management techniques do not penalize disentangled programs, we compare our performance with MPL*. In MPL*, all entanglement management techniques are disabled, including the read barrier and any potential overhead related to managing entangled objects is completely removed. This makes MPL* specialized for benchmarks that do not have entangled objects.

We evaluate MPL with MPL* on all such benchmarks from Figure 7.1 as well as multiple other programs from the Parallel ML Benchmark Suite, for a total of 30 benchmarks. Across these benchmarks, we find that our MPL, which is fully general, performs similarly to MPL*. Specifically, we observed less than 5% overhead across all benchmarks, on both 1 and 72 processors, and observed less than 4% average space overhead. These results confirm that MPL effectively takes advantage of the disentanglement hypothesis, because its entanglement management techniques do not compromise the efficiency of disentangled programs. Additionally, these results are a sanity check for our theory (Theorem 1), which bounds the extra work of tracking entanglement to $O(\epsilon)$, where ϵ is the amount of entanglement. As expected, the overhead on disentangled benchmarks is negligible.

7.3 Entanglement Management Overhead

To further investigate the cost of managing entanglement, we developed a stress test for which we can control the amount of entanglement. The total program work, live memory, and parallelism of this benchmark all stay approximately the same regardless of the amount of entanglement introduced. On this stress test, we varied the amount of entanglement and measured the overhead on

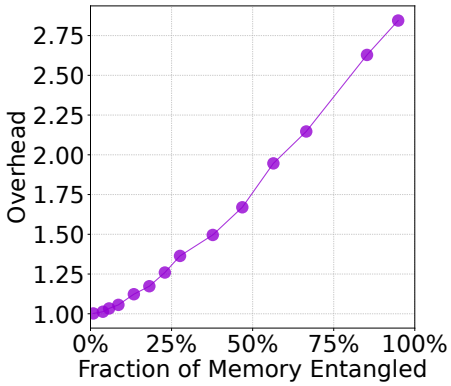

Figure 7.4: Stress test for entanglement management overhead

72 processors. Specifically, if the benchmark is $x\%$ entangled, then overhead is the ratio $T(x)/T(0)$, where $T(x)$ is the running time with $x\%$ entanglement. The results of this stress test are shown in Figure 7.4. First, we observe that the overhead increases linearly with the amount of entanglement. It is 1 (i.e., no overhead) at 0% entanglement and grows upto a factor of approximately 3 near 100% entanglement. This agrees with our theory: the work of entanglement tracking increases linearly with entanglement (Theorem 1). Furthermore, the constant factor hidden in the work bound $O(W + \epsilon)$ is relatively small (approximately 3 in this case)

7.4 Cross-Language Comparisons

In this section, we compare MPL with four other languages, C/C++, Go, Java, and multicore OCaml, all of which support nested fork-join parallelism in the same style as MPL. In Go, we use goroutines and channels to implement nested fork-join parallelism. In Java, we use parallel streams and the Java Fork/Join framework. (Note also that we account for Java warmup in our benchmarking methodology, discussed at the top of Chapter 7.) In OCaml, we use parallel primitives from the `domainslib` library.

We compare these languages on eight benchmarks, which come from the PBBS benchmark suite [12, 151] and the ParlayLib [46] library of parallel algorithms and data structures. We port these benchmarks from C/C++ into each of Parallel ML, Go, Java, and OCaml, preserving the underlying algorithms. For example, an array of structs in C/C++ is represented as (i) an array of tuples in Parallel ML and OCaml, (ii) an array of structs in Go, and (iii) an array of objects in Java. Where possible, we use external codes implemented by experts. In Java specifically, we use the `ConcurrentHashMap` and `parallelSort` implementations from the `java.util.concurrent` and `java.util.Arrays` libraries, respectively, for the `hash-dedup` and `msort-int64` benchmarks. Three benchmarks use lock-free data structures and have entanglement: `hash-dedup`, `linefit`, and `wc`.

Results The results of this comparison are shown in Figures 7.5 and 7.6. Figure 7.5 shows the time of each benchmark, as well as the ratio relative to our MPL; Figure 7.6 similarly reports space of each benchmark. Here, the abbreviations are **C** for C/C++, **M** for our MPL, **G** for Go, **J** for Java, and **O** for OCaml. For example, in the time figure, the column **J** is the running time (in seconds) for Java, and next to it, the column **J/M** is the ratio of Java's running time to MPL's running time. Higher ratios are favorable for MPL.

The results show that among memory-managed languages, MPL is faster and more space-efficient on all but two benchmarks. In particular, on benchmarks `hash-dedup` and `wc`, Java runs faster than MPL but consumes more space; on the same benchmarks, both Go and OCaml consume less space than MPL, but run slower than it. On all other benchmarks MPL performs better in terms of both speed and space consumption. On the `tokens` benchmark, we observe a

Time (s) on 72 processors:

	C/C++		MPL (Ours)		Go		Java		OCaml	
	C	$\frac{C}{M}$	M	$\frac{M}{M}$	G	$\frac{G}{M}$	J	$\frac{J}{M}$	O	$\frac{O}{M}$
hash-dedup	.043	**0.53**	.081	**1.00**	.127	**1.57**	.052	**0.64**	.174	**2.15**
linefit	.150	**1.01**	.149	**1.00**	.157	**1.05**	.405	**2.72**	1.18	**7.92**
mcss	.038	**0.47**	.080	**1.00**	.105	**1.31**	.507	**6.34**	.588	**7.35**
msort-int64	.058	**0.68**	.085	**1.00**	.280	**3.29**	.242	**2.85**	.489	**5.75**
primes	.072	**0.58**	.124	**1.00**	.189	**1.52**	.211	**1.70**	.166	**1.34**
sparse-mxv	.050	**1.04**	.048	**1.00**	.092	**1.92**	.089	**1.85**	.727	**15.15**
tokens	.020	**0.48**	.042	**1.00**	.484	**11.52**	.256	**6.10**	.947	**22.55**
wc	.058	**0.48**	.120	**1.00**	.171	**1.43**	.052	**0.43**	.910	**7.58**
geomean		**0.63**		**1.00**		**2.07**		**2.00**		**6.30**

Figure 7.5: MPL vs C++, Java, Go, and OCaml: time (seconds) on 72 processors. The time ratios are relative to MPL and show how fast it runs w.r.t. other languages (larger ratios favor MPL). The geomeans show the average of these ratios.

Space (GB) on 72 processors:

	C/C++		MPL (Ours)		Go		Java		OCaml	
	C	$\frac{C}{M}$	M	$\frac{M}{M}$	G	$\frac{G}{M}$	J	$\frac{J}{M}$	O	$\frac{O}{M}$
hash-dedup	1.40	**1.08**	1.30	**1.00**	1.10	**0.85**	7.70	**5.92**	1.10	**0.85**
linefit	8.80	**1.06**	8.30	**1.00**	8.40	**1.01**	23.0	**2.77**	33.0	**3.98**
mcss	4.80	**1.17**	4.10	**1.00**	4.90	**1.20**	27.0	**6.59**	4.20	**1.02**
msort-int64	1.40	**1.51**	.930	**1.00**	1.70	**1.83**	2.40	**2.58**	6.80	**7.31**
primes	.920	**2.30**	.400	**1.00**	.470	**1.17**	1.90	**4.75**	1.80	**4.50**
sparse-mxv	4.90	**1.09**	4.50	**1.00**	6.30	**1.40**	17.0	**3.78**	7.30	**1.62**
tokens	1.70	**1.42**	1.20	**1.00**	4.60	**3.83**	15.0	**12.50**	17.0	**14.17**
wc	2.50	**0.69**	3.60	**1.00**	3.60	**1.00**	5.60	**1.56**	1.90	**0.53**
geomean		**1.22**		**1.00**		**1.36**		**4.20**		**2.47**

Figure 7.6: MPL vs C++, Java, Go, and OCaml: space (GB) on 72 processors. The space ratios are relative to MPL and show the proportion of memory MPL saves, w.r.t. other languages. (larger ratios favor MPL). The geomeans average these ratios.

large performance gap between MPL and the other languages, where MPL is significantly faster and also consumes less space. This performance difference is due to the memory layout of a key data structure—specifically, an array with tuples for elements. MPL is able to "unbox" the tuples and flatten them into the array; note that MPL is based on MLton which performs optimizations such as data flattening. In Java, each element of the array is "boxed" and therefore incurs additional allocations. We confirmed that by pre-allocating the elements of the array and excluding this cost, the runtime of Java improves by approximately 2x. We similarly investigated the performance of Go and OCaml on this benchmark, and observed similar behavior.

We also compare against the C/C++ versions of these benchmarks. For our purposes, the C++ implementations serve as the "performance goal", as they have been independently developed and highly optimized to achieve the best parallel runtimes for these problems [46, 67]. For time performance, MPL is generally within a factor of 2 of C/C++, and in two cases matches C/C++ (`linefit` and `sparse-mxv`). This is excellent considering that the runtime of MPL the costs of automatic memory management.

We observe that MPL is also competitive in terms of space, with approximately 22% less space on average. This space difference is due to additional space consumed by the allocator used in the C/C++ benchmarks. In particular, the C/C++ benchmarks use an allocator provided by ParlayLib [46], which is primarily optimized for running time performance and incurs some space overhead to support fast parallel allocation. With the help of the ParlayLib authors, we were able replace the allocator with `jemalloc` [76] for some benchmarks. In particular, we were able to rerun `hash-dedup`, `msort-int64`, and `tokens` benchmarks using `jemalloc`, and observed that on these benchmarks, `jemalloc` yields similar time performance while using less space. (For these benchmarks, `jemalloc` uses 0.75x space relative to MPL on average; in comparison, the default ParlayLib allocator uses 1.4x space relative to MPL on this subset of benchmarks.) However, we also observed that some benchmarks crash with `jemalloc`, and we could not determine the source of the bug. We have relayed these findings to the ParlayLib authors.

Overall, our conclusion from these results is that MPL is generally fast, scalable, and space-efficient. Our experiments suggest that parallel functional programming can deliver the same performance and scalability as procedural and imperative languages.

8

Disentanglement Hypothesis for Futures

In the previous chapters, we described how disentanglement hypothesis can exploited to achieve provably and practically efficient memory management for fork-join programs. However, the fork-join model, while sufficient for many cases, can struggle to express certain classes of programs, such as those using pipelining [36, 165], or interaction, where parallelism is data-driven rather than control-driven [5, 123, 125, 126, 156].

For these reasons, many modern systems including Concurrent Haskell [115, 136], Habanero Java [97], Parallel ML (Manticore) [82, 83, 133, 165], Rust [144], TPL (a .NET library) [111], and X10 [59], support a more powerful construct for parallelism: futures. Futures allow you to create a parallel task and demand the result from the task at a later time when needed (hence the name "future"). Unlike fork-join which is a control-flow construct, futures are first-class values, which can be created, passed between function calls, or stored in memory, just like ordinary values. As a result, futures can effectively capture data-dependent parallelism, making possible speculative execution, pipelining [36, 165], and interactive applications [123, 125, 126, 156], which are difficult with fork-join.

The effectiveness of our disentanglement-based memory management techniques for fork-join programs raises the question: Can we generalize these techniques to programs with futures? Although a comprehensive answer to this question is beyond the scope of this thesis, we develop, in this chapter, the theoretical foundations for such a generalization. We consider a functional calculus with futures, Input/Output (I/O), and mutable state (references) and show that a broad range of programs written in this calculus only have disentangled objects. We then illustrate the potential advantages of using futures in a disentangled fashion, by providing concrete examples in Section 8.3. These examples demonstrate that **the disentanglement hypothesis applies even to programs that rely on dynamic dependencies and asynchronous interactions between tasks.**

These theoretical results do not directly follow from the disentanglement results of fork-join programs. Futures make computations first-class language citizens, allowing them to be stored and shared as values. Because parallel computations can share and synchronize with futures through language level primitives such as "get()", futures exhibit a complex dependency structure that evolves during runtime. Analyzing this dependency structure is more challenging than the series-parallel dependency structure of fork-join programs, where parallel tasks synchronize implicitly at the end of each task. To address this, we track object ownership and memory dependencies through a tree structure defined by spawned futures, and model synchronization between with futures as a "one-way dependency" that restructures this tree. This rewriting allows us to define and reason about disentanglement for futures.

8.1 Language

In this section, we consider a functional calculus that supports futures, I/O (input/output), and mutable references. We use this calculus to define disentanglement for programs with futures. To simplify theorems and their proofs, we define disentanglement as a program-level property, i.e., we say that a program satisfies disentanglement if it does not create any entangled objects. Although it is relatively straightforward to extend this calculus for defining per-object disentanglement as in Chapter 3, we focus on the entire program for clarity.

To define disentanglement, our semantics tracks the computation tree, which captures the control flow dependencies between the program threads and their memory actions such as allocations, reads, and writes. The computation tree is generated by the language semantics at each step of program evaluation. At a high level, we say that a computation tree satisfies disentanglement if the allocation actions of concurrent threads are oblivious to each other and a program evaluation satisfies disentanglement if the computation tree satisfies disentanglement at each step of the evaluation.

The semantics models parallelism by interleaving the evaluation of futures and their continuations. During the parallel evaluation of a future and its continuation, the semantics represents their actions as parallel in the computation tree. However, once the future finishes its evaluation, the semantics applies a join transformation on the tree. This transformation rewrites the computation tree to sequence the future's actions with the continuation's actions, capturing the idea that after the future has completed, its actions no longer need to be considered concurrent to the actions of the continuation. As we show in subsequent sections, the join transformation enables us to reason about disentanglement for futures.

8.1.1 Syntax

Our language contains constructs for functions, references, futures, and support for input/output operations. Figure 8.1 presents the syntax of the language.

$$
\begin{array}{rrcl}
\textit{Future names} & a, b & & \\
\textit{Memory Locations} & \ell & & \\
\textit{Types} & \tau & ::= & \texttt{bool} \mid \texttt{nat} \mid \tau \to \tau \mid \tau\ \texttt{fut} \\
\textit{Storables} & s & ::= & \texttt{true} \mid \texttt{false} \mid n \mid \texttt{fun}\ f\ x\ \texttt{is}\ e \mid \texttt{fcell}[a] \mid \texttt{ref}\ v \\
\textit{Values} & v & ::= & \ell \\
\textit{Memory} & \mu & \in & \textit{Locations} \rightharpoonup \textit{Storables} \\
\textit{Expressions} & e & ::= & v \mid s \mid x \mid e\ e \mid \texttt{fut}(e) \mid \texttt{fpoll}(e) \mid \texttt{get}(e) \mid \\
 & & & \texttt{ref}\ e \mid\ !\, e \mid e_1 := e_2 \mid \texttt{in\_nat}() \mid \texttt{out\_nat}(e) \\
\textit{Future Map} & \Delta & ::= & \emptyset \mid \Delta\ [a \blacktriangleright e] \mid \Delta\ [a \rhd v] \\
\textit{Action Trace} & t & ::= & \bullet \mid \mathbf{A}\ell \Leftarrow s \mid \mathbf{R}\ell \Rightarrow s \mid \mathbf{U}\ell \Leftarrow s \mid \mathbf{F}\ell \Rightarrow v \mid t \oplus t \\
\textit{Computation Trees} & T & ::= & \mathsf{Leaf}(t) \mid t \oplus (T \otimes T) \mid t \oplus_a T
\end{array}
$$

Figure 8.1: Syntax of λ^U

Types. The types include booleans, natural numbers, function types and the type τ `fut` for futures which evaluate an expression of type τ.

Storables and memory locations. To define disentanglement and precisely account for the actions on memory, the language distinguishes between ***storables*** and ***locations***. Storables include numbers, named recursive functions, and future cells. The language steps storables to locations and uses a memory store μ to map location to storables. A storable at a location may refer/point to other locations. We use $\mathcal{L}(s)$ to denote the locations referred to by the storable s. We represent locations with variables like ℓ, use $\mu(\ell)$ to denote the storable at location ℓ, and use $\mu[\ell \hookrightarrow s]$ to denote the allocation of location ℓ in the memory store μ (with the implicit requirement that $\ell \notin \text{dom}(\mu)$). Locations are the only irreducible form of the language. In our dynamics, we use this distinction between storables and locations to track all the program allocations.

Expressions. The expressions include the usual constructs for functions and references. The expression $\texttt{fut}(e)$ spawns a future to evaluate expression e. The language dynamics gives each future a name like a, b and other similar variables. For each future, the language allocates a future cell, which can be used by other threads to either 1) block on the future with the expression `get`, which returns the future's value when it finishes, or 2) poll the future with the expression `fpoll`, which returns true or false depending on whether the future has terminated. We denote the future cell for future a as $\texttt{fcell}[a]$. The expressions $\texttt{in\_nat}()$ and $\texttt{out\_nat}(e)$ support input and output operations for natural numbers. The language models an input as a non-deterministic step to a number and an output as a deterministic step that reads the argument and returns. This model captures the memory effects associated with these operations, which is sufficient for our goal of defining and reasoning about disentanglement.

8.1.2 Computation Trees

The computation tree records the memory actions taken during evaluation and organizes them according to their control flow dependencies. Each node of the tree represents a memory ***action*** taken by the program, which may be one of the following:

- $\mathbf{A}\ell \Leftarrow s$ is the allocation of location ℓ initialized with storable s.
- $\mathbf{R}\ell \Rightarrow s$ is a memory lookup (read) at location ℓ which returns storable s.
- $\mathbf{U}\ell \Leftarrow s$ is an update (write) which stores storable s at a mutable location ℓ.
- $\mathbf{F}\ell \Rightarrow v$ is a *synchronization* with the future whose future cell is at location ℓ which returns v.

The edges of the tree represent sequential ordering between the actions. For simplicity, we fuse sequentially taken memory actions into a single node of the tree and call it an action trace. An ***action trace*** contains a (possibly empty) series of actions composed by the operator $\oplus$, where the connective $\oplus$ emphasizes that actions within a trace are taken sequentially. Figure 8.1 shows the syntax of action traces and computation trees. We denote action traces with a lowercase variable like t and computation trees with an upper case variable like T.

When the evaluation starts, the computation tree only contains a single node. When a thread spawns a future, we add two leaves to the tree, one for storing the actions of the future and the second for storing the actions of the thread after the spawn. After a thread spawns a future, we refer to it as the ***continuation*** of that future.

A computation tree of the form $\mathsf{Leaf}(t)$ is a leaf and represents a sequential evaluation that performs actions in trace t. A computation tree of the form $t \oplus (T_1 \otimes_a T_2)$ represents an evaluation that performs the actions in trace t before spawning a future named a. The tensor $\otimes_a$ is called the ***spawn point*** of future a. The spawn point denotes that the actions of the future are in subtree T_1, actions of the continuation are in subtree T_2, and the respective actions are taken in parallel. A computation tree of the form $t \oplus_a T$ represents an evaluation that spawned a future named a, but the future has finished and "joined" with its continuation. The operator $\oplus_a$ is called the ***join point*** of future a. We describe the join operation in Section 8.1.4.

Small example. Figure 8.2 shows two computation trees of an evaluation where a thread *main* spawns a future a, which in turn spawns future b, and then thread *main* synchronizes with the future a to retrieve its result (location ℓ''). The figure shows two trees but we return to the right side tree later in the section. In the left tree, each box denotes an action trace and the edges between boxes denote the edges of the tree. The figure labels each box with the thread that performs its actions. After the *main* thread spawns the future a, it allocates the future cell $\texttt{fcell}[a]$ at location ℓ (see $\mathbf{A}\ell \Leftarrow \texttt{fcell}[a]$); the thread can now use location ℓ to synchronize with the future. The future a spawns future b and similarly allocates a cell for it at location ℓ' (see $\mathbf{A}\ell' \Leftarrow \texttt{fcell}[b]$). The future a then allocates some storable s at location ℓ'' (see $\mathbf{A}\ell'' \Leftarrow s$), which is the return value of the future. Its continuation (thread *main*) synchronizes with it and receives location ℓ'' (see $\mathbf{F}\ell \Rightarrow \ell''$).

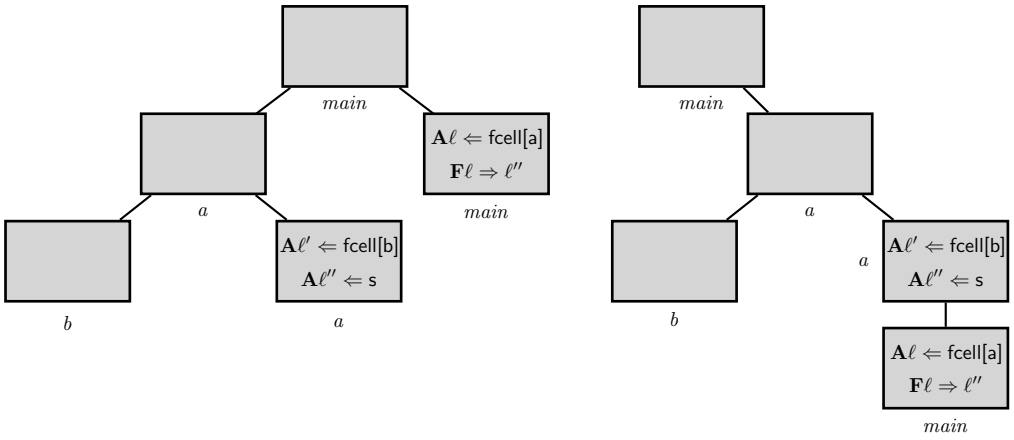

Figure 8.2: Two computation trees representing an evaluation where a thread *main* spawns a future named a, which in turn spawns future b, and then the main thread synchronizes with future a to retrieve its result (location ℓ''). We denote each node of the tree with a box containing a possibly empty action trace. The labels on the boxes denote the thread that performed the actions. The left and right trees show the tree structure without and with the join transformation. Without the join transformation, the left tree (mis-)characterizes the computation as entangled, as it represents the allocation ℓ'' of future a to be concurrent to the synchronized access of location ℓ'' by thread *main*. With the join transformation, the right tree correctly characterizes the computation to be disentangled as the allocation action is an ancestor of the synchronization action.

$\boxed{A \vdash t \ \mathit{de}}$

$$\frac{}{A \vdash \bullet \ \mathit{de}} \quad \frac{\mathcal{L}(s) \subseteq A}{A \vdash (\mathbf{A}\ell \Leftarrow s) \ \mathit{de}} \quad \frac{\ell \in A \qquad \mathcal{L}(s) \subseteq A}{A \vdash (\mathbf{R}\ell \Rightarrow s) \ \mathit{de}} \quad \frac{\ell \in A \qquad \mathcal{L}(v) \subseteq A}{A \vdash (\mathbf{F}\ell \Rightarrow v) \ \mathit{de}} \quad \frac{\ell \in A \qquad \mathcal{L}(s) \subseteq A}{A \vdash (\mathbf{U}\ell \Leftarrow s) \ \mathit{de}}$$

$$\frac{A \vdash t_1 \ \mathit{de} \qquad A \cup \mathsf{A}(t_1) \vdash t_2 \ \mathit{de}}{A \vdash t_1 \oplus t_2 \ \mathit{de}}$$

$\boxed{A \vdash T \ \mathit{de}}$

$$\frac{A \vdash t \ \mathit{de}}{A \vdash \mathsf{Leaf}(t) \ \mathit{de}} \quad \frac{A \vdash t \ \mathit{de} \qquad A \cup \mathsf{A}(t) \vdash T \ \mathit{de}}{A \vdash t \oplus_a T \ \mathit{de}}$$

$$\frac{A \vdash t \ \mathit{de} \qquad A \cup \mathsf{A}(t) \vdash T_1 \ \mathit{de} \qquad A \cup \mathsf{A}(t) \vdash T_2 \ \mathit{de}}{A \vdash t \oplus (T_1 \otimes T_2) \ \mathit{de}}$$

Figure 8.3: The figure defines the judgements $A \vdash T$ *de* and $A \vdash t$ *de*, which formalize disentanglement for a tree T and a node t respectively. The context A contains locations allocated by the ancestor actions of the tree/node.

8.1.3 Disentanglement

At a high level, disentanglement restricts concurrent threads from accessing each other's allocations. In the context of futures, disentanglement implies that a continuation is prohibited from accessing a future's allocations as long as the future is executing. However, if the continuation synchronizes with the future, disentanglement lifts the restrictions and allows the continuation to freely access the future's allocations. This is because a synchronization between the continuation and the future returns only after the future has terminated, rendering them non-concurrent.

We define disentanglement using the computation tree. The computation tree arranges the memory actions of program threads according to their control flow dependencies, i.e., it orders sequentially dependent memory actions in an ancestor-descendant relationship and keeps concurrent memory actions unrelated. A computation tree satisfies disentanglement when every action in the tree only mentions locations that are allocated by the ancestor actions of that action. An evaluation satisfies disentanglement if its computation tree maintains disentanglement at each step.

We formalize disentanglement for a tree with an inductive process using the judgement $A \vdash T$ *de*. The judgement's context A stores the set of locations allocated by the ancestor actions of tree T. The judgement checks that every location mentioned by an action of tree T is either present in the context A, or is allocated by some ancestor action in tree T. For a full tree T, if the judgement $\emptyset \vdash T$ *de* holds (i.e., with the empty context) then the tree T satisfies disentanglement.

Figure 8.3 defines the rules for the judgement $A \vdash T$ *de* for the tree and judgement $A \vdash t$ *de* for a node t of the tree. When the tree is of the form $\mathsf{Leaf}(t)$, the judgement checks the node t

which is an action trace. If the trace t is of the form $t_1 \oplus t_2$, then its rule checks the trace t_1 and subsequently checks the trace t_2 after extending the context A with the allocations in trace t_1. This is because actions in trace t_1 are ancestors to actions in trace t_2.

The rule for the allocation action $\mathbf{A}\ell \Leftarrow s$ checks that all locations in storable s are in the set A. The rule for the read action $\mathbf{R}\ell \Rightarrow s$ checks that both the location ℓ and locations in storable s are in set A. Similarly, the rule for the update action inspects both the location and the new storable. The rule for the synchronization action ($\mathbf{F}\ell \Rightarrow v$) checks the location ℓ and the locations in value v.

The rule for the form $t \oplus (T_1 \otimes T_2)$ checks that the trace t is disentangled ($A \vdash t$ *de*) and inspects the subtrees T_1 and T_2 after extending the context A with locations allocated by the trace t (denoted as $A(t)$). This is because actions of trace t are ancestors of actions in subtrees T_1 and T_2. Importantly, the rule does not include the locations allocated by tree T_2 to check tree T_1 and vice-versa. This is because their actions are not in an ancestor-descendant relationship.

When the tree is of the form $t \oplus_a T$, the judgement checks the trace t and the tree T. In a tree of this form, actions of trace t are ancestors to actions in tree T. Thus, the rule for this form checks the tree T after extending the context A with allocations of the trace t.

8.1.4 Joins

After a future terminates, its continuation can access its allocations without violating disentanglement. Our semantics represents this in the computation tree by transforming the tree after a future terminates. The semantics rearranges the tree such that memory actions of the future become ancestors of its continuation's actions, and they appear sequentially ordered. The semantics performs this ***join transformation*** at an evaluation step called ***join***. The join step is only a tool for reasoning about disentanglement and, in no way, affects the actual parallelism of the program.

Example. To illustrate the join transformation, we draw two trees in Figure 8.2. The two trees represent an evaluation in which a thread *main* spawns future a, which subsequently spawns future b, and then thread *main* synchronizes with the future to retrieve its result. The left tree does not incorporate the join transformation whereas the right tree does. In the left tree, the synchronization action by thread *main* ($\mathbf{F}\ell \Rightarrow \ell''$) and the allocation action ($\mathbf{A}\ell'' \Leftarrow s$) by future a are positioned concurrently in the tree, i.e., they are not in an ancestor-descendant relationship. Consequently, since the allocation of location ℓ'' is not an ancestor of the synchronization action, the left tree mistakes the evaluation as violating disentanglement. In contrast, the right tree satisfies disentanglement because of the join transformation. By sequencing the future's actions with those of the continuation, the join transformation ensures that the allocation action ($\mathbf{A}\ell'' \Leftarrow s$) for location ℓ'' becomes an ancestor of the synchronization action ($\mathbf{F}\ell \Rightarrow \ell''$).

By sequencing the future's actions before the continuation's actions, the join transformation represents that once a future finishes, it is no longer concurrent with the continuation. However, it is essential to note that any futures spawned by the completed future may still be

$$\frac{T_1 = \mathsf{Leaf}(t_1)}{\bowtie (a, T_1, T_2) = t_1 \oplus_a T_2}\ \textsc{Leaf} \qquad \frac{T_1 = t_1 \oplus_b T_1'}{\bowtie (a, T_1, T_2) = t_1 \oplus_b \bowtie (a, T_1', T_2)}\ \textsc{Join Point}$$

$$\frac{T_1 = t_1 \oplus (T_1' \otimes_b T_1'')}{\bowtie (a, T_1, T_2) = t_1 \oplus (T_1' \otimes_b \bowtie (a, T_1'', T_2))}\ \textsc{Spawn Point}$$

Figure 8.4: Function Join

executing and remain concurrent with the continuation. As a result, it is crucial for the join transformation to not sequence their actions with the continuation.

Join Function. Figure 8.4 shows the join transformation with the function $\bowtie$. The function takes arguments a, T_1, T_2, where a is the future that finished and trees T_1 and T_2 are the children of the spawn point of future a, i.e., tree T_1 is the tree containing the future's actions, and tree T_2 is the tree containing the continuation's actions. To perform the join, the function recurses down the tree T_1, following the actions of the future until it reaches the leaf. This leaf marks the end of the future's actions because it has finished. Then, the function sticks tree T_2 as a descendant of that leaf, making all of the future's actions ancestors of the continuation's actions. The function does not change the relationship between any actions corresponding to other threads in trees T_1 and T_2.

In the leaf case, when tree T_1 is a leaf of form $\mathsf{Leaf}(t_1)$, the function returns $t_1 \oplus_a T_2$, where the operator $\oplus_a$ marks the join point of future a. For tree T_1 of the form $t_1 \oplus_b T_1'$, which represents the join point of future b, the function returns the tree $t_1 \oplus_b \bowtie (a, T_1', T_2)$. This resulting tree maintains the relationship between trace t_1 and tree T_1', and also incorporates the join of tree T_2 with tree T_1'. For tree T_1 of the form $t_1 \oplus (T_1' \otimes_b T_1'')$, which represents the spawn point of future b, the function recurses on subtree T_1'' and leaves subtree T_1' unchanged. This is because the tree T_1' contains the actions of future b and the actions of the future a are in tree T_1''. By recursing down tree T_1'', the function makes all the actions of the future a ancestors to the actions of tree T_2.

Fork/Join. We note that our join here is similar to a "join" in a fork-join computation but there are some important differences. The join in fork-join is two-way because two sibling tasks finish their execution and join with each other. It requires that all tasks nested within the joining tasks also terminate before the join can proceed. As a result, join effectively eliminates all concurrency and parallelism within its scope. On the other hand, in the case of futures, the join is one-way because a future finishes and joins into its continuation. Furthermore, tasks spawned by the joining future can escape its scope. This key difference allows for concurrency and parallelism even after the join, as the spawned tasks can execute independently of the future and its continuation.

We can indeed observe these differences by considering the join function. For fork-join, a corresponding join function, say function J, is $\mathsf{J}(\mathsf{Leaf}(t_1), \mathsf{Leaf}(t_2)) = \mathsf{Leaf}(t_1 \oplus t_2)$. This function is relatively simple because both its arguments are guaranteed to be leaves in the

computation tree. Any tasks nested within their scope have finished. In contrast, the join function for futures operates on trees. This is because the continuation has not finished and the future, even though itself has finished, may have spawned other futures which are still executing. This ability of futures to spawn futures which continue to execute concurrently beyond the joining point introduces additional challenges for reasoning about concurrency and proving disentanglement.

8.1.5 Language Semantics

Our operational semantics steps a program state consisting of four components: (i) a future map Δ tracking the evaluation of futures, (ii) a memory store μ mapping locations to storables, (iii) a computation tree T, and (iv) an expression e. We write a program state as $(\Delta\,;\mu\,;T\,;e)$. Figure 8.5 and Figure 8.6 show the rules for the semantics. We split the rules into two figures for space reasons; Figure 8.6 shows the rules specific to futures, and Figure 8.5 shows rules for other features of the language.

Allocations and functions. The allocation rule ALLOC extends the memory μ with location ℓ mapped to storable s and records it in the leaf $\mathsf{Leaf}(t)$ as the allocation action $\mathbf{A}\ell \Leftarrow s$. Rules APPSL and APPSR for function application step the function and the argument respectively. The application rule APP applies the function to the argument. It substitutes recursive mentions of the function in its body e by location ℓ, and substitutes the variable x by the argument v.

References. Rules REFS, BANGS, UPDSL, and UPDSR evaluate their corresponding subexpressions. The rule BANG corresponds to dereferencing a mutable location ℓ and looks up the location ℓ in memory store μ and returns the stored value v. The rule records this in the leaf $\mathsf{Leaf}(t)$ as the read action $\mathbf{R}\ell \Rightarrow \texttt{ref}\ v$. The rule UPD corresponds to a destructive update and updates the memory location ℓ to refer to value v. The rule records this in the leaf $\mathsf{Leaf}(t)$ as the update action $\mathbf{U}\ell \Leftarrow \texttt{ref}\ v$.

Input/Output. The rule INPUT steps the expression $\mathsf{in\_nat}()$ to a non-deterministic natural number n. The natural number n will then be allocated in the memory store by the rule ALLOC. The rule's non-determinism models the effect of the input on evaluation and the allocation captures the effect of the input on memory and disentanglement. The rule OUTPUT takes a location ℓ, which stores a natural number n, and steps the location to a unit value. The rule extends the computation tree with the read action $\mathbf{R}\ell \Rightarrow n$. This step models an output to an environment and the read action captures the effect of the output on disentanglement. For brevity, we do not model I/O on other types but the language can be extended to support them.

Futures. The rule FSPAWN spawns a future. It steps the expression $\texttt{fut}(e)$ to the future cell $\texttt{fcell}[a]$, where a is an unused/fresh name. Each future's evaluation is tracked in the ***future map***, denoted Δ, which stores all future names and their expressions. We write $\Delta[a \blacktriangleright e]$ to

$$\frac{\ell \notin \mathrm{dom}(\mu)}{\Delta\,;\,\mu\,;\,\mathsf{Leaf}(t)\,;\,s \to \Delta\,;\,\mu[\ell \hookrightarrow s]\,;\,\mathsf{Leaf}(t \oplus (\mathbf{A}\ell \Leftarrow s))\,;\,\ell}\ \textsc{Alloc}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e_1 \to \Delta'\,;\,\mu'\,;\,T'\,;\,{e_1}'}{\Delta\,;\,\mu\,;\,T\,;\,(e_1\ e_2) \to \Delta'\,;\,\mu'\,;\,T'\,;\,(e_1'\ e_2)}\ \textsc{AppSL} \qquad \frac{\Delta\,;\,\mu\,;\,T\,;\,e_2 \to \Delta'\,;\,\mu'\,;\,T'\,;\,{e_2}'}{\Delta\,;\,\mu\,;\,T\,;\,(\ell_1\ e_2) \to \Delta'\,;\,\mu'\,;\,T'\,;\,(\ell_1\ e_2')}\ \textsc{AppSR}$$

$$\frac{\mu(\ell) = \mathtt{fun}\ f\ x\ \mathtt{is}\ e}{\Delta\,;\,\mu\,;\,\mathsf{Leaf}(t)\,;\,(\ell\ v) \to \Delta\,;\,\mu\,;\,\mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fun}\ f\ x\ \mathtt{is}\ e))\,;\,[\ell, v\ /\ f, x]e}\ \textsc{App}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e_1 \to \Delta\,;\,\mu'\,;\,T'\,;\,{e_1}'}{\Delta\,;\,\mu\,;\,T\,;\,(e_1 := e_2) \to \Delta\,;\,\mu'\,;\,T'\,;\,(e_1' := e_2)}\ \textsc{UpdSL}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e_2 \to \Delta\,;\,\mu'\,;\,T'\,;\,{e_2}'}{\Delta\,;\,\mu\,;\,T\,;\,(\ell_1 := e_2) \to \Delta\,;\,\mu'\,;\,T'\,;\,(\ell_1 := e_2')}\ \textsc{UpdSR}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e \to \Delta\,;\,\mu'\,;\,T'\,;\,e'}{\Delta\,;\,\mu\,;\,T\,;\,(\mathtt{ref}\ e) \to \Delta\,;\,\mu'\,;\,T'\,;\,(\mathtt{ref}\ e')}\ \textsc{RefS}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e \to \Delta\,;\,\mu'\,;\,T'\,;\,e'}{\Delta\,;\,\mu\,;\,T\,;\,(!\,e) \to \Delta\,;\,\mu'\,;\,T'\,;\,(!\,e')}\ \textsc{BangS}$$

$$\frac{\mu(\ell) = \mathtt{ref}\ v}{\Delta\,;\,\mu\,;\,\mathsf{Leaf}(t)\,;\,(!\,\ell) \to \Delta\,;\,\mu\,;\,\mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{ref}\ v))\,;\,v}\ \textsc{Bang}$$

$$\frac{}{\Delta\,;\,\mu_0[\ell \hookrightarrow s]\,;\,\mathsf{Leaf}(t)\,;\,(\ell := v) \to \Delta\,;\,\mu_0[\ell \hookrightarrow \mathtt{ref}\ v]\,;\,\mathsf{Leaf}(t \oplus (\mathbf{U}\ell \Leftarrow \mathtt{ref}\ v))\,;\,v}\ \textsc{Upd}$$

$$\frac{n :\ \mathsf{int}}{\Delta\,;\,\mu\,;\,\mathsf{Leaf}(t)\,;\,\mathsf{in\_nat}() \to \Delta\,;\,\mu\,;\,\mathsf{Leaf}(t)\,;\,n}\ \textsc{Input}$$

$$\frac{\mu(\ell) = n}{\Delta\,;\,\mu\,;\,\mathsf{Leaf}(t)\,;\,\mathsf{out\_nat}(\ell) \to \Delta\,;\,\mu\,;\,\mathsf{Leaf}(t \oplus \mathbf{R}\ell \Rightarrow n)\,;\,()}\ \textsc{Output}$$

Figure 8.5: Dynamics of λ^U(continued in Figure 8.6)

$$\frac{(a \textit{ fresh}) \qquad \Delta' = \Delta[a \blacktriangleright e]}{\Delta \,;\, \mu \,;\, \mathsf{Leaf}(t) \,;\, \mathtt{fut}(e) \to \Delta' \,;\, \mu \,;\, t \oplus (\mathsf{Leaf}(\bullet) \otimes_a \mathsf{Leaf}(\bullet)) \,;\, \mathtt{fcell}[a]} \textsc{FSpawn}$$

$$\frac{\Delta(a) \blacktriangleright e_1 \qquad \Delta \,;\, \mu \,;\, T_1 \,;\, e_1 \to \Delta' \,;\, \mu' \,;\, T_1' \,;\, e_1' \qquad (\Delta' = \Delta_s'[a \blacktriangleright e_1])}{\Delta \,;\, \mu \,;\, t \oplus (T_1 \otimes_a T_2) \,;\, e_2 \to \Delta_s'[a \blacktriangleright e_1'] \,;\, \mu' \,;\, t \oplus (T_1' \otimes_a T_2) \,;\, e_2} \textsc{FutS}$$

$$\frac{\Delta \,;\, \mu \,;\, T_2 \,;\, e_2 \to \Delta' \,;\, \mu' \,;\, T_2' \,;\, e_2'}{\Delta \,;\, \mu \,;\, t \oplus (T_1 \otimes_a T_2) \,;\, e_2 \to \Delta' \,;\, \mu' \,;\, t \oplus (T_1 \otimes_a T_2') \,;\, e_2'} \textsc{ContS}$$

$$\frac{\Delta = \Delta_s[a \blacktriangleright v] \qquad \bowtie (a, T_1, T_2) = T \qquad \Delta' = \Delta_s[a \triangleright v]}{\Delta \,;\, \mu \,;\, t \oplus (T_1 \otimes_a T_2) \,;\, e_2 \to \Delta' \,;\, \mu \,;\, t \oplus T \,;\, e_2} \textsc{FJoin}$$

$$\frac{\mu(\ell) = \mathtt{fcell}[a] \qquad \Delta(a) \triangleright v}{\Delta \,;\, \mu \,;\, \mathsf{Leaf}(t) \,;\, \mathtt{fpoll}(\ell) \to \Delta \,;\, \mu \,;\, \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a])) \,;\, \mathsf{true}} \textsc{PollT}$$

$$\frac{\mu(\ell) = \mathtt{fcell}[a] \qquad \Delta(a) \blacktriangleright e}{\Delta \,;\, \mu \,;\, \mathsf{Leaf}(t) \,;\, \mathtt{fpoll}(\ell) \to \Delta \,;\, \mu \,;\, \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a])) \,;\, \mathsf{false}} \textsc{PollF}$$

$$\frac{\Delta \,;\, \mu \,;\, T \,;\, e \to \Delta' \,;\, \mu' \,;\, T' \,;\, e'}{\Delta \,;\, \mu \,;\, T \,;\, \mathtt{get}(e) \to \Delta' \,;\, \mu' \,;\, T' \,;\, \mathtt{get}(e')} \textsc{GetS}$$

$$\frac{\mu(\ell) = \mathtt{fcell}[a] \qquad \Delta(a) \triangleright v}{\Delta \,;\, \mu \,;\, \mathsf{Leaf}(t) \,;\, \mathtt{get}(\ell) \to \Delta \,;\, \mu \,;\, \mathsf{Leaf}(t \oplus (\mathbf{F}\ell \Rightarrow v)) \,;\, v} \textsc{Get}$$

Figure 8.6: Dynamics of λ^U continued

extend map Δ with future a and use $\Delta(a) \blacktriangleright e$ to denote that a is mapped to expression e. The rule FSpawn extends the future map with the new future and also adds two empty leaves to the computation tree. The resulting tree is of the form $t \oplus (\mathsf{Leaf}(\bullet) \otimes \mathsf{Leaf}(\bullet))$, where the symbol $\bullet$ denotes the empty trace. The rule composes the leaves with the operator $\otimes_a$, marking the spawn point of future a. The left and right leaf will store the subsequent actions of the future and the continuation respectively.

The rules FutS and ContS step the program state if the computation tree is of the form $t \oplus (T_1 \otimes T_2)$. The rule FutS looks up the future name a and expression e_1 in future map Δ, and steps the expression e_1 with the left subtree T_1. The rule has a premise, the condition $\Delta' = \Delta'_s[a \blacktriangleright e_1]$ which guarantees that stepping e_1 does not change the future map for future a, i.e., $\Delta(a) = \Delta'(a)$. The rule has the premise because this rule is responsible for tracking the evaluation of future a. For the resulting state, the rule FutS maps the future a to expression e'_1. The rule ContS steps the continuation e_2 with the right subtree T_2. These rules can be interleaved non-deterministically to model parallel evaluation.

Once a future is fully evaluated to a value, the rule FJoin joins it with its continuation. The rule performs the join transformation on the computation tree, as described earlier in Section 8.1.4, and also updates the future map to mark that the future has joined. In the future map, we use an unshaded triangle to denote joined futures. The rule changes the map from $\Delta_s[a \blacktriangleright v]$ to $\Delta_s[a \triangleright v]$.

Polling and synchronization. The rules Poll, PollT, and PollF describe the semantics of polling. The rule Poll steps its argument subexpression. If the future being polled has joined, then the rule PollT steps $\texttt{fpoll}(\ell)$ to `true`; otherwise, the rule PollF steps $\texttt{fpoll}(\ell)$ to `false`. Since both the rules look up location ℓ in the memory store μ, they insert the read action $\mathbf{R}\ell \Rightarrow \texttt{fcell}[a]$ to the computation tree. Note that polling is a non-blocking primitive, as the expression `fpoll` always steps immediately.

Unlike the expression $\texttt{fpoll}(e)$, the expression $\texttt{get}(e)$ blocks until the future completes and then returns its the value. The rule GetS steps the argument expression e to a location ℓ. Then, once the future referred by location ℓ has joined, the rule Get retrieves the value of from the future map and returns it. The rule records this synchronization in the computation tree with the action $\mathbf{F}\ell \Rightarrow v$, where v is the return value of the future. Notice that the rule Get has the condition $\Delta_s[a \triangleright v]$ in the premise, asserting that it blocks until the future has joined.

8.2 Race Freedom and Disentanglement

In this section, we show that the determinacy-race-free programs of our language satisfy disentanglement. In the next section, Section 8.3, we illustrate how this result implies that we can express pipelining, dynamic programming, and interactive applications while satisfying disentanglement

$$\frac{}{F \vdash \bullet \ \textit{drf}} \quad \frac{\ell \notin F}{F \vdash (\mathbf{A}\ell \Leftarrow s) \ \textit{drf}} \quad \frac{\ell \notin F}{F \vdash (\mathbf{F}\ell \Rightarrow v) \ \textit{drf}} \quad \frac{\ell \notin F}{F \vdash (\mathbf{U}\ell \Leftarrow s) \ \textit{drf}} \quad \frac{\ell \notin F}{F \vdash (\mathbf{R}\ell \Rightarrow s) \ \textit{drf}}$$

$$\frac{F \vdash t_1 \ \textit{drf} \qquad F \vdash t_2 \ \textit{drf}}{F \vdash t_1 \oplus t_2 \ \textit{drf}}$$

$$\frac{F \vdash t \ \textit{drf}}{F \vdash \mathsf{Leaf}(t) \ \textit{drf}} \qquad \frac{F \vdash t \ \textit{drf} \qquad F \vdash T \ \textit{drf}}{F \vdash t \oplus_a T \ \textit{drf}}$$

$$\frac{F \vdash t \ \textit{drf} \qquad F \cup \mathsf{AW}(T_2) \vdash T_1 \ \textit{drf} \qquad F \cup \mathsf{AW}(T_1) \vdash T_2 \ \textit{drf}}{F \vdash t \oplus (T_1 \otimes T_2) \ \textit{drf}}$$

Figure 8.7: The figure defines the judgement $F \vdash T$ *drf*, where F is a set of locations that actions of T must not mention. The function AW takes a tree and returns the set of locations allocated/updated by it.

8.2.1 Determinacy Race Freedom

Determinacy Races. A ***determinacy race*** occurs when two concurrent threads access the same memory location, and one of those accesses modifies the location [130]. ***Determinacy race freedom*** is the program property that guarantees that every execution of the program is free of determinacy races. We say that a computation with no determinacy races is determinacy race free. For brevity, we write race and race freedom to mean determinacy race and determinacy race freedom.

We define race-freedom formally using the computation tree. The computation tree organizes the memory actions of program threads based on their control flow dependencies. It orders sequential memory actions in an ancestor-descendant relationship and keeps concurrent memory actions unrelated. The memory actions in the tree include modifying actions such as allocation ($\mathbf{A}\ell \Leftarrow s$) and update ($\mathbf{U}\ell \Leftarrow s$) and non-modifying actions such as read ($\mathbf{R}\ell \Rightarrow s$) and sync ($\mathbf{F}\ell \Rightarrow v$). A computation tree satisfies race freedom if no modifying action on a location is concurrent to another action, modifying or otherwise, on that location.

Figure 8.7 defines race freedom for a computation tree with an inductive process using the judgement $F \vdash T$ *drf*. The judgement's context F represents a set of "forbidden locations" that are modified by threads concurrent to those represented in tree T. The judgement $F \vdash T$ *drf* ensures that no action of tree T operates on a location in the set F.

Let's look at the rules for the judgement. The rule for the tree $\mathsf{Par}(t, T_1, T_2)$ shows how the forbidden set prohibits races between the concurrent trees T_1 and T_2. When checking tree T_1, the rule extends the forbidden set with locations that are modified by tree T_2. Specifically, let $\mathsf{AW}(T)$ represents the set of locations modified by tree T. Then the rule checks tree T_1 with the forbidden set $F \cup \mathsf{AW}(T_2)$, which ensures that actions of tree T_1 are forbidden from locations modified by tree T_2. The rule checks tree T_2 similarly.

The rule for the read action $\mathbf{R}\ell \Rightarrow s$ checks that location ℓ is not in the set F, checking that no concurrent action modified location ℓ. The rules for other actions also check the respective locations. These rules uncover an interesting perspective on the distinction between

race freedom and disentanglement. While race freedom only restricts where an action occurs, disentanglement goes a step further and also restricts what an action can store or retrieve. For example, the rule for checking disentanglement of a read action $\mathbf{R}\ell \Rightarrow s$ checks both the location ℓ and the locations in storable s (see Figure 8.3), whereas the rule for race freedom only checks the location ℓ, leaving the contents of storable s unrestricted. From this perspective, it is perhaps surprising that disentanglement applies to a broader class of programs (because it is implied by race freedom).

8.2.2 Determinacy Race Free Programs are Disentangled

We prove taking an arbitrary number of steps from an initial state, if the computation tree satisfies the *drf* property at each step, then it satisfies the *de* property after the final step. Thus, if the *drf* property holds for every step, the *de* property does as well.

Theorem 7 (DRF $\Rightarrow$ DE)**.** *For any* $\emptyset\,;\emptyset\,;\mathsf{Leaf}(\bullet)\,;e_0 \to^n \Delta\,;\mu\,;T_n\,;e_n$ *where* $\mathcal{L}(e_0)=\emptyset$*, if every intermediate tree* T_i *in* $\{T_1 \ldots T_n\}$ *satisfies* $\emptyset \vdash T_i$ *drf, then* $\emptyset \vdash T_n$ *de.*

To prove the theorem, we account for the two ways a thread can share an allocation with another thread. First, a thread can synchronize with another thread as a future, which may potentially return handles to handles to other futures because futures are themselves memory allocations (they are first class). Thus, to establish disentanglement, we must prove that a thread never gets a handle to a future that is spawned concurrently. Second, a thread can communicate with another thread through mutable effects. However, because the program satisfies determinacy race freedom, we can prove this form sharing does not occur. We formalize this by defining two properties, namely ***drfde*** and ***ok***.

The property *drfde* implies both disentanglement and race freedom and the property *ok* captures the structure of futures. For a state $\Delta\,;\mu\,;T\,;e$, we write the *drfde* property as the judgement $K\,;A\,;F \vdash_\mu \Delta\,;T\,;e$ *drfde* and the *ok* property as the judgement $A\,;\Delta \vdash_\mu K$ *ok*. The set A is a set of locations and set K is a set of futures, both of which a thread can access without violating disentanglement. The set F is the set of forbidden locations that a thread should not access or else the thread violates race freedom. These sets are empty for the full state, but for sub states of various threads in the program, they encode memory information relevant to disentanglement and race freedom. We discuss the properties in detail and prove them by induction.

Property ok. The judgement *ok* ($A\,;\Delta \vdash_\mu K$ *ok*) guarantees that the value of every terminated future in set K only refers to futures and locations within sets K and A. With this judgement, we can show that if a thread performs a synchronization action on a future to retrieve a value, then the value only contains locations and futures within sets K and A. We can define it formally as follows.

$$\frac{K \subseteq \mathrm{dom}(\Delta) \qquad \forall a \in K.\ \Delta(a) \rhd v \Rightarrow \mathcal{L}(v) \subseteq A \wedge \mathsf{Fut}(v,\mu) \subseteq K}{A\,;\Delta \vdash_\mu K\ \textit{ok}}$$

$\boxed{K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ T\ ;\ e\ \textit{drfde}}$

$$\frac{\mathcal{L}(e) \subseteq A \quad \mathsf{Fut}(e,\mu) \subseteq K \qquad \forall \ell \in A \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \qquad \forall \ell \in A.\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow a \in K}{K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ e\ \textit{drfde}}\ \textsc{(Base)}$$

$$\frac{F \vdash t\ \textit{drf} \qquad A \vdash t\ \textit{de} \qquad K\ ;\ A \cup \mathsf{A}(t)\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ e\ \textit{drfde}}{K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(t)\ ;\ e\ \textit{drfde}}\ \textsc{(Leaf)}$$

$$\frac{F \vdash t\ \textit{drf} \qquad A \vdash t\ \textit{de} \qquad \Delta(a) \rhd v \qquad \begin{array}{c} K\ ;\ A \cup \mathsf{A}(t)\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ v\ \textit{drfde} \\ K \cup \{a\}\ ;\ A \cup \mathsf{A}(t)\ ;\ F \vdash_\mu \Delta\ ;\ T\ ;\ e\ \textit{drfde} \end{array}}{K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ t \oplus_a T\ ;\ e\ \textit{drfde}}\ \textsc{(Joined)}$$

$$\frac{F \vdash t\ \textit{drf} \qquad A \vdash t\ \textit{de} \qquad \Delta(a) \blacktriangleright e_1 \qquad \begin{array}{c} K\ ;\ A \cup \mathsf{A}(t)\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ T_1\ ;\ e_1\ \textit{drfde} \\ K \cup \{a\}\ ;\ A \cup \mathsf{A}(t)\ ;\ F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta\ ;\ T_2\ ;\ e_2\ \textit{drfde} \end{array}}{K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ t \oplus (T_1 \otimes T_2)\ ;\ e_2\ \textit{drfde}}\ \textsc{(Par)}$$

Figure 8.8: Strengthening of disentanglement and race freedom with invariants on futures and memory

As a sanity check, the property *ok* ensures that all futures in K are in the future map Δ. The property then checks the values of terminated futures; recall that futures that have terminated are mapped with an unshaded triangle in the future map Δ. For each terminated future, the property checks that the return value v of the future only refers to locations and futures in sets A and K respectively. The property uses the function $\mathcal{L}(v)$ that returns all the locations mentioned in value v and asserts that all such locations are in the set A. The property uses the function $\mathsf{Fut}(v, \mu)$, which returns all futures referred by value v and asserts that they are in the set K. We define it formally as follows:

- $\mathsf{Fut}(e, \mu) = \bigcup_{\ell \in \mathcal{L}(e)} \mathsf{Fut}(\ell, \mu)$, where $\mathcal{L}(e)$ contains all locations mentioned by expression e.
- $\mathsf{Fut}(\ell, \mu) = \{a\}$, if $\mu(\ell) = \texttt{fcell}[a]$.

Property drfde. The *drfde* property implies disentanglement and determinacy-race-freedom, and in addition imposes restrictions on the memory locations mentioned by the expressions of various program threads. Given a subtree T and an expression e, the judgement *drfde* ($K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ T\ ;\ e$ *drfde*) enforces that (i) the tree T satisfies disentanglement w.r.t. the set A, i.e., $A \vdash T$ *de*, (ii) the tree T satisfies determinacy race freedom w.r.t. the set F, i.e., $F \vdash T$ *drf*, (iii) the expression e only mentions futures present in set K and locations present in set A, in addition to futures spawned and locations allocated within the subtree T, and (iv) all subtrees of tree T satisfy the judgement *drfde*.

Figure 8.8 shows the rules for the judgement $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ T\ ;\ e$ *drfde*. The rules encode both disentanglement and race freedom by creating the sets A and F in the same way as the definitions of judgements *de* and *drf* respectively.

The rule BASE applies to an empty leaf of the form $\mathsf{Leaf}(\bullet)$. The rule checks that all locations mentioned by expression e are in the set A. The rule uses the function $\mathsf{Fut}(e, \mu)$, which returns all futures mentioned by expression e, and asserts that they are in the set K. The rule also ensures that all locations that are in set A but not in set F, i.e. in set difference $A \setminus F$, point to locations in set A, i.e., the set $A \setminus F$ is closed under the memory pointer relation (highlighted in purple). Additionally, if a location in set A refers to a future, then the future is in set K. These four properties guarantee that no matter what the expression e does in the next step, it will not access a location outside set A or access a future outside set K, assuming the step does not access a location in set F (which is implied by race freedom).

The rule LEAF considers the case when the tree is a leaf, i.e., of the form $\mathsf{Leaf}(t)$. It checks that trace t satisfes the *drf* and *de* properties and defers to the BASE after extending the set A with the allocations in trace t. The rule PAR applies to a tree of the form $\mathsf{Par}(t, T_1, T_2)$. Recall that this tree represents the spawning of future a after actions in trace t, with tree T_1 recording the actions of the future, and tree T_2 recording the actions of the continuation. The rule checks the tree T_2 with expression e_2 after extending the set A to $A \cup \mathsf{A}(t)$, set F to $F \cup \mathsf{AW}(T_1)$ set K to $K \cup \{a\}$. These extensions represent that the tree T_2 and expression e_2 can mention locations allocated by the trace t and also access the future a. For the tree T_1, the rule uses future map Δ to retrieve the future's expression e_1 and checks $K \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta \,;\, T_1 \,;\, e_1$ *drfde*. The rule extends the set A with the allocations in tree t, but unlike for tree T_2, the rule does not extend the set K with future a. By excluding future a from set K, the rule prohibits the future from mentioning and accessing itself.

The rule JOINED shows the conditions for future a after it has joined with its continuation e. The tree in this case is of the form $t \oplus_a T$, where the operator $\oplus_a$ marks the join point of future a. Because the future has joined, its is mapped to a value v in the future map and the rule checks the value similar to rule LEAF.

Proof of Theorem. We can prove the theorem from the following lemma.

Lemma 7. *For any* $\Delta \,;\, \mu \,;\, T \,;\, e \to \Delta' \,;\, \mu' \,;\, T' \,;\, e'$ *if* $K \,;\, A \,;\, F \vdash_\mu \Delta \,;\, T \,;\, e$ *drfde,* $A \,;\, \Delta \vdash_\mu K$ *ok, and* $K \vdash T'$ *drf, then* $K \,;\, A \,;\, F \vdash_{\mu'} \Delta' \,;\, T' \,;\, e'$ *drfde and* $A \,;\, \Delta' \vdash_{\mu'} K$ *ok.*

The lemma states that if the *drfde* and the *ok* properties hold for a sub state $\Delta \,;\, \mu \,;\, T \,;\, e$ and if state takes a step to state $\Delta' \,;\, \mu' \,;\, T' \,;\, e'$, then the *drfde* and the *ok* properties holds for the resulting state assuming its tree T' satisfies race freedom, i.e., $F \vdash T'$ *drf*.

We can use the lemma to show that the full program state always satisfies disentanglement. For the full state, the sets K, A, and F are empty. From the lemma, we have that if a state that satisfies *drfde* takes a step such that the resulting state satisfies the *drf* property then the resulting state also satisfies the *drfde* property. Because the initial state satisfies the property *drfde* and we assume that all states satisfy the *drf* property, we have that all states satisfy the *drfde* property. Since the *drfde* property implies the *de* property, we have that race freedom implies disentanglement.

We prove the lemma by induction on the stepping relation $\Delta \,;\, \mu \,;\, T \,;\, e \to \Delta' \,;\, \mu' \,;\, T' \,;\, e'$. We cover the proof in the next subsection.

8.2.3 Proof of key Lemmas

Lemma 8. *For any* $\Delta\,;\mu\,;T\,;e \to \Delta'\,;\mu'\,;T'\,;e'$ *if* $K\,;A\,;F \vdash_\mu \Delta\,;T\,;e$ *drfde,* $A\,;\Delta \vdash_\mu K$ *ok, and* $A \vdash T'$ *drf, then* $K\,;A\,;F \vdash_{\mu'} \Delta'\,;T'\,;e'$ *drfde and* $A\,;\Delta' \vdash_{\mu'} K$ *ok.*

Proof. We prove the lemma by induction on the stepping relation.

Case FutA. We have $T = \mathsf{Leaf}(t)$ and $e = \mathtt{fut}(e'')$ and $\mu' = \mu$ and $T' = \mathsf{Par}(t, \mathsf{Leaf}(\bullet), \mathsf{Leaf}(\bullet))$ and $\Delta' = \Delta[a \blacktriangleright e'']$ and $e' = \mathtt{fcell}[a]$. Assume $K;A;F \vdash_\mu \Delta;\mathsf{Leaf}(t);e$ *drfde*. From inversion, we have $F \vdash t$ *drf*, $A \vdash t$ *de*, and $K;A \cup \mathsf{A}(T);F \vdash_\mu \Delta;\mathsf{Leaf}(\bullet);e$ *drfde*. The judgement $K\,;A\,;F \vdash_\mu \Delta'\,;\mathsf{Par}(t, \mathsf{Leaf}(\bullet), \mathsf{Leaf}(\bullet))\,;\mathtt{fcell}[a]$ *drfde* derives from the following:

- $F \vdash t$ *drf*, established above
- $A \vdash t$ *de*, established above
- $K \cup \{a\}\,;A \cup \mathsf{A}(T)\,;F \vdash_\mu \Delta'\,;\mathsf{Leaf}(\bullet)\,;\mathtt{fcell}[a]$ *drfde* by,
  - $\mathcal{L}(\mathtt{fcell}[a]) = \emptyset \subseteq A \cup \mathsf{A}(T)$
  - $\forall b \in \mathsf{Fut}(\mathtt{fcell}[a], \mu').\ b \in K \cup \{a\}$, from $\mathsf{Fut}(\mathtt{fcell}[a], \mu') = \{a\}$.
  - $\forall \ell \in (A \cup \mathsf{A}(T) \cup \mathsf{A}(\bullet)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(T) \cup \mathsf{A}(\bullet)$, from $(A \cup \mathsf{A}(T) \cup \mathsf{A}(\bullet)) = A \cup \mathsf{A}(T)$ and $\forall \ell \in (A \cup \mathsf{A}(T)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(T)$. The latter follows from inversion of $K\,;A \cup \mathsf{A}(T)\,;F \vdash_\mu \Delta\,;\mathsf{Leaf}(\bullet)\,;e$ *drfde*.
  - $\forall \ell \in (A \cup \mathsf{A}(T)).\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K$, from inversion of $K\,;A \cup \mathsf{A}(T)\,;F \vdash_\mu \Delta\,;\mathsf{Leaf}(\bullet)\,;e$ *drfde*.
- $K\,;A \cup \mathsf{A}(T)\,;F \vdash_\mu \Delta'\,;\mathsf{Leaf}(\bullet)\,;e''$ *drfde*. The proof is similar to the previous case.

From inversion on $A\,;\Delta \vdash_\mu K$ *ok*, $K \subseteq \mathrm{dom}(\Delta)$. Thus, $a \notin \Delta \Rightarrow a \notin K$. Applying Lemma 10 with $A\,;\Delta \vdash_\mu K$ *ok* and $\forall b \in K.\ \Delta(b) = \Delta'(b)$, we get $A\,;\Delta' \vdash_\mu K$ *ok*. By $\mu = \mu'$, $A\,;\Delta' \vdash_{\mu'} K$ *ok*.

Case Get. We have $T = \mathsf{Leaf}(t)$ and $e = \mathtt{get}(\ell)$ and $\mu' = \mu$, where $\mu(\ell) = \mathtt{fcell}[a]$ and $\Delta' = \Delta$, where $\Delta(a) \rhd v$ and $T' = \mathsf{Leaf}(t \oplus (\mathbf{F}\ell \Rightarrow v))$ and $e' = v$. Assumer $F \vdash T'$ *drf*. From applying inversion(s) [1] on $K\,;A\,;F \vdash_\mu \Delta\,;\mathsf{Leaf}(t)\,;\mathtt{get}(\ell)$ *drfde* we have,

- $F \vdash t$ *drf*
- $A \vdash t$ *de*
- $\mathcal{L}(\mathtt{get}(\ell)) \subseteq A \cup \mathsf{A}(t)$, which means $\ell \in A \cup \mathsf{A}(t)$
- $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$
- $\mathsf{Fut}(e, \mu') \subseteq K$. By definition $\mathsf{Fut}(e, \mu) = \{a\}$, which means $a \in K$.
- $\forall \ell \in (A \cup \mathsf{A}(T)).\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K$

Note that $\Delta = \Delta'$ and $\mu = \mu'$, so $A\,;\Delta' \vdash_{\mu'} K$ *ok* follows directly from $A\,;\Delta \vdash_\mu K$ *ok*. From inversion of $A\,;\Delta \vdash_\mu K$ *ok*, $a \in K$, and $\Delta(a) \rhd v$, we get:

- $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(T)$
- $\mathsf{Fut}(v, \mu) \subseteq K$

[1] Technically, by applying inversion on the judgement and applying it again on the derived judgement

The judgement $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(t \oplus (\mathbf{F}\ell \Rightarrow v))\ ;\ v$ *drfde* derives from the following:

- $F \vdash \mathsf{Leaf}(t \oplus (\mathbf{F}\ell \Rightarrow v))$ *drf*, from the assumption $F \vdash T'$ *drf* and $T' = \mathsf{Leaf}(t \oplus (\mathbf{F}\ell \Rightarrow v))$.
- $A \vdash \mathsf{Leaf}(t \oplus (\mathbf{F}\ell \Rightarrow v))$ *de*. We know $A \vdash t$ *de* from the above derivation, so we only need to show $A \cup \mathsf{A}(t) \vdash (\mathbf{F}\ell \Rightarrow v)$ *de*. From the definition, this follows as $\ell \in A \cup \mathsf{A}(t)$ and $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(T)$.
- $K\,; A \cup \mathsf{A}(t \oplus (\mathbf{F}\ell \Rightarrow v))\,; F \vdash_\mu \Delta\,; \mathsf{Leaf}(\bullet)\,; v$ *drfde*, from $A \cup \mathsf{A}(t \oplus (\mathbf{F}\ell \Rightarrow v)) = A \cup \mathsf{A}(t)$ and $K\ ;\ A \cup \mathsf{A}(t)\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ v$ *drfde* which derives from the following:
  - $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(t \oplus (\mathbf{F}\ell \Rightarrow v))$, from $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(T)$
  - $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, established above
  - $\mathsf{Fut}(v, \mu) \subseteq K$, established above
  - $\forall \ell \in (A \cup \mathsf{A}(T)).\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K$, established above

Case Bang. This step reads a reference's value from memory. The expression e is of the form $!\,\ell$ and it steps to location ℓ', such that $\mu(\ell) = \mathtt{ref}\ \ell'$. The tree T is a leaf of the form $\mathsf{Leaf}(t)$; this step extends it with the read action and creates the tree $T' = \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{ref}\ \ell'))$. The future map and the memory remain unchanged, i.e., $\Delta' = \Delta$ and $\mu' = \mu$. Thus, for the resulting state, the condition $ok{-}A\ ;\ \Delta' \vdash_{\mu'} K$ *ok*, follows directly from $A\ ;\ \Delta \vdash_\mu K$ *ok*.

Because the step is assumed to be race-free, we know $F \vdash T'$ *drf*. Recall that the *drf* property makes sure that no action tree T' mentions a location in F. Because the read action $\mathbf{R}\ell \Rightarrow \mathtt{ref}\ \ell'$ is in tree T', we know $\ell \notin F$. By applying inversion on $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(t)\ ;\ !\,\ell$ *drfde*, we get: $F \vdash t$ *drf*, $A \vdash t$ *de*, and $K\,; A \cup \mathsf{A}(t)\,; F \vdash_\mu \Delta\,; \mathsf{Leaf}(\bullet)\,; !\,\ell$ *drfde*. By applying inversion on $K\,; A \cup \mathsf{A}(t)\,; F \vdash_\mu \Delta\,; \mathsf{Leaf}(\bullet)\,; !\,\ell$ *drfde*, we have: $\ell \in A \cup \mathsf{A}(t)$, $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, $\mathsf{Fut}(e, \mu) \subseteq K$, and $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K$.

The judgement $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ T'\ ;\ e'$ *drfde* follows from applying Rule (8.2) with the following:

- $F \vdash T'$ *drf*, assumed.
- $A \vdash T'$ *de*. Note that T' is a leaf of the form $\mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{ref}\ \ell'))$. We know trace t is disentangled from $A \vdash T$ *de* and $T = \mathsf{Leaf}(t)$. To prove, that the read action $(\mathbf{R}\ell \Rightarrow \mathtt{ref}\ \ell')$ is disentangled, we need to show $\ell \in A$ and $\ell' \in A$ (both established above).
- $K\ ;\ A \cup \mathsf{A}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{ref}\ \ell'))\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ \ell'$ *drfde*. There is no allocation in this step, i.e., $A \cup \mathsf{A}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{ref}\ \ell')) = A \cup \mathsf{A}(t)$. We prove $K\,; A \cup \mathsf{A}(t)\,; F \vdash_\mu \Delta\,; \mathsf{Leaf}(\bullet)\,; \ell'$ *drfde* using Rule (8.1) with the following:
  - $\mathcal{L}(\ell') \subseteq A \cup \mathsf{A}(t)$, established above
  - $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, established above.
  - $\mathsf{Fut}(\ell', \mu) \subseteq K$. If ℓ' refers to a future cell, i.e. $\mu(\ell') = \mathtt{fcell}[a]$ for some a, then we know that $a \in K$ because: $\ell' \in A \cup \mathsf{A}(t)$ and $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K$. Thus, in this case, $\mathsf{Fut}(\ell', \mu) = \{a\} \subseteq K$. Otherwise, $\mathsf{Fut}(\ell', \mu) = \emptyset$, which is trivially a subset of K.
  - $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K$, established above

Case Upd. We have $\mu = \mu_0[\ell \hookrightarrow s]$ and $T = \mathsf{Leaf}(t)$ and $e = \ell := v$ and $\Delta' = \Delta$ and $\mu' = \mu_0[\ell \hookrightarrow \texttt{ref}\ v]$ and $T' = \mathsf{Leaf}(t \oplus (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v))$ and $e' = v$. Assume $F \vdash T'$ *drf*, $K\,;A\,;F \vdash_\mu \Delta\,;$ $\mathsf{Leaf}(t)\,;\ell := v$ *drfde* and $A\,;\Delta \vdash_\mu K$ *ok*. From inversion(s) of $K\,;A\,;F \vdash_\mu \Delta\,;\mathsf{Leaf}(t)\,;\ell := v$ *drfde*, we get the following:

- $F \vdash t$ *drf*
- $A \vdash t$ *de*
- $\mathcal{L}(e) \subseteq A \cup \mathsf{A}(t)$, which means $\ell \in A \cup \mathsf{A}(t)$ and $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(t)$.
- $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$
- $\mathsf{Fut}(e, \mu) \subseteq K$.
- $\forall \ell \in (A \cup \mathsf{A}(T)).\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow a \in K$

The judgement $K\ ;\ A\ ;\ F \vdash_{\mu'} \Delta\ ;\ \mathsf{Leaf}(t \oplus (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v))\ ;\ v$ *drfde* derives from the following:

- $F \vdash \mathsf{Leaf}(t \oplus (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v))$ *drf*, from $F \vdash T'$ *drf*.
- $A \vdash \mathsf{Leaf}(t \oplus (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v))$ *de*. We know $A \vdash t$ *de* from the above derivation, so we only need to show $A \cup \mathsf{A}(t) \vdash (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v)$ *de*. This follows from $\ell \in A \cup \mathsf{A}(t)$ and $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(T)$.
- $K\,;A \cup \mathsf{A}(t \oplus (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v))\,;F \vdash_{\mu'} \Delta\,;\mathsf{Leaf}(\bullet)\,;v$ *drfde*, from $A \cup \mathsf{A}(t \oplus (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v)) = A \cup \mathsf{A}(t)$ and $K\ ;\ A \cup \mathsf{A}(t)\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ v$ *drfde* which derives from the following:
  - $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(t \oplus (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v))$, from $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(T)$ ($\mathcal{L}(e) \subseteq A \cup \mathsf{A}(t)$).
  - $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu'(\ell)) \subseteq A \cup \mathsf{A}(t)$. Because μ and μ' only differ at ℓ, for other locations, this property follows from $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$. The location ℓ is updated to point to v. From $\mu(\ell) = v$ and $\mathcal{L}(v) \subseteq (A \cup \mathsf{A}(t))$, the property holds for ℓ too.
  - $\mathsf{Fut}(v, \mu') \subseteq K$, If v is a primitive value (not a location), then $\mathsf{Fut}(v, \mu') = \emptyset$ and the relation holds trivially. Otherwise, suppose $v = \ell'$ for some ℓ' and consider the following two cases:
    1. $\ell' \neq \ell$, i.e., $\mu'(\ell') = \mu(\ell)$. We have $\mathsf{Fut}(\ell', \mu) \subseteq \mathsf{Fut}(e, \mu) \subseteq K$. From $\mu'(\ell') = \mu(\ell')$, we get $\mathsf{Fut}(\ell', \mu') = \mathsf{Fut}(\ell', \mu) \subseteq K$
    2. $\ell' = \ell$. In this case $\mu'(\ell) = \texttt{ref}\ \ell$ and $\mathsf{Fut}(\ell, \mu') = \emptyset$ by definition.
  - $\forall \ell \in A \cup \mathsf{A}(t).\ \mu'(\ell) = \texttt{fcell}[a] \Rightarrow a \in K$. From Lemma 18 with $\Delta\ ;\ \mu\ ;\ T\ ;\ e \rightarrow \Delta'\ ;\ \mu'\ ;\ T'\ ;\ e'$, we have $\forall \ell \in \mathrm{dom}(\mu).\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow \mu(\ell) = \mu'(\ell)$. This implies the above property using $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow a \in K$ (established above).

The judgement $A\ ;\ \Delta \vdash_{\mu'} K$ *ok* derives from the following:

- $K \subseteq \mathrm{dom}(\Delta)$, from inversion of $A\ ;\ \Delta \vdash_\mu K$ *ok*.
- $\forall a \in K.\ \Delta(a) \blacktriangleright v \Rightarrow \mathcal{L}(v) \subseteq A \wedge \mathsf{Fut}(v, \mu') \subseteq K$. From inversion of $A\ ;\ \Delta \vdash_\mu K$ *ok*, we know $\forall a \in K.\ \Delta(a) \blacktriangleright v \Rightarrow \mathcal{L}(v) \subseteq A \wedge \mathsf{Fut}(v, \mu) \subseteq K$. Applying this to an arbitrary name $a \in K$, such that $\Delta(a) \blacktriangleright v'$, we know $\mathcal{L}(v') \subseteq A$ and $\mathsf{Fut}(v', \mu) \subseteq K$. To prove the property above, we need to show $\mathcal{L}(v') \subseteq A$ (which follows directly) and

$\mathsf{Fut}(v', \mu') \subseteq K$. If v' is a primitive location then $\mathsf{Fut}(v', \mu') = \emptyset$, so $\mathsf{Fut}(v', \mu') \subseteq K$ holds trivially. Otherwise, suppose $v' = \ell'$ for some ℓ' and consider the following two cases:

1. $\ell' \neq \ell$, i.e., $\mu'(\ell') = \mu(\ell)$. From $\mu'(\ell') = \mu(\ell')$, we get $\mathsf{Fut}(\ell', \mu') = \mathsf{Fut}(\ell', \mu) \subseteq K$.
2. $\ell' = \ell$. In this case $\mu'(\ell) = \texttt{ref}\ \ell$ and $\mathsf{Fut}(\ell, \mu') = \emptyset$ by definition.

Case Output.

$$\frac{\mu(\ell) = n}{\Delta\ ;\ \mu\ ;\ \mathsf{Leaf}(t)\ ;\ \mathsf{out\_nat}(\ell) \to \Delta\ ;\ \mu\ ;\ \mathsf{Leaf}(t \oplus \mathbf{R}\ell \Rightarrow n)\ ;\ ()}\ \textsc{Output}$$

We have $T = \mathsf{Leaf}(t)$ and $e = \mathsf{out\_nat}(\ell)$ and $\mu' = \mu$ and $\Delta' = \Delta$ and $T' = \mathsf{Leaf}(t \oplus \mathbf{R}\ell \Rightarrow n)$ and $e' = ()$. Assume $K\,; A\,; F \vdash_\mu \Delta\,; \mathsf{Leaf}(t)\,; \ell$ *drfde* and $A\,; \Delta \vdash_\mu K$ *ok*. We get $A\,; \Delta' \vdash_{\mu'} K$ *ok* directly because $\mu' = \mu$ and $\Delta' = \Delta$. To prove the judgment $K\,; A\,; F \vdash_\mu \Delta\,; \mathsf{Leaf}(t \oplus \mathbf{R}\ell \Rightarrow n)\,; ()$ *drfde*, we observe that from inversion on $K\,; A\,; F \vdash_\mu \Delta\,; \mathsf{Leaf}(t)\,; \ell$ *drfde*, we get $A \vdash t$ *de* and $K; A \cup A(t); F \vdash_\mu \Delta; \mathsf{Leaf}(\bullet); \ell$ *drfde*. And, from inversion on the latter we get $\mathcal{L}(\ell) \subseteq A \cup A(t)$. The judgement can be proved as follows:

- $A \vdash \mathsf{Leaf}(t \oplus \mathbf{R}\ell \Rightarrow n)$ *de*. We have $A \vdash t$ *de* from above and we only need to show $A \cup \mathsf{A}(t) \vdash \mathbf{R}\ell \Rightarrow n$ *de*. This follows from $\ell \in A \cup \mathsf{A}(t)$, which follows from above $\mathcal{L}(\ell) \subseteq A \cup \mathsf{A}(T)$.
- $F \vdash \mathsf{Leaf}(t \oplus \mathbf{R}\ell \Rightarrow n)$ *drf*. This follows from the assumption that the resulting tree satisfies *drf* (the program is *drf*).
- $K\ ;\ A \cup \mathsf{A}(t)\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ ()$ *drfde* by,
  - $\mathcal{L}(()) = \emptyset \subseteq A \cup \mathsf{A}(t)$
  - $\forall \ell \in (A \cup \mathsf{A}(t) \setminus F).\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, from $\forall \ell \in A \cup \mathsf{A}(t) \setminus F. \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, which follows from inversion of $K\ ;\ A \cup A(t)\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ \ell$ *drfde*..
  - $\mathsf{Fut}((), \mu) \subseteq K$, from $\mathsf{Fut}((), \mu) = \emptyset$

Case PollT We have $T = \mathsf{Leaf}(t)$ and $e = \texttt{fpoll}(\ell)$ and $\mu(\ell) = \texttt{fcell}[a]$ and $\mu' = \mu$ and $T' = \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \texttt{fcell}[a]))$ and $e' = \texttt{true}$. Assume $F \vdash T'$ *drf*. From inversion(s) on $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(t)\ ;\ \texttt{fpoll}(\ell)$ *drfde* we have,

- $F \vdash t$ *drf*
- $A \vdash t$ *de*
- $\mathcal{L}(\texttt{fpoll}(\ell)) \subseteq A \cup \mathsf{A}(t)$, which means $\ell \in A \cup \mathsf{A}(t)$
- $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$
- $\forall a \in \mathsf{Fut}(e, \mu').\ a \in K$.
- $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow a \in K$

Note that $\Delta = \Delta'$ and $\mu = \mu'$, so $A\ ;\ \Delta' \vdash_{\mu'} K$ *ok* follows directly from $A\ ;\ \Delta \vdash_\mu K$ *ok*. From $F \vdash \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \texttt{fcell}[a]))$ *drf*, we have $\ell \notin F$. The judgement $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \texttt{fcell}[a]))\ ;\ \texttt{true}$ *drfde* derives from the following:

- $F \vdash \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \texttt{fcell}[a]))$ *drf*, from the assumption $F \vdash T'$ *drf* and $T' = \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \texttt{fcell}[a]))$

- $A \vdash \mathsf{Leaf}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a]))$ *de*. We know $A \vdash t$ *de* from the above derivation, so we only need to show $A \cup \mathsf{A}(t) \vdash (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a])$ *de*. This follows from $\ell \in A \cup \mathsf{A}(t)$ (established above).
- $K \;;\; A \cup \mathsf{A}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a])) \;;\; F \vdash_\mu \Delta \;;\; \mathsf{Leaf}(\bullet) \;;\; \mathtt{true}$ *drfde*, from $A \cup \mathsf{A}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a])) = A \cup \mathsf{A}(t)$ and $K \,;\, A \cup \mathsf{A}(t) \,;\, F \vdash_\mu \Delta \,;\, \mathsf{Leaf}(\bullet) \,;\, \mathtt{true}$ *drfde* which derives from the following:
  - $\mathcal{L}(\mathtt{true}) \subseteq A \cup \mathsf{A}(t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a]))$, from $\mathcal{L}(\mathtt{true}) = \emptyset$
  - $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, established above
  - $\mathsf{Fut}(\mathtt{true}, \mu) \subseteq K$, from $\mathsf{Fut}(\mathtt{true}, \mu') = \emptyset$
  - $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K$, established above

Case PollF Proof is similar to the above case.

Case FutS. We have $e = e_2$ and $T = \mathsf{Par}(t, T_1, T_2)$ and $\Delta = \Delta_s[a \blacktriangleright e_1]$ and $e' = e_2$ and $T' = \mathsf{Par}(t, T_1', T_2)$ and $\Delta' = \Delta_s'[a \blacktriangleright e_1']$, where $\Delta \,;\, \mu \,;\, T_1 \,;\, e_1 \to \Delta' \,;\, \mu' \,;\, T_1' \,;\, e_1'$.

Assume $F \vdash T'$ *drf*, $A \,;\, \Delta \vdash_\mu K$ *ok*, and $K \,;\, A \,;\, F \vdash_\mu \Delta \,;\, T \,;\, e_2$ *drfde*. By inversion of $K \,;\, A \,;\, F \vdash_\mu \Delta \,;\, T \,;\, e_2$ *drfde*, we get the following:

- $F \vdash T$ *drf*, which implies $F \vdash t$ *drf*, $F \cup \mathsf{AW}(T_2) \vdash T_1$ *drf* and $F \cup \mathsf{AW}(T_1) \vdash T_2$ *drf*
- $A \vdash T$ *de*, which implies $A \vdash t$ *de*, $A \cup \mathsf{A}(t) \vdash T_1$ *de* and $A \cup \mathsf{A}(t) \vdash T_2$ *de*
- $a \notin K$
- $\Delta = \Delta_s[a \blacktriangleright e_1]$
- $K \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta \,;\, T_1 \,;\, e_1$ *drfde*
- $K \cup \{a\} \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta \,;\, T_2 \,;\, e_2$ *drfde*

First, apply the inductive hypothesis with:

1. $\Delta \,;\, \mu \,;\, T_1 \,;\, e_1 \to \Delta' \,;\, \mu' \,;\, T_1' \,;\, e_1'$, established above
2. $K \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta \,;\, T_1 \,;\, e_1$ *drfde*, established above
3. $F \cup \mathsf{AW}(T_2) \vdash T_1'$ *drf*, from inversion of $F \vdash T'$ *drf*, where $T' = \mathsf{Par}(g, T_1', T_2)$
4. $A \cup \mathsf{A}(t) \,;\, \Delta \vdash_\mu K$ *ok*, applying Lemma 11 with $A \,;\, \Delta \vdash_\mu K$ *ok* and $A \subseteq A \cup \mathsf{A}(t)$

to get $K \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_2) \vdash_{\mu'} \Delta' \,;\, T_1' \,;\, e_1'$ *drfde* and $A \cup \mathsf{A}(t) \,;\, \Delta' \vdash_{\mu'} K$ *ok*. To establish $K \,;\, A \,;\, F \vdash_{\mu'} \Delta' \,;\, \mathsf{Par}(t, T_1', T_2) \,;\, e_2$ *drfde*, we show the following:

- $F \vdash t$ *drf*, established above
- $A \vdash t$ *de*, established above
- $a \notin K$, established above
- $\Delta'(a) \blacktriangleright e_1'$, by definition
- $K \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_2) \vdash_{\mu'} \Delta' \,;\, T_1' \,;\, e_1'$ *drfde*, established above
- $K \cup \{a\} \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_1') \vdash_{\mu'} \Delta' \,;\, T_2 \,;\, e_2$ *drfde*. The proof has the following steps:
  1. $K \cup \{a\} \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta' \,;\, T_2 \,;\, e_2$ *drfde*. By the uniqueness of future names, $\mathsf{Fut}(T_1) \cap \mathsf{Fut}(T_2) = \emptyset$. Using this with $\forall a \notin \mathsf{Fut}(T_1).\ \Delta(a) = \Delta'(a)$, we get $\forall a \in \mathsf{Fut}(T_2).\ \Delta(a) = \Delta'(a)$. Applying Lemma 12 with $K \cup \{a\} \,;\, A \cup \mathsf{A}(t) \,;$

$F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta \,;\, T_2 \,;\, e_2$ *drfde*, we get $K \cup \{a\} \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta' \,;\, T_2 \,;\, e_2$ *drfde*.

2. $F \cup \mathsf{AW}(T_1') \vdash T_2$ *drf*, from inversion of $F \vdash \mathsf{Par}(g, T_1', T_2)$ *drf*.
3. $\mathsf{AW}(T_1) \subseteq \mathsf{AW}(T_1')$ and $\forall \ell \in \mathrm{dom}(\mu) \setminus (F \cup \mathsf{AW}(T_1')).\ \mu(\ell) = \mu'(\ell)$. Applying Lemma 14 with $\Delta \,;\, \mu \,;\, T_1 \,;\, e_1 \to \Delta' \,;\, \mu' \,;\, T_1' \,;\, e_1'$, we have $\mathsf{AW}(T_1) \subseteq \mathsf{AW}(T_1')$ and $\forall \ell \in \mathrm{dom}(\mu) \setminus (\mathsf{AW}(T_1') \setminus \mathsf{AW}(T_1)).\ \mu(\ell) = \mu'(\ell)$. From $(\mathsf{AW}(T_1') \setminus \mathsf{AW}(T_1)) \subseteq \mathsf{AW}(T_1') \subseteq (F \cup \mathsf{AW}(T_1'))$ with $\forall \ell \in \mathrm{dom}(\mu) \setminus (\mathsf{AW}(T_1') \setminus \mathsf{AW}(T_1)).\ \mu(\ell) = \mu'(\ell)$, we have $\forall \ell \in \mathrm{dom}(\mu) \setminus (F \cup \mathsf{AW}(T_1')).\ \mu(\ell) = \mu'(\ell)$.
4. $K \cup \{a\} \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_1') \vdash_{\mu'} \Delta' \,;\, T_2 \,;\, e_2$ *drfde*. From applying Lemma 13 with $K \cup \{a\} \,;\, A \cup \mathsf{A}(t) \,;\, F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta' \,;\, T_2 \,;\, e_2$ *drfde*, $F \cup \mathsf{AW}(T_1) \subseteq F \cup \mathsf{AW}(T_1')$, $F \cup \mathsf{AW}(T_1') \vdash T_2$ *drf*, and $\forall \ell \in \mathrm{dom}(\mu) \setminus (F \cup \mathsf{AW}(T_1')).\ \mu(\ell) = \mu'(\ell)$

The judgement $A \,;\, \Delta' \vdash_{\mu'} K$ *ok* follows from the following:

1. $K \subseteq \mathrm{dom}(\Delta')$, from inversion of $A \,;\, \Delta \vdash_\mu K$ *ok* and $\Delta \subseteq \Delta'$.
2. $\forall a \in K.\ \Delta'(a) \blacktriangleright v \Rightarrow \mathcal{L}(v) \subseteq A \wedge \mathsf{Fut}(v, \mu') \subseteq K$. From inversion of $A \cup \mathsf{A}(t) \,;\, \Delta' \vdash_{\mu'} K$ *ok*, we get $\forall a \in K.\ \Delta(a) \blacktriangleright v \Rightarrow \mathsf{Fut}(v, \mu') \subseteq K$. From $K \,;\, A \,;\, F \vdash_{\mu'} \Delta' \,;\, \mathsf{Par}(t, T_1', T_2) \,;\, e_2$ *drfde*, we get $K \cap \mathsf{Fut}(T_1) = \emptyset$. From $\Delta \,;\, \mu \,;\, T_1 \,;\, e_1 \to \Delta' \,;\, \mu' \,;\, T_1' \,;\, e_1'$, we have $\forall a \notin \mathsf{Fut}(T_1).\ \Delta(a) = \Delta'(a)$ and thus, $\forall a \in K.\ \Delta(a) = \Delta'(a)$. From inversion of $A \,;\, \Delta \vdash_\mu K$ *ok*, we get $\forall a \in K.\ \Delta(a) \blacktriangleright v \Rightarrow \mathcal{L}(v) \subseteq A$ and from $\forall a \in K.\ \Delta(a) = \Delta'(a)$, we get $\forall a \in K.\ \Delta'(a) \blacktriangleright v \Rightarrow \mathcal{L}(v) \subseteq A$

Case ContS. Proof is similar to the previous case.

Case FJoin. We have $T = \mathsf{Par}(t, T_1, T_2)$ and $e = e_2$ and $\Delta = \Delta_s[a \blacktriangleright v]$, and $\mu' = \mu$ and $T' = t \oplus G''$, where $\bowtie (a, T_1, T_2) = G''$, and $e' = e_2$ and $\Delta' = \Delta_s[a \rhd v]$.

From inversion on $K \,;\, A \,;\, F \vdash_\mu \Delta \,;\, \mathsf{Par}(t, T_1, T_2) \,;\, e_2$ *drfde*, we get:

- $A \vdash t$ *de*
- $F \vdash t$ *drf*
- $a \notin K$
- $\Delta(a) \blacktriangleright v$
- $K \,;\, A \,;\, F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta \,;\, T_1 \,;\, v$ *drfde*
- $K \cup \{a\} \,;\, A \,;\, F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta \,;\, T_2 \,;\, e_2$ *drfde*

The judgement $K \,;\, A \,;\, F \vdash_\mu \Delta' \,;\, t \oplus \bowtie (a, T_1, T_2) \,;\, e_2$ *drfde* derives from $A \vdash t$ *de*, $F \vdash t$ *drf*, and $K \,;\, A \cup \mathsf{A}(T) \,;\, F \vdash_\mu \Delta' \,;\, \bowtie (a, T_1, T_2) \,;\, e_2$ *drfde*, which follows from applying Lemma 9 with:

1. $a \notin K \cup \mathsf{Fut}(T_1) \cup \mathsf{Fut}(T_2)$, from uniqueness of names in T.
2. $K \cup \{a\} \,;\, A \cup \mathsf{A}(T) \,;\, F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta' \,;\, T_2 \,;\, e_2$ *drfde*, by Lemma 12 with $\forall b \in \mathsf{Fut}(T_2).\ \Delta(b) = \Delta'(b)$ and $K \cup \{a\} \,;\, A \,;\, F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta \,;\, T_2 \,;\, e_2$ *drfde*
3. $K \,;\, A \cup \mathsf{A}(T) \,;\, F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta' \,;\, T_1 \,;\, v$ *drfde*, by Lemma 12 with $\forall b \in \mathsf{Fut}(T_1).\ \Delta(b) = \Delta'(b)$ and $K \,;\, A \,;\, F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta \,;\, T_1 \,;\, v$ *drfde*.
4. $\mathsf{Fut}(T_1) \cap \mathsf{Fut}(T_2) = \emptyset$, from uniqueness of names in T.

5. $\Delta'(a) \rhd v$, by definition.

Lemma 9. *If*

1. $a \notin K \cup \mathsf{Fut}(T_1) \cup \mathsf{Fut}(T_2)$
2. $K \cup \{a\}\ ;\ A\ ;\ F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta\ ;\ T_2\ ;\ e_2$ *drfde*
3. $K\ ;\ A\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ T_1\ ;\ v$ *drfde*
4. $\mathsf{Fut}(T_1) \cap \mathsf{Fut}(T_2) = \emptyset$
5. $\Delta(a) \rhd v$

, then $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\bowtie (a, T_1, T_2)\ ;\ e_2$ *drfde*

Proof. We prove $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\bowtie (a, T_1, T_2)\ ;\ e_2$ *drfde* by induction on T_1:

1. Case $T_1 = \mathsf{Leaf}(g_1)$. In this case $\bowtie (a, T_1, T_2) = g_1 \oplus_a T_2$. The judgement $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ g_1 \oplus_a T_2\ ;\ e_2$ *drfde* derives from the following;
   - $F \vdash g_1$ *drf*. From inversion of $K\ ;\ A\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ T_1\ ;\ v$ *drfde*, we get $F \cup \mathsf{AW}(T_2) \vdash g_1$ *drf*. Using this and applying Lemma 16 with $F \subseteq F \cup \mathsf{AW}(T_2)$, we have $F \vdash g_1$ *drf*.
   - $A \vdash t_1$ *de*, from inversion of $K\ ;\ A\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ g_1\ ;\ v$ *drfde*
   - $a \notin K$, from premise (1) of the lemma.
   - $\Delta(a) \rhd v$, by definition
   - $K\ ;\ A \cup \mathsf{A}(T_1)\ ;\ F \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ v$ *drfde*. This follows from
     (a) $\mathcal{L}(v) \subseteq A \cup \mathsf{A}(T_1)$, from applying inversion twice on $K\ ;\ A\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(g_1)\ ;\ v$ *drfde*.
     (b) $\mathsf{Fut}(v, \mu) \subseteq K$, from applying inversion twice on $K\ ;\ A\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(g_1)\ ;\ v$ *drfde*.
     (c) $\forall \ell \in (A \cup \mathsf{A}(t_1)) \setminus (A \cup \mathsf{A}(t_1))$. $\mathcal{L}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, holds vacuously.
     (d) $\forall \ell \in A \cup \mathsf{A}(t_1)$. $\mu(\ell) = \texttt{fcell}[a] \Rightarrow a \in K$, from applying inversion twice on $K\ ;\ A\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(g_1)\ ;\ v$ *drfde*.
   - $K \cup \{a\}\ ;\ A \cup \mathsf{A}(t_1)\ ;\ F \vdash_\mu \Delta\ ;\ T_2\ ;\ e_2$ *drfde*, applying Lemma 15 with $K \cup \{a\}\ ;\ A\ ;\ F \cup \mathsf{AW}(\mathsf{Leaf}(g_1)) \vdash_\mu \Delta; T_2; e_2$ *drfde* and $K; A; F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta; \mathsf{Leaf}(g_1); v$ *drfde*.
2. Case $T_1 = g_1 \oplus_b T_1'$. In this case $\bowtie (a, T_1, T_2) = g_1 \oplus_b \bowtie (a, T_1', T_2)$. From inversion of $K\ ;\ A\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ g_1 \oplus_b T_1'\ ;\ v$ *drfde*, we get:
   - $F \cup \mathsf{AW}(T_2) \vdash g_1$ *drf*
   - $A \vdash g_1$ *de*
   - $b \notin K$
   - $\Delta(b) \rhd v'$
   - $K\ ;\ A \cup \mathsf{A}(T_1)\ ;\ A \cup \mathsf{A}(T_1) \vdash_\mu \Delta\ ;\ \mathsf{Leaf}(\bullet)\ ;\ v'$ *drfde*
   - $K \cup \{b\}\ ;\ A \cup \mathsf{A}(T_1)\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ T_1'\ ;\ v$ *drfde*

   By uniqueness of names in the graph T_1, we get $b \notin \mathsf{Fut}(T_1')$ and from $\mathsf{Fut}(T_1) \cap \mathsf{Fut}(T_2) = \emptyset$, we get $b \notin \mathsf{Fut}(T_2)$. The judgement $K; A; F \vdash_\mu \Delta; g_1 \oplus_b \bowtie (a, T_1', T_2); e_2$ *drfde* derives as follows:

- $F \vdash g_1$ *drf*. Using $F \cup \mathsf{AW}(T_2) \vdash g_1$ *drf* and applying Lemma 16 with $F \subseteq F \cup \mathsf{AW}(T_2)$, we have $F \vdash g_1$ *drf*.
- $A \vdash t_1$ *de*, established above
- $b \notin K$, established above
- $\Delta(b) \rhd v'$, established above
- $K \; ; \; A \cup \mathsf{A}(T_1) \; ; \; A \cup \mathsf{A}(T_1) \vdash_\mu \Delta \; ; \; \mathsf{Leaf}(\bullet) \; ; \; v'$ *drfde*, established above
- $K \cup \{b\} \; ; \; A \cup \mathsf{A}(T_1) \; ; \; F \vdash_\mu \Delta \; ; \bowtie (a, T_1', T_2) \; ; \; e_2$ *drfde*, by applying the inductive hypothesis with:
  (a) $a \notin K \cup \{b\} \cup \mathsf{Fut}(T_1') \cup \mathsf{Fut}(T_2)$, from $a \notin K \cup \mathsf{Fut}(T_1) \cup \mathsf{Fut}(T_2)$ and $b \in \mathsf{Fut}(T_1)$ and $\mathsf{Fut}(T_1') \subseteq \mathsf{Fut}(T_1)$.
  (b) $K \cup \{b\} \cup \{a\} \; ; \; A \cup \mathsf{A}(t_1) \; ; \; F \cup \mathsf{AW}(T_1') \vdash_\mu \Delta \; ; \; T_2 \; ; \; e_2$ *drfde*. By Lemma 17 with $b \notin \mathsf{Fut}(T_2)$ and $K \cup \{a\} \; ; \; A \; ; \; F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta \; ; \; T_2 \; ; \; e_2$ *drfde*, we get $K \cup \{a\} \cup \{b\} \; ; \; A \; ; \; F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta \; ; \; T_2 \; ; \; e_2$ *drfde*. Using this and $K \; ; \; A \; ; \; F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta \; ; \; T_1 \; ; \; v$ *drfde* with Lemma 15, we get $K \cup \{b\} \cup \{a\} \; ; \; A \cup \mathsf{A}(t_1) \; ; \; F \cup \mathsf{AW}(T_1') \vdash_\mu \Delta \; ; \; T_2 \; ; \; e_2$ *drfde*.
  (c) $K \cup \{b\} \; ; \; A \cup \mathsf{A}(t_1) \; ; \; A \cup \mathsf{A}(t_1) \vdash_\mu \Delta \; ; \; T_1' \; ; \; v$ *drfde*, established above.
  (d) $\mathsf{Fut}(T_1') \cap \mathsf{Fut}(T_2) = \emptyset$, from $\mathsf{Fut}(T_1) \cap \mathsf{Fut}(T_2) = \emptyset$
  (e) $\Delta(a) \rhd v$, by definition

3. Case $T_1 = g \oplus (T_1' \otimes_b T_1'')$. In this case $\bowtie (a, T_1, T_2) = g_1 \oplus (T_1' \otimes_b \bowtie (a, T_1'', T_2))$. From inversion of $K \; ; \; A \; ; \; F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta \; ; \; g_1 \oplus (T_1' \otimes_b T_1'') \; ; \; v$ *drfde*, we get:
   - $F \cup \mathsf{AW}(T_2) \vdash g_1$ *drf*
   - $A \vdash g_1$ *de*
   - $b \notin K$
   - $\Delta(b) \blacktriangleright e_1'$
   - $K \; ; \; A \cup \mathsf{A}(T_1) \; ; \; F \cup \mathsf{AW}(T_2) \cup \mathsf{AW}(T_1'') \vdash_\mu \Delta \; ; \; T_1' \; ; \; e_1'$ *drfde*
   - $K \cup \{b\} \; ; \; A \cup \mathsf{A}(T_1) \; ; \; F \cup \mathsf{AW}(T_2) \cup \mathsf{AW}(T_1') \vdash_\mu \Delta \; ; \; T_1'' \; ; \; v$ *drfde*

   By the uniqueness of names in the graph T_1, we get $b \notin \mathsf{Fut}(T_1') \cup \mathsf{Fut}(T_1'')$ and from $\mathsf{Fut}(T_1) \cap \mathsf{Fut}(T_2) = \emptyset$, we get $b \notin \mathsf{Fut}(T_2)$. The judgement $K \; ; \; A \; ; \; F \vdash_\mu \Delta \; ; \; g_1 \oplus (T_1' \otimes_b \bowtie (a, T_1'', T_2)) \; ; \; e_2$ *drfde* derives as follows:
   - $F \vdash g_1$ *drf*. Using $F \cup \mathsf{AW}(T_2) \vdash g_1$ *drf* and applying Lemma 16 with $F \subseteq F \cup \mathsf{AW}(T_2)$, we have $F \vdash g_1$ *drf*.
   - $A \vdash g_1$ *de*, established above
   - $b \notin K$, established above
   - $\Delta(b) \blacktriangleright e_1'$, established above
   - $K \; ; \; A \cup \mathsf{A}(T_1) \; ; \; F \cup \mathsf{AW}(\bowtie (a, T_1'', T_2)) \vdash_\mu \Delta \; ; \; T_1' \; ; \; e_1'$ *drfde*, from $K \; ; \; A \cup \mathsf{A}(T_1) \; ; \; F \cup \mathsf{AW}(T_2) \cup \mathsf{AW}(T_1'') \vdash_\mu \Delta \; ; \; T_1' \; ; \; e_1'$ *drfde* and $\mathsf{AW}(T_2) \cup \mathsf{AW}(T_1'') = \mathsf{AW}(\bowtie (a, T_1'', T_2))$.
   - $K \cup \{b\} \; ; \; A \cup \mathsf{A}(T_1) \; ; \; F \cup \mathsf{AW}(T_1') \vdash_\mu \Delta \; ; \; \bowtie (a, T_1'', T_2) \; ; \; e_2$ *drfde*, applying the inductive hypothesis with:
     (a) $a \notin K \cup \{b\} \cup \mathsf{Fut}(T_1'') \cup \mathsf{Fut}(T_2)$

(b) $K \cup \{b\} \cup \{a\}\ ;\ A \cup \mathsf{A}(T_1)\ ;\ F \cup \mathsf{AW}(T_1') \cup \mathsf{AW}(T_1'') \vdash_\mu \Delta\ ;\ T_2\ ;\ e_2$ *drfde*. By Lemma 17 with $b \notin \mathsf{Fut}(T_2)$ and $K \cup \{a\}\ ;\ A\ ;\ F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta\ ;\ T_2\ ;\ e_2$ *drfde*, we get $K \cup \{a\} \cup \{b\}\ ;\ A\ ;\ F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta\ ;\ T_2\ ;\ e_2$ *drfde*. Using this and $K\ ;\ A\ ;\ F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ T_1\ ;\ v$ *drfde* with Lemma 15, we get $K \cup \{b\} \cup \{a\}\ ;\ A \cup \mathsf{A}(t_1)\ ;\ F \cup \mathsf{AW}(T_1') \vdash_\mu \Delta\ ;\ T_2\ ;\ e_2$ *drfde*.

(c) $K \cup \{b\}\ ;\ A \cup \mathsf{A}(T_1)\ ;\ F \cup \mathsf{AW}(T_1') \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\ ;\ T_1''\ ;\ v$ *drfde*, established above.

(d) $\mathsf{Fut}(T_1') \cap \mathsf{Fut}(T_2) = \emptyset$, from $\mathsf{Fut}(T_1) \cap \mathsf{Fut}(T_2) = \emptyset$

(e) $\Delta(a) \rhd v$, by definition

□

□

8.2.4 Helper Lemmas

Lemma 10. *If* $K\ ;\ A \vdash_{\Delta,\mu} T\ ;\ e$ *defut and* $\forall a \in \mathsf{Fut}(T).\ \Delta(a) = \Delta'(a)$, *then* $K\ ;\ A \vdash_{\Delta',\mu} T\ ;\ e$ *defut, Similarly, if* $A\ ;\ \Delta \vdash_\mu K$ *ok and* $\forall a \in K.\ \Delta(a) = \Delta'(a)$, *then* $A\ ;\ \Delta' \vdash_\mu K$ *ok.*

Proof. Proof by induction on judgements $K\ ;\ A \vdash_{\Delta,\mu} T\ ;\ e$ *defut* and $A ; \Delta \vdash_\mu K$ *ok* respectively. □

Lemma 11. *If* $A\ ;\ \Delta \vdash_\mu K$ *ok and* $A \subseteq A'$, *then* $A'\ ;\ \Delta \vdash_\mu K$ *ok.*

Proof. Proof by induction on judgement $A\ ;\ \Delta \vdash_\mu K$ *ok.* □

Lemma 12. *If* $K\ ;\ A\ ;\ F \vdash_\mu \Delta\ ;\ T\ ;\ e$ *drfde and* $\forall a \in \mathsf{Fut}(T).\ \Delta(a) = \Delta'(a)$, *then* $K\ ;\ A\ ;\ F \vdash_\mu \Delta'\ ;\ T\ ;\ e$ *drfde,*

Proof. Proof by induction on judgement $K\ ;\ A\ ;\ F \vdash_\mu \Delta'\ ;\ T\ ;\ e$ *drfde.* □

Lemma 13. *If* $K ; A ; F \vdash_\mu \Delta ; T ; e$ *drfde,* $F \subseteq F'$, $F' \vdash G$ *drf, and* $\forall \ell \in \mathit{dom}(\mu) \setminus F'.\ \mu(\ell) = \mu(\ell')$, *then* $K\ ;\ A\ ;\ F' \vdash_{\mu'} \Delta\ ;\ T\ ;\ e$ *drfde*

Proof. Proof by induction on judgement $K\ ;\ A\ ;\ F' \vdash_{\mu'} \Delta\ ;\ T\ ;\ e$ *drfde.* □

Lemma 14. *If* $\Delta\ ;\ \mu\ ;\ T\ ;\ e \to \Delta'\ ;\ \mu'\ ;\ T'\ ;\ e'$, *then* $\mathsf{AW}(T) \subseteq \mathsf{AW}(T')$, $\mathit{dom}(\mu) \subseteq \mathit{dom}(\mu')$, $\Delta \subseteq \Delta'$ *and* $\forall \ell \in \mathit{dom}(\mu) \setminus (\mathsf{AW}(T') \setminus \mathsf{AW}(T)).\ \mu(\ell) = \mu'(\ell)$.

Proof. Proof by induction on stepping relation $\Delta\ ;\ \mu\ ;\ T\ ;\ e \to \Delta'\ ;\ \mu'\ ;\ T'\ ;\ e'$. □

Lemma 15. *If* $K ; A ; F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta ; T_2 ; e_2$ *drfde and* $K ; A ; F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta ; T_1 ; e_1$ *drfde, then* $K\ ;\ A \cup \mathsf{A}(\mathsf{hd}(T_1))\ ;\ F \cup \mathsf{AW}(\mathsf{tl}(T_1)) \vdash_\mu \Delta\ ;\ T_2\ ;\ e_2$ *drfde.*

Proof. Proof by induction on tree T_1. □

Lemma 16. *If* $F \vdash T$ *drf and* $F' \subseteq F$, *then* $F' \vdash T$ *drf.*

Proof. Proof by induction on $F \vdash T$ *drf*. □

Lemma 17. *If* $K\,;A\,;F \vdash_\mu \Delta\,;T\,;e$ *drfde and* $a \notin \mathsf{Fut}(T)$*, then* $K \cup \{a\}\,;A\,;F \vdash_\mu \Delta\,;T\,;e$ *drfde.*

Proof. Proof by induction on $K\ ;A\ ;F \vdash_\mu \Delta\ ;T\ ;e$ *drfde*. □

Lemma 18. *If* $\Delta\,;\mu\,;T\,;e \to \Delta'\,;\mu'\,;T'\,;e'$ *then* $\forall \ell \in \mathit{dom}(\mu).\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow \mu(\ell) = \mu'(\ell)$.

Proof. Proof by induction on $\Delta\ ;\mu\ ;T\ ;e \to \Delta'\ ;\mu'\ ;T'\ ;e'$ □

8.3 Applications of Futures with Disentanglement

Because it combines futures, state, and I/O, our calculus enables us to express a broad range of applications in a disentangled fashion. In this section, we present four examples to illustrate this. First, in Section 8.3.1, we use our language to express a parallel tree merge algorithm and show that futures express pipelining in a succinctly and efficiently. In Section 8.3.3, we present a PDF viewer, demonstrating the language's ability to handle asynchrony by utilizing futures and state to proactively compute and cache results, improving responsiveness. Then, we consider a web server involving interaction (Section 8.3.2), and lastly we express a dynamic programming algorithm leveraging data-dependent parallelism (Section 8.3.4).

All of these programs satisfy disentanglement, and thus the disentanglement hypothesis, because they do not have any entangled objects. This was surprising to us—we did not expect that the complex dependency structure of parallel programs with futures, which allows asynchronous and data-dependent dependencies between threads/tasks, could satisfy the disentanglement hypothesis. The ability to express such programs in a disentangled fashion suggests that our memory management techniques could be generalized to a wider range of parallel programs, including those presented in this section.

8.3.1 Pipelining

Pipelining is a fundamental technique in the design of parallel algorithms that can meaningfully reduce the parallel depth (span). For example, pipelining was used by Paul et al. to improve parallel operations on balanced trees [135] and by Cole to give a $O(\lg n)$ span parallel mergesort algorithm [64]. Implementing pipelined algorithms, however, is quite challenging, because the programmer has to carefully manage the rather complex, producer-consumer-like data dependencies between parallel computations. Blelloch and Reid-Miller [36] showed that pipelined algorithms can be expressed at quite a high level by using functional programming extended with futures.

Figure 8.9 shows the code for a pipelined tree merge from Blelloch and Reid-Miller [36], adapted to our language [2]. The `tree` datatype is a binary search tree whose branches are of

[2]The example in Blelloch and Reid-Miller [36] uses a non-strict semantics for forcing futures whereas our futures are strict.

```
type α tree =
  Empty
| Node of α * (α tree) fut * (α tree) fut

merge :: α tree → α tree → α tree
fun  merge t1 t2 =
  case (t1, t2) of
    (Empty, _) → t2
  | (_, Empty) → t1
  | (Node (v, l, g), _) →
      let s = split v t2
          ll = fut merge (get l) (#1 s)
          gg = fut merge (get g) (#2 s)
      in
        Node (v, ll, gg)
```

```
split :: α → α tree →
  ((α tree) fut * (α tree) fut)
fun split k t =
  case t of
    Empty →
    (fut Empty, fut Empty)
  | Node (v, l, g) →
    if k < v then
      let s = fut split k (get l)
      in
        (fut #1 s,
          fut Node (v, #2 s, g))
    else
      let s = fut split k (get g)
      in
        (fut Node (v, l, #1 s),
          fut #2 s)
```

Figure 8.9: Pipelined merge with futures. We define the function `#1 x = get (fst x)` and `#2 x = get (snd x)`, where `fst` and `snd` project out the first and the second component of a pair.

future type. The `merge` function returns the non-empty tree if one of the trees is empty. In the case where both trees are non-empty, the function splits the second tree by using the key at the root of the first tree and recursively merges the two "halves" from the two trees. These recursive merges run inside futures, allowing them to proceed in parallel. Because the function `split` also returns the recursive portion of its result inside a future, the recursive calls to `merge` can run in a pipelined fashion with the `split`. This is possible because each node of the tree is wrapped inside future and is demanded by the `get` expression as needed.

Given two balanced trees of depth $O(\lg n)$, the parallel `merge` runs in $O(\lg n)$ span or parallel time. With fork-join parallelism, however, the best parallel merge runs in $O(\lg^2(n))$ span.

8.3.2 Web Server

Our web server listens for clients on its socket and spawns futures to handle their requests. Each future services exactly one client and as it does so, the future tracks the relevant information from their requests and aggregates it into a log object. Our current example simplifies the log to only include the name of the client and request count, but in practice, the log could contain many different statistics that we omit. When a client terminates the connection, the corresponding future produces a log of its interaction with the client and completes its evaluation.

The server synchronizes with the futures to obtain their logs. However, in order to respond

```
type socket
type log = {name : string, requests: int}
start_socket : unit → socket
listen : socket → unit
accept : socket → socket option
process : socket → log

fun server () =
let
  server_sock = start_socket ()
  val _ = listen(server_sock)
  fun handle_clients (clients : log fut set) =
    let
      completed = filter (fn c ⇒ fpoll c) clients
      logs = map (completed, fn c ⇒ get c)
    in
      (logs, Set.diff (clients, completed))
  fun loop (clients  : log fut set) =
    let
      (logs, remaining_clients) = handle_clients (clients)
      (* Process logs as desired *)
    in
      case accept (server_sock) of
        NONE ⇒ loop (remaining_clients) (* No new clients *)
      | SOME client_sock ⇒
        let val f = fut (process (client_sock))
        in loop (Set.add (remaining_clients, f))
in
  loop (Set.empty ())

type request
recv : socket → request option
service : request → unit
name : socket → string

fun process (c) =
  let fun loop (req, cnt) =
    case req of
      NONE ⇒ {name = name (c), count = cnt}
    | SOME req ⇒
      service (req); loop (recv (c), cnt + 1)
  in
    loop (recv (c), 0)
```

Figure 8.10: A server with pollable futures and disentanglement

to incoming clients efficiently, the server never blocks or waits on a future. The server achieves this by regularly polling the futures it spawned, filtering out the ones that have terminated, and synchronizing only with terminated futures. By only synchronizing with the terminated futures, the server ensures that it gets the logs immediately without creating any interruption.

Figure 8.10 shows the code for the server. The function `server` initializes a socket and proceeds to listen for incoming clients by calling the function `listen`. Subsequently, the server calls the function `loop`. The function `loop` accepts new clients, spawns futures to handle their requests, and collects and processes logs. The function `loop` maintains the spawned futures in a set named `clients`. Each time it accepts a client, the loop spawns a future to (concurrently) execute the function `process`, which services the requests of the client and returns the log.

Before spawning a future for a new client, the loop checks on the other clients by calling the function `handle_clients`. The function `handle_clients` takes the set of futures, filters those that have finished servicing their clients, and aggregates their logs. To filter out completed futures, the function uses the non-blocking primitive `fpoll`, which returns `true` for terminated futures (see line 14). Subsequently, the function uses the primitive `get` to obtain the logs. Note that each use of `get` returns immediately because the function only calls them on terminated futures. After aggregating the logs, the function removes the terminated futures from the set of futures, and returns the logs and the running futures back to the loop.

The function `loop` processes the logs it receives. We leave the log processing abstract as it varies with the use case. Then, it proceeds to process a new client, adds it to the set of clients, and repeats.

Overall the application uses the ability to store futures in a set and to poll them for managing clients without ever blocking. The application does not use any mutable effects and thus is free from determinacy races by construction. From the result that determinacy-race-free programs are disentangled (Theorem 7), we get that application satisfies disentanglement. This is interesting because the server is interactive and involves communication between threads, and performs I/O, but it remains disentangled. The key point is that the server thread obtains a log only after the corresponding future has terminated, which, as we have shown, does not violate disentanglement.

8.3.3 PDF viewer with disentanglement

We consider a PDF viewer that accepts a page number from the user and displays the corresponding page of the PDF on the screen. Before displaying a requested page, the viewer must first render the page, i.e., it must interpret the bytes in the PDF and generate a visual representation, like a pixel array, which it can then display. Our viewer renders pages in an optimized manner, as it not only renders the pages as requested by the user, but also anticipates the user's navigation actions and renders adjacent pages in the background. It spawns futures to perform this proactive rendering and stores the futures in an array indexed by the appropriate page numbers. In this way, futures and array implement a proactive memoization/caching mechanism, allowing the viewer to display rendered pages efficiently and without delay.

Figure 8.11 shows the code for the viewer. The function `viewer` allocates the "page array", a

```
type pdf = {raw_data : bytes array,
  num_pages : int,
  page_offsets : int → int}
val render : pdf * int → page
val display : page → unit
val getClick : unit → int

fun viewer (p : pdf) =
let
  page_arr = tabulate (#num_pages p) NONE
  fun loop () =
    let pnum = getClick () in
      case page_arr[pnum] of
        NONE ⇒
          pg = render (p, pnum);
          display (pg);
          page_arr[pnum] := SOME (fut (pg));
          fill_page_arr (pnum - δ, pnum + δ) page_arr p
        SOME f ⇒
          display (get f)
in
  loop ()

fun fill_page_arr (l, r) page_arr p =
  let
    fun fill i =
      case page_arr[i] of
        NONE ⇒ page_arr[i] :=
          fut (render (p, i))
      | SOME _ ⇒ ()
  in
    foreach (l, r) (fn i ⇒ fill (i))
```

Figure 8.11: PDF viewer with disentanglement

future option array indexed by the page numbers of the PDF. Initially all elements of the array are `NONE`. The function then runs the `loop` function, which repeatedly awaits for the user to request a page number by calling the function `getClick`. To complete the user request, the function `loop` looks up the page number in the array and if it finds `NONE`, the loop prepares the page by calling function `render`, displays the page to the user, and writes it to the page array at this page number. After completing the user request, the function `loop` proactively renders δ number of pages near the current page by calling the function `fill_page_arr`, which spawns futures to render the given range of pages and writes the corresponding futures in the page array. Note that function `fill_page_arr` takes constant time because it returns immediately after spawning futures that render pages in the background. This allows the loop to be ready promptly for the next user request.

The code uses the stateful array to track futures and also to memoize the already prepared pages. Note that the array is accessed exclusively by a single thread, the thread that runs the functions `viewer`, `loop`, and `fill_page_arr`. The futures spawned by the thread never read or write to the array. Thus, the code does not exhibit determinacy races because all the writes in the code are visible only to the writer itself. Thus, from Theorem 7, it satisfies disentanglement.

8.3.4 Futures and references for dynamic programming

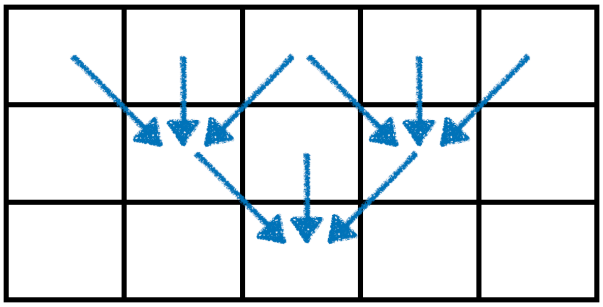

Figure 8.12: An illustration of data-dependent parallelism in a DP matrix: two paths can proceed in parallel regardless of all the other elements in their respective rows.

We consider a dynamic programming algorithm that tabulates an $M \times N$ matrix by computing a cell's value from the value of its neighboring cells in the row above the cell. Specifically, it computes the value at a cell (i, j) by applying an abstract function `f` to the values at cells $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$. The algorithm exemplifies a common pattern and has various applications, such as seam carving and sequence alignment [27, 153]. In seam carving, for instance, the function `f` takes the minimum of the neighbor's values and adds a constant factor to compute the value of a cell.

To tabulate the matrix, we could implement an algorithm that proceeds in a row-by-row manner and fills each row in parallel (because cells of a row do not depend on each other). This algorithm, however, does not exploit parallelism across the rows. In particular, once three

```
1  fun f: α * α * α → α
2  val id : α
3  val arr = array2D (M, N, NONE)
4
5  fun lookup (i, j) =
6      if i < 0 || j < 0 || j >= M then id (* Out-of-bounds reads return id *)
7      else
8        case arr[i][j] of
9        | SOME r → get r
10       | NONE → raise Impossible
11
12 fun compute_cell (i, j) =
13    f (lookup (i - 1, j - 1), lookup (i - 1, j), lookup (i - 1, j + 1))
14
15 (*Initialize the array row-by-row in parallel*)
16 val _ = seq_fill M
17   (fun i → par_fill N
18     (fun j → arr[i][j] <- SOME (future (compute_cell (i, j)))))
19 val result = lookup (M - 1, N - 1)
```

Figure 8.13: Dynamic programming with futures, state, and disentanglement

consecutive cells of row i are computed, the middle cell in the next row $i + 1$ can be computed without waiting for the rest of the cells of row i. Because such "vertical" parallelism depends on the data flow, it is impossible to express with fork-join parallelism, but is naturally expressible by using a combination of futures and state. Figure 8.12 illustrates the flow of data and parallelism present in this application.

Figure 8.13 shows an algorithm where futures unleash the data dependent parallelism across rows. The algorithm represents each cell of the matrix with a future, which waits for the neighbors to complete and then computes the cell's value by using the function `compute_cell`. The algorithm starts by initializing each cell of a mutable $M \times N$ array `arr` with `NONE`. It then fills the array with futures. To do so, the algorithm proceeds in a row-by-row manner and writes the futures of a row in parallel, ensuring that a future is spawned only after the futures it depends on have been spawned. Each future executes the function `compute_cell` which synchronizes with the neighboring cells by calling the function `lookup`, a function that handles boundary conditions around the edges of the matrix, waits for a cell to finish using expression `get`, and returns its value.

The algorithm satisfies disentanglement but this is not easy to establish by reasoning about the the memory allocations of the program. One potential concern arises from the fact that futures read handles to other futures from the array, and since these handles themselves are allocated concurrently, reading them could create entanglement. However, we can see that each future only reads those indices of the shared array which are tabulated before the future is spawned. Thus, no future witnesses the concurrent updates of the array (see line 10) and the

code satisfies determinacy race freedom. Using the result that race freedom implies disentanglement (Theorem 7), we get that the code satisfies disentanglement.

9 Related Work

9.1 Parallel Memory Management

Since its early days in the Lisp language, automatic memory management has come a long way and has become a popular and a prominent feature of modern programming languages. The book by Jones et al. [101] discusses many garbage-collection techniques incorporating parallelism, concurrency, and real-time features. There is, however, relatively little work on the problem of parallel memory management for functional languages that support fork-join parallelism, where programs may create many (e.g., millions of) fine grained threads, which are scheduled by a (usually highly nondeterministic) scheduler.

Within the world of parallel functional programming, we can distinguish between two main architectural approaches to memory management, none of which have (until this work) established space and work bounds.

The first approach uses processor-local or thread-local heaps combined with a shared global heap that must be collected cooperatively [13, 26, 71–73, 116]. This design is employed by the Doligez-Leroy-Gonthier (DLG) parallel collector [71, 72] and the Manticore garbage collector [26, 108]. They enforce the invariant that there are no pointers from the shared global heap into any processor-local heap and no cross pointers between processor local-heaps. To maintain this invariant, all mutable objects are allocated in the shared global heap and (transitively reachable) data is promoted (copied) from a processor-local heap to the shared global heap when updating a mutable object. The Glasgow Haskell Compiler (GHC) uses a garbage collector [116] that follows a similar architecture but also employs techniques similar to Domani et al. [73]. The collector allows pointers from global to local heaps and relies on a read barrier to promote (copy) data to the global heap when accessed. Recent work on a multicore

memory manager for OCaml uses several techniques to reduce the cost of promotions [159]. Because promotions require copying reachable objects from a shared pointer and can be triggered by scheduler actions, none of these approaches can guarantee space and work/time bounds. Indeed, promotions have proved to be expensive in practice [89, 158].

The second approach is due to more recent work on disentanglement [3, 89, 138, 174]. In that work, the authors associate heaps with tasks rather that system-level threads or processors and organize the memory as a dynamic hierarchy that can be arbitrarily deep and grows and shrinks as the computation proceeds. Pointers between heaps that have ancestor-descendant relationships are allowed but cross pointers between concurrent heaps are not allowed. Therefore, disentangled parallel programs can return the result of a child task and migrate threads without copying (promoting) data and concurrent threads can share the data allocated by their ancestors. More recently, techniques were proposed to prevent undefined behavior in case of entanglement by safely terminating the program when entanglement occurs [176]. The primary focus of work on disentanglement so far has been to develop the dynamic memory architecture consisting of a tree of heaps and does not offer any guarantee on space usage.

One of our key contributions is to support unrestricted mutation in the hierarchical heap architecture, and do so without breaking the key advantages of the architecture: independent allocation, reclamation and promotion-free sharing. We propose techniques for dynamically tracking "entanglements" that are created by cross-pointers between concurrent heaps. For garbage collection, we use both concurrent and a hybrid collector. The concurrent collector never moves objects and is similar to standard mark-sweep collectors [102] except that it only collects a single heap (region) of memory. The hybrid collector moves non-entangled objects and leaves the entangled objects in place. To accomplish this, our hybrid collector uses techniques that are similar to the "mostly copying" collector of Bartlett [30, 31] and the Customizable Memory Manager of Attardi et al [24, 25], which was designed as a garbage collector for C++ programs. The basic approach is to track "quasi" or "ambigious" pointers, which may be pointers to objects and prevent these objects from moving during garbage collection. Hosking [96] also presents a mostly copying algorithm; this work allows for concurrency but relies on a stop-the-world phase to collect roots. Our approach applies ideas from these to hierarchical heaps and entanglement but differs in the specific barriers used and specific ways of handling pinned objects. These culminate in techniques that allow threads to allocate, access, and reclaim memory independently without using locks or stop-the-world pauses.

Nearly all of the work on parallel functional languages organizes memory as a hierarchy of heaps; this general idea goes back to 1990s [11, 106, 131, 180]. More recent work includes Sequoia [77] and Legion [32]. These techniques are also remotely related to region-based memory management in the sequential setting [81, 88, 93, 143, 149, 169, 173], which allows objects to be allocated in specific regions, which can be deallocated in bulk. MPL's approach differs from prior work because in MPL hierarchical heaps are fully dynamic and are managed (created, maintained, destroyed, and collected) without programmer intervention.

9.2 Cost Bounds

Memory Management. There are many provably space and work efficient algorithms for garbage collection for uniprocessor computing models. Similar provable algorithms for multiprocessors or parallel systems are more scarce. One notable exception is the algorithm of Blelloch and Cheng [39, 61], which is able to achieve tight space and time bounds. The algorithm, however, is primarily meant for real-time garbage collection and has several shortcomings, including its complex synchronization and load-balancing techniques, and its relatively liberal space usage [28]. Follow-up work has overcome some of these limitations, though sometimes by assuming the uniprocessor model [28, 137]. These real-time algorithms may be used in conjunction with our heap scheduling algorithm in real-time applications.

Scheduling. Nearly all modern parallel programming languages today rely on a scheduler to distribute threads over hardware resources so that the programmer does not have to control it manually. This is important as, manual thread scheduling can be very challenging, especially for multiprogrammed environments.

Early results on scheduling goes back to 1970s, beginning with the work of mathematician Brent [55], whose results were later generalized to greedy schedulers [22, 74]. Blumofe and Leiserson [48], and later Arora, Blumofe, and Plaxton [21], proved that randomized work stealing algorithm can generate efficient greedy schedules "on-the-fly", also on multiprocessor systems. More recent work extended these techniques to account for the cost of thread creation [4, 6, 170] and aspects of responsiveness or interactivity [121–124]. Our implementation is based on a variant of the work stealing algorithm based on private deques [2].

The space consumption and local properties of various scheduling algorithms have been widely studied [3, 9, 37, 40, 44, 47, 62, 110, 128, 165], but, none of these works consider garbage collection and the impact of thread scheduling on garbage collection. For example Blumofe and Leiserson [48] establish space bounds, but assume a restricted form of "stack-allocated computations" that use work stealing, where all memory is allocated on the stack, and all memory allocated by a function call is freed upon returning from that call. They show that $P-$core executions consume as much as $O(S_1 \cdot P)$ space, where S_1 is the space usage of a sequential execution, under stack allocation. Stack allocation is a rather unrealistic assumption for most programming languages, because even non-managed languages such as C/C++ permit heap allocated objects. They assume instead that programs follow a specific allocation strategy, typically "stack allocation", where objects that are allocated by deeper calls cannot be returned without being copied explicitly.

One of our key contributions is that our bounds account for heap allocated objects in work-stealing. To do this, we define sequential space in an unordered fashion, which considers executing two sides of a parallel pair in either order (i.e., left before right, and right before left). Although the unordered depth-first execution differs from traditional left-first and depth-first sequential executions, this difference is fundamental to how work-stealing schedulers operate, and seems unavoidable in this context. We are not aware of any space bounds for heap allocated objects for work-stealing schedulers.

Space Bounds in Scheduling. There has been significant research on understanding the space behavior of parallel programs. Considering programs with manual memory allocation, Burton in 1996 [57] and later Burton and Simpson [152] showed that P-core executions can consumes $O(P \cdot R)$ total space, where R is the space of the sequential execution. Blelloch, Gibbons, and Matias [37] showed that parallel depth-first schedules can deliver tighter space bounds of $O(R + P \cdot S)$, where S is the span of the computation. While this approach delivers excellent space performance, it can be inefficient in practice, because of high scheduling overheads and related factors due to frequent data migrations between processors [129]. Narlikar and Blelloch [128, 129] have proposed algorithms that can combine theoretical and practical efficiency. All of these works assume manual memory management and apply only under the C/C++ model where all the malloc/free operations are given by the programmer. In this thesis, we consider a more general model, where memory is automatically managed without any programmer intervention.

Cost Semantics. To establish our bounds, we use a cost semantics that keeps track of work, space usage, and yields a task tree of the computation. The task tree allow us to reason about the intrinsic properties of the computation (threads/concurrency created and their relationships). For our bounds, we use a notion of reachability that accounts for the different orders in which parallel pairs may be executed in a sequential computation. This notion is quite interesting: it does not account for all interleavings of the parallel pairs, but just two specific ones, where the left completes before the right starts, and the right completes before the left starts.

Cost semantics have proved to be an effective tool for reasoning about non-trivial properties of the computation. The idea of cost semantics goes back to the early 90s [142, 145] and has been shown to be particularly important in high-level languages such as lazy (e.g., [145–147]) and parallel languages (e.g., [4, 6, 35, 41, 164]). Aspects of our cost semantics resemble prior cost semantics used in parallel languages [4, 6, 41, 164], though the specifics such as our use of task trees and our specific notion of reachability measure differ.

9.3 Parallel Programming Languages

In this thesis, we implement and evaluate our techniques as part of the MPL (MaPLe) compiler for Parallel ML, which is a functional language that extends Standard ML with parallelism, and supports references and destructive updates [3, 89, 138, 174, 178]. Parallel ML builds on a rich history of research on parallel functional programming languages, including parallel Haskell [58, 103, 116] Manticore [82, 83] MultiMLton [157, 183], SML# [133], and multicore OCaml [158] projects. MPL differs from these projects in its emphasis on theoretical guarantees on efficiency and implementations that can match the bounds in practice. Even though it is not aimed at functional programming, the Rust language supports type-safe programming and can ensure memory safety without using a garbage collector [144].

In addition to the functional languages mentioned above, there has also been extensive research on procedural parallel languages. Systems including Cilk/Cilk++ [49, 85, 99], Cilk-

F/L [154, 155], I-Cilk [121], and Intel TBB [100] extend C/C++ with task parallelism and require manual memory management. Deterministic Parallel Java [53], Fork-Join Java [109], and Habanero Java [97], extend the Java language to support parallelism and support automatic memory management. The X10 [59] is designed with concurrency and parallelism from the beginning and uses automatic memory management. The Go language is designed from grounds up with concurrency in mind. The Rust language supports type-safe programming and can ensure memory safety without using a garbage collector [144].

9.4 Disentanglement and Futures

Disentanglement. Research on disentanglement originates with two key theoretical results for fork-join programs. Acar et al. [3] showed that purely functional fork-join programs satisfy program-level disentanglement. Westrick et al. [174, 178] generalized this result to include effects and showed that fork-join parallel programs remain disentangled as long as the mutable effects are performed without creating determinacy races. This thesis generalizes the applicability of disentanglement in two important ways.

First, we redefine disentanglement as a continuous object-level property rather than a program-level property, which enables us to propose the disentanglement hypothesis. This shift is important because it allows us to exploit the disentanglement hypothesis for provably and practically efficient memory management of fork-join programs.

Second, we extend the theory of disentanglement to include futures and prove that determinacy-race-free programs with futures do not have any entangled objects. Importantly, our generalization of disentanglement for futures is consistent with fork-join, i.e., if we restrict our language to fork-join, the class of programs that satisfy our definition of disentanglement is the same. The theoretical results for futures are strictly more general because futures are more expressive than fork-join (we can implement fork join constructs using futures [162]).

From Fork-Join Parallelism to Futures. Fork-join parallelism has proved to be an effective model for many parallel computations [50, 84, 109]. But it has limitations, especially when it comes to computations where parallel tasks run asynchronously until a data-driven condition, e.g., based on input from the user, is satisfied. Muller et al.'s work shows that futures can provide this kind of asynchrony, which is pervasive in interactive applications [121–123, 126, 127, 156]. This line of work observed that, when combining asynchronous interaction and fine-grained compute-heavy parallelism, it is necessary to assign higher *priorities* in the scheduler to the interactive threads to maintain responsiveness. Priorities are orthogonal to the theory of disentanglement we explore in this thesis, and so we did not add them to this theory to keep the focus on disentanglement.

Researchers have therefore proposed futures for increasing the expressiveness of parallel languages. Futures were invented in the 1970s [29] and were brought to their current form by Halstead in the 1980s [91]. Today, many concurrent and parallel programming systems support futures, including Cilk-F/L [154, 155], I-Cilk [121], Concurrent Haskell [92, 112, 116,

136], Habanero Java [97], Parallel ML [7, 17, 19, 82, 83, 89, 133, 138, 157, 163, 174], OCaml [69, 114], Rust [144], and TPL (a .NET library) [111].

Futures can also be challenging to manage in the run-time system of a programming language. For example, data locality of parallel programs with futures can significantly worsen when they execute in parallel [1] and restricting their expressiveness can improve the data locality [95, 165].

Exploiting Disentanglement Hypothesis for Futures. The motivation for generalizing the disentanglement hypothesis to futures comes from its effectiveness in the fork-join setting. We believe that our memory management techniques for fork-join can be generalized to futures in a provably efficient fashion. First, because our result establishes race-free programs are disentangled, we anticipate that the disentanglement hypothesis will hold for parallel programs with futures. The result implies that objects become entangled only because of races and as we have observed for fork-join, races typically only involve a small portion of the memory. This is important because our memory manager in MPL is designed to exploit the disentanglement hypothesis and its efficiency relies on the observation that entanglement is rare. Second, our semantics for futures defines the heap tree at each step (Section 8.1.5), and the join step for futures (Section 8.1.4) is analogous to the surrender step in our coscheduling algorithm. This suggests that our coscheduling algorithm could be extended to create and surrender heaps for futures, assigning heaps to processors in a similar fashion. However, one engineering challenge is extending the specialized task scheduling infrastructure of MPL to support futures, which have a more complex dependency structure than fork-join.

Race Freedom. As we established in this thesis, disentanglement is implied by freedom from determinacy races [78, 130]. Determinacy races, also called general races, cause nondeterminism and are considered bugs for programs that are intended to be deterministic [130]. Absence of determinacy races guarantees determinism: in every execution, the executed instructions and their execution order are the same. Determinacy races are different from *data races*, which only occur when a critical section of the code is not executed atomically. Unlike data races, determinacy races are quite conservative and can include accesses that produce deterministic outcomes. For example, atomic fetch-and-add operations by concurrent threads do not cause a data race because each increment is performed atomically. However, such operations cause a determinacy race because the execution order is nondeterministic.

There is a lot of work checking for race freedom and bounding the impact of races, partly because data races usually cause incorrect behavior [8, 54, 70]. Many algorithms for race detection in fork-join parallel programs have been proposed [33, 60, 78, 79, 117, 139, 140, 171, 179]. More recent work considers race detection for futures [179].

There has also been significant research on race detection for more general concurrent programs [80, 104, 132, 148, 160, 181]. Such programs differ from task-parallel programs, because they use coarse-grained threads and synchronize in an unstructured manner, using locks and other synchronization primitives. The two classes of programs therefore typically require dif-

ferent approaches for reasons of efficiency, soundness (ability to correctly detect races), and completeness (ability to detect all races).

Deadlock. The expressiveness of futures can make them harder to use safely. One important concern is deadlock: with futures, it is possible to create cyclic dependencies in a computation that prevent the computation from making progress. Cogumbreiro et al. identify this problem and formulate two properties called "known joins" and "transitive joins" that can be enforced by restricting the expressive power of joins [63, 172] to prevent deadlock. They achieve this by enforcing a discipline on the use of futures, i.e., they forbid tasks from synchronizing with certain other futures. Loosely speaking, the known joins property states that task A is allowed to sync with future B only if B is spawned by one of the ancestors of A. The transitive joins work relaxes the known joins restriction by adding transitivity, i.e., if task A can synchronize with B and B can synchronize with C, then A is allowed to synchronize with C. The known joins property is naturally satisfied in disentangled executions—in a disentangled execution, a task only knows about those locations/futures that are allocated/spawned by its ancestors. Thus, we have separately proven in our paper that disentanglement implies deadlock freedom [20]. Prior work has shown that determinacy race freedom implies known joins.

10 Conclusion and Future Work

10.1 Conclusion

In this thesis, we presented novel techniques for addressing the challenge of parallel memory management in functional programs. Our memory management techniques are provably work and space efficient, leading to practically efficient and scalable parallel programming. The key technical innovation behind our techniques is coscheduling, which partitions the memory into smaller heaps and schedules (assign) these heaps on processors in such a way that allows each processor to independently manage its own heaps. The provable bounds are crucial for not only for guaranteed efficiency, but also for precisely specifying the behavior of the memory manager across different types of programs, including deterministic and nondeterministic programs.

The guiding principle for our approach is the disentanglement hypothesis, which states that most objects are disentangled, i.e., they are only accessed by tasks that are sequentially dependent on the task that allocated them. To exploit the hypothesis, we organize the memory into a tree of heaps, which mirrors the structure of parallel tasks, and enables efficient management of the disentangled objects. Then, we develop techniques for tracking and managing the special and rare case of entangled objects. These techniques incur overhead only for entangled objects, ensuring that allocations and accesses of disentangled objects observe no overhead due to them.

A fundamental reason for the prevalence of disentangled objects is the fact that entangled objects arise from races on mutable objects. In contrast, immutable objects—which are common in functional programming—do not lead to entanglement. Our memory manager is optimized to handle immutable objects, using bump allocation for fast allocations and ensuring barrier-free access. Supporting immutability efficiently is a key goal for our approach, as our memory management techniques are explicitly designed to avoid overheads for immutable objects.

Building on memory management techniques for the heap tree, we developed a coscheduling algorithm that integrates task scheduling with memory management. By coscheduling the computation with memory, we ensure that related tasks and their associated memory are managed together on the same processor. This integration reduces the overheads of memory management and improves precision—when a processor garbage collects, it traces all the heaps that are assigned to it, accurately accounting for all the inter-heap pointers between the heaps assigned to it. Coscheduling guarantees work and space bounds because it clusters heaps such that each cluster represents a sequential execution.

To demonstrate the practicality of our techniques, we developed the the MPL compiler suite and showed that it can be used to implement sophisticated non-blocking and wait-free concurrent data structures, as well as parallel algorithms that use them. Our experiments show that MPL performs well, incurring relatively small overheads compared to sequential runs, and scaling well as the number of cores increase. Notably, MPL delivers performance and compactness simultaneously: parallel runs on dozens of cores usually consume less memory than sequential runs and deliver significant speedups. We also perform a more modest comparison with several other languages including Go, Java, Multicore OCaml, and C++. Our results show that MPL is competitive with these languages, establishing that functional languages can deliver both performance and scalability, while offering important safety benefits.

10.2 Future Work

There are several promising directions for future research based on the concepts presented in this thesis.

Span Bounds. In this thesis, we primarily focused on establishing the provable bounds for work and space for our memory management techniques. However, the *span* of our memory management techniques is another crucial metric for analyzing the efficiency of the runtime system. Span represents the longest chain of sequentially dependent instructions in a parallel computation. Bounding the span is important for establishing an upper bound on the end-to-end execution time of the program.

One promising direction is designing a **span-aware collection policy** and extending our analysis for deriving span bounds of memory managed programs. Currently, we perform garbage collection of each heap sequentially. Instead, future work could implement a parallel garbage collector that runs in $O(D)$ span, where D is the longest chain of pointers in memory. The collection policy could then take advantage of the parallel garbage collector and ensure that the span of the program, including the cost of parallel garbage collection, is $O(D+S)$, where S is the span of the computation. Achieving work, space, and span bounds simultaneously, while preserving the practical benefits of independent allocation and garbage collection, would be an excellent theoretical result.

Interactive Applications and Futures. Another promising area for generalizing our memory management techniques is their application to interactive applications. Interactive applications, including real-time systems or data-driven software, introduce new challenges for memory management due to their need for responsiveness. Although our work primarily focuses on the end-to-end running time or throughput of applications, there is potential to apply the disentanglement hypothesis for interactive applications.

As proved in this thesis, when interactive applications are expressed using futures, they satisfy the disentanglement hypothesis. Integrating our coscheduling techniques and performing independent memory management for interactive applications is an exciting direction for future research. Independent garbage collection is particularly powerful, because it requires no more than one processor to be paused at any time, and can be used to eliminate stop-the-world pauses and optimize for low latency. Exploiting the disentanglement hypothesis in such interactive environments is an exciting avenue for research.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[3] Umut A. Acar, Guy Blelloch, Matthew Fluet, Stefan K. Muller, and Ram Raghunathan. Coupling memory and computation for locality management. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[4] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)*, 26:e23, 2016.

[5] Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. Dag-calculus: A calculus for parallel computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 18–32, 2016.

[6] Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 769–782, 2018. ISBN 978-1-4503-5698-5.

[7] Umut A. Acar, Jatin Arora, Matthew Fluet, Ram Raghunathan, Sam Westrick, and Rohan Yadav. Mpl: A high-performance compiler for parallel ml, 2020. https://github.com/MPLLang/mpl.

[8] Sarita V. Adve. Data races are evil with no exceptions: technical perspective. *Commun. ACM*, 53(11):84, 2010.

[9] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, R. K. Shyamasundar, and Katherine A. Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*, pages 229–240, 2007.

[10] T. R. Allen and D. A. Padua. Debugging Fortran on a shared memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, August

1987.

[11] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 600–608 vol.2, Oct 1990. doi: 10.1109/FSCS.1990.89581.

[12] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. The problem-based benchmark suite (pbbs), V2. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 445–447. ACM, 2022. doi: 10.1145/3503221.3508422. URL https://doi.org/10.1145/3503221.3508422.

[13] Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, pages 21–30, 2010.

[14] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989. URL http://www.cs.princeton.edu/fac/~appel/papers/143.ps.

[15] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[16] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, January 1996. URL ftp://daffy.cs.yale.edu/pub/papers/shao/stack.ps.

[17] Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space efficient parallel functional programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[18] Jatin Arora, Sam Westrick, and Umut A. Acar. Replication instructions for Article: Efficient Parallel Functional Programming with Effects, March 2023. URL https://doi.org/10.5281/zenodo.7824069.

[19] Jatin Arora, Sam Westrick, and Umut A. Acar. Efficient parallel functional programming with effects. *Proc. ACM Program. Lang.*, 7(PLDI):1558–1583, 2023. doi: 10.1145/3591284. URL https://doi.org/10.1145/3591284.

[20] Jatin Arora, Stefan K. Muller, and Umut A. Acar. Disentanglement with futures, state, and interaction. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. doi: 10.1145/3632895. URL https://doi.org/10.1145/3632895.

[21] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[22] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[23] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, October 1989.

[24] Giuseppe Attardi and Tito Flagella. A customisable memory management framework. Technical Report TR-94-010, International Computer Science Institute, Berkeley, 1994. URL ftp://ftp.icsi.berkeley.edu:/pub/techreports/1994/tr-94-010.ps.Z. Also Proceedings of the USENIX C++ Conference, Cambridge, MA, 1994.

[25] Giuseppe Attardi, Tito Flagella, and Pietro Iglio. A customisable memory management framework for C++. *Software Practice and Experience*, 28(11):1143–1183, November 1998. URL ftp://ftp.di.unipi.it/pub/Papers/attardi/SPE.ps.gz.

[26] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC)*, pages 51–57, 2011.

[27] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, page 10–es, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781450378369. doi: 10.1145/1275808.1276390. URL https://doi.org/10.1145/1275808.1276390.

[28] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[29] Henry G. Baker and Carl E. Hewitt. The incremental garbage collection of processes. AI memo 454, MIT Press, December 1977.

[30] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988. URL http://www.research.digital.com/wrl/techreports/88.2.ps. Also in Lisp Pointers 1, 6 (April–June 1988), 2–12.

[31] Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical note, DEC Western Research Laboratory, Palo Alto, CA, October 1989. URL ftp://ftp.digital.com/pub/DEC/WRL/research-reports/WRL-TN-12.ps. Sources available in ftp://ftp.digital.com/pub/DEC/CCgc.

[32] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2012. doi: 10.1109/SC.2012.71.

[33] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 133–144, 2004.

[34] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX, page 117–128, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581133170. doi: 10.1145/378993.379232. URL https://doi.org/10.1145/378993.379232.

[35] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 226–237. ACM, 1995.

[36] Guy Blelloch and Margaret Reid-Miller. Pipelining with futures. *Theory of Computing Systems*, 32(3):213–239, 1999.

[37] Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[38] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[39] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.

[40] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, 2004.

[41] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, 1996.

[42] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994.

[43] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 12–23, 1997.

[44] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, 2011.

[45] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP '12*, pages 181–192, 2012.

[46] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib - A toolkit for parallel algorithms on shared-memory multicore machines. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 507–509. ACM, 2020. doi:

10.1145/3350755.3400254. URL https://doi.org/10.1145/3350755.3400254.

[47] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[48] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[49] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[50] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996.

[51] Robert L. Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *ACM POPL*, 2011.

[52] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 97–116, 2009.

[53] Robert L Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *First USENIX Conference on Hot Topics in Parallelism*, 2009.

[54] Hans-Juergen Boehm. How to miscompile programs with "benign" data races. In *3rd USENIX Workshop on Hot Topics in Parallelism, HotPar'11, Berkeley, CA, USA, May 26-27, 2011*, 2011.

[55] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974. ISSN 0004-5411.

[56] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, page 261–270, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336178. doi: 10.1145/2767386.2767436. URL https://doi.org/10.1145/2767386.2767436.

[57] F.W. Burton. Guaranteeing good memory bounds for parallel programs. *IEEE Transactions on Software Engineering*, 22(10):762–773, 1996. doi: 10.1109/32.544353.

[58] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France,*

January 16, 2007, pages 10–18, 2007.

[59] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005.

[60] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, 1998.

[61] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 125–136, Snowbird, Utah, June 2001. ACM Press.

[62] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 207–216, New York, NY, USA, 2008. ACM.

[63] Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco T. Vasconcelos, and Max Grossman. Deadlock avoidance in parallel programs with futures: why parallel tasks should not wait for strangers. *Proc. ACM Program. Lang.*, 1(OOPSLA):103:1–103:26, 2017.

[64] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[65] Intel Corp. Knights landing (knl): 2nd generation intel xeon phi processor. In *Intel Xeon Processor E7 v4 Family Specification*, 2017. https://ark.intel.com/products/series/93797/Intel-Xeon-Processor-E7-v4-Family.

[66] Christopher M. Dawson and Michael A. Nielsen. The solovay-kitaev algorithm. *Quantum Inf. Comput.*, 6(1):81–95, 2006. doi: 10.26421/QIC6.1-6. URL https://doi.org/10.26421/QIC6.1-6.

[67] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Trans. Parallel Comput.*, 8(1):4:1–4:70, 2021. doi: 10.1145/3434393. URL https://doi.org/10.1145/3434393.

[68] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. Pac-trees: Supporting parallel and compressed purely-functional collections, 2022.

[69] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming*, pages 98–117, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89719-6.

[70] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. *SIGPLAN Not.*, 53(4):242–255, jun 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192421. URL https://doi.org/10.1145/3296979.3192421.

[71] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Portland, OR, January 1994. ACM Press. URL ftp://ftp.inria.fr/INRIA/Projects/para/doligez/DoligezGonthier94.ps.gz.

[72] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993. URL file://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/publications/concurrent-gc.ps.gz.

[73] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 76–87, Berlin, June 2002. ACM Press. URL http://www.cs.technion.ac.il/~erez/publications.html.

[74] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[75] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '91*, pages 580–588, November 1991.

[76] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd. 01 2006.

[77] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0.

[78] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, June 1997.

[79] Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2005.

[80] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. *SIGPLAN Not.*, 44(6):121–133, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542490.

[81] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. Linear regions are all you need. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*, March 2006.

[82] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008.

[83] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[84] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[85] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, 2009.

[86] Marcelo J. R. Gonçalves. *Cache Performance of Programs with Intensive Heap Allocation and Generational Garbage Collection*. PhD thesis, Department of Computer Science, Princeton University, May 1995.

[87] Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Record of the 1995 Conference on Functional Programming and Computer Architecture*, June 1995.

[88] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 282–293, Berlin, June 2002. ACM Press. ISBN 1-58113-463-0.

[89] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 81–93, 2018.

[90] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17. ACM, 1984.

[91] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[92] Kevin Hammond. Why parallel functional programming matters: Panel statement. In *Reliable Software Technologies - Ada-Europe 2011 - 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings*, pages 201–205, 2011.

[93] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[94] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In Jennifer L. Welch, editor, *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001. doi: 10.1007/3-540-45414-4_21. URL https://doi.org/10.1007/3-540-45414-4_21.

[95] Maurice Herlihy and Zhiyu Liu. Well-structured futures and cache locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 155–166, New York, NY, USA, 2014. ACM.

[96] Antony Hosking. Portable, mostly-concurrent, mostly-copying garbage collection for multi-processors. volume 2006, 01 2006. doi: 10.1145/1133956.1133963.

[97] Shams Mahmood Imam and Vivek Sarkar. Habanero-java library: a java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*, pages 75–86, 2014.

[98] Intel. Intel threading building blocks, 2011. https://www.threadingbuildingblocks.org/.

[99] *Intel Cilk++ SDK Programmer's Guide*. Intel Corporation, October 2009. Document Number: 322581-001US.

[100] TBB. *Intel(R) Threading Building Blocks*. Intel Corporation, 2009. Available from http://www.threadingbuildingblocks.org/documentation.php.

[101] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[102] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook : The Art of Automatic Memory Management*. CRC Press, 2012.

[103] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010.

[104] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 157–170. ACM, 2017.

[105] Alexei Yu Kitaev, Alexander Shen, Mikhail N Vyalyi, and Mikhail N Vyalyi. *Classical and quantum computation*. Number 47. American Mathematical Soc., 2002.

[106] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 262–273, New York, NY, USA, 1993. ACM. ISBN 0-8186-4340-4. doi: 10.1145/169627.169724. URL http://doi.acm.org/10.1145/169627.169724.

[107] Ananya Kumar, Guy E. Blelloch, and Robert Harper. Parallel functional arrays. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 706–718. ACM, 2017.

[108] Matthew Le and Matthew Fluet. Partial aborts for transactions via first-class continuations. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 230–242, 2015. ISBN 978-1-4503-3669-7.

[109] Doug Lea. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*, pages 36–43, 2000.

[110] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *TOPC*, 2(3):17:1–17:42, 2015.

[111] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, 2009.

[112] Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 107–118, 2007.

[113] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings*, volume 8304 of *Lecture Notes in Computer Science*, pages 206–220. Springer, 2013. doi: 10.1007/978-3-319-03850-6_15. URL https://doi.org/10.1007/978-3-319-03850-6_15.

[114] LWT. Lwt ocaml. GitHub, 2022. URL https://github.com/ocsigen/lwt.

[115] Simon Marlow. Parallel and concurrent programming in haskell. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, volume 7241 of *Lecture Notes in Computer Science*, pages 339–401. Springer, 2011.

[116] Simon Marlow and Simon L. Peyton Jones. Multicore garbage collection with local heaps. In Hans-Juergen Boehm and David F. Bacon, editors, *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, pages 21–32. ACM, 2011.

[117] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33, 1991.

[118] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996. doi: 10.1145/248052.248106. URL https://doi.org/10.1145/248052.248106.

[119] Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 196–203, 2013.

[120] MLton. MLton web site. http://www.mlton.org, n.d.

[121] Stefan Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. Responsive parallelism with futures and state. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[122] Stefan K. Muller and Umut A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 71–82, 2016.

[123] Stefan K. Muller, Umut A. Acar, and Robert Harper. Responsive parallel computation: Bridging competitive and cooperative threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 677–692, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8.

[124] Stefan K. Muller, Umut A. Acar, and Robert Harper. Competitive parallelism: Getting your priorities right. *Proc. ACM Program. Lang.*, 2(ICFP):95:1–95:30, July 2018. ISSN 2475-1421.

[125] Stefan K. Muller, Umut A. Acar, and Robert Harper. Types and cost models for responsive parallelism. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '18, 2018.

[126] Stefan K. Muller, Sam Westrick, and Umut A. Acar. Fairness in responsive parallelism. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2019, 2019.

[127] Stefan K. Muller, Kyle Singer, Devyn Terra Keeney, Andrew Neth, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. Responsive parallelism with synchronization. *Proc. ACM Program. Lang.*, 7(PLDI):712–735, 2023.

[128] Girija J. Narlikar and Guy E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(1):138–173, 1999. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/314602.314607.

[129] Girija Jayant Narlikar. *Space-efficient scheduling for parallel, multithreaded computations*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1999.

[130] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[131] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. ISSN 1061-7264. doi: 10.1145/289918.289920. URL http://doi.acm.org/10.1145/289918.289920.

[132] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In Rudolf Eigenmann and Martin C. Rinard, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, pages 167–178. ACM, 2003.

[133] Atsushi Ohori, Kenjiro Taura, and Katsuhiro Ueno. Making sml# a general-purpose high-performance language, 2018. Unpublished Manuscript.

[134] OpenMP 5.0. *OpenMP Application Programming Interface, Version 5.0*, November 2018. Accessed in July 2018.

[135] Wolfgang Paul, Uzi Vishkin, and Hubert Wagener. Parallel dictionaries on 2–3 trees. In *International Colloquium on Automata, Languages, and Programming*, pages 597–609. Springer, 1983.

[136] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008.

[137] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices*, 43(6):33–44, 2008.

[138] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 392–406, New York, NY, USA, 2016. ACM.

[139] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 368–383. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-16611-2.

[140] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, 2012.

[141] Dan Robinson. Hpe shows the machine — with 160tb of shared memory. *Data Center Dynamics*, May 2017.

[142] Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, 1989.

[143] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, August 1967.

[144] Rust Team. Rust language, 2019. URL https://www.rust-lang.org/.

[145] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, Imperial College, September 1990.

[146] David Sands. Complexity analysis for a lazy higher-order language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*, pages 361–376, London, UK, 1990. Springer-Verlag.

[147] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Principles of Programming Languages*, pages 355–366, 1995.

[148] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[149] Jacob T. Schwartz. Optimization of very high level languages (parts i and ii). *Computer Languages*, 2–3(1):161–194,197–218, 1975.

[150] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP '13*, pages 135–146, New York, NY, USA, 2013. ACM.

[151] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, 2012. ISBN 978-1-4503-1213-4.

[152] D.J. Simpson and F.W. Burton. Space efficient execution of deterministic parallel programs. *IEEE Transactions on Software Engineering*, 25(6):870–882, 1999. doi: 10.1109/32.824415.

[153] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. Proactive work stealing for futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 257–271, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295735. URL https://doi.org/10.1145/3293883.3295735.

[154] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. Proactive work stealing for futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 257–271, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6225-2. doi: 10.1145/3293883.3295735. URL http://doi.acm.org/10.1145/3293883.3295735.

[155] Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. Scheduling I/O latency-hiding futures in task-parallel platforms. In Bruce M. Maggs, editor, *1st Symposium on Algorithmic Principles of Computer Systems, APOCS 2020, Salt Lake City, UT, USA, January 8, 2020*, pages 147–161. SIAM, 2020. doi: 10.1137/1.9781611976021.11. URL https://doi.org/10.1137/1.9781611976021.11.

[156] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. Priority scheduling for interactive applications. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 465–477, 2020.

[157] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Multimlton: A multicore-aware runtime for standard ml. *Journal of Functional Programming*, FirstView: 1–62, 6 2014.

[158] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Retrofitting parallelism onto ocaml. *Proc. ACM Program. Lang.*, 4(ICFP):113:1–113:30, 2020.

[159] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Retrofitting parallelism onto ocaml. *arXiv preprint arXiv:2004.11663*, 2020.

[160] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 387–400. ACM, 2012.

[161] A. Sodani. Knights landing (knl): 2nd generation intel xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Aug 2015.

[162] Daniel Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009.

[163] Daniel Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, May 2009. URL https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf.

[164] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008.

[165] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 91–100, New York, NY, USA, 2009. ACM.

[166] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 218–231. ACM Press, 1990.

[167] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. Pam: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18. ACM, February 2018. doi: 10.1145/3178487.3178509. URL http://dx.doi.org/10.1145/3178487.3178509.

[168] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. PAM: parallel augmented maps. In Andreas Krall and Thomas R. Gross, editors, *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 290–304. ACM, 2018. doi: 10.1145/3178487.3178509. URL https://doi.org/10.1145/3178487.3178509.

[169] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997. URL http://www.diku.dk/research-groups/topps/

activities/kit2/infocomp97.ps.

[170] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS*, 36(3):10:1–10:51, September 2014.

[171] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 83–94, 2016.

[172] Caleb Voss, Tiago Cogumbreiro, and Vivek Sarkar. Transitive joins: a sound and efficient online deadlock-avoidance policy. In Jeffrey K. Hollingsworth and Idit Keidar, editors, *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 378–390, 2019.

[173] David Walker. On linear types and regions. In *Proceedings of the First workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*, London, January 2001. URL http://www.diku.dk/topps/space2001/program.html#DavidWalker.

[174] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2020.

[175] Sam Westrick, Jatin Arora, and Umut A. Acar. Entanglement detection with near-zero cost. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022. doi: 10.1145/3547646. URL https://doi.org/10.1145/3547646.

[176] Sam Westrick, Jatin Arora, and Umut A. Acar. Entanglement detection with near-zero cost. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2022, 2022.

[177] Sam Westrick, Larry Wang, and Umut A. Acar. DePa: Simple, provably efficient, and practical order maintenance for task parallelism. *CoRR*, abs/2204.14168, 2022. doi: 10.48550/arXiv.2204.14168. URL https://doi.org/10.48550/arXiv.2204.14168.

[178] Samuel Westrick. Efficient and Scalable Parallel Functional Programming Through Disentanglement. 10 2022. doi: 10.1184/R1/21313731.v1. URL https://kilthub.cmu.edu/articles/thesis/Efficient_and_Scalable_Parallel_Functional_Programming_Through_Disentanglement/21313731.

[179] Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. Parallel determinacy race detection for futures. In Rajiv Gupta and Xipeng Shen, editors, *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, pages 217–231. ACM, 2020. doi: 10.1145/3332466.3374536. URL https://doi.org/10.1145/3332466.3374536.

[180] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Kr-

ishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[181] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 221–234. ACM, 2005.

[182] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990. ISSN 0164-1212. doi: https://doi.org/10.1016/0164-1212(90)90084-Y. URL https://www.sciencedirect.com/science/article/pii/016412129090084Y.

[183] Lukasz Ziarek, K. C. Sivaramakrishnan, and Suresh Jagannathan. Composable asynchronous events. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 628–639, 2011.