THE UNIVERSITY OF CHICAGO

IMPLICITLY PARALLEL SCRIPTING AS A PRACTICAL AND MASSIVELY SCALABLE PROGRAMMING MODEL FOR HIGH-PERFORMANCE COMPUTING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
TIMOTHY ARMSTRONG

CHICAGO, ILLINOIS
JUNE 2015
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. v

LIST OF TABLES .................................................................. vii

ACKNOWLEDGMENTS ............................................................... viii

THESIS .................................................................................. ix

1 INTRODUCTION ................................................................. 1
   1.1 Goals and Scope of Dissertation ................................... 4
   1.2 Swift Programming Language Background .................... 6

2 THE SWIFT EXECUTION MODEL ............................................. 10
   2.1 Overview .................................................................. 10
   2.2 Notation .................................................................. 12
   2.3 Data Types and Data Structures ................................... 13
   2.4 Sequential Semantics with Nondeterministic Task Order ........ 24
   2.5 Determinism of Execution Model ................................. 34
   2.6 Extensions to Semantics ............................................. 44
   2.7 Mapping Swift to Execution Model ............................... 49
   2.8 Limitations and Future Work ...................................... 52
   2.9 Related Work .......................................................... 54

3 A MASSIVELY PARALLEL RUNTIME FOR DATA-DRIVEN TASK PARALLELISM 59
   3.1 Runtime Architecture ................................................ 60
   3.2 Task Queue ................................................................ 69
   3.3 Data Store ................................................................ 79
   3.4 Dependency Engine .................................................. 81
   3.5 Evaluation .................................................................. 82
   3.6 Runtime Support for Heterogeneous Tasks ....................... 87
   3.7 Related Work ........................................................... 88

4 COMPILING SWIFT FOR MASSIVE PARALLELISM .................. 90
   4.1 Compiler Architecture ................................................. 91
   4.2 Compiler Frontend ..................................................... 91
   4.3 Optimization Goals for Data-driven Task Parallelism ............ 93
   4.4 Intermediate Representation ...................................... 94
   4.5 STC Optimizer ........................................................ 103
   4.6 Compiler Postprocessing ............................................ 115
   4.7 Related Work on Compiler Optimization ......................... 118
5 EVALUATION ................................................................. 121
  5.1 Method for Large-Scale Experiments ............................. 124
  5.2 Discussion and Analysis of Large-Scale Experiments ............. 125
  5.3 Contribution of Individual Optimizations ......................... 127
  5.4 Memory Management Overhead ...................................... 129
  5.5 Compilation Time .................................................... 130

6 CONCLUSION ............................................................. 132
  6.1 Future Work .......................................................... 133

APPENDICES ............................................................... 134

A SWIFT/T LANGUAGE ..................................................... 135
  A.1 Semantics by Example ............................................... 135
  A.2 Grammar ............................................................... 147

B TCL FOR HIGH PERFORMANCE COMPUTING ......................... 150

C COMPILER INTERMEDIATE REPRESENTATION AND OPTIMIZATIONS . 153
  C.1 Intermediate Representation Interpreter .......................... 153
  C.2 Ordering of Optimization Passes ................................... 153
  C.3 Value Numbering Pseudocode ...................................... 157

REFERENCES ............................................................. 159
# LIST OF FIGURES

1.1 Conceptual overview of abstractions and systems .......................... 5
2.1 Trace graph for data-driven task parallelism ............................ 10
2.2 Swift code and trace graph for simple application ....................... 11
2.3 Operations semantics for execution model .................................. 30
2.4 Equivalence of concurrent and interleaved execution ..................... 45
2.5 Deadlocking Swift/T example program ..................................... 52
2.6 Swift/T example program where deadlocking depends on optimization level ... 53
3.1 Distributed services view of Swift/T runtime ............................. 60
3.2 Runtime process layout on distributed-memory system .................... 64
3.3 Runtime architecture showing coordination of worker processes ......... 64
3.4 Matching algorithm for tasks in ADLB server ............................ 70
3.5 Scalability of previous version of Swift/T ................................ 71
3.6 Request queue data structures ............................................ 72
3.7 Work queue data structures ............................................... 73
3.8 Pseudocode for work stealing algorithm with asynchronous probes .... 78
3.9 Efficiency of request queue data structure ................................ 83
3.10 Efficiency and scalability of work queue data structure ................ 84
3.11 Efficiency and scalability of request and work queue data structures .... 85
3.12 Single server task queue throughput .................................... 86
3.13 Throughput and scaling of runtime system for varying task durations ... 87
4.1 Swift/T toolchain ......................................................... 90
4.2 STC compiler architecture ............................................... 91
4.3 Swift code fragment illustrating wavefront pattern ....................... 92
4.4 Sample Swift/T program and IR for recursive Fibonacci algorithm .... 95
4.5 Grammar for IR-1 ....................................................... 96
4.6 Opcodes for IR-1 ....................................................... 97
4.7 Type system and variables used in STC intermediate representation .... 98
4.8 Alias analysis in Dead Code Elimination ................................ 106
4.9 Intermediate representation with instructions in reverse dataflow order ... 111
4.10 Intermediate representation with instructions reordered into dataflow order ... 111
4.11 Intermediate representation optimized after reordering .................. 112
4.12 Task optimizations for Swift/T code fragment .......................... 115
5.1 Effect of optimization on application speedup and scalability .......... 123
5.2 Impact of optimizations on runtime operations ........................... 124
5.3 Incremental impact of optimizations on runtime operation counts ...... 125
5.4 Operation counts with single optimizations disabled ..................... 128
5.5 Impact of unoptimized and optimized reference counting ............... 129
A.1 Swift/T Example - Hello World ......................................... 135
LIST OF TABLES

3.1 Runtime task operations ............................................. 61
3.2 Runtime data operations ............................................. 63
3.3 Comparison of work queue data structures ........................ 74
3.4 Comparison of related work on task matching .................... 75

5.1 STC Compile Times .................................................. 130
ACKNOWLEDGMENTS

This research was supported by the U.S. DOE Office of Science under contract DE-AC02-06CH11357 and by NSF award ACI 1148443. Computing resources were provided in part by NIH through the Computation Institute and the Biological Sciences Division of the University of Chicago and Argonne National Laboratory, under grant S10 RR029030-01, and by NSF award ACI 1238993 and the state of Illinois through the Blue Waters sustained-petascale computing project.
THESIS

That a simple, deterministic, high-level programming language built around the abstraction of monotonic variables can offer high efficiency and scalability for a broad range of large-scale computing applications on massively parallel distributed-memory computer systems.
CHAPTER 1
INTRODUCTION

In recent years, large-scale computation has become an indispensable tool in many fields, including those that have not traditionally used high-performance computing. These include data-intensive applications such as machine learning and scientific data crunching and compute-intensive applications such as high-fidelity simulations. Programming models and languages play a central role in enabling such large-scale computation by abstracting the computational capabilities of a network of computers in ways that enable programmers to build parallel applications without dealing with the full complexity of a distributed system of computers. Examples of widely-used programming models include MapReduce [25], which abstracts a distributed computation as map and reduce functions applied to streams of data tuples to enable automatic scaling and fault tolerance and MPI [87], which abstracts a range of network architectures with standardized point-to-point and collective messaging operations.

This dissertation is motivated by a number of problems that are not addressed well with current programming models (and/or the current implementations of those programming models). In this dissertation I demonstrate that a high-level programming language built on top of the data-driven task parallelism execution model can feasibly address many of these problems for certain applications. In this execution model, massive numbers of concurrently executing tasks are dynamically assigned to execution resources, with synchronization and communication handling using intertask data dependencies.

One motivating problem is ease of programming for irregular, task-parallel applications running at large scale on high-performance computing systems with hundreds or thousands of cores. The traditional development model for high-performance computing requires close cooperation between domain experts and parallel computing experts to build applications that efficiently run on large-scale distributed-memory systems, with careful attention given to
low-level concerns such as distribution of data, load balancing, and synchronization. Many real-world applications, however, are amenable to generic approaches to these concerns: a robust, scalable implementation of algorithms for these tasks combined with a simple and accessible parallel programming model could enable these applications to be rapidly constructed and scaled up.

Another motivating problem is the increasing prevalence of heterogenous computing hardware, including GPUs and other accelerators. Use of heterogeneous hardware in a distributed-memory computing system can offer greatly improved performance and energy efficiency, particularly for floating-point intensive problems [31]. Distributed memory systems comprising many nodes with attached GPUs or other accelerators are challenging to program because multiple programming models must be composed to build a complete application, for example MPI for internode parallelism, OpenMPI [65] for intranode CPU parallelism, and CUDA [63] or OpenCL [40] for GPU programming. Furthermore, applications must manage synchronization and data movement across multiple memory spaces. Variants of the data-driven task parallel execution can address these challenges of utilizing heterogenous, distributed-memory resources with transparent data movement between devices and dynamic data-aware task scheduling. Recent work has explored implementing this execution model with libraries and conservative language extensions to C for distributed-memory and heterogenous systems [8, 17, 21, 81] and has shown that performance can match or exceed performance of code directly using the underlying interfaces (e.g., message passing or threads). One reason for this success is that sophisticated algorithms for load balancing (e.g., work stealing) or data movement, usually impractical to reimplement for each application, can be implemented in an application-independent manner. Another reason is that the asynchronous execution model is effective at hiding latency and exploiting available resources in applications with irregular parallelism or unpredictable task runtimes.

A third and final set of problems comes from the hardware constraints and limitations
that are increasingly hard to ignore: energy consumption and unreliability of hardware, particular for future Exascale high-performance computing systems [29]. Unreliability of consumer-grade hardware has been a feature of commodity clusters used by industry for at least a decade, so frameworks like MapReduce allow computations to continue running in the presence of failures. However, the most widely-used high-performance computing applications and programming models avoid the overhead of fault tolerance mechanisms aside from periodic checkpoints and still depend to a great extent on reliable hardware. This approach to fault tolerance is unlikely to be sufficient for future Exascale systems [20].

A programming model that allows computation and data to be migrated and makes data dependencies explicit may enable lightweight and scalable approaches to fault tolerance and energy efficiency by allowing migration of computation and recomputation of intermediate data upon failures.

Swift/T [96] is a scalable and high performance implementation of a high-level implicitly parallel programming language that aims to make writing massively parallel code for data-driven task parallelism as easy and intuitive as sequential scripting in languages such as Python. It has been developed since 2011 under the ExM project at the University of Chicago and Argonne National Laboratory [6]. The language design is based on the original Swift programming language implementation [92], which works well for executing scientific workflows on wide-area distributed systems but cannot support extremely scalable and efficient execution because of its unscaleable centralized design and relatively high-overhead task dispatch mechanisms. Implementing a high-level language like Swift efficiently and scalably is challenging because the programmer specifies little beyond data dependencies, which are implicitly defined by function composition or reads and writes to variables and data structures such as associative arrays. Thus, data movement, parallel task management, and memory management are left entirely to the language’s compiler and runtime system. Since large-scale applications may require execution rates of hundreds of millions of tasks per
second on many thousands of cores, this complex coordination logic must be implemented both efficiently and scalably, which requires sophisticated compilers and runtime systems.

1.1 Goals and Scope of Dissertation

My overall goal with this dissertation is to present abstractions and implementations for a high-level programming model that enables construction of realistic parallel applications that execute efficiently and scalably on massively parallel computing systems.

This work touches on multiple levels of the programming language implementation, from high-level language semantics to implementation of runtime algorithms such as reference counting. Figure 1.1 provides a visual overview of the different levels of abstraction, and shows how they apply to a concrete implementation of a programming language. The levels of abstraction provide a way to separate and reason about the different levels of the system, and also to demonstrate the broader applicability of the work to other contexts, particularly the related task-parallel execution models. The highest level of abstraction is the Swift language, in which parallelism is implicit, determinism is guaranteed, no manual memory management is required, and the data model is simple and uniform. The execution model describes the capabilities needed to implement this, including task creation, data-dependent task execution, and monotonic data types, while the concrete APIs provide a concrete interface, and implementation details such as reference counting functions to manage memory. These abstractions correspond to intermediate stages in the compilation process: the input Swift program, the internal intermediate representation used for optimization, and the generated output code for execution.

The main contributions of the dissertation are as follows:

- Abstract semantics for the data-driven tasks execution model with monotonic lattice data types.
Figure 1.1: Conceptual overview of abstractions and systems under discussion, from high-level language to low-level parallel runtime, illustrating how abstractions of different levels correspond to interfaces between different components of the system.

- Design and implementation of an efficient and scalable runtime system for this execution model, including load balancing by work-stealing, a distributed data store, and data notification mechanisms.

- Techniques for efficiently compiling the Swift language for this style of runtime system, including compiler optimization techniques that allow a range of applications developed in Swift language to efficiently execute on massively parallel distributed-memory systems.

Additional elements are also important to achieving the ultimate goal of an efficient and scalable programming language, but are not core contributions of the dissertation, including:
• Logistics of running compiled Swift code on current high-performance computing software environments.

• Aspects of compiler implementation outside of the intermediate representation and optimization (i.e., semantic analysis, translation to the intermediate representation, and code generation).

1.2 Swift Programming Language Background

The Swift programming language [92] has been successful in enabling scientific programmers without particular expertise in parallel programming to harness parallel computing resources. The original implementation of Swift (called Swift/K because it is based on a system called Karajan) was designed for coordination of large-scale distributed computations that make use of multiple autonomous computing resources distributed over varied administrative domains, such as clusters, clouds, and grids. Swift/K focused on reliability and interoperability with many systems at the expense of performance: execution of the program logic is confined to a single shared-memory master node, with calls to external executable applications dispatched to execution resources as parallel tasks over an execution provider such as Coasters [41] or Falkon [72]. Even in favourable circumstances with a fast execution provider executing tasks on a local cluster, at most 500–1000 tasks per second can be dispatched by Swift/K. This rate is insufficient for applications with more demanding performance needs such as a high degree of parallelism or short task duration. Optimizations to the language interpreter, network protocols, and other components could increase throughput, but a single master architecture ultimately limits scaling and is unsuitable for applications with tasks with durations of hundreds of milliseconds or less or with a high degree of parallelism (more than several thousand parallel tasks) [67, 57]. Thus, in order to address the needs of many demanding parallel applications, a high-performance implementation of Swift would require script execution and task management to be parallelized and
distributed across many nodes.

The Swift/T language’s syntax and semantics are derived from the Swift language. We describe syntax and semantics in more detail in Appendix A. Swift/T focuses on high-performance fine-grained task parallelism, such as calling foreign functions (including C and Fortran) with in-memory data and launching kernels on GPUs and other accelerators [47]. These foreign functions are integrated into the Swift/T language as typed leaf functions that encapsulate computationally intensive code, leaving parallel coordination, task distribution, and data dependency management to the Swift/T implementation.

1.2.1 Swift Language Semantics

Swift has been successful at enabling many programmers to parallelize their applications because Swift programs are expressed with control and data structures that are familiar and accessible to programmers of imperative languages, even though the the language is pervasively parallel. In most cases, the semantics of Swift programs remain the same if it is executed sequentially, allowing programmers to write programs without reasoning about concurrency.

In spite of Swift’s history and its somewhat unusual programming model, no serious attempt was made previously to document or formalize the semantics. This is problematic, because it makes it difficult to ensure that the language semantics and implementation are complete and correct, and also makes it difficult to extend the language while remaining consistent with existing language features.

The main features that characterize Swift are:

- Implicit parallelism and relaxed execution ordering constraints: program statements can execute out of order whenever input data is available.
- Use of data types such as single-assignment variables with the property of monotonicity,
which can ensure that results of computations are deterministic even with nondeterministic scheduling of tasks. This is discussed in more depth in the next section.

- Control structures, including conditional if/switch statements and loop constructs, that are semantically related to the equivalent imperative constructs, but are adapted for implicit parallelism and monotonic data.

Chapter 2 describes formal semantics for an execution model that captures monotonic data structures and nondeterministic execution of parallel tasks, then shows how Swift’s high-level control structures can be translated into it.

1.2.2 Monotonic Data in Swift

Swift can guarantee deterministic execution even with implicit parallelism because its standard data types are monotonic; that is, they cannot be mutated in such a way that information is lost or overwritten. A monotonic variable starts off empty, then incrementally accumulates information until it is frozen, whereupon it cannot be further modified.

Programs that attempt to overwrite data will fail at runtime (or compile time if the compiler determines that the write is definitely erroneous). Swift programs using only monotonic variables are deterministic by construction, up to the order of side-effects such as I/O. For example, the output value of an arbitrarily complex function involving many data and control structures is deterministic, but the order in which debug print statements execute depends on the nondeterministic order in which tasks run. Further nondeterminism is introduced only by non-Swift/T code, library functions such as \texttt{rand()}.  

Basic Swift variables are single-assignment I-vars [62], which are frozen when first assigned. All basic scalar primitives in Swift are semantically I-vars: ints, floats, booleans, and strings. Files can also be treated as I-vars, with an I-var in the language mirroring a file in the file-system that it is mapped to. Assigning a mapped file variable in Swift then results in a file appearing at that path.
Composite monotonic data types can be incrementally assigned in parts but cannot be overwritten. The only composite data types Swift originally supported were structs and associative arrays, both of which are monotonic [93]. The *associate array* is the most complex and heavily used composite data type in Swift. Integer indices are the default, but other index types including strings are supported. The array can be assigned all at once (e.g., `int A[] = f();`), or in parts (e.g., `int A[]; A[i] = a; A[j] = b;`). The array lookup operation `A[i]` will return when `A[i]` is set. An incomplete array lookup does not prevent progress; other statements can execute concurrently.

The Swift language guarantees that variables are automatically *frozen* when the implementation is sure that no more writes will occur. This allows Swift code to query properties such as the size of arrays and ensures that reads of non-existent array keys will eventually fail. Automatic freezing is implemented jointly with compiler analysis and runtime support in both Swift/K and Swift/T. Semantics of freezing have been subtle and poorly documented in the past, particular for Swift/K. This is problematic because programs often depend on an array being frozen to make progress: programs can deadlock if the implementation does not correctly infer that an array can be frozen. Therefore, it is particularly valuable that language semantics clarify this behavior.
CHAPTER 2

THE SWIFT EXECUTION MODEL

In this chapter we introduce an execution model for *data-driven task parallelism* and formalize its semantics. This provides a solid basis for reasoning about the semantics of Swift and implementation of compilers and runtimes for Swift. We will show that the execution model has desirable properties. In particular, subject to reasonable constraints, the result of a computation in the execution model is deterministic even when tasks concurrently execute in a nondeterministic manner while accessing a shared data store.

2.1 Overview

In data-driven task parallelism, a program is organized into *task definitions* with explicit inputs. A *task* is a runtime instantiation of a task definition with inputs bound to specific data. Once executing, tasks run to completion and are not preempted. Tasks communicate by reading and writing *shared data* that resides in a shared data store. Shared data is also the main means of synchronization: execution of a task can be made dependent on shared data so that the task does not run until its input data is ready.

To visualize the execution model, we will use a graphical notation for *trace graphs* that show tasks, shared data, and their relationships and runtime. A trace graph such

![Trace graph](image)

Figure 2.1: Trace graph showing task and data dependencies at runtime in data-driven task parallelism, forming a spawn tree rooted at task $a$. Data dependencies on shared data defer execution of tasks until the variables are frozen.
blob models[], res[];

foreach m in [1:N_models] {
    models[m] = load(sprintf("model%i.data", m));
}

foreach i in [1:M] {
    foreach j in [1:N] {
        // initial quick evaluation of parameters
        p, m = evaluate(i, j);
        if (p > 0) {
            // run ensemble of simulations
            blob res2[];
            foreach k in [1:S] {
                res2[k] = simulate(models[m], i, j, k);
            }
            res[i][j] = summarize(res2);
        }
    }
}

// Summarize results to file
foreach i in [1:M] {
    file out=sprintf("output%i.txt", i)>
    out = analyze(res[i]);
}

(a) Implicitly parallel Swift/T code.

(b) Visualization of optimized parallel tasks and data dependencies for parameters M = 2
N = 2 S = 3. Tasks and data are mapped dynamically to compute resources at run-
time.

Figure 2.2: Swift source code and data-driven task trace graph of optimized code for a simple
application.

as Figure 2.1 illustrates a single runtime execution of a program. A single static graph
cannot completely describe a data-driven tasks program because the dependencies emerge
dynamically at runtime, and tasks may selectively read, write, or spawn based on values
computed at runtime. The trace graph never has cycles: in the case of deadlocks, deadlocked
tasks will have fewer in-edges than data dependencies.

Each task can spawn asynchronous child tasks, resulting in a spawn tree of tasks as in
Figure 2.1. Shared data items can be read or written by any task that obtains a reference
to the data. Parent tasks can pass data to their child tasks at spawn time, for example
small data such as numbers or short strings, along with references to arbitrary shared data.
Shared data items provide a means for coordination between multiple tasks. For example,
a task can spawn two tasks, passing both a reference to a shared data item, which one task
reads and the other writes. Data dependencies, which defer the execution of tasks, are
the only way to synchronize between tasks. The execution model permits a task to write (or
not write) any data it holds a reference to, allowing many runtime data dependency patterns beyond static task graphs.

An example of how Swift may be translated to the execution model is shown in Figure 2.2. The illustrated application is an amalgam of several real scientific applications that runs an ensemble of simulations for many parameter combinations. The code executes with implicit parallelism, ordered by data dependencies. Data dependencies are implied by reads and writes to scalar variables (e.g. $p$ and $m$) and associative arrays (e.g. $\text{models}$ and $\text{res}$). Swift/T semantics allow functions (e.g. $\text{load}$, $\text{evaluate}$, and $\text{simulate}$) to execute in parallel when execution resources are available and data dependencies are satisfied. This example illustrates the additional expressivity of the execution model over some common alternatives such as static task graphs or dataflow networks. Simulations are conditional on runtime values: data-driven task parallelism allows dynamic runtime decisions about what tasks to create. The diagram shows an optimized translation to data-driven task parallelism. An unoptimized version would comprise more variables and tasks.

2.2 Notation

Before we start defining the execution model, we will list some of the standard notation used and define some less standard notation.

To express logical formulas, we use standard notation $\text{TRUE}$, $\text{FALSE}$, $\land$ (and), $\lor$ (or), and $\neg$ (not), $\Rightarrow$ (implies), $\forall$ (for all), $\exists$ (exists), with the usual meanings. Standard set theoretical notation is also used, e.g. $\in$ (in), $\notin$ (not in), $\cup$ (union), ($\text{intersection}$ $\subseteq$ (strict subset), $\subseteq$ (subset), $\{x \mid p(x)\}$ (set-builder notation: all $x$ for which predicate $p(x)$ is $\text{TRUE}$). $2^X$ is the set of all subsets of $X$: all possible combinations of elements of $X$. $\mathbb{N}$ is the set of natural numbers: $\mathbb{N} = \{1, 2, \ldots\}$.

To denote a $\text{function type}$ for function $f$ mapping domain $X$ to range $Y$ we use the notation $f : X \to Y$. The cartesian product is used for tuples, e.g. $f : X \times Y \to A \times B$
In addition to sets, we sometimes need ordered collections. \(\langle x_1, ..., x_n \rangle\) denotes an ordered tuple or sequence with \(n\) elements. The \(\cdot\) operator denotes concatenation of sequences: \(\langle x_1, ..., x_n \rangle \cdot \langle y_1, ..., y_n \rangle = \langle x_1, ..., x_n, y_1, ..., y_n \rangle\). \(S(X)\) is the set of all sequences of members of set \(X\): \(S(X) = \{\langle x_1, ..., x_n \rangle \mid n \in \mathbb{N} \land (\forall i \in [1, n], x_i \in X)\}\). The \textbf{Perms} function \(\text{Perms} : S(X) \rightarrow 2^{S(X)}\) enumerates all permutations of a sequence.

\textbf{Multisets} are sets that allow duplicate elements. Notation \(x \mid n \mid \in X\) means that element \(x\) has multiplicity \(n\) in multiset \(X\). \(\cup\) is the union operation for multisets, which adds the multiplicities in the two sets. Multiset subtraction \(-\) similarly subtracts the multiplicities.

\textbf{Finite maps} are functions mapping of keys to values. E.g. a finite map \(M\) from a key type \(K\) to a value type \(V\) is denoted as \(M : K \rightarrow V\). We use several pieces of notation to express finite maps. First, the value for a key \(k\) can be obtained by evaluating the finite map as a function: \(M(k)\). If \(k\) is not in the domain of the map, this is undefined. We can express a map by enumerating the keys and values: \(M = [k_1 \mapsto v_1, ..., k_n \mapsto v_n]\), or extend an existing map with \(k \mapsto v\) with notation: \(M[k \mapsto v]\). Extending a map with an existing key overwrites the previous mapping. The set of defined keys in \(M\), i.e. its domain is denoted as \(\text{dom}(M)\).

\section{Data Types and Data Structures}

In this section we will describe the basic data types used in our Swift semantics.

\textbf{Definition 2.3.1.} Let \(\mathcal{R}\) be an opaque \textbf{data representation} that can capture any form of data representation - e.g. in-memory data structures, binary strings, etc.

\textbf{Definition 2.3.2.} Let \(\mathcal{V} \subseteq \mathcal{R}\) be the set of \textbf{atomic values}. We will leave the full extent of \(\mathcal{V}\) unspecified, but it can be assumed to include data types such as integers, floating point numbers, strings. All values in \(\mathcal{V}\) must support equality comparison.

\textbf{Definition 2.3.3.} The \textbf{bottom value} \(\bot \in \mathcal{V}\) represents a value that is not yet present.
Definition 2.3.4. The top value $\top \in V$ represents an invalid value.

Out of these atomic values, we can build more complex data structures. We model all data structures as the combination of a data representations $r \in R$ with a data type that is used to interpret and manipulate the data structure.

Definition 2.3.5. Let $\mathcal{T}$ be the set of data types. Each data type $t \in \mathcal{T}$ is composed of four functions that implement operations on the data structure: $t = \langle \text{read}, \text{frozen}, \text{update}, \text{freeze} \rangle$. For convenience subscript notion is used to refer to functions for type $t$: $\text{read}_t$, $\text{frozen}_t$, $\text{update}_t$, and $\text{freeze}_t$.

Definition 2.3.6. Let $\mathcal{D} = R \times \mathcal{T}$ be the set of all typed data structures.

We will now describe the expected behavior of the functions associated with a type. As we proceed through the definitions, we will define a number of properties that a type must have to be considered a valid type.

Definition 2.3.7. $\mathcal{VT} \subset \mathcal{T}$ is the set of valid types.

Definition 2.3.8. A path is a value $p \in V$ that identifies some part of the data structure.

All of a type’s functions take a path as an argument. The way paths are treated is quite general and multiple kinds of path can be implemented within the general framework. One kind is a set of paths that describe disjoint parts of a data structure, e.g. subscripts $A[0]..A[n]$ for an array $A$ of size $n$. Another kind is a path that describes some property of a data structure. e.g. “length” could refer to the length of the array, or the path “length > 5” could be a boolean predicate that is true if the length of the array is greater than 5. In general, paths do not have to be disjoint or be in any way related to the internal structure of the representation.

Definition 2.3.9. $\langle \rangle \in V$ is a null path referring to the entire data structure.
Definition 2.3.10. The **read function** allows information to be extracted from the data structure. It takes two arguments: the data structure and a path value that identifies that information to be extracted from the data structure and returns a value. The function type is \( \text{read}_t : \mathcal{R} \times \mathcal{V} \rightarrow \mathcal{V} \)

**Property 2.3.1.** Reads of all paths in \( \top \) must return \( \top \): \( \forall p \in \mathcal{V}, t \in \mathcal{V}\mathcal{T} \Rightarrow \text{read}_t(\top, p) = \top \).

Definition 2.3.11. The **frozen function** is used to check whether a particular path is frozen, i.e. the value will not change. The function type is \( \text{frozen}_t : \mathcal{R} \times \mathcal{V} \rightarrow \{\text{True}, \text{False}\} \).

**Property 2.3.2.** For valid types, a frozen root path guarantees that all paths are frozen: \( \forall r \in \mathcal{R}, \forall p \in \mathcal{V}, (t \in \mathcal{V}\mathcal{T} \land \text{frozen}_t(r, \langle \rangle)) \Rightarrow \text{frozen}_t(r, p) \)

**Property 2.3.3.** Any path with read value of \( \top \) permanently remains in that state and is frozen. \( \forall r \in \mathcal{R}, \forall p \in \mathcal{V}, t \in \mathcal{V}\mathcal{T} \Rightarrow (\text{read}_t(r, p) = \top \Rightarrow \text{frozen}_t(r, p)) \)

Definition 2.3.12. The **update function** allows information to be added to the data structure to produce a new data structure. It takes three arguments: a data structure representation, a path, and a value that the path should be updated to. It returns a new data structure representation. The function type is \( \text{update}_t : \mathcal{R} \times \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{R} \). If the update is invalid, \( \text{update} \) returns \( \top \) to represent an invalid state.

Definition 2.3.13. The **freeze function** freezes a data structure, which prevents any future updates to the data structure. The function type is \( \text{freeze}_t : \mathcal{R} \rightarrow \mathcal{R} \).

**Property 2.3.4.** For valid types, updating or freezing \( \top \) must always result in \( \top \). \( \forall p, v \in \mathcal{V}, t \in \mathcal{V}\mathcal{T} \Rightarrow \text{update}_t(\top, p, v) = \top \land \text{freeze}_t(\top, v) = \top \).

**Property 2.3.5.** Once freeze is called on a typed data structure with a valid type, it should be frozen, not be updatable, and reads should always return the same values as before it was
frozen. Formally, $\forall r \in \mathcal{R}, \forall p, v \in \mathcal{V}, t \in \mathcal{VT} \Rightarrow$

$$
frozen_t(freeze_t(r), p) \land 
update_t(freeze_t(r), p, v) = \top \land 
read_t(freeze_t(r), p) = read_t(r, p)
$$

### 2.3.1 Equivalence of Data Structures

We will now define some additional properties of valid types that hold over sequences of updates. We will first set up some definitions that allow us to talk about equivalent representations and the possible successor representations that can be reached by applying the update function to an initial representation.

**Definition 2.3.14.** $update^*$ is a function that applies a sequence of updates defined by a sequence of (subscript, value) pairs. The type is $update^*_t : \mathcal{R} \times S(\mathcal{V} \times \mathcal{V}) \rightarrow \mathcal{R}$ and the definition is:

$$
update^*_t(r, updates) = \begin{cases} 
  r & \text{if } updates = \langle \rangle \\
  update^*_t(r', updates') & \text{otherwise}
\end{cases}
$$

where $(\langle p_1, v_1 \rangle) \cdot updates' = updates$

$$
r' = update_t(r, p_1, v_1)
$$

**Definition 2.3.15.** $update^+_t$ is a function that enumerates all possible data values that can be obtained by applying $update_t$ to $r$, i.e. the transitive closure of the update function. The type is: $update^+_t : \mathcal{R} \rightarrow 2^{\mathcal{R}}$. We define $update^+_t(r) = \{update^*_t(r, s) \mid s \in S(\mathcal{V} \times \mathcal{V})\}$.

For most data types, only a small subset of all possible update sequences result in a valid state. By Property 2.3.4, if a prefix of a sequence of updates results in state $\top$, any
subsequent updates are irrelevant as the state will remain $\top$.

Now that we have definitions that can express all of the possible states that a typed data structure can get into, we can reason about the equivalence of two data structures. There are cases where two distinct representations of a data structure may be functionally identical. For example, a set data type that supports adding items and checking if items are in the set may use different representations for the same set of items depending on the order in which they were added. If a hash table is used as the representation, the exact location of elements may depend on the order in which they were added and how collisions were resolved by the hash table implementation.

**Definition 2.3.16.** $\sim_t$ is an equivalence relation for typed data structures with type $t$. If two representations are equivalent, then read/frozen will always return the same value after applying the same sequence of updates to both representations.

Formally: $\forall r_1, r_2 \in \mathcal{R}, r_1 \sim_t r_2 \iff \forall s \in S(\mathcal{V} \times \mathcal{V}), \forall p \in \mathcal{V},$

\[
\begin{align*}
\text{read}_t(\text{update}^*_t(r_1, s), p) &= \text{read}_t(\text{update}^*_t(r_2, s, p)) \land \\
\text{frozen}_t(\text{update}^*_t(r_1, s), p) &= \text{frozen}_t(\text{update}^*_t(r_2, s, p))
\end{align*}
\]

At this point, some additional properties of valid types need to be defined to constrain their behavior further.

**Property 2.3.6.** Valid types must handle the null path so that the read returns the complete information about the typed data structure, i.e. enough information to reconstruct an equivalent representation\(^1\). Formally, $t \in \mathcal{V} \mathcal{T} \Rightarrow \exists f : \mathcal{V} \rightarrow \mathcal{R}$ s.t. $\forall r \in \mathcal{R}, r \sim_t f(\text{read}_t(r, \langle \rangle))$.

**Property 2.3.7.** After frozen has returned TRUE, the path should remain frozen and all reads should return the same value. The only exception is if an invalid update occurred,

\(^1\) This constraint is not essential to later proofs, but forces a read to the root path to return all information in the data structure, as one would intuitively expect it to behave.
which makes the data entire structure invalid. In that case, by Property 2.3.1, read must return \(\top\). Formally, \(\forall r \in R, \forall p \in V, \forall v \in VT, \text{frozen}_t(r, p) \Rightarrow \forall r' \in update^+_t(r)\):

\[
\text{frozen}_t(r', p) \land (r' = \top \lor \text{read}_t(r', p) = \text{read}_t(r, p))
\]

### 2.3.2 Lattice Data Types

In addition to the basic restrictions on data types we have already described, it is possible to define types with certain further restrictions that lead to useful properties. Previous work has shown that lattice-based data structures provide a framework that enables deterministic parallelism with data structures shared between threads [48]. A wide range of useful data structures can be fit into the lattice framework [79, 50].

In this section we describe restrictions on a data type’s functions that are necessary but not sufficient for a data type to be a valid lattice. For brevity we will refer to these data types as lattice data types even though not all lattice data types meet the requirements to be a lattice.

**Definition 2.3.17.** \(t\) is a **lattice data type** \(t \in \mathcal{LT} \subset VT\) if the \(update_t\) function conforms to the following commutativity property: \(\forall \langle r, s \rangle \in R \times S(V \times V), t \in \mathcal{LT} \Rightarrow \forall s' \in \text{Perms}(s), update^*_t(r, s) \sim update^*_t(r, s') \land update^*_t(r, s) = \top \iff update^*_t(r, s') = \top\).

Informally, this means that the application of a sequence of updates can be reordered in any way yet still obtain an equivalent final result.

Given the previous restrictions on typed data structures in general, we can now prove a significant property of lattice data types: reads of frozen paths always return a consistent value, regardless of ordering of read and update calls.

Consider a lattice data type \(t \in \mathcal{LT}\). Given \(\langle r, s \rangle \in R \times S(V \times V)\), and \(s' \in \text{Perms}(s)\). Let \(r_{\text{final}} = update^*_t(r, s)\) and \(r'_{\text{final}} = update^*_t(r, s')\). Let \(s_{\text{prefix}}\) and \(s'_{\text{prefix}}\) be any prefixes of \(s\) and \(s'\) respectively. Let \(r_{\text{prev}} = update^*_t(r, s_{\text{prefix}})\) and \(r'_{\text{prev}} = update^*_t(r, s'_{\text{prefix}})\).
Theorem 2.3.1. The **consistent frozen reads** theorem is: \( \forall p \in \mathcal{V}, \ \text{frozen}_t(r_{\text{prev}}, p) \land \text{frozen}_t(r'_{\text{prev}}, p) \Rightarrow \text{read}_t(r_{\text{prev}}, p) = \text{read}_t(r'_{\text{prev}}, p) \lor r_{\text{final}} = r'_{\text{final}} = \top \)

This is important because it tells us that the order in which updates are applied does not affect the overall result in a specific way: either all reads are consistent or the inconsistency is detected and the data structure ends up in the invalid \( \top \) state.

**Proof.** We observe that, since the final representations are obtained by applying a sequence of updates to the previous representations, that \( r_{\text{final}} \in \text{update}_t^+(r_{\text{prev}}) \) and \( r'_{\text{final}} \in \text{update}_t^+(r'_{\text{prev}}) \).

For the left hand side of the implication to be true, \( \text{frozen}_t(r_{\text{prev}}, p) \) and \( \text{frozen}_t(r'_{\text{prev}}, p) \). Therefore, by Property 2.3.7, both final representations must be frozen: \( \text{frozen}_t(r_{\text{final}}, p) \) and \( \text{frozen}_t(r'_{\text{final}}, p) \).

Definition 2.3.17 lets us divide the final states \( r_{\text{final}} \) and \( r'_{\text{final}} \) into two cases. In the first case, \( r_{\text{final}} = r'_{\text{final}} = \top \), which satisfies the right hand side of the implication in the theorem. In the other case, \( r_{\text{final}} \neq \top \) and \( r'_{\text{final}} \neq \top \). \( r_{\text{final}} \sim_t r'_{\text{final}} \) by Definition 2.3.17 because \( t \) is a lattice type. By Property 2.3.7, \( \text{read}(r_{\text{final}}, p) = \text{read}(r_{\text{prev}}, p) \) and \( \text{read}(r'_{\text{final}}, p) = \text{read}(r'_{\text{prev}}, p) \). \( r_{\text{final}} \sim_t r'_{\text{final}} \), therefore \( \text{read}(r_{\text{prev}}, p) = \text{read}(r_{\text{final}}, p) = \text{read}(r'_{\text{final}}, p) = \text{read}(r'_{\text{prev}}, p) \), and the right hand side of the implication is satisfied. \( \square \)

### 2.3.3 Lattice Data Type Examples

For the purposes of illustration, we will define some lattice data types in the framework described so far.

In these definitions we make some omissions for the sake of brevity. We omit “error handling” cases where values are out of the expected range of the functions: in those cases, it can be assumed that the function’s value is \( \top \) for read and update or \( \text{FALSE} \) for frozen. We omit definitions of frozen: it can be universally implemented by augmenting the representation with a \( \text{TRUE}/\text{FALSE} \) value indicting whether the data structure is frozen.
Definition 2.3.18. $T_\bot$ is the **bottom type**, used to represent an uninitialized value without an assigned type:

$$read_\bot(r, p) = \bot,$$

$$update_\bot(r, p, v) = \bot,$$

$$frozen_\bot(r, p) = \text{FALSE}$$

Definition 2.3.19. $T_{I\text{var}}$ is the **I-var type**: a data structure containing a single value that can be assigned at most once.\(^2\)

$$read_{I\text{var}}(r, p) = r$$

$$update_{I\text{var}}(r, p, v) = \begin{cases} v & \text{if } r = \bot \\ \top & \text{if } r \neq \bot \end{cases}$$

$$frozen_{I\text{var}}(r, p) = \begin{cases} \text{FALSE} & \text{if } r = \bot \\ \text{TRUE} & \text{if } r \neq \bot \end{cases}$$

Definition 2.3.20. $T_{\langle t_1, \ldots, t_n \rangle}$ is the **tuple type**: a tuple of other lattice data types. We represent the state as a tuple of representations. I.e. $r = \langle r_1, \ldots, r_n \rangle$. If $r = \bot$ then for convenience we assume $r_i = \bot$.

---

2. I-vars are alternatively referred to as futures in Swift
\[\text{read}_{t_1,\ldots,t_n}(r,p) = \begin{cases} 
\langle \text{read}_{t_i}(r_i,\langle\rangle) \mid i \in [1,n] \rangle & \text{if } p = \langle\rangle \\
\text{read}_{t_i}(r_i,p') & \text{if } p = i \cdot p' \land i \in [1,n] 
\end{cases}\]

\[\text{update}_{t_1,\ldots,t_n}(r,p,v) = \begin{cases} 
\langle \text{update}_{t_i}(r_i,\langle\rangle) \mid i \in [1,n] \rangle & \text{if } p = \langle\rangle \land v = \langle v_1,\ldots,v_n \rangle \land \\
\text{update}_{t_j}(r_j,p') & \text{if } j = i, r_j \text{ otherwise} \mid j \in [1,n] \land \\
\text{update}_{t_j}(r_j,p') & \text{if } j = i, r_j \text{ otherwise} \mid j \in [1,n] \land \\
\forall i \in [1,n], \text{update}_{t_i}(r_i,\langle\rangle, v_i) \neq T
\end{cases}\]

\[\text{frozen}_{t_1,\ldots,t_n}(r,p) = \begin{cases} 
\forall i \in [1,n], \text{frozen}_{t_i}(r_i,\langle\rangle) & \text{if } p = \langle\rangle \\
\text{frozen}_{t_i}(r_i,p') & \text{if } p = i \cdot p' \land i \in [1,n] 
\end{cases}\]

**Definition 2.3.21.** \(T_{t[n]}\) is the **fixed-size array type.** These arrays are also referred to as I-structures in the literature. It is simply a special case of the tuple type with the same type parameter repeated: \(T_{t[n]} = T_{t^n}\)

**Definition 2.3.22.** \(T_{t[]}\) is the **associative array type:** a finite map of keys to structures of lattice data type \(t\). This is represented as a finite map from keys to representation. I.e. \(r = [k_1 \mapsto r_1,\ldots,k_n \mapsto r_n]\). If \(r = \bot\) then for convenience we treat it as an empty mapping.
We also assume that the finite map gives a result of ⊥ if the key is not in the domain.

\[
read_t[\ ](r, p) = \begin{cases} 
[k \mapsto \text{read}_t(r', \langle \rangle) \mid k \mapsto r' \in r] & \text{if } p = \langle \rangle \\
\text{read}_t(r(k), p') & \text{if } p = k \cdot p'
\end{cases}
\]

\[
update_t[\ ](r, p, v) = \begin{cases} 
r[k \mapsto \text{update}_t(r(k), \langle \rangle, v') \mid k \mapsto v' \in v] & \text{if } p = \langle \rangle \land \\
\forall k \mapsto v' \in v, \text{update}_t(r(k), \langle \rangle, v') \neq \top & \text{if } p = k \cdot p'
\end{cases}
\]

\[
frozen_t[\ ](r, p) = \begin{cases} 
\text{FALSE} & \text{if } p = \langle \rangle \\
frozen_t(r(k), p') & \text{if } p = k \cdot p'
\end{cases}
\]

Additional collection types can be defined, including unordered \textit{sets} and \textit{multisets}. For brevity we omit the full definitions.

\textbf{Definition 2.3.23.} \(T_{\text{counter}}\) is the \textit{counter type}, which supports incrementing by a non-
negative number.

\[
\text{read}_{\text{counter}}(r, p) = \begin{cases} 
    r & \text{if } p = \langle \rangle \\
    r \neq \bot \land r \geq x & \text{if } p = \geq x \\
    \bot & \text{if } v < 0
\end{cases}
\]

\[
\text{update}_{\text{counter}}(r, p, v) = \begin{cases} 
    v & \text{if } r = \bot \\
    r + v & \text{if } r \neq \bot
\end{cases}
\]

\[
\text{frozen}_{\text{counter}}(r, p) = \begin{cases} 
    \text{FALSE} & \text{if } p = \langle \rangle \\
    r \neq \bot \land r \geq x & \text{if } p = \geq x
\end{cases}
\]

It is possible to define a range of numeric types along the same lines as the counter type with other operations:

- A data type that tracks the maximum value updated so far with the (also commutative) maximum operation.
- A counter that uses the (also commutative) multiplication operation with positive numbers.
- A counter that allows additions and subtractions, but does not support the \(\geq\) threshold operation.

**Definition 2.3.24.** \(T_{\text{File}}\) is the **file I-var type**: a data structure that mirrors the state of a file in the file system and allows it to be treated as a single-assignment I-var. The file
tracks the file path and the status of the file (whether it is present in the file system).

\[
\text{read\textsubscript{File}}(r, p) = \begin{cases} 
\bot & \text{if } r = \bot \\
\text{filepath} & \text{if } p = 'path' \land r = (\text{status}, \text{filepath}) \\
\text{status} & \text{if } p = 'status' \land r = (\text{status}, \text{filepath}) \\
r & \text{if } p = \langle \rangle \\
(U\text{NSET}, v) & \text{if } p = 'path' \land r = \bot \\
(S\text{ET}, \text{filepath}) & \text{if } p = 'status' \land r = (U\text{NSET}, \text{filepath}) \\
\top & \text{otherwise} 
\end{cases}
\]

\[
\text{update\textsubscript{File}}(r, p, v) = \begin{cases} 
(U\text{NSET}, v) & \text{if } p = 'path' \land r = \bot \\
(S\text{ET}, \text{filepath}) & \text{if } p = 'status' \land r = (U\text{NSET}, \text{filepath}) \\
\top & \text{otherwise} 
\end{cases}
\]

\[
\text{frozen\textsubscript{File}}(r, p) = \begin{cases} 
\text{FALSE} & \text{if } r = \bot \\
= (S\text{ET}, \text{filepath}) & \text{if } p = \langle \rangle \\
= (\text{status}, \text{filepath}) & \text{if } p = 'path'
\end{cases}
\]

2.4 Sequential Semantics with Nondeterministic Task Order

This section will lay out operational semantics for a basic version of the execution model without concurrency, where tasks execute in a sequential but nondeterministic order. The basic semantics also avoids any complications related to distributed execution by treating all state as global. This basic semantics is relatively easy to reason about and, as we will later show, can be extended to capture additional facets such as concurrency.

The total system state is comprised of three components: \(\langle D, W, W^{\text{fin}} \rangle\). \(D\) is the state of the data store, while \(W\) is the set of all tasks in the system, and \(W^{\text{fin}}\) is the set of tasks that have finished. We will define these components over the following sections.
2.4.1 Data Store

Let us set up the state of shared data in the system.

**Definition 2.4.1.** Let $D$ be the *data store state*. The data store is a key-value store with typed values represented as a finite map. Each item of data in $D$ is a mapping $key \mapsto d$, with $key \in \mathcal{V}$, and $d \in D$. $\mathcal{DS} = \mathcal{V} \rightarrow D$ is the set of all possible data store states.

We can refer to any path of any data item with an $(key, path)$ pair.

**Definition 2.4.2.** We define the set of all possible *data store paths* as $\mathcal{P} = \mathcal{V} \times \mathcal{V}$.

With these definitions of data store and path and our typed data structures, we can then define functions that let us access and modify the data store state:

**Definition 2.4.3.** The *dscreate* function creates a typed data structure in the data store. It has type $dscreate : \mathcal{DS} \times \mathcal{V} \times \mathcal{T} \rightarrow \mathcal{DS} \cup \{\top\}$. It updates the data store with a new key and type, returning the new data store state:

$$
\text{dscreate}(D, k, t) = \begin{cases} 
D[k \mapsto (t, \bot)] & \text{if } k \notin \text{dom}(D) \land t \neq t_\bot \\
D[k \mapsto (t, \text{update}_t^*(\bot, r_{prev}))] & \text{if } t_{prev} = t_\bot \land t \neq t_\bot \\
\top & \text{otherwise}
\end{cases}
$$

where $(t_{prev}, r_{prev}) = D(k)$

dscreate changes the type from $t_\bot$, a transition that can only happen once, and applies any accumulated updates that were stored while the type was $t_\bot$. Storing updates means that dscreate can commute with update functions, i.e. dscreate does not need to specify a type for a key before it is written. This ensures determinism in a wider range of scenarios and simplifies later proofs.

An more restricted implementation could avoid storing updates if programs are constructed so that dscreate for key $k$ is always invoked before dsupdate for key $k$. In practice
this is a reasonable constraint (and one that Swift/T follows) because the type of a key is almost always known before it is read or written.

**Definition 2.4.4.** The \( \text{dsupdate} \) function allows a typed data structure in the data store to be updated. It has type \( \text{dsupdate} : \mathcal{DS} \times \mathcal{P} \times \mathcal{V} \rightarrow \mathcal{DS} \cup \{\top\} \) and is defined as:

\[
\text{dsupdate}(D, p, v) = \begin{cases} 
D[k \mapsto \{T_\perp, \langle\langle p, v\rangle\}\}] & \text{if } k \notin \text{dom}(D) \\
D[k \mapsto \{T_\perp, r \cdot \langle\langle p, v\rangle\}\}] & \text{if } t = T_\perp \\
\top & \text{if } r' = \top \\
D[t \mapsto r'] & \text{otherwise}
\end{cases}
\]

where \( \langle k, p' \rangle = p \), \( \langle t, r \rangle = D(k) \), and \( r' = \text{update}_t(r, p', v) \)

The cases handle storing updates to keys which were not yet created, invalid updates, and valid updates to created keys.

**Definition 2.4.5.** The \( \text{dsread} \) function to reads from a typed data structure in the data store. It has type \( \text{dsread} : \mathcal{DS} \times \mathcal{P} \rightarrow \mathcal{V} \) and is defined as:

\[
\text{dsread}(D, p) = \begin{cases} 
\text{read}_t(r, p') & \text{if } k \in \text{dom}(D) \\
\bot & \text{otherwise}
\end{cases}
\]

where \( \langle k, p' \rangle = p \) and \( \langle t, r \rangle = D(k) \)

**Definition 2.4.6.** The \( \text{dsfrozen} \) function checks if data store path is frozen. It has type
\( dsfrozen : DS \times P \rightarrow \{ \text{True, False} \} \) and is defined as:

\[
ds\text{frozen}(D, p) = \begin{cases} 
\text{frozen}_t(r, p') & \text{if } k \in \text{dom}(D) \\
\text{False} & \text{otherwise}
\end{cases}
\]

where \( \langle k, p' \rangle = p \) and \( \langle t, r \rangle = D(k) \)

**Lemma 2.4.1.** Given that all data store types are lattice types, i.e. if for every \( t \) provided to \( \text{dscreate} \), \( t \in \mathcal{LT} \). Then \( \text{dscreate} \) and \( \text{dsupdate} \) operations commute: applying the operations in any order will give an equivalent same final data store state.

*Proof.* The create and update operations are applied to each key independently, so if the operations for each key commutes, then all operations commute. Therefore we can consider the sequence of calls for each key independently.

If there are two \( \text{dscreate} \) operations, the final result will be \( \top \). Either the argument to one \( \text{dscreate} \) is \( T \bot \), which results in \( \top \), or the second \( \text{dscreate} \) fails, resulting in \( \top \). Once the data store state is \( \top \), both \( \text{dscreate} \) and \( \text{dsupdate} \) always result in \( \top \). Therefore we only need consider sets of operations with a single \( \text{dscreate} \).

Because \( \text{dsupdate} \)s are saved until the first \( \text{dscreate} \) occurs, a sequence with a single \( \text{dscreate} \) and zero or more \( \text{dsupdate} \)s will be reordered so the \( \text{dscreate} \) is applied first. So the only difference between sequences is the order in which \( \text{dsupdate} \)s are applied. The commutativity of \( \text{dsupdate} \) follows directly from the commutativity of update (Definition 2.3.17).

\( \square \)

**Lemma 2.4.2.** Given that all data store types are valid types, i.e. if for every \( t \) provided to \( \text{dscreate} \), \( t \in \mathcal{VT} \). After \( ds\text{frozen} \) has returned \( \text{True} \) for a data store path, it should remain frozen and all reads should return the same value. The only exception is if an invalid update occurs, which makes the entire data store invalid. \( \forall p \in \mathcal{P} \), \( ds\text{frozen}(D, p) \Rightarrow \forall D' \in \mathcal{DS} \)
\[ dsupdatecreate_t^+(r). \]

\[ dsfrozen_t(D', p) \land (D' = \top \lor dsread_t(D', p) = dsread_t(D, p)) \]

2.4.2 Data Store Permissions

In this section we will introduce the idea of permissions that constrain whether an entity is allowed to read or write keys in the data store. Adding permissions enables reasoning about how tasks can interact and when it is safe to automatically freeze data structures. These permissions implement a basic effect system similar to other systems previously described in the literature [27, 56, 75].

Definition 2.4.7. A data store handle is a reference to a data store key with an associated permission. \( \mathcal{H} \) is the set of all data store handles, where each handle \( h \in \mathcal{H} \) is a pair \( h = \langle \text{key}, \text{permission} \rangle \), where \( \text{key} \) is the key of a data store item, and \( \text{permission} \in \{\text{Read}, \text{Write}\} = \mathcal{RW} \). \( \text{permission} \) determines whether the data referenced can be read or written through the handle: the READ permission is needed to use \( dsread \) on the key and the WRITE permission is needed to apply \( dsupdate \) to the key.

Now we assume that there is a way of encoding data store handles into a value.

Definition 2.4.8. The function \( \text{refs} : \mathcal{V} \to 2^\mathcal{H} \) extracts the set of data store handles encoded in a value. We do not specify the exact encoding, but require that \( \text{refs}(\bot) = \text{refs}(\top) = \emptyset \).

Definition 2.4.9. The function \( \text{all_refs} : \mathcal{D} \to 2^\mathcal{H} \) extracts all data store handles that could be extracted from a typed data structure.

\[ \text{all_refs}(\langle r, t \rangle) = \bigcup_{p \in \mathcal{V}} \text{refs}(\text{read}_t(r, p)) \]

3. \( dsupdatecreate^+ \) is the transitive closure of \( dscreate \) and \( dsupdate \) operations
**Property 2.4.1.** We will further constrain the behavior of valid types so that any handles added to a data structure must be explicitly added through the update. Given \( r \in \mathcal{R}, p \in \mathcal{P}, v \in \mathcal{V}, \) and \( r^\prime = \text{update}_t(r, p, v) \), then if \( t \in \mathcal{VT} \), then \( r^\prime = \top \) or \( \text{all_refs}((t, r^\prime)) \subseteq \text{all_refs}((t, r)) \cup \text{refs}(v) \)

### 2.4.3 Tasks

Now let us set up the state of tasks in the system.

**Definition 2.4.10.** Let use define **tasks**. \( \mathcal{W} \) is the set of all possible tasks. The letter \( t \) is already taken for types, so we use the letter \( w \) for “work”. Any given task \( w \in \mathcal{W} \) is a tuple \( \langle f_w, \text{handles}_w, \text{wait-data}_w, \text{read-data}_w \rangle \). \( f \) is a black box function that can represent executable code combined with data. \( \text{handles} \subseteq \mathcal{H} \) is a set of permissions to access data store keys. \( \text{wait-data} \) and \( \text{read-data} \) are both sets of paths, i.e. \( \text{wait-data} \subseteq \mathcal{P} \) and \( \text{read-data} \subseteq \mathcal{P} \).

**Definition 2.4.11.** A task’s function \( f \) maps a tuple of values to sequences of creates and updates and a set of new tasks. It has type \( f : S(\mathcal{V}) \to S(\mathcal{V} \times \mathcal{T}) \times S(\mathcal{P} \times \mathcal{V}) \times 2^\mathcal{W} \).

**Definition 2.4.12.** Let \( W \subseteq \mathcal{W} \) be the multiset of all work tasks in the program’s execution, including those that have yet to be executed and those that have already been executed.

**Definition 2.4.13.** Let \( W^{\text{fin}} \) be the multiset of all finished tasks. Tasks are added to \( W^{\text{fin}} \) as execution of the program proceeds.

Note that these are all multisets so the same task can appear multiple times in a given set. Subtraction of multisets accounts correctly for the fact that some of these tasks may have completed, while others may not have completed.

### 2.4.4 Sequential Operational Semantics

The initial state of the system is typically \((\emptyset, \{w_0\}, \emptyset)\): no data and only an initial task.
Given system state $S = \langle D, W, W^{\text{fin}} \rangle$:

\[
W^{\text{pending}} = W - W^{\text{fin}}
\]

\[
W^{\text{runnable}} = \{ w | n | \in W^{\text{pending}} \land \forall p \in \text{wait-data}_w, dsfrozen(D, p) \}
\]

\[
\text{Select-Nondet}
\]

\[
\begin{align*}
S \rightarrow \langle S, w \rangle & \quad \text{if } w \in W^{\text{runnable}} \\
\langle S, w \rangle \rightarrow \langle S, w, \langle dsread(D, p) \mid p \in \text{read-data}_w \rangle \rangle & \quad \text{Read-Data}
\end{align*}
\]

\[
\langle D, W^{\text{fin}}, w, rvals \rangle \rightarrow \langle D, W^{\text{fin}} \uplus \{ w \mid_1 \}, C, U, W^{\text{new}} \rangle & \quad \text{Exec-Task}
\]

\[
\langle D, W^{\text{fin}}, w, rvals \rangle \rightarrow \text{error} & \quad \text{Exec-Task-Fail}
\]

\[
\langle D, W^{\text{fin}}, w, rvals \rangle \rightarrow \langle D, W^{\text{fin}} \uplus \{ w \mid_1 \}, C, U, W^{\text{new}} \rangle & \quad \text{Apply-Create}
\]

\[
\langle D, W^{\text{fin}}, \{ \langle k, t \rangle \} \cdot C, U, W^{\text{new}} \rangle \rightarrow \langle dscreate(D, k, t), W^{\text{fin}}, C, U, W^{\text{new}} \rangle & \quad \text{Apply-Update}
\]

\[
\langle D, W^{\text{fin}}, \{ \} \cdot U, W^{\text{new}} \rangle \rightarrow \langle dsupdate(D, k, p, v), W^{\text{fin}}, \{ \} \cdot U, W^{\text{new}} \rangle & \quad \text{Apply-Data-Fail}
\]

\[
\langle D, W^{\text{fin}}, \{ \} \cdot \{ \} \cdot W^{\text{new}} \rangle \rightarrow \text{error} & \quad \text{Apply-Tasks}
\]

\[
D \neq \top \rightarrow \langle D, W^{\text{fin}}, \{ \} \cdot \{ \} \cdot W^{\text{new}} \rangle \rightarrow \langle D, W \uplus W^{\text{new}}, W^{\text{fin}}, \text{FREEZE} \rangle & \quad \text{FREEZE}
\]

\[
\langle D, W^{\text{fin}}, \text{FREEZE} \rangle \rightarrow \langle [ k \mapsto \text{freeze}(D, k, d) \mid k \mapsto d \in D ], W, W^{\text{fin}} \rangle & \quad \text{FREEZE}
\]

\[
\text{freez}(D, k, \langle t, r \rangle) = \begin{cases} 
(t, r) & \text{if } \exists w \in W^{\text{pending}}, \text{has-perm}(w, \text{key}, \text{WRITE}) \\
(t, \text{freeze}(t, r)) & \text{otherwise}
\end{cases}
\]

Figure 2.3: Small-step operational semantics for execution model. \textit{validate} and \textit{has-perm} are defined in Definitions 2.4.15 and 2.4.16 respectively.
The system evolves in a stepwise way with the operational semantics shown in Figure 2.3. In this basic operational semantics, one task executes at a time, with no concurrency. At each step of execution, an arbitrary runnable task is selected nondeterministically, its inputs are read from the data store, the task is executed and the system state updated. Validation occurs to ensure that the task only did valid actions it was allowed to do and that updates did not put the data store into the invalid $\top$ state.

**Definition 2.4.14.** An *error state*, denoted with *error*, is used in place of a valid system state to indicate that an error has occurred.

Checking the validity of a task’s actions is somewhat complex. Invalid creates or updates will result in an *error* state for the entire system. It is necessary to terminate execution in these cases because a previous task may have read a now-invalid value from the data structure and proceeded to make any number of additional writes. Errors could possibly be more selectively propagated by tracking the provenance of each value and invalidating all values dependent on this newly invalid value. This is left for future work.

Permissions for data store keys are somewhat more subtle. Permissions to a key are always initially obtained by a task invoking dscreate for a new key. Attempting to create the same key twice causes an error, so only one task can successfully create a key and obtain the initial permissions. After this, a key’s permissions can be transferred from parent to child task or through the data store. At each step we want to detect any violations of permissions: if a task accesses any keys it does not have permissions for, if it creates a child task with permissions it does not have, or if it tries to transfer permissions to the data store that it does not have.
Definition 2.4.15. The **validate function** determines whether a task’s actions are valid according to the permissions it has for the data store. We need to assume that all types are valid types because checking that no invalid references were transferred to a data structure relies on Property 2.4.1.

\[
\text{validate}(w, rvals, creates, updates, W^{new}) = \\
\text{read}_\text{keys} \times \{\text{READ}\} \subseteq \text{perms} \land \\
\text{update}_\text{keys} \times \{\text{WRITE}\} \subseteq \text{perms} \land \\
\forall (k, p, v) \in \text{updates}, \text{refs}(v) \subseteq \text{perms} \land \\
\forall w' \in W^{new}, \text{handles}_{w'} \subseteq \text{perms}
\]

where \(\text{create}_\text{keys} = \{k \mid (k, t) \in \text{creates}\}\)

\(\text{read}_\text{keys} = \{k \mid (k, p) \in \text{read-data}_w\}\)

\(\text{update}_\text{keys} = \{k \mid (k, p, v) \in \text{updates}\}\)

\(\text{perms} = \text{handles}_w \cup \bigcup_{v \in rvals} \text{refs}(v) \cup \\
\{\langle k, \text{perm} \rangle \mid \langle k, t \rangle \in \text{create}_\text{keys}, \text{perm} \in \text{RW}\}\)

Corollary 2.4.1. The outputs of a task: \(\langle \text{creates}, \text{updates}, W^{new} \rangle\) and the result of the validate check is a deterministic function of the task \(w\) and \(rvals\).

This is true by construction and can be verified by inspection of the EXEC-TASK rule and the validate function. The root source of any nondeterminism in the semantics is the nondeterministic selection of the next task to run. This nondeterminism can flow through to the order in which dsupdates, dscreates, and new tasks are applied to the system state, which then can flow to task definitions \(w\) and \(rvals\) in subsequent applications of \(\text{run}_\text{task}\).
Definition 2.4.16. The has-perm function determines whether a task can read or write a key directly or indirectly.

\[
has-perm(w, key, perm) = \exists (\text{root\_key, perm}) \in \text{handles}_w, \\
\text{has-perm-via(root\_key, key)} \\
\text{where has-perm-via(src, dst) = (src = dst) } \lor \\
\exists (\text{ref, perm}) \in \text{all\_refs}(D(src)), \text{has-perm-via(ref, dst)}
\]

2.4.5 Selection of Data Store Keys

One issue that has not yet been addressed is how data store keys are chosen. It is highly useful to have a mechanism to generate unique keys - this makes it easier to express many programs. In practice, unique key generation is usually implemented in a nondeterministic way, e.g. by handing out keys in a first-come-first-served manner. This complicates proof of determinism, because nondeterministic values become part of the system state and may be accessible to tasks.

For the purposes of proof, we assume that keys are generated in a deterministic way. This does not sacrifice any generality because unique keys can be generated deterministically. One possible scheme is to generate a unique task identifier based the task’s location in the spawn tree, e.g. the first task is 0, the second child task of 0 is 0.1, the first child task of 0.1 is 0.1.0, and so forth. Given a unique task identifier, a unique data key can be generated by generating a unique key within a task and appending it to the unique task identifier. Leiserson et. al. refer to this method of labelling tasks as spawn pedigree [53]. A similar technique is already used in Swift/K for the purposes of uniquely identifying variables in restart logs [93].

It would also be possible to extend the proof to nondeterministic key generation by
defining an equivalence relation over system states that is not sensitive to key values and by placing constraints on tasks that nondeterministic key values are not directly inspected or modified.

2.5 Determinism of Execution Model

Now that we have specified the operation semantics, the question remains: how does the execution model behave? In broad terms, there are three possible outcomes for the execution of a program: it can terminate with a valid state, it can terminate in the error, or it can not terminate and run for an unbounded number of steps. The execution model has nondeterminism built into the task selection, so it is far from obvious that two executions starting from the same initial state will give the same result. In this section we will prove that this, in fact, is largely the case. Specifically, we will show that:

- a program that terminates in a valid state on one execution will terminate in equivalent valid states on all executions, i.e. deterministically.
- programs that terminate in error or do not terminate on one execution may have a different outcome (error or non-termination) on different executions.
- unless the task selection policy guarantees that runnable tasks execute within a bounded number of steps.

Definition 2.5.1. An execution trace is a sequence of the tasks \( \langle w_0, w_1, ..., w_n \rangle \) in the order they were executed by iteratively applying the step function. This sequence has a corresponding sequence of states \( \langle s_0, s_1, ..., s_{n+1} \rangle \). \( s_i \) is the state before \( w_i \) was executed. If \( s_{n+1} \) is error, then the trace terminated in an error. Any non-error states are a tuple \( s_i = \langle D_i, W_i, W_i^{\text{fin}} \rangle \).

Note that the execution trace is simply an ordering of \( W_i^{\text{fin}} \).
Definition 2.5.2. A terminating execution trace is an execution trace of length \( n \) where either \( s_{n+1} = \text{error} \) or \( W^{\text{runnable}}_{n+1} = \emptyset \). \( \langle D_{n+1}, W_{n+1}, W^{\text{fin}}_{n+1} \rangle \) is the final state of the system: executing another step results in the same state.

Definition 2.5.3. Let \( P^\text{read}_i \) be the paths read up to step \( i \) of an execution trace:
\[
P^\text{read}_i = P^\text{read}_{i-1} \cup \text{read-data}_i,
\]
where \( \text{read-data}_i \) is the read-data value of \( w_i \).

For the purposes of our proofs, we will assume that all read data is explicitly waited for by the task that reads it.

Property 2.5.1. Tasks wait before reading. Given system state \( \langle D, W, W^{\text{fin}} \rangle \), \( \forall w \in W \), \( \text{read-data}_w \subseteq \text{wait-data}_w \)

Lemma 2.5.1. Consider an execution trace \( \langle w_0, w_1, \ldots, w_n \rangle \) in which Property 2.5.1 is true for all tasks. All tasks in the trace make consistent reads: each task that reads a path \( p \) sees the same value. Formally, \( \forall p \in P^\text{read}_n, \exists v, \forall 0 \leq i \leq n, p \notin P^\text{read}_i \vee v = dsread(D_i, p) \)

Proof. Let us assume for the purpose of induction that for all paths \( p \in P^\text{read}_{i-1}, \)
\[
\bullet \ dsfrozen(D_{i-1}, p) \text{ and } \\
\bullet \ \exists v, \forall 0 \leq j \leq i - 1, p \notin P^\text{read}_j \vee v = dsread(D_j, p)
\]

\( i = 0 \) is the base case for the induction. We only need consider paths read by \( w_0 \). All paths read are frozen by Property 2.5.1. There were no previous reads, so any values read by \( w_0 \) are consistent.

First we will prove the first part of inductive assumption for \( i \). All paths in \( P^\text{read}_{i-1} \) are frozen by the inductive assumption. By Property 2.5.1, all new paths read were frozen in state \( D_i \). Therefore \( \forall p \in P^\text{read}_i, dsfrozen(D_i, p) \).

Now we will prove the second part of inductive assumption for \( i \). For any new path read \( p \), either \( p \notin P^\text{read}_{i-1} \), or \( p \in P^\text{read}_{i-1} \). In the first case, \( v = dsread(D_i, p) \) satisfies the condition. In the second case, \( v = dsread(D_{i-1}, p) \), which must be defined by the inductive
assumption. \( dsfrozen(D_{i-1}, p) \) by the inductive assumption. We also know that no invalid updates occurred before the final task \( w_n \) read its inputs, because otherwise execution would have terminated in error before \( w_n \) was run. Therefore by Lemma 2.4.2, \( dsread(D_i, p) = v \) and the condition is satisfied.

**Definition 2.5.4.** Let \( V : P \rightarrow V \) be the **consistent value map** for a trace \( \langle w_1, w_2, ..., w_n \rangle \) that captures the values read by tasks. Because of Lemma 2.5.1, each path read in the trace maps to a unique value.

\[
V(p) = \begin{cases} 
  dsread(D_n, p) & \text{if } p \in P_n^{read} \\
  \bot & \text{otherwise}
\end{cases}
\]

**Definition 2.5.5.** Let \( F \) be the **frozen path set** for a trace \( \langle w_1, w_2, ..., w_n \rangle \) with all paths that were frozen at some point in the trace (excluding the final state). Because of Lemma 2.4.2, once a path is frozen it is always frozen. Therefore we can define it as

\[
F = \{ p \mid dsfrozen(D_n, p) \}
\]

**Lemma 2.5.2.** Consider two traces that start with the same initial state and task: \( \langle w_0, w_1, ... \rangle \) and \( \langle w_0', w_1', ... \rangle \). For every task \( w_i \) in the first trace, either \( w_i \) also appears in the second trace as \( w'_j \) or there is at least one **common ancestor task** \( w_k \) with \( 0 \leq k < i \) that is an ancestor task of \( w_i \) that appears in the trace as \( w'_j \).

**Proof.** Let us prove it by induction over \( i \). The base case is \( i = 0 \). The lemma is trivially true because the traces start with the same task, i.e. \( w_0 = w'_0 \).

Let us assume the lemma is true for \( 0 \) up to \( i - 1 \). There are two cases: either \( w_i \) is in the second trace or it is not. The first case directly satisfies the lemma for \( i \). In the second case, \( w_i \) is not in the second trace and was spawned by a predecessor task \( w_k \) \((0 \leq k < i)\) to which the inductive assumption is applicable. \( \square \)
Lemma 2.5.3. Consider a task $w_i$ in the scenario where $w_i$ is not in the second trace. Let $w_k (0 \leq k < i)$ be the last common ancestor of $w_i$ (the common ancestor of $w_i$ with maximum $k$). There is also a task $w_l (k < l \leq i)$ that is a child of $w_k$. Either $i = l$ or $w_l$ is an ancestor of $w_i$. $w_l$ is the first differing ancestor. $w_l$ is not in the second trace: either $w_l$ is in $W^{\text{pending}}_m$ or it was never spawned by $w_k$.

**Proof.** The previous lemma traced the lineage of $w_i$, which is not in the second trace, back to $w_k$, which is in the second trace. A task meeting the criteria for $w_l$ must exist in this lineage. There are two possibilities for $w_l$ in the second trace. If $w_l \in W'_m$, but is not in the second trace, then it must be in $W^{\text{pending}}_m$. If $w_l \not\in W'_m$, it was never spawned by $w_k$. □

Definition 2.5.6. A trace has **bounded waiting** if no task waits for more than $c$ steps to run in the trace. Formally, $\exists c$ such that $\forall i, \forall w_{|n|} \in W^{\text{runnable}}_i$ and $w_{|m|} \in W^{\text{fin}}_i \Rightarrow \exists j \leq i + c, w_{|n+m|} \in W^{\text{fin}}_j$.

Corollary 2.5.1. Any trace that terminates in a valid state after $n$ steps has bounded waiting, because $W^{\text{runnable}}_{n+1} = \emptyset$ and therefore no task waited for more than $n$ steps.

Corollary 2.5.2. Any trace that terminates in **error** does not have bounded waiting.

Definition 2.5.7. A trace $\langle w_0, w_1, ... \rangle$ is **divergent** at task $w_i$ from a second trace $\langle w'_0, w'_1, ... \rangle$ in which $V$ and $V'$ are the respective consistent values and $F$ and $F'$ are the respective frozen paths if either:

- $w_i$ is not in the second trace, i.e. $\nexists j$ such that $w_i = w'_j$.
- $w_i$ read a different value, i.e. $\exists p \in \text{read-data}_i$ such that $V(p) \neq V'(p)$.

Corollary 2.5.3. If a task $w$ is in two traces, and produces different outputs in the two traces, the traces are divergent from each other at $w$.

**Proof.** The outputs are a deterministic function of the inputs, so by Corollary 2.4.1 $w$ must have read different values in the two traces. □
Lemma 2.5.4. Differing values implies earlier divergence. Given two traces, \( \langle w_0, w_1, \ldots \rangle \) and \( \langle w_0, w'_1, \ldots \rangle \) where the second trace has bounded waiting, If \( \exists p, i, j, \) such that \( p \in P_i^{read} \) and \( p \in P_j^{read} \) and \( V(p) \neq V'(p) \), then \( \exists k < i \) such that the first trace is divergent from the second trace at \( w_k \).

Proof. First we observe that, by Property 2.5.1, the path \( p \) was frozen at steps \( i \) and \( j \) the two traces.

Let \( O \) be the set of dsupdate and dscreate operations applied before step \( i \) in the first trace. Let \( O' \) be the set of all dsupdate and dscreate operations applied at any step in the second trace. \( V(p) \neq V'(p) \), so by Lemma 2.4.1 \( O \neq O' \).

Suppose, for the sake of contradiction, that \( O \subset O' \). Then applying just the operations in \( O \) in the second trace would lead to the same value for \( p \) in the second trace, which must also be frozen. Applying the additional operations in \( O' \) would either lead to the same value for \( V'(p) \) or a data store state of \( \top \) and therefore error by Lemma 2.4.2. Neither of these is true, therefore we have a contradiction and \( O \not\subset O' \).

This means that there is an \( o \) that is in \( O \) but not \( O' \). This \( o \) must have come from a task \( w_j \) with \( j < i \) where the first trace is divergent. \( \square \)

Lemma 2.5.5. Given two traces, \( \langle w_0, w_1, \ldots \rangle \) and \( \langle w_0, w'_1, \ldots \rangle \) where the second trace has bounded waiting, if there is path \( p \) in \( wait-data_i \) for \( w_i \) in the first trace where \( dsfrozen(D_i, p) \) and \( p \notin F' \), then \( \exists j < i \) such that the first trace is divergent at \( w_j \).

Proof. Let \( p = \langle k, p' \rangle \).

Case 1: \( k \) is not writable at step \( i \) of first trace.

Case 1a: \( k \) was not created by the same task in both traces.

We know there is some \( j < i \) where \( w_j \) in the first trace output a dscreate call for \( k \). If this task is not in the other trace, it is divergent. If it is in other trace, it is the only task in the second trace that created \( k \): otherwise the double create would have caused an error and the second trace would not meet the bounded waiting criteria.
**Case 1b:** $k$ was created by the same task in both traces.

$k$ is not frozen in the other trace and therefore is writable. There must be some difference in the second trace that keeps it writable. Either this is a task that has a write permission for some key that it didn’t in the first trace, or a data structure holds a write permission for some key that it didn’t in the first trace. Because the write permission for $k$ originated in the same task in both traces, there must be some unbroken chain of permission propagation via data structures and tasks that links it back to the task that created $k$. There must be some task $w_j$ in the first trace that did not propagate a write permission that it did in the second trace. $j < i$ because $k$ was not writable by step $i$ in the first trace and therefore no tasks directly or indirectly held the write permission for $k$.

**Case 2:** $k$ is writable at step $i$ of first trace.

When the path $k$ is not frozen with a freeze operation, then the path must be frozen in the first trace purely from the set of operations applied $O$. $\text{dsfrozen}$ follows the same commutativity rules as $\text{dsread}$ in this case, so by the same argument used in Lemma 2.5.4, the first trace is divergent at $w_j$ with $j < i$.

**Lemma 2.5.6.** Given two traces, $\langle w_0, w_1, ... \rangle$ and $\langle w_0, w'_1, ... \rangle$, if there is a task $w_i$ ($i > 0$) in the first trace that is not in the second trace and the second trace has bounded waiting, then $\exists j < i$ such that the first trace is divergent at $w_j$.

**Proof.** By Lemma 2.5.2, there is a common ancestor task $w_k$ of $w_i$ and a first differing ancestor $w_l$. This lemma gives us two cases to consider:

- $w_l$ was spawned but never ran: $\exists n, \forall m \geq n, w_l \in W_m^{\text{pending}}$. Because we assumed bounded waiting, the task cannot be in $W_m^{\text{runnable}}$ indefinitely. Therefore there is a path $p \in \text{wait-data}_l$ where $p \notin F', \text{ but } p \in F$ because $w_l$ ran in the first trace. By Lemma 2.5.5, the first trace is divergent from the second at a task $w_j$ with $j < l \leq i$.

- $w_l$ was never spawned: the output of $w_k$ was different between the two traces. By
Corollary 2.4.1, a read to path $p$ by $w_k$ yielded a different value in the two traces. Therefore $V(p) \neq V'(p)$ and the first trace is divergent from the second at $w_k$ with $k < i$.

\[ \square \]

Lemma 2.5.7. Given two traces $\langle w_0, w_1, ..., w_n \rangle$ and $\langle w'_0, w'_1, ... \rangle$, where the second has bounded waiting. If the first trace is divergent from the second at $w_i$, then the initial states of the two traces $\langle D_0, \{w_0\}, \emptyset \rangle$ and $\langle D'_0, \{w'_0\}, \emptyset \rangle$ are different.

Proof. We prove this by induction on $i$. The base case $i = 0$ is trivial: the traces can only diverge at $w_0$ if the initial states are different: either $w_0 \neq w'_0$ or $D_0 \neq D'_0$. Our inductive assumption is that the lemma is true for $0$ up to $i - 1$.

If $w_i$ is not in the second trace, then by Lemma 2.5.6, the first trace is divergent at the second at $w_j$ with $j < i$. If $w_i$ read different values in the second trace, then by Lemma 2.5.4, the first trace is divergent at $w_j$ with $j < i$. In both cases we can apply the inductive assumption to prove it for $i$.

\[ \square \]

Lemma 2.5.8. If a trace $\langle w_0, w_1, ..., w_n \rangle$ terminates with error but another trace $\langle w'_0, w'_1, ... \rangle$ does not terminate with error and has bounded waiting, then the first trace is divergent from the second trace at a task $w_i$.

Proof. There are two categories of immediate reasons why $w_n$ failed:

- $w_n$ did something that is invalid in isolation, e.g. wrote something it had no permission for, read or wrote a bad path, updated with a bad value, or created an invalid child task.

- $w_n$ output a dscreate/dsupupdate that conflicted with one or more dscreates/dsupdates output by a set of other tasks. The order in which the dscreate/dsupdate operations were applied does not matter because of Lemma 2.4.1.
$w_n$ did not fail in the second trace, so either:

- $w_n$ was not in the second trace.
- $w_n$ produced different outputs.
- $w_n$ produced the same outputs, but at least one other task $w_i$ ($0 \leq i < n$) in the first trace was not present in second trace or produced different outputs.

In all of these cases a task $w_i$ produced different outputs, and therefore was divergent by Corollary 2.5.3, or a task $w_i$ was not present and therefore is divergent by definition.

**Definition 2.5.8.** Two data store states $D$ and $D'$ are equivalent: $D \sim_{DS} D'$ iff $\forall (k, r, t) \in D$, $\exists (k, r', t') \in D'$, s.t. $t = t'$ and $r \sim r'$, and vice-versa.

**Theorem 2.5.1.** Execution is deterministic if program terminates successfully and all data store types are lattice types. Given any two execution traces with same initial state $\langle D_0, W_0, W_{fin}^0 \rangle$. If the first trace terminates with a valid state $\langle D_{n+1}, W_{n+1}, W_{fin}^{n+1} \rangle$, then the second trace will terminate with an equivalent valid state $\langle D_{m+1}', W_{m+1}', W_{fin}^{m+1}' \rangle$, where $n$ and $m$ are the number of tasks in the first and second trace. That is, $D_{n+1} \sim_{DS} D_{m+1}'$, $W_{n+1} = W_{m+1}'$, and $W_{fin}^{n+1} = W_{fin}^{m+1}'$.

**Proof.** Let us assume for the sake of contradiction that the theorem is false. There are then three possible outcomes of the second trace:

1. It terminates in a different state after $m$ steps.
2. It terminates in error after $m$ steps.
3. It does not terminate.

In case 1, there are three overlapping subcases: $D_{n+1} \sim_{DS} D_{m+1}', W_{fin}^{n+1} \neq W_{fin}^{m+1}'$, or $W_{n+1} \neq W_{m+1}'$. The first subcase in fact subsumes the remaining two cases. In the second
subcase, $D_{n+1} \overset{D_{S}}{\sim} D'_{m+1}$ follows from Lemma 2.5.6. In the third subcase, we assume $W^\text{fin}_{n+1} = W^\text{fin}_{m+1}$ (otherwise it is subsumed by second subcase). Therefore $W^\text{pending}_{n+1} \neq W^\text{pending}_{m+1}$. By Lemma 2.4.1, if the final data store state is different, then a different set of dscreate/dsupdate operations was applied. This means that either a task that appears in both traces $w_i = w'_j$ produced different output or a task in one trace $w_i$ or $w'_j$ is not in the other. I.e. one trace is divergent from the other at some task. Both traces complete normally, so by Corollary 2.5.1, they both have bounded waiting. Lemma 2.5.7 can therefore be applied to obtain the contradiction for case 1 that the initial states of the two traces are different.

In case 2, by Lemma 2.5.8, the second trace is divergent from the first.

In case 3, the second non-terminating trace is infinitely long, so must either include tasks not in the first trace (in which case we apply Lemma 2.5.6 as before) or include duplicates of one of the tasks in the first trace. If all tasks from the second trace are in the first trace, yet there are infinitely many, then one of the tasks is producing additional tasks as output. Therefore the second trace must be divergent from the first.

Since the first trace has bounded waiting and the second trace is divergent in cases 2 and 3, we can apply Lemma 2.5.7 to obtain the contradiction that the initial states of the traces are different.

Now it remains to consider the remaining possible outcomes of execution: non-termination and termination with error. Lemma 2.5.1 implies that if two traces start with the same state and one does not terminate or ends in the error state, then the other does not terminate in a valid state. Ideally both traces would have the same outcome (i.e. non-termination implies non-termination and an error implies an error). We will show that this is true under some assumptions and not under others.

**Theorem 2.5.2.** If there are task functions that do not terminate execution, then two traces with the same starting state can have different outcomes: non-termination and error.
Proof. Consider a system with two tasks eligible to run: one that does not terminate, and another that causes an error. If the first is selected, execution does not terminate. If the second is selected, execution terminates in an error.

From now on we will assume that computation of task functions terminate.

**Theorem 2.5.3.** If tasks are selected to run according to an arbitrary policy, then two traces with the same starting state can have different outcomes: non-termination and error.

**Proof.** Consider a system with a single I-var $X$ and three types of task:

- Task A: assigns $X ← 0$
- Task B: assigns $X ← 1$
- Task C: once $X$ is frozen, if $X = 1$, spawn Task C

If the system is in a state with only A and B runnable and C pending, there are two possible outcomes depending on the order in which tasks are selected to run. If task A and task B are both run, then the double assignment to $X$ gives error. However, Task B is run before Task A and if a copy of Task C is always chosen before Task A, then execution will not terminate because Task B will never run. The execution trace would be $⟨B, C, C, C, C, C, C, ...⟩$.

Note that this is consistent with realistic task selection policies like last-in first-out (LIFO). Other task selection policies can avoid this problem. What we really need to avoid the divergent behavior of error versus non-termination is a task choice policy that guarantees that tasks become runnable within a bounded number of steps.

**Definition 2.5.9.** A task selection policy for choosing tasks to execute from $W_{\text{Runnable}}$ guarantees **bounded waiting** if $\exists c$ such that $\forall w \in W_{\text{Runnable}}$, $w$ will be executed within $c$ steps.

An example of a policy with bounded waiting is first-in first-out (FIFO): $c$ is bounded by the number of currently runnable tasks.
Theorem 2.5.4. If a trace \( \langle w_0, w_1, ..., w_n \rangle \) terminates in \textbf{error}, then another trace \( \langle w_0, w'_1, ... \rangle \) with the same initial state and bounded waiting also terminates in \textbf{error}.

\textit{Proof.} Theorem 2.5.1 implies that the second trace does not terminate successfully (otherwise the first trace would have terminated successfully). Therefore we only need eliminate the possibility that the second trace does not terminate. Let us suppose, for the sake of contradiction, that this is the case.

By Lemma 2.5.8, there is a task \( w_i \) where the first trace is divergent from the second. We have bounded waiting in the second trace, so by Lemma 2.5.7 the initial states were different - a contradiction. \( \Box \)

Corollary 2.5.4. If a trace with bounded waiting does not terminate, then any other trace with the same initial state also does not terminate.

As an final diversion, we will quickly look at an alternative task selection policy: random task choice, where tasks are selected with equal probability. This is is a superficially attractive model and can avoid non-termination in some circumstances. In the above example in Theorem 2.5.3 with tasks A, B, and C, the probability of eventually selecting A is \( \frac{1}{2} + \frac{1}{4} + ... \), which converges to 1. However, if Task C was modified to spawn enough children that the number of runnable tasks at step \( i \) is at least \( 2^i \), then the probability of eventually executing Task A is \( < \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... = \frac{\pi^2}{6} - 1 \approx 0.645 \). This possible non-termination a problem in theory only: a task queue that grows every step will cause any real system to eventually run out of memory, a kind of \textbf{error} the model does not capture!

2.6 Extensions to Semantics

There are a number of extensions or modifications to the core semantics that are useful in modelling more realistic implementations of the execution model or reasoning about transformations of programs in the execution model.
2.6.1 Concurrent Execution of Tasks

An execution model that only allows tasks to execute sequentially is of limited use to us. However, the properties of the sequential execution model generalize to the same execution model extended with concurrent execution of tasks. We will consider two extensions and argue informally that this is the case.

The first extension to consider is concurrent execution of tasks with atomic reads and writes of values at the beginning of the end. That is, at each step of execution, a task can be started, its inputs read and the task added to the set of running tasks, or a running task can finish and have its outputs written. This behavior can be accurately simulated within the original execution without overlapping if each task is split into two tasks. The initial task reads inputs and spawns the second continuation task passing along the values it read. The continuation task then computes the original function and produces outputs. Figure 2.4 illustrates this transformation. Any valid concurrent trace of tasks can be mapped to a
sequential trace of these split tasks. The sequence of reads and writes is identical in both
cases and therefore the end state is identical.

The second extension is to allow tasks to read and write values throughout their execution,
rather than just at the beginning and end. Tasks must still follow rules about not reading any
non-frozen data. A similar argument to concurrent execution of tasks applies - a task that
does multiple reads and writes throughout its execution can be broken up into an initial task
and a sequence of continuation tasks, each of which only reads at its beginning and writes
at its end. Therefore everything we have proven about tasks that execute sequentially with
atomic reads and writes also applies to tasks that read and write values throughout their
execution.

The second extension also allows a single task to follow multiple references, i.e. issue a
read for something it has a handle, then issue another read to a newly obtained reference.

2.6.2 Removing Waits

In some cases constraints imposed on behavior of programs and valid system states are
overly strict in order for the sake of simplicity. For example, Property 2.5.1 requires tasks to
explicitly wait for data to be frozen before it is read. There are various ways that programs
can break this rule without affecting behavior.

A \langle key, path \rangle pair can be removed from a task’s wait-data without altering program
behavior if, at the time the task is created, \textit{frozen}(key, path) is \textsc{true} in all possible traces
of the program. This is the case when:

- an ancestor task wrote the key and the data type’s semantics guarantee it is frozen, or
- the task was in wait-data of an ancestor, or
- either of these things is true for a task the current task is dependent on via data, or
• in general, any chain of spawn and data dependencies leads to a task with \( \langle \text{key}, \text{path} \rangle \) in \textit{wait-data} or a task which wrote the key in such a way that it is frozen.

2.6.3 Garbage Collection and Freezing

The semantics so far described do not directly describe a practical implementation of a data store. Directly implementing the semantics is problematic. No information is removed from the data store, so its size could grow boundlessly. Furthermore, a naïve implementation of the algorithm to detect freezing would require processing a number of keys up to the size of the data store at every step.

In our semantics, the data store monotonically increases in size as a consequence of the lattice semantics and the fact that keys are not removed: information is not removed from the data store. To allow practical implementation, unneeded data structures need to be removed: garbage collection [44] is needed. The problem superficially is straightforward: if no tasks have permissions for a key, then the associated data structure can be garbage collected. This can be formulated in a similar way to freezing: if no pending tasks have \textsc{Read} or \textsc{Write} permissions to access a key (directly or via another data structure), then garbage collection has no effect on the programs execution. However, the semantics of \textsc{dscreate} present a problem: \textsc{dscreate} must return \( \top \) if the key was previously used. To faithfully implement this, the system needs to track the set of keys that have been used at any point, which could require storing an unbounded amount of information.

Suppose that all information associated with a key is removed when it becomes inaccessible, i.e. \( D(k) \) is reset to \( \langle T_\perp, \langle \rangle \rangle \). This can lead to a race condition where a second \textsc{dscreate} races with the garbage collection of the key. This problem can be worked around by constructing programs to avoid this kind of data race. For example, the program could be constructed so that \textsc{dscreate} is invoked at most once for any key, for example by using a unique key generation algorithm like the one described in Section 2.4.5. Alternatively, pro-
grams could reuse keys but include sufficient synchronization so that each key is guaranteed to be reset before it is reused, e.g. if all uses of one version of a key are in predecessor tasks of the task that invokes `dscreate` to create a new version of the key.

Many approaches to garbage collection, and therefore freezing, are known that are more efficient than a naïve algorithm of scanning all references at each step [44]. One approach is to use reference counting to track the number of active read/write references to each key. The could be implemented in the execution model by updating reference counts every time a task duplicates or releases a reference. The major limitation of reference counting is that it fails to reclaim memory in the presence of cycles of references. In Swift/T this is not a problem because the type system prevents construction of reference cycles.

### 2.6.4 Event Handlers and Iteration over Collections

One pattern that cannot be directly expressed in the basic execution model is iteration over a collection of values in a data structure, where elements are gradually added to the collection. For example, given an associative array, we may wish to run a task every time a key is assigned. We can model this with **event handlers**, similar to the event handlers proposed for LVish [51].

Let $E \subseteq V$ be an **event space** associated with a typed data structure $(r, t)$. $E$ is logically a set of paths into the data structure. For example, $E$ could simply be all the possible keys that might be assigned in an associative array. an **event handler function** $f$ is defined for the event space, a task $f(e)$ should be created whenever a path $e \in E$ becomes frozen. When the event handler is initially created, tasks must be created for all already-frozen events.

Extending the execution model with event handlers would complicate proofs. However, it is possible to simulate event handlers in the existing model and therefore show that the determinism results also apply to the model extended with event handlers. We propose two ways to do this.
The first, simpler, way, is to simulate an event handler by enumerating every value \( e \in E \) and creating a task \( f(e) \). \( E \) is an infinite set in many cases, so an infinite number of tasks will remain unexecuted when a program terminates.

A second way of simulating event handlers avoids the need for an infinite number of tasks in the system state. Let us assume that there is an arbitrary ordering of values in \( V \) and that a successor function \( succ(v) \) is defined. Then we can simulate an event handler for event space \( E \) in data structure \( (r, t) \) with the following process:

- Modify \( t \) so that the following paths implementing boolean predicates are defined:
  
  \( x \in E \), \( \exists e \in E \text{ s.t. } e > x \), and \( \exists e \in E \text{ s.t. } e < x \)

- When the event handler is created, choose an arbitrary value \( v_0 \in I \), and create three tasks that will run when the following subscripts are frozen: A) \( e_0 \in E \), B) \( \exists e \in E \text{ s.t. } e > e_0 \), and C) \( \exists e \in E \text{ s.t. } e < e_0 \)

- Task A runs the actual event handler function for \( e_0 \)

- Task B creates two more tasks for the subscripts \( succ(e_0) \in E \) and \( \exists e \in E \text{ s.t. } e > succ(e_0) \). Thus if a path in the event space \( x > e_0 \) becomes frozen, new tasks will be created one-by-one until the actual event handler task for \( x \) is created.

- Task C is the mirror-image of task B.

### 2.7 Mapping Swift to Execution Model

In Section 2.1 we described the relationship between Swift code and the execution model and illustrated in Figure 2.2 how code might be translated in an optimized way. Appendix A gives an overview of the syntax and semantics of Swift/T.

Swift/T compiles to a subset of valid programs in this execution model, and relieves a programmer of many burdens they would face if programming to the execution model.
directly. In this section we will sketch the strategy we use for translating Swift programs to the execution model. The execution model has many similarities to Swift and provides the main features needed to implement Swift, but there remains a significant gap in semantics between of Swift and the execution model. The abstract execution model is much lower level than high-level programming models such as the Swift language: there is no high-level syntax and fewer protections against expressing incorrect code that will cause runtime errors or produce incorrect results. A task can easily produce invalid output, for example writing data that does not exist, or a cause a race condition if shared data is read without synchronizing using data dependencies.

The abstract execution model omits some complications that arise when implementing the model in practice. In our abstract execution model, we assume that each task executes an arbitrary function with no side-effects beyond well-defined outputs to allow reasoning about the behavior. In practice the majority of tasks in a Swift application fit this model. However, in realistic applications tasks can execute arbitrary code that performs arbitrary computation, I/O, and runtime operations such as spawning tasks or reading/writing data. A real implementation also requires explicit bookkeeping for memory management and correct freezing of variables. Programming errors could result in memory leaks, prematurely freed data, or deadlocks.

There are many possible ways to translate a Swift program to the execution model. We approached the problem by initially implementing a naïve strategy that directly translates each program variable to a runtime shared data structure, and each function call or operation to an asynchronous task. This is clearly sub-optimal given that shared data structures and tasks incur runtime overhead to create, but makes it simple to correctly produce a correct implementation. Compiler optimizations, described in a later chapter, are used to get from the inefficient naïve implementation to an efficient implementation.
The naïve translation of Swift to the execution model is as follows:

- Swift data types all meet the requirements for lattice data types.
- For each block of code and each declared variable and temporary intermediate variable, dscreate is used to create a corresponding shared data structure in the store.
- Each statement in a block separated by `;` is spawned as a task with inputs every variable referenced and outputs any variables assigned in the statement.
- Every input read by a task is explicitly waited for.
- Each function call (including Swift functions, command-line applications, and foreign language functions) is executed in a separate task. For recursive function calls, this functions as a trampoline, enabling tail recursion without stack growth.
- If the statement has an expression with multiple subexpressions, a task is spawned to compute each subexpression and store the result to a temporary shared data structure.
- Conditional if and switch statements are implemented as a task that waits for then reads a condition then executes the appropriate block.
- Wait statements are implemented as a task that executes the block of code with the variables waited added to the task’s `wait-data`.
- Parallel foreach loops are implemented by spawning a task per iteration or by using recursive splitting.
- Foreach loops over containers are implemented using event handlers.
- Ordered for loops are implemented as a chain of tasks, with each iteration spawning the next iteration. Each iteration waits for the loop condition before executing.
- The READ/WRITE handles for each task are determined with static analysis by inspecting inputs and outputs of statements and blocks of code. We rely on the fact that functions explicitly declare inputs and outputs and cannot write their inputs.
int A[];
A[0] = size(A);
trace(A[0]);

Figure 2.5: Example Swift code that deadlocks in Swift/T but could potentially execute successfully.

2.8 Limitations and Future Work

2.8.1 Permissions Model

The described mapping works well and is used in Swift/T.

The semantics contributed in this dissertation formalize the automatic freezing behavior using a system of permissions and have enabled improvements to the automatic freezing behavior in both Swift/T and Swift/K to handle far more cases correctly.

However, there are some limitations that arise as a consequence of how automatic freezing only happens at key granularity. In this section I will discuss some limitations that arise when the value of a path in a data structure depends in some way on the value of a different path in the same data structure.

For example, the program in Figure 2.5 deadlocks (i.e. terminates with unexecuted tasks) in the current version of Swift/T because size(A) only returns when the whole of A is frozen and A is never frozen because assignment of A[0] only happens after size(A) returns.

However, basic inspection reveals that it would be entirely reasonable for the program to print “1”: it is straightforward to infer statically that the statement will assign exactly one subscript, 0, of the array. Exploiting this in the execution model would require extensions to support a partially frozen state where the set of paths to be assigned (or the number of paths to be assigned) is fixed but the values are not yet known. Achieving this would require more granular tracking of what effects unexecuted tasks like A[0] = size(A).

More subtle issues occur with multi-dimensional arrays. In Swift/T, these are implemented as nested arrays with outer arrays holding references to inner arrays. Inner arrays
```swift
int A[];
int N = 10;

foreach i in [0:N] {
    foreach j in [0:N] {
        if (i == 0) {
            // Random integer from 1 to 100
            A[i][j] = randint(1,100);
        } else {
            // Depends on entire previous inner array
            int prev_sum = sum(A[i-1]);
            A[i][j] = f(A[i-1][j], prev_sum);
        }
    }
}
```

Figure 2.6: Example Swift code that deadlocks at some optimization levels in Swift/T but executes successfully at high optimization levels.

are not frozen until the outer arrays are frozen

4, so as long as a write reference to the outer arrays is held, no inner arrays can be frozen.

In the code shown in Figure 2.6 data dependencies flow in one way: from \( i - 1 \) to \( i \). However, each subarray \( A[i] \) is not frozen until the root array \( A \) is frozen, which will not happen if the code holds onto a writable reference to \( A \). At lower optimization levels, a writable reference to \( A \) is held so that \( A[i][j] \) can be assigned in the innermost loops. This prevents subarray \( A[i] \) with \( i > 0 \) from being filled in because \( \text{sum}(A[i-1]) \) cannot be computed until \( A[i-1] \) is frozen. At higher optimization levels this example runs to completion in Swift/T because writes to the outer array \( A \) are hoisted out of the inner loop body, which allows the outer array to be frozen before all assignments to the inner arrays finish.

Having progress guarantees depend on code optimizations that are best-effort is undesirable: the language semantics are implementation-dependent. There is potential to address this in the execution model with more granular tracking of which paths of a data structure are writable by each task, formulated in such a way that progress is predictable and well-

4. a simplification - if the inner array is assigned in whole, then the inner array may be frozen
defined. We leave this to future work, but note that there is relevant work in the literature on more sophisticated effect systems that may be applicable to this problem [13, 27, 56, 75].

2.8.2 In-place Updates and Linear Types

A practical limitation of the execution model is that lattice types do not allow in-place destructive updates to data. In the context of high-performance distributed systems working with large data such as matrices, this is a major efficiency issue.

Fortunately, there is substantial existing work on linear type systems that could be exploited to allow in-place updates without giving up desirable properties like determinism [16]. The essence of linear type systems is that the value of linearly typed variables is only allowed to be read once. This means that any memory associated with the variable can safely be reused once the variable is read, thereby enabling in-place or destructive updates. In the context of data-driven tasks, this could be implemented through restrictions on duplication and transfer of READ handles to linearly typed keys.

2.9 Related Work

A fundamental and challenging problem in parallel programming is how to reason about the behavior of concurrent executing threads of execution with shared, mutable state, given the exponential number of potential interleavings of the concurrent threads. This problem is unavoidable with implicitly parallel programming models like Swift, because implicit parallelism leads to a high degree of implicit concurrency, with program variables and data structures visible to any thread within the scope. In such situations, if concurrent threads are allowed to make arbitrary mutations to shared state, it can become extremely difficult to reason about what values will be read by other threads: indeed, it is nondeterministic and can vary from one execution of the same program to another.

A general approach to dealing with this problem is to restrict the ways in which shared
state can be read and written. Restricting writes to shared state can directly reduce the size of the space of possible states by disallowing some state transitions. Restricting reads to shared state can prevent different states from being distinguished, which can help states converge rather than diverge. A particularly strong guarantee is that all reads will return the same value on every execution of the program regardless of how execution of the threads is interleaved. If this is guaranteed, it is straightforward to construct deterministic programs.

Single-assignment data is a well-known way of achieving determinism in parallel programs that trivially fulfills this requirement because each data storage location only ever has a single value and all reads to that location are guaranteed to return that value. This is generally implemented by blocking threads of execution until the location is written. Single-assignment data has been used by many systems over at least three decades, for example I-vars and arrays of I-vars [7].

In recent years, multiple research groups [24, 51, 79] have pursued such an approach by defining data types with a property of *monotonicity* that is more general than single-assignment data. Intuitively, monotonicity means that a write to the data type cannot overwrite information from a previous write. Constraining data types to be a lattice (or have some properties of a lattice), is a natural way to achieve monotonicity. This can guarantee that writes are be commutative, i.e. applying a set of writes results in the same final state regardless of order. This in itself does not guarantee determinism because readers can still observe the order of state transitions. Additional constraints can prevent intermediate states or the order in which writes were applied from being observable by readers.

Recent work has also explored monotonic data types for distributed systems, where they can enable eventual consistency in distributed systems with replicated state. Conflict-free replicated data types [79] (CFRDT) for distributed systems allow different members of a distributed system to have different views of shared state, but guarantees that the views will converge to a consistent state under certain additional constraints. This allows a scal-
able, high-performance implementation of these data types where writes to are made to local replicas of the data and asynchronously replicated to other replicas. Observation of intermediate states is allowed, so reads are nondeterministic, but the lattice structure guarantees convergence to a particular state. Kuper [49] compares CFRDTs and LVars in detail.

2.9.1 Comparison with LVars

Our semantics has many similarities with and several differences from the LVars/LVish semantics of Kuper et al. [51]. This section describes the major differences between their LVar and our lattice data types variables and in operational semantics of the execution models.

The two semantics were developed concurrently: our work is a formalization and extension of the behavior of Swift [93, 96] and Turbine [95]. Previous work on CFRDT and LVars was very helpful in guiding our formalization. Our overall approach to specifying the semantics is slightly different from LVars: we tend to rely on “black box” functions rather than lambda calculus and reduction rules. This reflects its history as a formalization of the runtime system described later in the thesis, and makes the connection to the runtime system API described in the next chapter much clearer.

One significant difference is that LVars are required to be a join semi-lattice, while our lattice data types only meet some of the requirements. The difference is that puts in LVars are idempotent while operations on our lattice data types are not required to be idempotent. That is, applying the same put a second time has no effect in LVars, but can have an effect in some lattice data types.

The idempotence requirement has practical implications. Idempotence means that tasks can be reexecuted safely, which can be leveraged in some implementations of work-stealing [58]. However, idempotence is a barrier to implemention of some behaviors. For example, if I-var \( V \) is updated with \( V \leftarrow x \) then \( V \leftarrow y \), in our lattice data types (and the original work on I-vars) the result is always \( \top \). However, idempotence in LVars requires that that \( V = x \) if
$x = y$ and $V = \top$ otherwise. We believe that non-idempotent I-vars are generally preferable: programs are easier to reason about if double assignments consistently cause an error.

Another data type that cannot be implemented directly in LVars is a counter with atomic increments. The LVar formalism is still, however, equivalently powerful in a setting with no duplication of work: it is possible to simulate non-idempotent operations in an LVars by tagging every put value with a unique identifier, for example, using our unique key generation method. The unique identifier can simply be ignored by the implementation, but this gets around the idempotence requirement by guaranteeing that two put arguments are never identical. Kuper et. al. also identified the limitations of idempotence in subsequent work and have proposed extensions to LVars with non-idempotent operations [50].

LVars and join semi-lattices also support a join operation that merges two values. So far, we have not introduced a join operation for lattice data types, only incremental updates. In contrast, all puts in LVars are (formally) joins. A join operation can be defined for lattice data types. Consider two lattice data structures with type $t$ constructed with update sequences $u_1, u_2 \in S(V \times V)$. Let $r_1 = t, update_t^*(\top, u_1)$ and $r_2 = t, update_t^*(\top, u_2)$. Then $join_t(r_1, r_2) = update_t^*(\top, u_1 \cdot u_2)$. The associativity and commutativity of join follows from commutativity of update. This connection is very similar to a connection drawn by Shapiro et. al between op-based and state-based CRDTs. They prove that each CRDT variant can be emulated by the other in a distributed system [79]. We have not explored use of join operations yet in practice, but it is almost certainly applicable to implementing distributed lattice data structures.

Another difference of LVars is that the get operation is a blocking threshold reads, while we treat all reads as non-blocking. The same effect is achieved by our initial assumption that tasks wait before reading (Property 2.5.1), however, within the model we can easily describe programs that relax this restriction. The get operation is formalized differently: in the LVars formalism it takes a threshold set that explicitly enumerates lattice values. In
practice, though, LVars is implemented with operations that merely behave in an equivalent way to avoid the need to enumerate the sets.

Both our semantics and LVars define a store, which is a map from keys (or locations) to data structures. There are some differences in how these are handled. LVars’ `new` operations uses a nondeterministic method for generating unique keys. This complicates proof of determinism of LVars, because proofs must reason about the equivalence of configurations in which keys are were chosen differently. In contrast, we greatly simplify our proofs assuming that keys are provided by tasks and argue that it is possible to deterministically generate unique keys based on the lineage of the task. It is clearly possible in principle to generalize our determinism results to systems with nondeterministic key generation given constraints on the behavior that prevents functions from “looking into” the values of keys. However, specifying these behavioral constraints formally is difficult when we do not describe in any detail the internal structure or behavior of black-box functions and data types.

The treatment of freezing and quiescence in LVish is quite different to our handling of freezing. The key difference is that our permissions system allows exact determination of when no more updates to data structures are possible, so they can be automatically frozen. LVish relies on the programmer to freeze variable and only guarantees “quasi-determinism” because races between put and freeze operations are possible.
A MASSIVELY PARALLEL RUNTIME FOR DATA-DRIVEN TASK PARALLELISM

Swift/T’s runtime system, Turbine, provides the required runtime support for massively scalable data-driven task parallelism on a MPI-2 or MPI-3 communication layer [87].

The runtime system is a descendant of the ADLB [57] load-balancing library. ADLB implemented a distributed task queue that allowed MPI processes to enqueue (put) and dequeue (get) tasks from a distributed global queue. Tasks in ADLB had an associated priority and optionally targeted, in which case they would be routed to the specified rank. ADLB task payloads are arbitrary binary data, which, with some creativity, allowed a range of parallel applications to be implemented using its simple set of primitives. The task queue was split between multiple ADLB servers. Load balancing was achieved by redirecting put and get requests to another server if a server was low on memory or if a server had no available worker. Early versions of Turbine [95] built on ADLB to add more functionality, including the distributed data store and data-dependent task release.

Since then Turbine has been enhanced in many ways to make it a complete and scalable distributed language runtime for Swift. This enhancements include improvements to task queue performance and scalability, work-stealing to rebalance work between servers, richer data functionality, and support for garbage collection through reference counting.

This thesis contributes several important extensions and optimizations that greatly improve the runtime’s efficiency and scalability over earlier incarnations:

- Techniques to improve scalability of server-to-server operations and analysis of the problems involved
- Extensions to work-stealing algorithms to improve scalability
3.1 Runtime Architecture

The Turbine/ADLB distributed runtime can be viewed as a distributed system that allows many execution resources or workers to cooperate in executing massively parallel applications. It enables coordination between workers through three core services that are implemented efficiently and scalably: a distributed data store to store shared data, a distributed task queue to distribute work, and a distributed dependency engine that resolves data dependencies of tasks. Figure 3.1 illustrates the interactions between these services.

The remainder of the chapter will describe the primitive operations that are supported by the services in Section 3.1.1 and discuss division of processes into servers and worker in Sections 3.1.2 and 3.1.3. This leads into the implementation of data store, task queue, and
Table 3.1: Runtime task operations. Some advanced options and operations are omitted.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Returns</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>payload, type, parallelism, priority, location</td>
<td>-</td>
<td>Enqueue a task</td>
</tr>
<tr>
<td>DPUT</td>
<td>payload, type, parallelism, priority, location, dependencies</td>
<td>-</td>
<td>Enqueue a task with data dependencies</td>
</tr>
<tr>
<td>GET</td>
<td>type, payload</td>
<td>payload</td>
<td>Get payload data for a task with given type. Blocks until available</td>
</tr>
<tr>
<td>AMGET</td>
<td>type, num_tasks, request_handle</td>
<td>request_handle</td>
<td>Issue asynchronous requests for tasks</td>
</tr>
<tr>
<td>AGET_TEST</td>
<td>request_handle</td>
<td>payload?</td>
<td>Check if asynchronous request completed. Returns payload if completed, otherwise returns nothing</td>
</tr>
<tr>
<td>AGET_WAIT</td>
<td>request_handle</td>
<td>payload</td>
<td>Block until asynchronous request completes, then return payload</td>
</tr>
</tbody>
</table>

dependency engine in Sections 3.2, 3.3, and 3.4 respectively.

### 3.1.1 Runtime Operations

The runtime services provide a number of operations that are required to implement distributed execution of Swift. This section lists the main operations that runtime servers must implement to provide context for later discussion about implementation.

Runtime Task Operations

Table 3.1 lists task operations that support adding and removing tasks from the queue. The *payload* of each task is arbitrary binary data that can be interpreted in an application-dependent way. Two PUT operations support adding tasks. DPUT enqueues a task in the dependency engine and PUT enqueues a task for immediate execution, bypassing the dependency engine. PUT is equivalent to DPUT with *dependencies* = ∅. GET is the basic way to get a task from the queue of the desired *type*, while the remaining methods enable non-blocking asynchronous gets. Currently only concurrent non-blocking gets of the same type per worker are supported.

A task has several attributes, specified when the task is added to the task queue with a
The attributes influence which tasks are matched to a worker that issues a *put operation*. The *type* is one of a fixed set of task types specified at startup time that is used to segregate different tasks into distinct types, e.g. CPU and GPU tasks. The *parallelism* indicates the number of workers the task should execute on - tasks with *parallelism* > 1 require assembling a team of multiple workers. Tasks with a higher integer *priority* are matched preferentially, although global priority order is not guaranteed because it would require prohibitively expensive global synchronization. The *task location* describes which workers the task should be sent to. The location has multiple elements: *rank*, the MPI rank of a worker if a task is *targeted* or ANY if a task is *untargeted*; *strictness*, HARD if the task must go to the specified rank or SOFT if the task can go to another rank if the target is busy; and *accuracy*, RANK if the task is targeted to the exact rank, or NODE if the task is targeted to any rank on the same node.

Tasks are matched to workers with the following rules.

- Targeted tasks are first matched to the targeted workers if idle. More accurate targeted tasks (e.g. targeted to a rank) are matched before less accurate (e.g. targeted to a node).

- Untargeted tasks are then matched to any idle workers.

- Soft targeted tasks are then matched to any idle workers.

- If multiple tasks in one of the above categories could be matched to a worker, the highest priority task is matched.

**Runtime Data Operations**

Table 3.2 lists out data operations that allow creating, reading, writing, subscribing to, and reference counting of shared data items in the data store. These operations are closely connected to the data model described in Chapter 2.
Table 3.2: Runtime data operations. Some advanced options and operations are omitted.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Returns</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>type, read_refcount, write_refcount</td>
<td>id</td>
<td>Create a typed data store item</td>
</tr>
<tr>
<td>MULTICREATE</td>
<td>create_params</td>
<td>ids</td>
<td>Create multiple data store items</td>
</tr>
<tr>
<td>RETRIEVE</td>
<td>id, path, refcount_decr, referand_acquire</td>
<td>value, type</td>
<td>Retrieve value of path of data store item. Caller can release refcounts or acquire refcounts if the value refers to another data store item</td>
</tr>
<tr>
<td>STORE</td>
<td>id, path, type, value, refcount_decr, referand_transfer</td>
<td>-</td>
<td>Store to a path of a shared data item. Caller can release refcounts or pass in refcounts if the value refers to another data store item.</td>
</tr>
<tr>
<td>INSERT ATOMIC</td>
<td>id, path</td>
<td>success, value</td>
<td>Attempt to acquire the right to assign a path of a datum. success is TRUE if the caller wins this right. If the path was assigned, value is set to the existing value. This enables idempotent creation of nested data items, e.g. nested arrays</td>
</tr>
<tr>
<td>REFCOUNT INCR</td>
<td>id, refcount_change</td>
<td>-</td>
<td>Increment or decrement refcount</td>
</tr>
<tr>
<td>SUBSCRIBE</td>
<td>id, path</td>
<td>subscribed</td>
<td>Subscribe to data item, or path of data item. If frozen, returns TRUE. If not yet frozen, returns FALSE and sends notification to caller once frozen.</td>
</tr>
<tr>
<td>COPY</td>
<td>from_id, from_path, to_id, to_path</td>
<td>-</td>
<td>Copy data from one path to another once frozen. Data types must match.</td>
</tr>
</tbody>
</table>

3.1.2 Server and Worker Processes

In order to implement services, MPI processes are divided into two roles: workers and servers. The system can be scaled up arbitrarily by proportionally adding processes of both types. Worker processes execute any program logic, coordinating with each through the services. These services are used by invoking remote data and task operations on the servers. These operations are implemented with remote procedure calls (RPCs), where an initial request message is sent to the server and the operation completes once a response message is received. Currently all operations invoked by workers are synchronous: the worker does not resume running the code that invoked the RPC until a response is received and the RPC is completed. Each server process checks for incoming request messages, processes the requests, then responds to them in a server loop. Any worker can communicate and
Figure 3.2: Runtime process layout on distributed-memory system. Processes are divided into workers and servers, which are then mapped onto the processes of multi-core systems.

Figure 3.3: Runtime architecture showing distributed worker processes coordinating through task and data operations. Ready/waiting tasks and shared data items are stored on servers, with each server storing a subset of tasks and data. Servers must communicate to redistribute tasks through work-stealing, and to request/receive notifications about data availability on other servers.
synchronize with any server, but direct communication between workers is not supported, as shown in Figure 3.3. Figure 3.2 shows a common way of distributing servers: one server per node.

This design dedicates processes (and therefore typically CPU cores) to exclusively working as servers. The obvious disadvantage is that these cores cannot run any application code. For example, if 5% of cores are dedicated to running servers, then the maximum achievable utilization for CPU cores running application work is reduced to 95%.

Thus, the design decision deserves some justification. This approach is one of multiple ways to implement services like the distributed task queue and data store while maintaining the quick response times needed for remote task and data operations. We can immediately rule out any designs that depend on worker threads (or processes) handling requests: in Swift/T the worker threads may perform arbitrary computation that will tie up a thread for an arbitrarily long amount of time. Thus we need a way to implement operations that does not require two-sided involvement amongst worker threads to avoid the scenario where one thread waits for an arbitrarily long period of time for the other thread.

Adding server processes solves this problem by introducing an additional class of processes to intermediate between workers: processing of an operation on a server does not require cooperation from a worker.

The alternative solutions that we have considered all have downsides. One possible alternative is to use one-sided communication primitives, such as the one-sided communication primitives provided in MPI. However, these primitives impose a narrower interface with more restrictions and still require a dedicated thread to guarantee asynchronous progress in many cases [101]. Another alternative is to have separate communication threads that could process incoming requests when needed but yield to computation threads. However, current MPI implementations do not readily support this - blocking MPI operations use busy waiting, so communication threads would compete with computation threads for CPU time.
Overall we believe that the benefits of the server/worker model outweighs the overhead of dedicated cores. As future architectures include more and more cores per node, one core per node is an increasingly minor overhead. Even today, with 16 to 32-core nodes common on most modern high performance computing systems [82], the overhead is fairly low: 3–6%. Other systems [21, 101] have used dedicated communication threads for asynchronous handling of messages for essentially the same reasons.

The server/worker architecture imposes challenges that must be addressed to achieve high performance and scalability. All communication is intermediated through the servers, so their responsiveness is critical to the overall throughput of the system. Workers often depend on servers to make progress, for example if they are waiting to receive input data, so high response times from a single server can hinder the progress of many workers.

To enable progress, any blocking operations invoked on servers should be short-lived - on the order of microseconds, because any time a server spends processing an operation is time that other pending operations are delayed. In particular, it is important that a server should not block on operations that may take a significant amount of time, such as I/O or a response from another process. Sections 3.2 and 3.3 describe how task and data operations are implemented efficiently on servers, achieving throughputs of several hundreds of thousands of operations per server per second.

While processing individual server operations efficiently is necessary, it is not sufficient. Two major obstacles to scalability remain: data hot spots and inter-server synchronization.

Data hot spots occur when one server must service a disproportional number of operations, for example, if it owns data that receives a disproportional number of reads or writes. Performance of the entire system can be limited by the performance of that individual server. In the worst cases, “convoys” can form where many workers sit idle waiting for responses from an overloaded server. This is a challenging problem with a substantial associated literature, largely focusing on distributed databases and distributed hash tables. Common solutions
include randomized data distribution [26], data replication [38] and data migration [52]. We
will not discuss this problem in detail, leaving it to future work.

Inter-server synchronization occurs when a server has to invoke an operation on another
server, for example to redistribute tasks, or to send subscriptions/notifications. Care must
be taken to avoid deadlocks and to avoid servers blocking waiting for responses. The issues
are discussed in the next section.

3.1.3 Server to Server Synchronization

As previously mentioned, server to server synchronization has the potential to inhibit scala-
bility, performance, and reliability of the system if not designed and implemented carefully.
If servers become non-responsive to workers for any length of time, this can lead to significant
overall performance degradation because workers are often unable to make progress without
data or tasks from servers.

We will discuss two particular issues with server-to-server operations and what has been
done to address them in the runtime system: deadlock and cascading delays.

The possibility of deadlock arises when a server sends a message to invoke an operation
on another server and enters a state where it is unable to make progress on serving some/all
other requests before it receives a response message from the other server. A deadlock can
occur if the other server simultaneously makes a request and also enters such a state. If
neither server is able to make progress on the other’s request, then a deadlock will occur.
Deadlocks can also occur with longer cycles involving multiple servers.

Cascading delays occur when one server’s poor response times (typically due to being
overloaded) leads to another server exhibiting poor response times, which can quickly be-
come contagious. In the case where all servers are lightly loaded and can quickly process
any pending operations, invoking an operation on another server will cause a slight delay
in processing a server’s own requests, but this will be quickly caught up. However, if a
server is heavily loaded, cascading delays can result when another server’s progress becomes dependent on the heavily loaded server and it is unable to serve more of its own requests until the heavily loaded server responds. This phenomenon is insignificant at small scale but is potentially catastrophic under certain circumstances at scales of 10,000+ cores. This is further exacerbated by effects at the level of the MPI implementation and network stack: long unexpected message queues in MPI [89] and network contention effects [12] can both markedly reduce throughput.

The possibility of both deadlocks and cascading delays are avoided if both the initial request and the response are sent asynchronously, so that the requesting server continues to make progress on other requests, including any requests from the other server, even if the response message is delayed. We call these asynchronous RPCs. This is typically implemented on the server by saving whatever state is necessary to process the response when received, then returning to the main server loop.

We implemented asynchronous RPCs for the most common server-to-server requests in the runtime, including subscribes, notifications, reference count updates, and work stealing probes. All server-to-server requests in the runtime are initiated with a small sync message. The size of sync messages is capped so that fixed-size receive buffers can be preposted by servers and that sync messages can be sent through low-overhead protocols such as the “short” and “eager” protocols used by BlueGene/Q [37] and Cray XE [71] networks. The data for many RPC requests can be packed fully into the sync message and handled entirely asynchronously by the recipient server. Otherwise additional messages can be sent with the remaining data.

Another option in some cases is to give workers responsibility for executing operations on the behalf of a server. For example, if a worker invokes an operation on server A that triggers a data notification to be sent to server B, server A can instruct the worker to send the notification. Currently, this technique is used for server-to-server data notifications.
(responses to subscribes), server-to-server data copies (if the data is small), and server-to-server data reference count decrements.

These techniques are not easily applicable in all cases. It is not always possible to push work to workers and breaking operations into multiple asynchronous steps requires carefully saving all intermediate state and ensuring that no race conditions result from intermediate operations modifying the same data structures. Some server-to-server operations are still implemented as synchronous RPCs because of this difficulty.

Server-to-server synchronous RPCs require a protocol to avoid potential deadlocks among servers. When issuing a synchronous RPC, a server goes into a loop where it checks alternately, with non-blocking MPI functions, for the response to its RPC and an incoming request from an other server. If a response is received, the RPC is completed and it can exit the loop. However, when an incoming request from another server is received, a decision is made about whether to respond immediately or to defer the response. If the response is as simple as sending a single response message, it can always be completed without risking deadlock. These simple synchronous RPCs are always processed immediately. However, other complex synchronous RPCs require cooperation from both sides to complete (e.g. multiple rounds of communication). A deadlock scenario for these is possible if two servers (or a cycle of servers) are attempting to complete the other’s request. To avoid this we use a tiebreaking protocol for complex synchronous RPCs. While waiting for a response to its own synchronous RPC, a server always processes requests from higher ranks and defers processing of requests from servers with lower ranks. This prevents a cycle forming.

3.2 Task Queue

Implementation of an efficient and scalable task queue hinges on efficient task matching algorithms and data structures to maximize throughput per server and scalable work distribution algorithms to handle load imbalances between servers.
Figure 3.4: Matching algorithm for tasks in ADLB server. HANDLEGet and HANDLEPut are called from the main loop of the server when get and put requests are received from workers.

3.2.1 Task Matching

Our task matching algorithms and data structures handle matching tasks to workers on a single server. The initial versions were inherited from the original ADLB implementation but have evolved over time as more demanding applications have required performance improvements and new features.

Figure 3.4 provides pseudocode that outlines the matching process for single-work tasks.¹ HANDLEGet and HANDLEPut are called from the server loop when get and put requests are received from workers. These functions match up incoming requests and work to unmatched requests and work stored on the server.

The main challenge lies in selecting data structures that support efficient storage and lookup of unmatched requests and work. A single server stores the task queue for many workers, so achieving high throughput and low latency is needed to provide each worker with a steady stream of work as it demands it.

The original ADLB implementation of task matching stored requests and work in two separate linked lists. This required linear search of the lists for most get and put operations [19].

---

¹ We omit detailed discussion of task matching for tasks with parallelism > 1, which uses a somewhat different approach because it needs to assemble multiple workers into a team. Wozniak, Peterka, Armstrong, et al. [97] give a high-level description.
This meant that get and put operations had time complexity $O(n)$, where $n$ is the number of work tasks and requests respectively. The number of requests is bounded by the number of workers per server but the number of enqueued work tasks can grow very large in workloads with surplus parallelism. Thus linked lists perform acceptably only for applications with limited surplus parallelism. Earlier versions of Swift/T demonstrated poor efficiency and scalability when work queues grew long. Previous experiments revealed efficiency and scalability degradation at scale even with 100 second tasks, as shown in Figure 3.5 [96]. This was caused jointly by limitations of earlier versions of both work queue data structures and work-stealing algorithms.

In order to improve the efficiency of matching and support new features, we have gradually improved the data structures used to achieve $O(\log(n))$ or even, in many common cases, $O(1)$ time complexity for put and get operations.

The request queue tracks unmatched requests. It needs to support two lookup in two ways: by target, for targeted tasks that must be sent to a specific worker, and by type, for untargeted tasks that can be sent to any worker. Figure 3.6 shows a data structure that supports insertion and lookup of requests in $O(1)$ time for both targeted and untargeted work. A worker with MPI rank $w$ will only request tasks from the server with MPI rank $(w \mod \text{num\_servers}) + \text{num\_workers}$. A worker can also have multiple outstanding get
Figure 3.6: Request queue data structures for ADLB server 0 in a 16 node, 32 core per node system with 16 ADLB servers and 496 workers. Targeted and untargeted work is matched to requests by indexing into the `worker_requests` and `type_requests` arrays respectively. Each worker can have multiple concurrent requests (tracked by the `count`) field, but they must all be of the same type. Matching incoming work to existing requests requires $O(1)$ time with this data structure.

requests of the same type from the AMGET operation.

The work queue tracks unmatched work. It more intricate and needs to support more complex matching rules. The data structures are illustrated in Figure 3.7. Tasks are stored unordered in a single main work array. Fast lookup of tasks is enabled by the `targeted_work` and `untargeted_work` arrays of priority queues. Finding a matching task for a request requires at most two lookups (via `targeted_work` and `untargeted_work`).

All work queue operations require worst-case $O(\log(n))$ time, where $n$ is the number of work tasks in the queue. Specifically, it takes $O(1)$ time to index into each array and locate the appropriate priority queue (a binary heap), $O(1)$ time to find the highest-priority task at the head of the heap and $(O(\log(n))$ worst-cast amortized time to insert into or delete from the binary heap. Items can be inserted or removed from the main work array in $O(1)$ amortized time (another array is used to track unused entries in the array). In the (very common) case where all priorities are equal, no heap entries need to be rearranged to maintain the heap property. Therefore heap insertion/deletion and therefore the overall complexity of
Figure 3.7: Work queue data structures for ADLB server 0 in a 16 node, 32 core per node system with 16 ADLB servers and 496 workers. These data structures support fast matching of requests to tasks with $O(\log(n))$ time complexity.

Put and get operations improve to $O(1)$ time. Using array-based data structures also gives good memory locality properties, so the constant factors associated with these operations are low on typical modern computer systems with multiple levels of cache.

A significant advantage of this design - a main work array with indexes - is that soft targeted tasks can be implemented straightforwardly by including a reference to the task in multiple indexes. If a reference is removed from one index, the entry in the other priority queue is not immediately removed: it becomes stale. Stale entries are cleaned up when they are encountered by a later lookup. In order to avoid retrieving incorrect tasks via stale priority queue entries, the lookup operation checks that the task in the main work array exists, has the expected type, target and priority: if not, the stale entry is removed and lookup is attempted again. It is possible that, by coincidence, a stale entry refers to a different task with matching type, target, and priority. In this case it is unproblematic if this task is returned despite being accessed through a stale entry.

It is also straightforward to further extend the work queue with additional indexes.
Table 3.3: Comparison of work queue data structures. We assume multiple queues per type/rank are used. \( n \) is the number of enqueued tasks.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Deque</th>
<th>Linked lists (ADLB)</th>
<th>Heaps (pointers)</th>
<th>Heaps (work array)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enqueue - no priorities</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Dequeue - no priorities</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Enqueue - priorities</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(log(n)) )</td>
<td>( O(log(n)) )</td>
</tr>
<tr>
<td>Dequeue - priorities</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(log(n)) )</td>
<td>( O(log(n)) )</td>
</tr>
<tr>
<td>Dequeue - soft targeting</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(log(n)) )</td>
</tr>
</tbody>
</table>

For example, we implemented node-aware task targeting by adding an additional index \texttt{node\_targeted\_work} analogous to \texttt{targeted\_work} that supports lookup by a unique node index. For simplicity, we have not shown this index in our figures.

Table 3.3 compares this data structure with alternative implementation strategies. Deque is a traditional double-ended queue that can implement FIFO or LIFO policies. Finding the maximum priority task, however, requires scanning all elements of the deque. The linked list data structure used in initial versions of ADLB has similar operation complexity. The heap (pointers) structure was used in an earlier version of the runtime system. In this approach, both targeted and untargeted heaps held pointers to soft targeted tasks. Removing a soft targeted task was unfortunately a worst-case \( O(n) \) operation because if a task was found and removed via one heap, removing the invalid pointers from the other heap required linearly scanning it. This problem could have been solved by adding a reference count to the task, but this would have deferred freeing the potentially large amount of memory used to store the task payload.

Task Matching Related Work

Previous work on task scheduling and matching has involved devising task-matching algorithms on more varied policies and performance constraints. Table 3.4 summarizes the different characteristics of existing solutions and our problem. LIFO and FIFO scheduling policies can be implemented very efficiently with deque data structures in shared-memory [1, 35] or
Table 3.4: Comparison of related work on task matching.

<table>
<thead>
<tr>
<th>Task types</th>
<th>FIFO/LIFO Work Stealing</th>
<th>Priority Scheduling</th>
<th>Grid Scheduling</th>
<th>Swift/T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling</td>
<td>Online</td>
<td>Online</td>
<td>Ahead of time/periodic</td>
<td>Online</td>
</tr>
<tr>
<td>Per-task overhead</td>
<td>ns/µs</td>
<td>ns/µs</td>
<td>ms/s</td>
<td>ns/µs</td>
</tr>
<tr>
<td>Prioritization</td>
<td>Heuristic (depth/breadth-first)</td>
<td>Numeric priority</td>
<td>Arbitrary policy</td>
<td>Numeric priority</td>
</tr>
<tr>
<td>Locality awareness</td>
<td>Heuristic (inertia)</td>
<td>Heuristic (inertia)</td>
<td>Arbitrary policy</td>
<td>Explicit (multiple levels and strictnesses)</td>
</tr>
</tbody>
</table>

distributed-memory [28]. These these deque-based approaches do not readily support priority scheduling or locality-aware scheduling beyond the heuristic of inertia where tasks stay in their original location until stolen. Priority scheduling can also be implemented very efficiently, although there is a somewhat higher overhead than priority queues [3, 61, 94]. However, again, pure priority scheduling only supports locality-aware scheduling to the same limited extent. Grid scheduling systems can implement more sophisticated scheduling policies but have a much higher per-task overhead and generally need to schedule ahead-of-time scheduling or in periodic batches [55, 73, 86].

Outside of task scheduling, implementation of the MPI standard [87] requires matching posted receives to messages according to a moderately complex set of rules. Similarly to our task matching problem, messages can be matched by tag and by source rank. However, no prioritization is supported and no support for hierarchies of locations. Common MPI implementations such as MPICH and OpenMPI use linked lists, similar to the original ADLB implementation, but recent research has investigated indexing schemes to improve scalability with long message or receive lists [103]. This work is not directly applicable to our task matching problem because it is specifically tailored for MPI message matching and cannot be easily extended handle priorities or multiple levels of locations.
3.2.2 Work Stealing

The per-server task matching algorithm only solves the problem of matching work on an individual server to that server’s own workers. If a server runs out of work to match to its workers, then work needs to be found on another server. In practice it is very common for load imbalances such as this to arise: moving work from overloaded to underloaded servers efficiently is critical.

Work stealing is a method for balancing work that has attractive properties: it is can be implemented efficiently and scalably for shared memory [1, 35] and distributed memory [28, 54, 77, 84] systems.

Implementing work stealing in our runtime system presented some additional challenges not directly addressed by previous work. Most or all of the work-stealing literature deals with a single uniform task type. However, to support heterogenous task parallelism with heterogeneous workers, multiple task types need to be distributed. This requires generalization of the workstealing algorithm to deal with per-type idleness and leads to scenarios where a thief has opportunities to steal tasks of a type that it has no idle workers for, but there is an imbalance in queue length. We also needed to factor the stealing into multiple asynchronous messages for the reasons described in Section 3.1.3.

Figure 3.8 shows pseudocode for the work stealing algorithm. The functions are called from the main server loop either periodically or in response to messages received from other servers. The algorithm requires two message round-trips to complete a successful steal, but only uses short, asynchronous messages that do not block progress of either server until it confirms that matching work is present on a victim. This general algorithm is parameterized in several ways: the steal rate limit implemented by CURRENTSTEALINTERVAL, the number of concurrent outstanding probes, and the selection of work to send in SELECTWORK.

One problem that remains with work-stealing is how to detect when there is no more runnable work in the system. Scioto [28] implements a distributed termination detection
algorithm where a spanning tree is used to scalably aggregate reports of which workers are idle. Steals during termination detection need special handling: if a worker that has reported itself to be idle successfully steals from a victim, the victim must report itself as not idle during the current termination check. The Scioto algorithm is further based closely on a much older algorithm by Francez and Rodeh [34].

The presence of multiple task types complicates termination detection: it is neither sufficient to check that all workers are idle (because work needed on one server may be on a server which has no matching work), nor to check that all task queues are empty (because work could be present that will never be matched to a worker). We use a centralized variant of the termination-detection algorithm where each server tracks which of its workers are idle, and a single master server is responsible for contacting all other servers. The master server initiates the termination check when it is idle for more than a certain time. It contacts each other server in sequence. The multiple task type problem is solved by having the master server collect counts of tasks and requests for each work type to see if any tasks match requests on other servers.

### 3.2.3 Task Queue Limitations and Future Work

The current task queue implementation is effective at solving the task distribution problem that we were faced with. However, room for improvement remains in some situations.

The work stealing algorithm is suboptimal when some work types are executed only by a limited number of workers, which are only associated with a subset of servers. The work stealing algorithm currently does not avoid work going to servers with no workers of the correct type. This problem arises from how the system is layered: the server only knows what work type a worker has currently requested and has no knowledge of what types workers will request in future. This presents challenges in efficiently getting work to the correctly workers. In the current implementation, work will find eventually find workers that can run
MaybeSteal()
1 // Called periodically by server loop
2 if StealRateLimit() and RequestQueueSize() > 0
3 SendProbe()

StealRateLimit()
1 timeSinceSteal = Now() - lastStealTime
2 if timeSinceSteal > CurrentStealInterval()
3 timeSinceSteal = Now()
4 return True
5 else
6 return False

SendProbe()
1 if NumOutstandingProbes() > stealConcurrencyLimit
2 return
3 victim = RandomOtherServer()
4 // Avoid repeated probes without response
5 if HaveOutstandingProbe(victim)
6 return
7 SendProbe(victim)
8 AddOutstandingProbe(victim)

HandleProbe(thief)
1 // Called on the victim when a steal probe is received
2 // The victim sends back counts of work types to thief
3 victimCounts = WorkQueueTypeCounts()
4 SendResponse(thief, victimCounts)

HandleResponse(victim, victimCounts)
1 // Called on the thief when the response is received
2 counts = WorkQueueTypeCounts()
3 SendResponse(thief)
4 // Only steal if work will match pending requests
5 if WorkMatchesRequests(victimCounts)
6 InitiateSteal(victim)
7 RemoveOutstandingProbe(victim)

InitiateSteal(victim)
1 thiefCounts = WorkQueueTypeCounts()
2 SendInitiateSteal(victim, thiefCounts)
3 // Receive the work units victim decides to send
4 // This is a synchronous, blocking operation
5 work = ReceiveWork(victim)
6 MatchWork(work)

HandleInitiateSteal(thief, thiefCounts)
1 // Called on victim if thief initiates steal
2 // Select work units based on thief v. victim counts
3 work = SelectWork(thiefCounts)
4 SendWork(thief, work)

SelectWork(thiefCounts)
1 work = []
2 victimCounts = WorkQueueTypeCounts()
3 foreach t ∈ WorkTypes()
4 if thiefCounts[t] = 0
5 // Always steal if thief out of this work type
6 steal = True
7 else
8 // Attempt to balance given opportunity
9 imbalance = (victimCounts[t] - thiefCounts[t]) ÷ victimCounts[t]
10 steal = imbalance ≥ MinStealImbalance
11 if steal
12 count = (victimCounts[t] - thiefCounts[t]) ÷ 2
13 work = work + SelectRandom(t, count)
14 return work

Figure 3.8: Pseudocode for work stealing algorithm with asynchronous probes. These functions are called from the server loop, either periodically or in the event of incoming messages.
it, but work stealing may shuffle it around other servers multiple times beforehand.

Like other work stealing algorithms based on random stealing, its performance is sub-optimal when work is scarce: idle workers will repeatedly attempt steals even though no work is available. Throttling of steal attempts ameliorates this problem adequately, but the “lifelines” workstealing algorithm [77] offers a promising approach to better addressing this in future work. In lifeline-based work stealing, after some number of failed steal attempts, thieves set up lifelines to a small number of peers and go idle. If those peers obtain work, the work is sent along the lifelines. Lifeline-based work stealing can decrease overhead and enable faster distribution of work in situations where work is scarce. This is a promising direction, and our the request queue mechanism could be used to implement lifelines fairly straightforwardly.

### 3.3 Data Store

The runtime’s data store implements a distributed data store with semantics based on the abstract data store described in Section 2.4.1.

Data store keys are 64-bit integers and the key space is partitioned round-robin between servers. Multiple placement strategies are possible when a worker calls CREATE. The current strategy used is to place the data on the nearest server, which improves data locality, but can lead to problems with load imbalance.

Each data store key has an associated type tag. For compound data structures, additional type information about members is stored in various ways. For example, each associative array include key and value type tags, while a global registry of struct types is maintained, with a struct type identifier supporting retrieval of information about struct field types. Lattice semantics are implemented: attempting to double-assign a path, or modify a data structure with no write references will cause a runtime error. Paths into a data structure are represented with a string of bytes that is interpreted in a way specific to each data type.
A range of lattice data types are implemented: single-assignment scalar values, dynamic associative arrays, multi-sets, and structs with a fixed number of named and typed fields. The details correspond to the data types described in Section 2.3.3.

Garbage collection and automatic freezing are supported by a reference counting mechanism, described in the next section. Data-dependent task release is based on a key/path pair becoming frozen. This is implemented efficiently through a subscription mechanism: any process in the system can subscribe to a key/path pair. These subscriptions are tracked by the server that the key maps to, and when the frozen state is entered, a notification message is set to the subscriber.

### 3.3.1 Reference Counting

Correct reference counting depends on both the runtime and the application code running on the runtime system because the runtime has no visibility into what data store references are retained by the application code. Responsibility is replaced on client to correctly increment and decrement according to how many references they are using. In general, reference counts can be incremented or decremented with the `REFCOUNTINCR` operation, but other operations take arguments that specify reference counts to increment or decrement. This supports many common cases. E.g. it is common to decrement the `READ` reference count at the same time as a `RETRIEVE` operation. Storing or retrieving a reference to a different data store key also requires bookkeeping to transfer reference counts from the caller to the callee or vice-versa.

Once a reference has been stored in a data store structure, intervention is required from the runtime in order to correctly manage reference counts: the data store needs to generate increment/decrement messages as appropriate when owner refs are copied, or when the containing structure is freed. To support certain optimizations, a reference stored in a data store item can own multiple `READ` and `WRITE` references to a referand, which allows
the reference to be duplicated by decrementing only a local reference count, rather than manipulating the (possibly remote) reference count of the referand.

3.4 Dependency Engine

The dependency engine is perhaps the most straightforward of the three services. Its sole responsibility is to accept data-dependent tasks which come with a list of dependencies and release them to the task queue when those dependencies are available.

The implementation is relatively straightforward: when a new task is added to the dependency engine, it subscribes to all input data so that the engine will be notified once the input data is ready. Once notifications have been received for all input data, the task is released to the task queue.

In the current runtime architecture, the dependency engine is implemented in the server process. Tasks are added to the dependency engine with a RPC from a worker. Subscribes and notifications are transmitted directly from the data store to the dependency engine within a server process when the subscribed-to data is in the same process. Tasks are moved from the dependency engine to the task queue by copying a pointer from one data structure to another.

Two significant optimizations have been implemented in the engine:

- Multiple concurrent subscribes to the same data from a single engine are combined into a single shared subscription to avoid redundant subscriptions. A hash table in the engine tracks current subscriptions the tasks that are waiting for the notification.

- Results of recent remote notifications are cached to avoid repeated subscribes to the same data. A least-recently-used policy is used to evict cache entries.

In the original Turbine architecture [95], the dependency engine was implemented using a third dedicated class of “engine” processes. This architecture had some disadvantages. First
and foremost, deciding how to allocate processes between three classes considering various performance bottlenecks was considerably harder than allocating them into two classes. There was also an efficiency disadvantage in some cases where tasks were transmitted three times rather than twice: from a worker to an engine to a server back to a worker. The primary advantage of dedicated engines was that engines could actually execute some short-running tasks themselves: local data-dependent tasks could be executed on engines without ever sending them through the distributed task queue. However, the efficiency gain was typically minimal and it prevented load balancing for those tasks, often leaving some engines significantly overloaded.

3.5 Evaluation

In this section we will evaluate the performance of the distributed task queue, focusing on the task matching and work-stealing performance: aspects of the runtime system which are critical to scaling for even simple applications.

Evaluation was conducted on two different Cray XE6 systems: Beagle2 at the University of Chicago and Blue Waters at NCSA at the University of Illinois. Both systems share a common network interconnect - Gemini, but differ in CPU and memory setup.

The Gemini interconnect has a 3-dimensional torus topology with each node connected to 6 neighbours and offers high point-to-point bandwidth (up to 5GB/s between nodes) and low latency [4].

Beagle2 is a 728-node Cray XE6 system with 32 cores and 64GB RAM per node. The 32 cores are divided between two AMD Opteron 6380 “Abu Dhabi” processors. Each processor has a nominal clock speed of 2.5GHz, 8 x 64KB shared L1 instruction caches, 16 x 16KB L1 data caches, 8 x 2MB shared L2 caches and 2 x 8MB shared L3 caches [23].

Blue Waters is a hybrid Cray XE6/XK7 system. We only used the XE6 nodes, of which there are 22,640, each with 32 cores and 64GB RAM. The 32 cores are divided between two
Figure 3.9: Efficiency of request queue data structure with task mixes of varying targeting parameters.

AMD 6276 “Interlagos” processors. Each processor has a nominal clock speed of 2.3GHz, 8 x 64KB shared L1 instruction caches, 16 x 16KB L1 data caches, 8 x 2MB shared L2 caches and 2 x 8MB shared L3 caches [60]

3.5.1 Task Matching Data Structure Evaluation

To quantity the performance of the data structures for various workloads, I ran a series of benchmarks that simulated incoming tasks and requests and measured the average time per request queue or work queue operation.

In experiments with a work queue, the initial work queue size was varied to understand how very large work queues affected performance. In all workloads there are equal numbers of adds and removes, so the queue size for the duration of the experiment is directly related to the initial queue size.

The data structures were testing in isolation without additional overheads that arise from network communication and memory allocation for tasks - the workload and tasks were pregenerated and stored in memory before timing began. This also avoided to need to generate random numbers during the timed benchmark, which would add measurable overhead and affect results.

Multiple workloads were tested, with the following variants in different combinations:

EQUAL: all priorities are equal
Figure 3.10: Efficiency and scalability of work queue data structure with task mixes of varying priority and targeting.

**Uniform_Random:** priorities are randomly chosen with a uniform distribution

**Untargeted:** all tasks are untargeted

**Targeted:** all tasks are targeted

**Rank/Node:** the accuracy of targeting used

**Soft:** soft targeting is enabled

**Equal_Mix:** an equal mix of untargeted and Rank_Targeted tasks are used

In each experiment, the time taken to complete 5 million operations was measured, with the mean taken to obtain the time per operation in nanoseconds. 1 million unique operations and tasks were pregenerated for each benchmark. Each task had a 256B payload. Five trials of each experiment were conducted, with the results of the first “warmup” trial discarded.

Initially the request queue and work queue were tested in isolation. Figure 3.9 shows per-
Figure 3.11: Efficiency and scalability of request and work queue data structures with task mixes of varying priority and targeting.

Performance results for the request queue. Matching strict rank targeted tasks is very efficient but slightly more overhead is associated with matching node and soft targeted tasks. Figure 3.10 show performance results for the work queue with varying queue lengths. Finally, the request and work queue were tested together using the algorithm in Figure 3.4, with results shown in Figure 3.11. Matching a task requires both a Put and a Get operation, so times from the final experiment can be doubled to obtain the time taken to match a task.

The scalability curves for the work queue are similar to combined curves for work and request queues. The varying request queue performance accounts for moderate constant offsets between the two sets of curves. Overall everything is efficient - the worst workloads with very large queue sizes (over 100 million) result in an average time per operation of under a microsecond and more typical workloads result in average time per operation of 50 to 100 nanoseconds. With EQUAL priorities, performance is almost independent of queue
Figure 3.12: Single server task queue throughput measured in GET/PUT operations per second. The top figure shows performance for different task mixes with queue length of 512 tasks. The bottom figure shows performance by queue length for the UNIFORMRANDOM/EQUALMix task mix. The queue length scalability results size, except for EQual.Equal_Mix, where performance suddenly drops then stabilizes, probably because of caching effects in the memory hierarchy as the working set size of the test grows. With UNIFORM_RANDOM priorities, the average time scales logarithmically and performs well even with a million tasks in the queue. Both NODE and SOFT targeting add a roughly constant factor overhead, which is unsurprising since the logic and data structures involved are slightly more complex.

The measured throughputs for the data structure only are not achievable in the context of the server/worker architecture, where overheads form the network, MPI stack, server loop, and other sources are added. Figure 3.12 shows performance measurements for the same task mixes running with a typical server/worker setup on Beagle. Experiments were run on 1, 2, 4, and 8 nodes with a single server serving all workers. Throughput is not greatly affected by the queue length or the task mix: our queue designs can deliver consistent performance in a wide range of scenarios. The MPI stack and network communication is the dominant overhead in our task queue design: there is a big drop in throughput going from one to two nodes when the network becomes involved and a smaller drop in throughput as the number of nodes increases further.

We can draw two main conclusions from the single server experiment. First, running a
server per node will maximize per-server task throughput because the communication will be local. Second, to get further significant improvements in task queue throughput requires bypassing the MPI stack and network or avoiding the requirement of a synchronous RPC per task queue operation.

3.5.2 Scalability Evaluation

Large-scale scaling experiments were conducted with an earlier version of Swift/T on the Blue Waters supercomputer. These experiments used a version of Swift/T that did not incorporate all of the work-stealing and task-matching algorithm improvements described in this chapter. Unfortunately we do not have access or time allocation to repeated the experiments with a more recent version of Swift/T- we believe that if the experiments were conducted again today, they would demonstrate even better scaling and throughput.

Figure 3.13 illustrates the scalability and task throughput of Swift/T programs using the runtime system. Swift/T achieved a peak throughput of 1.47 billion tasks/s on 524,288 cores running the Sweep benchmark described later in Section 5. Tasks of 1 ms or more achieve high efficiency when the servers are lightly loaded and queuing delays are minimal.

3.6 Runtime Support for Heterogeneous Tasks

Additional extensions to the runtime have been implemented to support further task types.
The first major extension is support for parallel MPI tasks, where a task is executed in parallel on a specified number of workers. This requires additional support from the task matching system to assemble a “team” of the desired size from idle workers. New MPI 3.0 communicator creation functions are used to dynamically construct MPI communicators for the worker team [97].

The second major extension is support for pluggable asynchronous task executors that support execution of multiple concurrent tasks on an execution resource external to Swift/T. The runtime will dispatch a task to the executor, then run a provided callback function when the task completes successfully or fails. A pluggable executor for Coasters [41] supports execution of command-line tasks on a range of remote resources. The GeMTC framework, which was integrated with Swift/T to support dispatching tasks to GPUs, was implemented using an early prototype of this functionality [47].

### 3.7 Related Work

Execution models and runtime systems combining task parallelism with data flow for HPC applications have been a topic of interest for several groups [32]. Tarragon [22] and DaGuE [14] implement efficient parallel execution of explicit dataflow DAGs of tasks from within an MPI program. ParalleX [45] provides a programming model through a C++ library that encompasses globally addressable data and futures, with the ability to launch tasks based on dataflow. StarPU [8] and OmPSS [17] both provide lower-level library and pragma-based interfaces for executing tasks with data dependencies on CPUs and accelerators on distributed-memory clusters.

Habanero Java [83] and Habanero C [21] support asynchronous task parallelism with data dependencies on shared-memory nodes. Extensions to Habanero C support some inter-node parallelism with integration between MPI primitives and Habanero C. X10 supports asynchronous task parallelism, but synchronization is based on a finish statement and termination
detection algorithms, instead of data-dependencies [84].

The Asynchronous Dynamic Load Balancer (ADLB) [57], the basis of our runtime system, is highly scalable and has been successfully used by large-scale physics applications. However, it does not support shared global data and the task queue has performance limitations that were addressed with this work.

Scioto [28] is a library for distributed memory dynamic load balancing of tasks, similar to ADLB. Scioto implements work-stealing among all nodes instead of the server-worker design of ADLB. Scioto’s efficiency is impressive, but it does not provide features required for Swift/T such as task priorities, work types, and targeted tasks.

Recent work on systems such as Sparrow [67], CloudKon [76], and Apollo [15] has attempted to improve throughput of task schedulers in cloud computing to enable workloads composed of “tiny tasks” on large clusters. These systems must deal with problems such as unreliability of workers and the need to enforce scheduling policies for shared resources. As a result of this and other implementation choices, they are unable to achieve anywhere near the efficiency of our runtime system: typical per-task overhead is 10s to 100s of milliseconds.

The MATRIX task scheduler [91] has similar goals to our own work: implementing high-performance distributed task scheduling with policies such as data-aware scheduling. However, the system architecture is built on a general-purpose key-value store, which is not specifically designed for high-performance task scheduling. Reported task throughputs are limited to several thousand tasks per second.
CHAPTER 4

COMPILING SWIFT FOR MASSIVE PARALLELLISM

STC is a whole-program optimizing compiler for Swift/T. Figure 4.1 illustrates how STC fits into the Swift/T toolchain. We discuss the overall design of STC in Section 4.1. STC generates code in the Tcl scripting language that calls into the C functions implementing the distributed runtime described in Chapter 3.

Compiling to Tcl add performance overhead, but was a pragmatic choice. Tcl features including powerful string manipulation allowed rapid development of the compiler, and easy implementation of extension functions. We discuss the general challenges of using interpreted languages like Tcl in a HPC environment in Appendix B. For current applications, the overhead of interpreting Tcl, as opposed to executing compiled code, has not been a major bottleneck; but this may change in future with finer-grained parallelism. For that reason, STC has a modular code generator and is retargetable.

Within STC we have implemented optimizations aimed at reducing communication and synchronization without loss of parallelism (Section 4.3). An intermediate representation for the program captures the execution model (Section 4.4), allowing optimization of synchronization, shared data, and reference counting (Sections 4.5.3, ??, 4.6.2, respectively).

Figure 4.1: The STC compiler is in the middle of the Swift/T toolchain and translates high-level Swift code into execution code for the Turbine runtime.
4.1 Compiler Architecture

The STC compiler uses a architecture that divides the compilation process into multiple phases: a frontend phase that parses and validates input code before translating it to an intermediate representation (IR) using the strategy described in Section 2.7, a middle-end phase that optimizes and otherwise transforms the IR, and a backend phase that generates the final output code based on the intermediate representation. The middle-end phase is further subdivided into optimization, which iteratively rewrites the original IR (IR-1) produced by the frontend and post-processing, which augments IR-1 with additional information for reference counting and data transfer between tasks that is required for code generation. STC’s phases are illustrated in Figure 4.2. Almost all modern compilers follow similar designs.

4.2 Compiler Frontend

The compiler frontend translates Swift to an intermediate representation, while checking for invalid Swift code and programmer errors.

The details of the frontend are mostly unremarkable. The syntax is specified with an ANTLR3 grammar [70] that translates the input text into an abstract syntax tree (AST). Typechecking is performed on the AST to ensure the validity of the program. Some basic semantic analysis is also performed that generates warnings and errors for simple errors in
foreach i in [1:N] {
    foreach j in [1:M] {
        a, b, c = A[i-1][j-1], A[i-1][j], A[i][j-1];
        A[i][j] = h(f(g(a)), f(g(b)), f(g(c)));
    }
}

Figure 4.3: Swift code fragment illustrating wavefront pattern.

variable usage, including some double assignments of single-assignment variables and reads of unassigned variables. After an initial pass over the program AST to locate all global function and variable definitions, a second pass walks all function ASTs to translate each function AST to intermediate representation. The general strategy described in 2.7 works to translate Swift into the compiler's intermediate representation because the intermediate representation closely mirrors the abstract execution model.

An annotation mechanism is supported in the frontend that allows additional information or implementation details to be passed through to later stages of the compiler. Supported annotations include:

- Semantic information about functions for the optimizer, e.g. @pure if the function is deterministic and free of side-effects.

- Task parameters for the runtime system: @location to specify the required or preferred location where the function should execute, @dispatch to specify the task type used for matching to workers (this must be coupled with a declaration of a new task type), @priority to specify the task priority, and @par to specify the degree of parallelism for parallel tasks.

- Optimization requests that override the default, e.g. @unroll to force loop unrolling.
4.3 Optimization Goals for Data-driven Task Parallelism

To optimize a wide range of data-driven task parallelism patterns, we need compiler optimization techniques that can understand the semantics of task parallelism and monotonic variables in order to perform major transformations of the task structure of programs to reduce synchronization and communication at runtime, while preserving parallelism. Excessive runtime operations impair program efficiency because tasks waste time waiting for communication; they can also impair scalability by causing bottlenecks for data or task queues.

The implicitly parallel Swift/T code in Figure 4.3 illustrates the opportunities and challenges of optimization. The code specifies a dynamic, data-driven wavefront pattern of parallelism, where evaluation of cell values is dynamically scheduled based on data availability at runtime, allowing execution to adapt to variable task latencies. Two straightforward transformations give immediate improvements: representing input parameters such as i and j as regular local variables rather than shared monotonic variables and hoisting the lookups of A[i-1] and A[i] out of the inner loop body. The real challenge, however, is in efficiently resolving implied data dependencies between loop iterations. The naïve approach uses three data dependencies per input cell; but with this strategy, synchronization can quickly become a bottleneck. Smarter approaches can identify common inputs of neighboring cells to avoid redundant data reads, or defer task spawns until input data is available: if the task for (i - 1, j) spawns the task for (i, j), only grid cell A[i][j + 1] must be resolved at runtime since both other inputs were available at (i - 1, j). The characteristics of the f, g, and h functions also affect performance of different parallelization schemes. Fusing f and g invocations is a clear improvement because no parallelism is lost; but, depending on function runtimes and other factors, the optimal parallel structure is not immediately obvious. To maximize parallelism, we would implement the loop body invocations as three independent f(g(...)) tasks that produce the input data for a h(...) task. To minimize runtime overhead, on the
other hand, we would merge these four tasks into a single task that executes the \( f(g(\ldots)) \) calls sequentially.

### 4.4 Intermediate Representation

The STC compiler uses a medium-level intermediate representation (IR) that captures the execution model of data-driven task parallelism. Two IR variants are used by stages of the compiler, as shown in Figure 4.2. IR-1 is generated by the compiler frontend and then optimized. IR-2 includes additional information for code generation: explicit bookkeeping for reference counts and data passing to child tasks. Sample IR-1 code for a parallel, recursive Fibonacci calculation is shown in Figure 4.4.

#### 4.4.1 Structure of Intermediate Representation

This section will describe the structure of the basic STC intermediate representation, IR-1. Figure 4.4 provides an illustrative example of IR-1 with unoptimized and optimized versions of the same program.

Figure 4.5 shows the grammar describing the structure of the IR-1 intermediate representation. Each IR program is comprised of a number of functions. The functions are either external functions, such as foreign functions or calls to external applications, or intermediate representation functions. The IR function called `__entry` is the program’s entry point.

Each IR function is structured as a tree of blocks. Each block is represented as a sequence of statements. Statements are either composite conditional statements or single IR instructions operating on input/output variables, giving a flat, simple-to-analyze representation. Control flow is represented with high-level structures: `if` statements, `foreach` loops, `do/while` loops, and so forth. The statements in each block execute sequentially, but blocks
main () {
    int n = argv("n"); // Get command line argument
    int f = fib(n);
    // Print result once computation finishes
    printf("fib(%i)=%i", n, f);
}

(int o) fib (int i) {
    if (i == 0) {
        o = 0;
    } else if (i == 1) {
        o = 1;
    } else {
        // Compute fib(i-1) and fib(i-2) concurrently
        o = fib(i - 1) + fib(i - 2);
    }
}

(a) Swift/T code for recursive Fibonacci.

main () {
    declare $int v_n, int f // variables for block
    CallExtLocal argv [ v_n ] [ "n" ] // call to argv
    Call fib [ f ] [ v_n ] // fib runs asynchronously
    wait (f) { // execute block once f is frozen
        declare $int v_f
        LoadScalar v_f f // Load value of f to v_f
        CallExtLocal printf [ ] [ "fib(%i)=%i" v_n v_f ]
    }
}

(b) IR-1 optimized at -O2.

Figure 4.4: Sample Swift/T program and corresponding IR for recursive Fibonacci algorithm. The IR comprises two functions: main and fib functions. IR instructions include Swift function calls (e.g. Call fib), foreign function calls (e.g. CallExtLocal printf), immediate arithmetic operations (e.g. LocalOp <eq_int>), data-dependent arithmetic operations (e.g. AsyncOp <minus_int>), and reads and writes of shared data items (LoadScalar and StoreScalar, respectively). Control flow constructs used include conditional if statements, and wait statements for data-dependent execution.
Figure 4.5: Grammar describing structure of the IR-1 variant of the STC intermediate representation.
Figure 4.6: Grammar describing operation codes for the IR-1 variant of the STC intermediate representation.
Figure 4.7: Type system and variables used in STC intermediate representation.

within some control-flow structures execute asynchronously and some IR instructions spawn asynchronous tasks. Data-dependent execution is implicit in some asynchronous IR instructions or explicit with `wait` statements that execute a code block after a set of variables is frozen. The use of high-level control flow instead of, e.g., a general control flow graph, is often helpful: the tree structure simplifies some passes, and the code generator can emit specialized code for, e.g., parallel loops.

The type and variable system is shown in Figure 4.7. Variables are identified by name (`⟨id⟩`) and have associated type and storage information. Fields, such as instruction input arguments, which can either be a variable or a constant, can be represented with `⟨arg⟩`.

Variables are either dynamic single-assignment values stored locally in the task (TASKLOCAL variables) or references to shared data that identify data on a remote process. A reference is either the initial reference to a variable allocated in the block (SHARED) or an alias to the initial reference (SHAREDALIAS). Aliases are created by duplicating a reference, acquiring a reference stored in a data structure, or by creating a reference to a path of a data structure.

`⟨prim-type⟩` represents any of the primitive types supported, while more complex types are constructed in a number of ways. Arrays and structs are constructed with the Array and Struct type constructors, while the Shared type constructor introduces a degree of indirection via the data store. For example, `IntType` is a scalar integer, `Shared IntType` is a reference to

<table>
<thead>
<tr>
<th>Argument (argument)</th>
<th>Declaration/Reference (variable declaration/reference)</th>
<th>Identifier (identifier)</th>
<th>Types (types)</th>
<th>Primitive Types (primitive types)</th>
<th>Reference to Shared Type (reference to shared type)</th>
<th>Associative Array Type with Primitive Type Key (associative array type with primitive type key)</th>
<th>Structure Type (structure type)</th>
<th>Variable Storage (variable storage)</th>
<th>Literal Constant (literal constant)</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨arg⟩ := ⟨var⟩</td>
<td>⟨const⟩</td>
<td>⟨id⟩ := ⟨string-literal⟩</td>
<td>⟨type⟩ := ⟨prim-type⟩</td>
<td>⟨shared-type⟩</td>
<td>⟨array-type⟩</td>
<td>⟨bag-type⟩</td>
<td>⟨struct-type⟩</td>
<td>TaskLocal</td>
<td>Shared</td>
</tr>
<tr>
<td>⟨var⟩ := ⟨id⟩ ⟨type⟩ ⟨storage⟩</td>
<td>⟨const⟩ := ⟨int-literal⟩</td>
<td>⟨bool-literal⟩</td>
<td>⟨float-literal⟩</td>
<td>⟨string-literal⟩</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨id⟩ := ⟨string-literal⟩</td>
<td>⟨const⟩ := ⟨int-literal⟩</td>
<td>⟨bool-literal⟩</td>
<td>⟨float-literal⟩</td>
<td>⟨string-literal⟩</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨type⟩ := ⟨prim-type⟩</td>
<td>⟨shared-type⟩</td>
<td>⟨array-type⟩</td>
<td>⟨bag-type⟩</td>
<td>⟨struct-type⟩</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨prim-type⟩ := ‘int’</td>
<td>‘bool’</td>
<td>‘float’</td>
<td>‘string’</td>
<td>‘blob’</td>
<td>‘file’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨shared-type⟩ := ⟨type⟩ ‘*’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨array-type⟩ := ⟨type⟩[⟨prim-type⟩]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨bag-type⟩ := bag(⟨type⟩)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨struct-type⟩ := Struct (⟨id⟩ ⟨type⟩)*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨storage⟩ := TaskLocal</td>
<td>Shared</td>
<td>SharedAlias</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨const⟩ := ⟨int-literal⟩</td>
<td>⟨bool-literal⟩</td>
<td>⟨float-literal⟩</td>
<td>⟨string-literal⟩</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

98
an integer stored in the data data, and \textit{Shared IntType} is a reference to a reference to an integer. Numeric, string and boolean literal values are captured with the \textit{\langle const\rangle} production.

Shared data types are based on the lattice data types described in Section 2.3.3: I-vars, associative arrays, and tuples (used to implement structs). Additional data types have been implement but are not shown. These include various counters, unordered multisets and file I-vars.

In order to specify IR semantics, pseudocode for an IR interpreter is provided in Section C.1.

4.4.2 Properties of IR Structures

This section describes various computed properties of different IR structures that are used in the compiler analysis.

Concepts

- ExecContext
- ExecTarget - describe. Async versus sync
- ExecContext validity

Variable Properties

IR variables can be declared in several ways. Most variables are explicitly declared, either as local variables for a given block or as input/output arguments to a function. Some other variables are implicitly defined by control flow structures. For example, foreach loops define a loop variable that is bound to a different value for each loop iteration.
In the STC compiler, global variables that are initialized at startup are also supported, but we omit description of these for simplicity.

Variables declared in a block or construct are logically visible in all descendant blocks in the IR tree. Some variable types cannot be passed from parent to child tasks: these are inaccessible across task boundaries - any IR in which a child task reads a variable from a parent task that cannot be passed across task boundaries is invalid.

Each variable has three components as shown in Figure 4.7: the variable name, which must be unique within an IR function; the variable data type; and the variable storage, which describes what kind of storage needs to be allocated for the variable. TaskLocal variables only need local storage on the stack/heap to store a value for the duration of the task. Shared variables need storage to be allocated in the data store so that the variable is accessible to other tasks and also local storage for a handle to that. SharedAlias variables are aliases to data store entries, so only need the local handle to be allocated. This information is useful for compiler passes - Shared variables always refer to data store entries that were allocated in a known block, while SharedAlias variables may alias entries allocated elsewhere.

- Rules about when can read/write
- Cannot write function inputs
- Initialization - value and alias variables (also others).

Validity

STC contains a pass that validates that the IR meets these criteria.

Instruction Properties

For the purposes of IR semantics and analysis, each instruction has a number of inherent and derived properties.
An instruction’s *inputs* are any program variables or literal constants that are read by the instruction. They are not written by the instruction. An instruction’s *outputs* are any program variables where the value is modified by instruction or where a write reference count is simply consumed. The inputs and outputs include all variables that are read or written in any ways by the instruction. An instruction’s *modified outputs* is the subset of outputs that are actually modified by the instruction in any way beyond reference count updates. An instruction’s *initialized outputs* is the subset of outputs that are initialized by the instruction. Initialization is described in Section 4.4.2.

Each instruction may or may not have *side effects*, where the instruction has some effect on something other than it’s output variables - e.g. if it performs I/O. If an instruction does not have side-effects, it can be safely removed if the output variable is not needed. When this is not known, the conservative assumption is that an instruction has side-effects. Information about an instruction being side-effect free depends on the particular opcode. Some built-in opcodes are always side-effect free. For other opcodes, additional information about functions, executors, etc, from other sources such as the compiler frontend and configuration can determine if an instruction may have side effects.

Some instructions are *idempotent*: if executed multiple times, they have the same effect as if they were executed only once.

- Piecewise assigned outputs - outputs where value is only partially assigned
- ExecMode
  - Sync/async, if async further information about location
  - Blocking inputs - for async instructions, inputs which must be frozen before output will be assigned
  - Closed outputs - for sync instructions, outputs closed immediately once instruction executed
Control Flow Structure Properties

- Exec target
- Parent and child blocks
- Task boundary (sync continuations)
- Construct defined vars
- Required vars (forDeadCodeElim variant)
- Child context - what context is based on parent
- MustRunLast - flag that can be set to force continuation to be run last in block. When adding continuation to block, this is respected
- isConditional
- isLoop
- executesBlockOnce - whether it executes each enclosed block exactly once
- Blocking vars - vars that the continuation waits for
- Closed vars - superset of blocking vars, also including any construct-defined vars that are closed upon initialization (may be added by optimization passes)

Conditional Properties

Conditional statements - if and switch statements - have related properties that are used in analysis. All conditionals in the IR are exhaustive - at least one branch will be taken (even if that branch is empty). If the condition value is known, there will be a predicted branch that is taken.
4.5 STC Optimizer

STC’s optimizer In subsequent sections we will describe basic initialization and alias analyses (Sections 4.5.1 and 4.5.2) that are used in multiple optimization passes, then describe individual optimization passes, which we categorize into traditional optimizations (Section 4.5.3), shared data optimizations (Section 4.5.4) and task parallelism optimizations(Section ??). For reference, the ordering of optimization passes is documented in Section C.2 of the appendices.

4.5.1 Initialization

The initialization analysis allows determining

4.5.2 Aliases

- Basic definition - when an alias variable refers to part or whole of full thing

- getAliases() function and .aliases module

4.5.3 Adaption of Traditional Optimizations

The foundation of our suite of optimizations is a range of traditional optimization techniques adapted from conventional compilers [59] to our intermediate representation and execution model in general. This required substantial changes to many of the techniques, particularly to generalize them to monotonic variables, and also to be able to effectively optimize across task boundaries and with concurrent semantics.

Value Numbering

STC includes a powerful value numbering [59] (VN) analysis. that discovers congruence relations in an IR function between various expression types, including variables, array cells,
constants, arithmetic expressions, and function calls. Annotations on functions, including standard library and user functions, assist this optimization. For example, the annotation `@pure` asserts that a function output is deterministic, and it has no side-effects. The VN pass identifies congruence relations for each IR block. Value congruence, for example, 
\[ \text{retrieve}(x) \cong^V y \ast 2 \cong^V 6, \]
means that multiple expressions have the same value. Alias congruence, for example 
\[ y \cong^A z \cong^A A[0], \]
means that IR variables refer to the same runtime shared data. Alias congruence implies value congruence. A relation for a block \( B \) applies to \( B \) and all descendant blocks, because of the monotonicity of IR variables. A set of expressions congruent in \( B \) defines a congruence class.

STC’s VN implementation visits all IR instructions in an IR function with a reverse postorder tree walk. Each IR instruction, for example, `StoreInt A 1`, can yield congruence relations: in this case \( A \cong^V \text{store}(1) \) and \( 1 \cong^V \text{retrieve}(A) \). These new relations are added to the known relations, perhaps merging existing congruence classes. For example, if \( B \cong^V \text{store}(1) \), then \( A \cong^V B \). Erroneous user code that double-assigns a variable forces VN to abort, since the correctness of the analysis depends on each variable having a consistent value. Congruence relations in a block always apply to descendant blocks. We also propagate congruence relations upward to parent blocks in the case of conditional statements. For example, if \( x \cong^V 1 \) on both branches of an if statement, it is propagated to the parent. We create temporary variables if necessary to do this, for example, if \( x \cong^V \text{retrieve}(A) \) and \( y \cong^V \text{retrieve}(A) \) on the branches, a new variable \( t_1 \) is assigned \( x \) and \( y \) on the respective branches, so that \( t_1 \cong^V \text{retrieve}(A) \) in the parent.

After the initial VN analysis, IR transformations can use the congruence information. The basic VN optimization replaces variables with the canonical congruence class member: inputs using value congruence classes, and outputs using alias congruence classes. The canonical member is chosen based on the expression type (e.g., constants are preferred) and other factors (e.g., the first variable to be computed is preferred). Variables are thereby
replaced with constants, and redundant computations or shared data loads can be avoided.

STC’s VN analysis supports constant folding [59], where expressions with constant arguments can be evaluated during the VN tree walk. Constant results can then be propagated by using congruence relations, allowing constant folding of further expressions and merges of congruence classes. STC supports binding key-value command-line arguments to constants to compile specialized versions of a program.

Detailed pseudocode for the value numbering pass is provided in Section C.3.

Dead Code Elimination

Dead code elimination (DCE) eliminates unneeded code that is never executed or that computes unneeded results. This includes both unexecuted user code and dead code from earlier optimization passes. VN, for example, eliminates uses of redundant variables but depends on DCE to later eliminate any IR instructions that became redundant as a result.

STC’s dead code elimination is done at the procedural level. The analysis used is closely related to a traditional algorithm that uses use-def chains [46]. However, STC’s IR required some adjustments. First, the traditional algorithm eliminates particular definitions of (generally scalar) mutable variables, while all variables in STC’s IR are either dynamic single-assignment variables or more complex data structures. This means that generally any assignment to a variable may flow to any use, excepting some cases of conditional control flow or reads/writes to disjoint parts of a data structure.

The dead code elimination analysis determines which variables in the function can be eliminated without affecting the overall behavior of the function, similarly to how the traditional algorithm determines which definitions of a variable can be eliminated. For each function a dependence graph between variables is built to determine which variables are live and cannot be eliminated. An edge from $v_1$ to $v_2$ implies that if $v_1$ is live then $v_2$ is live. A pre-order tree traversal is used to build the dependency graph (although the order
of traversal does not matter).

The following variables are added to the live set:

- Global variables
- Function output arguments
- Required variables of all control-flow constructs.
- Input and output variables of instructions with side-effects

The following edges are added to the dependency graph:

- From the first modified output to each input variable.
- From the first modified output to each read output.
- If there are \( n > 1 \) modified outputs, from each modified output \( i \) to output \((i + 1) \mod n\) to form a ring
- From a variable \( v_1 \) to a variable \( v_2 \) if a write to \( v_1 \) may affect the value of \( v_2 \).

Note that we could equivalently add more edges from every modified output to every input variable and every modified output, but the connectivity of this graph is the same.

The analysis also is extended to analyse aliasing and reads and writes to components of data structures. A similar analysis would conservatively assume that every aliased variable or data structure was live, however this assumption limited the optimization’s ability to eliminate unneeded data structures such as nested arrays or struct. For example, in the example in Figure 4.8, value numbering can replace the argument to \texttt{trace} with constant 0, but removing the assignment to \( A[0][0] \) is trickier: \( A[0] \) is represented by an alias variable in the IR and therefore dead code elimination requires alias analysis to determine that it only aliases the unused variable \( A \).

```c
1 int A[ ][ ];
2 A[0][0] = 0;
3 trace(A[0][0]);
```

Figure 4.8: Example illustrating need for alias analysis in Dead Code Elimination.

Therefore, we need to analyze aliases to more accurately determine whether a write to IR
variable $v_1$ may flow to a read of IR variable $v_2$. The do this through component relationships. A component relationship from a whole variable to a part variable is represented as a sequence of elements, each of which can be:

- A subscript, such as an array key or struct field name, which indicates the part is a component of the whole. This is either a constant value or $?$ if not constant. E.g. $\langle A, 0 \rangle$ and $\langle A, ? \rangle$ are subscripts of array $A$.

- A reference, $\ast$, which indicates that a reference must be traversed, e.g. $\langle x, \ast \rangle$ is the variable obtained by dereferencing $x$.

The dead code elimination pass collects two kinds of information on components. First, it collects modified components for an instruction, e.g. if an array subscript of $A$ is assigned the component may be $\langle A, 0 \rangle$. If the entire variable is assigned, an entry is for the entire variable. Second, it collects component alias information, where a variable is a part of a component.

From the component alias information, a component graph can be built. A component graph $G$ allows us to query which other variables might alias a component of variable $(v_1, c_1)$. I.e. we can compute $\text{maybeAliases}(G, v_1, c_1)$, and add an edge to the dependency graph from $v_2$ to $v_1$ for each $v_2 \in \text{maybeAliases}(G, v_1, c_1)$ where $(v_1, c_1)$ is a modified component.

Once the complete dependency graph and live variable set are built, the complete set of live variables is computed with the usual approach of depth-first search on the dependency graph starting from each of the live variables, marking each reachable variable as live. Live variables are only visited once, so this depth first search takes time proportional to the number of variables in the function. The set of variables to eliminate is computed by subtracting the set of required variables from the set of all variables in the function. Eliminating a variable entails removing the variable declaration, any instructions with the variable as output, and any other appearances of the variable, e.g. in the wait list for a wait statement.
Finally, any empty control-flow constructs, e.g. conditionals with no statements inside, are removed in a pre-order pass over the tree.

If any changes were made to the IR tree, the pass does another iteration of dead code elimination for the function. In some cases removal of variables enables removal of control-flow constructs, which allows removal of more variables. This process will always terminate because the total count of control flow constructs and variables is reduced at each step.

Function Inlining

*Function inlining* is an important optimization that creates interprocedural optimization opportunities for later passes and eliminates function call overhead. The function inlining transformation replaces a call to a function with a copy of that function’s code. Swift scripts are often small enough that the entire script be inlined into a single function, allowing optimization across the whole program.

STC’s inlining pass uses several simple heuristic to identify function call sites where inlining is likely beneficial. Overall the goal is to maximize the benefits derived from inlining without excessive code growth. Functions with a single call site are always inlined, because this does not increase the size of the program overall. Otherwise, a simple heuristic is used:

\[
\text{function IR instruction count} \times \# \text{ call sites} < 500.
\]

The pass first constructs a graph of functions from call site to caller, then removes any call sites not satisfying the heuristic criteria. It does depth first search starting at each remaining candidate call site to identify recursive function calls, which could lead to infinite cycles of inlining. When a cycle is detected, it is broken by removing the last edge visited.

The inlining transformation is implemented by copying the function body to the call site with input/output arguments replaced with the input/output variables of the function call instruction. Any local variables in the function are renamed to avoid name conflicts. If any function input arguments are labelled with “WaitFor True”, then the inlined function body
is wrapped in a Wait for those arguments.

Loop Invariant Hoisting

Loop invariant hoisting is important for many Swift/T scripts, where redundant operations such as array accesses occur inside nested parallel foreach loops.

Loop Unrolling

Loop unrolling performs loop unrolling for range loops. The main benefit of this is to allow optimization across loop boundaries. To unroll a loop, an unroll factor $u$ must be selected. Unrolling is implemented by splitting the loop into two copies. The first has the loop body duplicated $u$ times and the stride of the loop increased by a factor of $u$. The second has the same stride as the original loop and executes any leftover iterations that are not a multiple of $u$. Loops are unrolled completely (i.e. $u$ is the same as number of iterations) if they are determined to have $<16$ iterations. Other loops are unrolled by a factor of $u \leq 8$, limited by a simple heuristic that caps code growth at 256 IR instructions per unrolled loop.

Control-flow Fusion

A control-flow fusion pass fuses multiple control-flow constructs into one. This is performed for range loops with identical bounds, foreach loops over the same array, and conditional statements with the same condition. In all cases, this allows optimization across the fused blocks. For foreach and range loops, this also can reduce runtime overhead associated with the loop.
4.5.4 Shared Data Optimizations

We devised further optimizations that exploit the properties of Swift’s data model, in particular the lattice data types. These optimizations reduce runtime operations and to assist other optimizations by simplifying the IR.

Frozen Variable Analysis

Frozen variable analysis (FVA) detects which I-vars, monotonic arrays, and so forth are frozen at each statement. A variable is frozen if an IR instruction freezes it directly (e.g., writing an I-var) or within a `wait` statement for that variable. Alias congruence relations from VN are used to enhance the analysis. Data dependency analysis also allows freezing to be inferred in further situations. For example, if I-var x is the output of an operation with input y, then y must be frozen inside `wait(x) { ... }`. FVA allows inlining of `wait` continuations and strength reduction, whereby statements using expensive runtime data or task operations are replaced with ones that use fewer or no runtime operations, for example by skipping runtime data-dependency checks or executing an operation within the current task context.

Instruction Reordering

One peculiarity of the IR is that instructions that read a variable are allowed to precede instructions that write the same variable if the instructions execute asynchronously. For example, the following is valid IR:

```
In this example, multiple instructions reading variables are after the corresponding instructions that write them. This prevents frozen variable analysis from converting the variables to task-local variables. For example, b cannot be converted to a task-local variable because it is read by `plus_float` before it is assigned by `StoreScalar`. 
```
Figure 4.9: Intermediate representation with instructions in reverse dataflow order.

The example is somewhat contrived, but similar situations occur frequently in real programs. Sometimes user code has statements out of dataflow order and sometimes optimization passes that perform transformations such as inlining continuations result in blocks of code with instructions out of dataflow order. In the above example, if code was directly generated from it, relatively expensive runtime mechanisms would be used to resolve data dependencies, even though there is nothing that fundamentally requires the use of runtime dependency resolution.

To address these ordering problems, we have an instruction reordering pass that attempts to reorder instructions into dataflow order. For example, the above example would be transformed to:

```c
() @main () {
    declare float* a, float* b, float* c, float r

    StoreScalar a 1
    CallExtLocal rand [ r ] [ ]
    StoreScalar b r
    AsyncOp plus_float [ c ] [ a b ]
    CallExt trace [ ] [ c ]
}
```

Figure 4.10: Intermediate representation with instructions reordered into dataflow order.

The frozen variable analysis and value numbering passes can then optimize it to simple
code that executes in a single task with reduced runtime performance overhead:

```c
() @main () {
    declare float c, float r
    CallExtLocal rand [ r ]
    LocalOp plus_float [ c ] [ 1 r ]
    CallExtLocal trace [] [ c ]
}
```

Figure 4.11: Intermediate representation optimized after reordering pass.

Instruction reordering is implemented by constructing, for an IR block, a graph of IR instructions with edges to indicate some kind of possible data dependence between them. All pairs of instructions are checked (so the pass is $O(n^2)$ in the size of the block) to see if there is a dependence between them. The analysis is conservative in assuming dependence to avoid reordering in some invalid way. Forward edges (where the dependence matches current statement order) are always kept if there is some kind of dependence. Backward edges are removed as needed to prevent cycles.

The graph is then topologically sorted to produce a new ordering of instructions.

Store Coalescing

*Store coalescing* combines writes to shared composite data types such as arrays and structs into single store operations. It is applied when a variable is written multiple times in a block, for example if multiple indices of an array are assigned.

Argument Localization

*Argument localization* addresses inefficiencies in the default function calling convention, where arguments are passed as references to shared data that may not be frozen, which often leads to unnecessary data dependencies, reads, and writes to shared data. This overhead is significant, especially for short functions. The same essential problem exists with
values passed between sequential loop iterations. We address this problem with an analysis that identifies when code cannot make progress without an input being frozen. The code is transformed so that the input is passed as a regular value, rather than a reference to shared data, and then add wait statements to function call sites where necessary.

4.5.5 Task Parallelism Optimizations

We implemented a further set of transformations, specific to data-driven task parallelism, that rearrange the task structure of the program to reduce runtime operations and assist further optimization. These must avoid reducing worthwhile parallelism that has granularity to justify task creation overhead.

Two properties of IR instructions can decide whether a transformation reduces worthwhile parallelism. First, an instruction is long running if it executes synchronously in the current task for a long or unbounded time. STC’s optimization passes avoid serializing execution of long-running instructions that could run in parallel, but attempt to merge short-running tasks where the overhead of parallel execution is not justified. The exact boundary between short and long-running instructions is difficult to define and somewhat arbitrary. In practice, however, most instructions fall clearly in one category or the other: simple built-in functions such as arithmetic, string operations, and runtime operations versus computationally intensive user tasks. Second, an instruction is progress enabling if execution of the instruction may fulfill data dependencies of other tasks: for example, a shared data write. The optimizer avoids deferring execution of progress enabling code by a long or unbounded amount of time. For example, it will not add direct or indirect dependencies from a long-running instruction to a progress-enabling instruction. Several criteria are used by the optimizer to determine if an intermediate representation instruction is long running or progress enabling. Any function call or built-in operations that is not explicitly specified as short running by hardcoded compiler rules or a user function annotation is assumed to be long running. Ex-
ecution of a long running instruction counts as progress, so all long running instructions are progress enabling. The optimizer also conservatively assumes that any operation that assigns a dataflow variable releases parallel work and is therefore progress-enabling.

Asynchronous Op Inlining

Asynchronous op inlining is a variant of inlining where an asynchronous built-in operation (e.g., an arithmetic operation or array lookup) is expanded to a wait statement plus non-asynchronous IR instruction, allowing later optimizations to manipulate task structure.

Task Coalescing

Task coalescing is a family of techniques that reconfigure the IR task structure. One effective technique, which we call task pushdown, is to resolve data dependencies between tasks by relocating statements, such as wait statements and data-dependent IR instructions, to descendant blocks in the IR tree where input variables are assigned. This can enable the sequence of transformations in Figure 4.12c, where VN, FVA, and DCE eliminate shared data items, completing conversion of data dependency to task spawn edges. Task coalescing also merges tasks, for example nested or sibling wait statements, when it can determine that the transformation will not prevent progress at runtime.

Pipeline Fusion

Another optimization is pipeline fusion, illustrated in Figure 4.12d. A commonly occurring pattern is a sequentially dependent set of function calls: a “pipeline.” We can avoid runtime task dispatch overhead and data transfer without any reduction in parallelism by fusing a pipeline into a single task. For tasks with short duration or large input/output data, this method saves much overhead. As a generalization, a fused task will spawn dependent tasks if a pipeline “branches.” STC’s pipeline fusion was inspired by the like-named technique from
1  |  a = f1(); b = f2(a);
2  |  c, d = f3(a, b); e = f4(f5(c);
3  |  f = f4(f5(d); g = f6(e, f);

(a) Swift/T code fragment

(b) Unoptimized version, passing data as shared data and perform synchronization

(c) After wait pushdown and elimination of shared data in favor of parent-to-child data passing

(d) After pipeline fusion merges tasks

Figure 4.12: Traces of execution showing optimization of task and data dependencies in a Swift/T code fragment.

streaming languages [39], which is similar in concept but not similar in implementation. Streaming languages have static task graphs with dynamic flows of data, whereas data-driven task parallelism has dynamic task graphs with discrete data. In streaming languages, pipeline fusion trades off pipeline parallelism for lower overhead. In Swift/T, there is no pipeline parallelism to lose, but the more dynamic execution model requires more analysis to identify valid opportunities.

4.6 Compiler Postprocessing

This section covers the postprocessing phase of the compiler in which STC augments IR-1 with the follow additional information to form IR-2:
• Places where the code generator must ensure that data is passed from a parent to a child task

• Places where reference counts must be incremented or decremented in order to correctly implement memory management and freezing for shared variables.

4.6.1 Extensions to IR for Variable Passing and Reference Counting

• Block: Initial reference counts for variables

• Instruction reference counts (read/write)
  - in - any reference counts consumed by instruction
  - out - any reference counts returned by instruction

• Control flow
  - Variable passing modes
  - Passed variables
  - Write passed variables

4.6.2 Memory Management and Freezing Optimizations

Efficient memory management is a challenging issue in a distributed, parallel context, especially in the highly dynamic execution model of data-driven task parallelism, because references to data may be spread across a large number of concurrent processes. The classic memory management problem is generally formulated as the problem of detecting when no direct or indirect references to a data item are held by executing program. In a programming model with monotonic data, such as Swift/T, the variable freezing problem can be formulated similarly as detection of when no remaining references to a data item will be used to write to it (i.e., are read-only references).
We tackle both problems with automatic distributed reference counting, by giving each shared data item read/write reference counts \((\text{refcount})\). When the write refcount drops to zero, the variable is frozen and cannot be written; when both refcounts drop to zero, the variable can be deleted. This design is multipurpose: for example, an I-var starts with one write reference, which is decremented upon assignment to freeze it. In the case of arrays, the compiler must determine which statements may read or write each variable, and write refcounts are incremented and decremented as active tasks that could modify the array are spawned and complete. A well-known weakness of reference counting is that it cannot handle cycles of references. The Swift/T data model does not permit such reference cycles, so this problem is avoided.

Two postoptimization passes over the IR add all necessary refcount operations. The first pass identifies where read and write references are passed from parent to child tasks. For example, if the array A is declared in a parent block and written within a wait statement, a passed write reference is noted. The second pass performs a postorder walk over the IR tree to add reference counting operations.

Naïve reference counting strategies that update reference counts every time a reference is passed into an instruction or goes out of scope are problematic: even in seemingly straightforward code, these reference count updates can more than double the number of data operations executed.

STC uses a range of techniques to address this problem. The basic analysis computes the number of read/write increments/decrements \((\text{incrs}/\text{decrs})\) in each IR block (i.e. four integers per block). Increments accrue from copied references to a variable (e.g., if a reference is passed to an instruction or into an annotated child block), and decrements accrue from references that is acquired from newly initialized variables, variables passed from the the parent block, or from instructions that return references. Refcount operations are placed in the block only after this counting process completes. The basic placement strategy puts
incrs and decrs at the start and end of the block, respectively, thus ensuring that refcounts are not dropped to zero too early during execution.

This framework supports several optimizations. *Merging* incrs/decrs is achieved by the use of counters and an alias analysis that detects when two program variables refer to the same reference counted data item. *Cancellation* of incrs/decrs can happen, for example, when an incr for a reference passed to a single child task cancels out a decr for the variable going out of scope in the parent. This is subject to a pass over the block to verify that the reference is not used after being handed to the child. Refcount incrs or decrs can be *piggybacked* on other data operations, such as variable creation or reads. With a distributed runtime, the piggybacked operation is essentially free because it requires no additional synchronization and minimal additional communication (a few bytes). Unplaced incrs/decrs can be *hoisted* to the parent, subject to conditions: the incr/decr is not in a loop. If in a conditional, the incr/decr must occur on all branches; and if a decr, the child block must be executed synchronously within the parent. In combination, these techniques allow reference counting overhead to be reduced greatly. In cases where the number of readers/writers is determined statically, such as static task graphs, all incrs/decrs are merged, cancelled, or piggybacked, which eliminates the need for reference counting operations. In cases of large parallel loops, reference counting overhead is amortized over the entire loop with batching.

### 4.7 Related Work on Compiler Optimization

Other authors have studied optimization of parallel and distributed applications using a wide range of techniques.

Hardware data-flow-based languages and execution models received significant attention in the past, but there is a resurgence of interest in dataflow models. Previous work has optimized data flow languages with arrays: SISAL [78] and Id [88]. This work emphasized generating low-level machine code, rather than code for a distributed runtime system. For
example, Id targets shared-memory dataflow machines. The SISAL runtime used fork-join parallelism, so compilation necessarily eliminated some potential parallelism.

Optimization techniques have been proposed for distributed-memory or task-parallel functional languages. Compiler optimizations have been implemented for Eden’s distributed memory extensions for Haskell [68]. Optimizations have been implemented that specialize communication operations using analysis of communication patterns in Concurrent ML program [74].

Research on the PGAS family of programming languages [2, 9, 84] has resulted in optimization techniques for explicitly parallel programs executing in a partitioned global address space with async/finish synchronization. This work makes uses of intermediate representation constructs that represent remote or asynchronous execution, and optimizations that understand and optimize these constructs.

Other authors have described parallel intermediate representations (IRs), that are sequential IRs with parallel extensions [99].

Other compiler techniques relevant to task parallelism have been proposed in other contexts. Task creation and management overhead is a core challenge of task parallelism. Zhao et al. reduce this overhead by safely eliding or reducing strength of synchronization operations [100]. Arandi et al. show benefits from compiler-assisted resolution of task data dependencies with a shared-memory runtime [5]. Jagannathan’s communication-passing transformation [42] moves operations to execute at the place and time their inputs are produced. Previous work has addressed compile-time reference counting optimization for sequential or explicitly parallel languages [43, 69].

Distributed computing researchers have explored optimization of distributed data-dependency driven workflows. This work has focused on scheduling the workflows with constraints of resource availability and data movement cost [80, 98, 18], typically assuming that a static task graph is available. Our work focuses on finer-grained parallelism in conjunction with
a high-level, more general programming model, with the short duration of tasks making runtime overhead, in contrast, a dominant concern.
CHAPTER 5
EVALUATION

To characterize the impact of different optimization levels, we chose five benchmarks that capture common patterns of asynchronous task parallelism. **Sweep** is a parameter sweep with two nested loops and completely independent tasks with uneven task durations governed by a log-normal distribution, requiring dynamic assignment of tasks to resources. **Reduce-Tree** is a synthetic application comprising a massive reduction tree with the same structure as a recursive Fibonacci calculation. At full scale, the results of billions of asynchronously-executing tasks are reduced to a single result. **UTS** (Unbalanced Tree Search) is a benchmark that simulates a recursive search procedure with a highly irregular structure, requiring efficient load balancing [64]. The core of UTS in Swift/T is a six line recursive function that calls into the serial C code performing the UTS computation. The serial code executes until it has processed 1 million tree nodes or accumulated 128 unprocessed nodes. **Wavefront** is an application with the wavefront pattern in Figure 4.3 that executed a single function call to compute each grid cell, with runtime following a log-normal distribution with mean 5ms. **Annealing** is a science application comprising an iterative simulated annealing optimization algorithm implemented in ∼500 lines of Swift/T and a simulation implemented in ∼2,000 lines of C++. The objective function of the algorithm is a large ensemble of simulations, with up to 10,000-way parallelism, which is multiplied by the parallelism derived from multiple annealing runs for different parameters. Task runtimes are irregular and vary as a run progresses, requiring highly dynamic load balancing to redistribute tasks, especially to keep workers busy as straggler tasks from each objective function evaluation complete.

We implemented baseline versions of four benchmarks as C programs that directly use the ADLB [57] runtime library. These baselines aim to be equivalent to what a knowledgeable user familiar with ADLB would write. We strived to implement the ADLB baselines efficiently and scalably, but in a straightforward manner, that is, without any overly complex
parallelization schemes. The Sweep ADLB baseline statically partitioned the outer loops between nodes, with up to four processes per node inserting tasks. The UTS ADLB baseline uses the same heuristics as were used in the Swift/T version and avoids all shared data operations, with each task spawning tasks directly. In the ReduceTree ADLB baseline, each task \( f(n) \) spawned two child tasks to compute \( f(n - 1) \) and \( f(n - 2) \), and a third data-dependent task to sum the results. The Wavefront ADLB baseline used a master process to manage data dependencies and launch work tasks.

To pare down Swift/T configurations to a manageable number, we grouped optimizations into four levels: \textbf{O0}: naïve compilation strategy with no optimization; \textbf{O1}: basic redundancy-reducing optimizations, namely, value numbering, constant folding, dead code elimination, loop fusion, frozen variable analysis, and refcount optimizations; \textbf{O2}: more aggressive optimizations, namely, asynchronous op inlining, task coalescing, arg localization, and hoisting; and \textbf{O3}: the remaining optimizations, namely, function inlining, pipeline fusion, loop unrolling, and instruction reordering. Automatic memory management was enabled in all cases.

We expect that the O0 baseline configuration will perform poorly because of the nature of the Swift language, where language features such as implicit parallelism, transparent data movement, and monotonic data structures do not have general-purpose implementations that map efficiently to lower-level hardware and software interfaces. O1 is a stronger baseline configuration that includes basic compiler optimizations of the kind that would appear in most compilers. O1 also includes frozen variable analysis because it is critical in supporting the removal of shared data items. The O1 optimizations subsume most basic optimizations that could be implemented in the frontend or code generator, for example specialization of operations with literal constant arguments.
Figure 5.1: Application speedup and scalability at different optimization levels. X axes show scale in cores. Primary Y axes show application throughput in application-dependent terms. Secondary Y axes show problem size or degree of parallelism where applicable.
Figure 5.2: Impact of optimizations on # of runtime operations issued to servers. Each additional level of optimizations reduces the number of operations required to execute the Swift program, which can lead to better throughput and scalability.

5.1 Method for Large-Scale Experiments

We evaluated the optimizations by running our benchmark applications at different combinations of scale and optimization levels. A prerelease version of Swift/T 0.6\textsuperscript{1} was used for the experiments. We have made the benchmark source code publically available\textsuperscript{2}, with the exception of our collaborators’ annealing code. We ran the benchmarks on the Cray XE6 nodes of the Blue Waters supercomputer [60], which have 2 AMD Interlagos model 6276 CPUs with 32 cores total with clock speeds of >2.3 GHz and 64 GB of memory. We assigned one core per node to act as a server while the remainder were workers. Figure 5.1 shows application speedup, measured with the metric appropriate to each benchmark to quantify how rapidly and efficiently compiled Swift/T parallel coordination code can generate and distribute work. We increased the scale of each application at each optimization

\textsuperscript{1} http://swift-lang.org/Swift-T
\textsuperscript{2} https://github.com/swift-lang/exm-stc/tree/master/bench/suite
level until it failed to continue scaling.

To better understand the effect of optimizations, we instrumented the servers with performance counters that collect aggregate statistics with low overhead, including counts of each operation type invoked on servers and statistics about tasks and workstealing. Figure 5.2 shows operation counts summarized into several categories of runtime operations. *Data Creates* create new shared data items, *data loads* and *data stores* read and write them, and *data subscribes* implement notifications for data dependencies. *Task puts* and *task gets* add and remove tasks from the distributed task queue. *Refcount* operations are standalone refcount operations. *Server* operations include workstealing attempts and other communication between servers.

5.2 Discussion and Analysis of Large-Scale Experiments

These experimental results show that all applications benefit markedly from basic optimization at O1, but further optimizations often, but not always, provide further benefits of similar magnitude. By comparing Figure 5.1 with Figure 5.2, we see that reduction in operation counts leads directly to application speedup. This means that the effectiveness of the compiler optimizations is not specific to our runtime system: some or all of these runtime operations will be bottlenecks to throughput and scaling in any task-parallel runtime system.
ReduceTree shows good scaling at all optimization levels with no scaling bottlenecks. Because of the short duration of the tasks, however, the superior efficiency of the code at O2 and O3 leads to an order of magnitude higher throughput compared with that of O0. UTS shows nearly perfect scaling, with performance of O2, O3, and ADLB nearly indistinguishable. O3 and ADLB were more economical with data operations than O2, but throughput was limited by computation of the UTS update hash function rather than work distribution. At lower optimization levels, input data to the UTS function quickly became a bottleneck and prevented further scaling. Our simple recursive UTS implementation reached a scale 4.7x larger than the previous largest reported UTS run, which was achieved by an X10 work-stealing algorithm [84].

The strong scaling results for Annealing show O0 and O1 failing to scale beyond a certain point as data operations became a bottleneck, while O2 and O3 continued scaling up until the point when work was relatively scarce. Figure 5.1f shows that O3 suffered a collapse in throughput when moving to 32,768 cores that was not suffered by O2, despite O2 using slightly more runtime operations. Work is underway to fix this problem, which we believe is caused by workers frequently transitioning from busy to idle in such a manner that the work stealing algorithm causes excessive congestion.

STC at O3 is competitive with the hand-coded ADLB baselines. In the case of UTS, O3 uses measurably more runtime operations, but this does not impact throughput to any great extent. In some cases it scales better because STC’s dynamic, recursive partitioning of foreach loops is more friendly to the runtime’s work-stealing algorithms than the static partitioning used by the ADLB baseline. In the Wavefront benchmark, O3 gradually overtook the ADLB baseline: the optimized code was less efficient but more scalable because management of data dependencies was automatically balanced between nodes.
5.3 Contribution of Individual Optimizations

We also conducted some experiments to understand the contribution of individual optimizations. It is impractical to fully explore the space of all combinations of optimizations, so we conducted some narrow experiments to gain some additional understanding. We look at the incremental contribution of optimizations in annealing to understand the progression from O0 to O3 and we experiment with disabling individual optimizations to see how critical they are.

5.3.1 Annealing Incremental Contributions

To illustrate better how each optimization pass described can contribute to overall speedup in a complex example, we analyzed the incremental contribution of each optimization level in smaller-scale runs of the Annealing benchmark. The results in Figure 5.3 show that frozen variable analysis and hoisting were effective at eliminating shared data and in hoisting shared array accesses out of loops. FVA is intimately connected to monotonic variables, while hoisting is a member of a widely used family of optimization techniques, a fact that clearly illustrates how both conventional and dataflow-specific optimizations are required for data-driven task parallelism. These optimizations also rely on basic optimizations, particularly VN and DCE to clean up and remove redundancy after major transformations of the IR.

In other benchmarks, other optimizations proved to be critical. In UTS, the task coalescing optimization was able to transform the dataflow logic of the UTS tree search procedure into purely recursive function calls with all intermediate data passed directly to child tasks. Wavefront benefited greatly from loop unrolling, which allowed loads of neighboring array cells to be shared by multiple loop iterations.
Figure 5.4: Operation counts with single optimizations disabled and all remaining optimizations enabled. Y axis shows the number of operations as a percentage of the baseline (all optimizations enabled). High bars indicate that the optimization removed had a significant impact and remaining optimizations were unable to compensate for it. Some data is missing where the atypical combination of optimizations lead to a compile error.
Figure 5.5: Impact of unoptimized and optimized reference counting for memory management, normalized to the total count of runtime operations without memory management. Each bar includes previous optimizations.

5.3.2 Take-one Experiment

It is also helpful to understand which optimizations are most critical to the results - that is - which ones have the biggest impact if removed. In some cases optimizations can have a big impact, but be somewhat redundant to some extent with a combination of other optimizations. In other cases, optimizations are difficult to substitute. To understand this, we conducted an experiment where we ran each benchmark with all optimizations enabled save one that was disabled and compared operation counts to a baseline where all optimizations were enabled. Figure 5.4 shows the results. DCE and VN were bother critical, which is perhaps unsurprising given their generality. VN also had an outsized effect since other optimizations such as FVA are built on top of it. Almost all optimizations (except for algebra) made some contribution in at least one benchmark. The more complex annealing benchmark benefited from the widest range of optimizations.

5.4 Memory Management Overhead

We also performed experiments to understand the effectiveness of reference counting optimization, in particular to understand the overhead of automatic memory management and how much it could be mitigated. We ran scaled-down instances of the benchmarks under multiple configurations: Off, where read reference counts are not tracked and memory is never
Table 5.1: Compile time in seconds for STC at different optimization levels.

<table>
<thead>
<tr>
<th></th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sweep</td>
<td>0.8</td>
<td>1.1</td>
<td>0.9</td>
<td>2.3</td>
</tr>
<tr>
<td>ReduceTree</td>
<td>0.8</td>
<td>1.1</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>UTS</td>
<td>0.9</td>
<td>1.5</td>
<td>1.5</td>
<td>1.4</td>
</tr>
<tr>
<td>Annealing</td>
<td>2.1</td>
<td>3.6</td>
<td>3.9</td>
<td>5.5</td>
</tr>
<tr>
<td>Wavefront</td>
<td>0.9</td>
<td>1.4</td>
<td>1.4</td>
<td>3.3</td>
</tr>
</tbody>
</table>

freed; **Unopt**, where all reference counting optimizations are disabled; and different levels where reference counting optimizations are incrementally enabled. All other optimizations were enabled; hence, for some benchmarks shared data was optimized out in all cases.

Figure 5.5 shows that the reference counting optimizations are effective: the overhead measured in operations from by automatic memory management after optimization is at most 12.2%. The additional operations only cause a proportional decrease in speed if the application is bottlenecked on runtime operations. In practice, this means that the overhead of automatic memory management leads to a 0–12% increase in the minimum task granularity that can be supported. This granularity increase is small enough that automatic memory management in Swift is very viable for large-scale computing. This is important because Swift semantics can require creation of many runtime data items for data dependencies, which cannot always be optimized out; automatic memory management is required to support the high-level programming model.

### 5.5 Compilation Time

In order for an optimizing compiler to be useful in practice, the compilation process must be fast enough that the program can be recompiled. In order to demonstrate the practicality of the compiler, we timed compilation for various optimization levels and programs.

Compilation times were obtained by invoking the *stc* compiler from the command line and timing with the Linux `time` utility. Benchmarks were conducted on a laptop computer
with an Intel Core i5-4200U processor running at 1.6GHz, with 8GB RAM and a solid state drive. STC was run on the OpenJDK 1.7.0_u75 Java Virtual Machine. The compilation was run several times to warm up caches before times were recorded. Table 5.1 shows the timing results.

The compile times are all in the range of 1–5 seconds, which is sufficiently fast for most or all purposes. Swift programs are almost always short scripts of < 1000 lines, so we believe that these compile times are reflective of what will be seen in practice.

There are many opportunities in STC to optimize compile times: so far little effort has been put into optimizing compile times, beyond avoiding optimization algorithms with excessive runtimes, so we believe that compilation could be sped up greatly without any drastic changes or loss in effectiveness.
CHAPTER 6

CONCLUSION

We have described a suite of optimization techniques that can be applied to improving efficiency and scalability of distributed-memory task-parallel programs expressed in a high-level programming language. We applied these techniques to a particularly challenging case: high-level implicitly parallel scripts in the Swift/T programming language. However, these techniques are not specific to Swift/T: we expect that they could also be applied to other task-parallel programming models.

Our performance results support two major claims: that applying a wide spectrum of compiler optimization techniques can greatly improve performance and that compiler optimization can allow high-level implicitly parallel code to drive fine grained task-parallel execution at massive scales, rivaling the efficiency and scalability of hand-written parallel coordination code for common patterns of parallelism at scales from tens of cores to half a million cores for a range of task-parallel application patterns including iterative optimization, tree search, and parallel reductions.

The system described in this dissertation has been used for production science applications running on up to 8,000 cores in production and over 100,000 cores in testing. Application of both compiler and runtime techniques was essential to reaching this scale. The programming model offers a combination of ease of development and scalability that has proven valuable for developers who need to rapidly develop and scale up applications. and do not have the time, expertise, or need to implement, optimize and debug applications in a lower-level distributed-memory programming model like MPI.
6.1 Future Work

Workers in the Swift/T runtime currently use mainly synchronous RPCs to implement task and data operations, which means workers spend much time waiting for responses to messages, yet communication latency could be better masked with overlapped asynchronous operations. It is non-trivial to determine when it is safe to overlap data or task operations. The STC compiler provides the infrastructure that would allow implementation of analyses to determine when operations could be safely overlapped.

The intermediate representation and optimization techniques that we describe in this paper can provide the foundation for further research, both in compiler optimization, and in combined runtime/compiler approaches. For example, opportunities exist to implement further techniques from the extensive compiler optimization literature. More sophisticated control and data flow analyses could bring further incremental improvements to many applications, while more specialized techniques, such as for affine nested loops [10], would aid certain applications such as wavefront.

The compiler infrastructure presents opportunities for cross-layer optimization between the compiler and the distributed runtime. Past work [90] has identified opportunities for runtime systems to optimize data placement and movement for data-intensive applications given hints about future workload, which could be provided by compile-time analysis of operations that can be overlapped.
Appendices
APPENDIX A
SWIFT/T LANGUAGE

A.1 Semantics by Example

In this section I will introduce the Swift/T language through a series of examples that illustrate the semantics and syntax of the language. The examples use version 0.8.0 of Swift/T, available online at http://swift-lang.org/Swift-T/[85].

A.1.1 Hello World

We begin with the Swift/T version of the classic hello world program. For this example, we only need two lines of code: the import statement, which imports the builtin io module, then the next statement, which invokes the printf function (which is in the io module) to print a string.

```swift
1 import io;
2 3 printf("Hello World");
```

Figure A.1: Swift/T Example - Hello World

Next, we will add a slight twist to it, adding another printf.

```swift
1 import io;
2 3 printf("Hello World");
4 printf("Goodbye World");
```

Figure A.2: Swift/T Example - Hello/Goodbye World

Here Swift’s implicit parallelism starts to show: in Swift, the statements are allowed to run in any order because there is no data dependency between them: the program might print Hello World after Goodbye World!
A.1.2 Variables and Scalar Data Types

Variables in Swift are strongly and statically typed: each variable has a type assigned at compile time and automatic conversion between types happens in very few cases. The basic data types in Swift, which are treated as scalar values, are: `int` - 64-bit integer, `float` - double-precision floating point, `string` - unicode string, `boolean` - boolean value, `void` - no value (used for signalling), `file` - file variable. Scalar variables are single-assignment I-vars: after it is declared, it can be assigned at most once. Assigning a variable twice leads to a runtime error.

```swift
// Declaration then assignment
int x;
x = 0;
printf(x);

// Combined declaration and assignment
float y = 2.0 + toFloat(x);

// Use before assignment is valid
string z;
printf(z);
z = "The quick brown fox jumped over the lazy dog";
```

Figure A.3: Swift/T Example - Data Types

Variables can be assigned without being explicitly declared. If an variable name that has not previously been declared is assigned, Swift creates a new variable in the current scope with a type matching the expression on the right hand side of the assignment. This technique can be used in many but not all cases. For example, automatic declaration can be used below for `x` and `condition`, but `y` requires an explicit declaration because the assignments are both in inner scopes.

A.1.3 Dataflow Execution

As mentioned earlier, Swift is implicitly parallel, with program execution ordered by data dependencies. This means that any two operators, function calls, or other parts of a Swift
import io;

// x is automatically declared as a string variable
x = "Hello" + " " + "World";

// x is automatically declared as a boolean
condition = true;

if (condition) {
    y = x;
} else {
    y = "";
}

// Error! y is not defined in this scope
printf(y);

Figure A.4: Swift/T Example - Automatic Declaration

Program can execute in parallel so long as there is no direct or indirect data dependency between them.

In the following example, the two calls to f can execute in parallel because neither depends on data produced by the other. The call to g, however, cannot execute in parallel with either f call because it depends on the data produced by them.

x = f(0);
y = f(1);
z = g(x, y);
printf("%i %i %i", x, y, z);

Figure A.5: Swift/T Example - dataflow parallelism between statements

Different subexpressions of the same expression can also be evaluated in parallel. For example, this implements the same pattern of parallelism as the previous example, despite the calls to f and g being embedded in the same expression.

printf("%i", g(f(0), f(1)));

Figure A.6: Swift/T Example - dataflow parallelism between expressions
A.1.4 Conditional Statements

Code in Swift can be conditionally executed using the `if` and `switch` statements.

The `if` statement’s syntax is identical to many imperative programming languages, such as C, but it executes in a data-dependent manner consistent with the rest of Swift. The condition of the `if` statement is evaluated in parallel with other statements in the enclosing block. Once the value of the condition is computed, the appropriate branch of the `if` statement is executed.

To illustrate how the `if` statement behaves in an implicitly parallel context, consider the below code that executes two computationally intensive simulation functions in parallel. After they finish, it compares the results and print a message depending on the outcome. The programmer does not have to write code to explicitly synchronize and gather the results from the two computations. Rather, the required synchronization happens automatically as part of the evaluation of the the condition of the `if` statement, so that the message is printed once the outcome is known.

```swift
import random;

float f1, f2;

f1 = simulationA();

f2 = simulationB();

if (f1 > f2) {
    printf("Simulation A won!")
} else {
    printf("Simulation B won!")
}
```

Figure A.7: Swift/T Example - conditional execution with if statement
A.1.5 Data-dependent Control Flow

Swift programmers can add explicit dependencies into their program with two different constructs: the `wait` statement, and the `=>` chaining operator. These explicitly make statements depend on data, so that the statements execute only after the data is available, even if the statements do not actually depend on the data directly.

This functionality can be used to add delays to a program or to print messages reporting progress.

```swift
import sys;

// Chaining of multiple statements
printf("Going to sleep") =>
sleep(1) =>
printf("Woke up") =>
sleep(1) =>
printf("Woke up again");

x = compute_something();

// The following forms are equivalent:
x => printf("Done!");

wait (x) {
    printf("Done!");
}
```

Figure A.8: Swift/T Example - data-dependent control flow

These two constructs differ subtly in several ways. `=>` waits on a statement, while `wait` waits on an expression as its argument. Not all statements support chaining - the statements must produce some kind of output variable. `=>` can have any statement on it’s right hand side, while `wait` must be followed by a block enclosed in curly braces.

Most Swift functions have at least one output argument. Many functions that do not produce any output data have a `void` output argument that contains no actual information, but signals when the function has finished executing.
A.1.6 Foreach Loops and Arrays

Foreach loops are tied closely with Swift arrays, so we will introduce them both at the same time.

Arrays in Swift are associative arrays: maps of keys to values. The value type can be any Swift type. The default key type is `int` and other scalar key types such as strings are supported. Associative arrays with integer keys can also be viewed as sparse arrays: arrays which have integer keys that do not need to be contiguous. There are multiple ways to declare and initialize arrays:

```swift
// Declaring array mapping integers to strings
string A[int];
string A[];

// Declaring array mapping strings to integers
int A2[string];

/*
Equivalent ways to initialize an array with the numbers from 1 to 4
*/
B = [1, 2, 3, 4]; // List of values (keys 0-3 are implied)
B = [1:4]; // Integer range (keys 0-3 are implied)
B = [1:4:1]; // Integer range with explicit step of 1
B = { 0: 1, 1: 2, 2: 3, 3: 4 }; // Explicit keys

// Assigning piece-by-piece
B[0] = 1;
B[1] = 1;
B[2] = 1;
B[3] = 1;

// Declaring two-dimensional nested array
string C[][];
C[0][0] = "top-left";
C[0][1] = "top-right";
C[1][0] = "bottom-left";
C[1][1] = "bottom-right";
```

Figure A.9: Swift/T Example - array declarations

The workhorse control flow construct in most Swift programs is the `foreach` loop, which enables parallel iteration over the members of Swift data structures, including arrays. Iterations of foreach loops are independent and execute in parallel provided that data dependen-
Iteration over an array constructed with the \[\text{begin:end:step?}\] syntax is the idiomatic way to iterate over a range of integers in Swift. In Swift/T this is guaranteed to be implemented without construction of an intermediate array. For example, we can build an array by iterating over a range of integers, then iterate over the constructed array:

```
import io;
import stats;
import sys;

// Get keyword command-line argument n, with default value of 100
int n = parseInt(argv("n", "100"));

float harmonic[];

// Compute the harmonic series
foreach i in [1:n] {
    harmonic[i] = 1 / toFloat(i);
}

// Iterate over values and indices
foreach x, i in harmonic {
    printf("H[\%i] = \%f", i, x);
}

printf("sum = \%f", sum(harmonic));
```

Figure A.10: Swift/T Example - basic foreach loops

In the above example, the iterations of each loop will execute in an arbitrary order: the results will almost certainly not print in ascending order.

### A.1.7 Swift Functions

So far we have only shown examples with Swift code at the top level of the program. Swift code can also be enclosed in functions for encapsulation and reuse. Swift functions must declare types and names of their input and output arguments.

Functions return values by assigning the output arguments in the function body. Recursive function calls are allowed and tail recursion is supported in Swift/T: tail recursive calls
of unlimited depth will not cause Swift/T to run out of stack space.

```python
import io;
import sys;

x_val = parseInt(argv("x", "5"));
f1, f2 = fact2(x_val);
printf("fact(%i) = %i", x_val, f1);
printf("fact_tail(%i) = %i", x_val, f2);

/*
   Recursive implementation of factorial.
*/
(int result) fact(int x) {
  if (x == 0) {
    result = 1;
  } else {
    result = x * fact(x - 1);
  }
}

/*
   Tail-recursive implementation of factorial.
*/
(int result) fact_tail(int x, int accum) {
  if (x == 0) {
    result = accum;
  } else {
    result = fact_tail(x - 1, accum * x);
  }
}

/*
   Compute factorial with both implementations, to illustrate
   multiple output arguments.
*/
(int r1, int r2) fact2(int x) {
  r1 = fact(x);
  r2 = fact_tail(x, 1);
}
```

Figure A.11: Swift/T Example - basic Swift functions computing factorials

Function bodies can begin executing as soon as the function is called, regardless of the
state of their input and output arguments. In the below example, assignment of each function
input is delayed a different amount. The print statements in the function will execute at
approximately one-second intervals once inputs are assigned.

```plaintext
import io;
import sys;

print_three(string x, string y, string z) {
  printf("%s", x);
  printf("%s", y);
  printf("%s", z);
}

a = "Now";

sleep(1) =>
  b = "Later" =>
  sleep(1) =>
  c = "Even later";

print_three(a, b, c);
```

Figure A.12: Swift/T Example - delayed assignment of Swift function arguments illustrating execution of Swift function body before arguments are all assigned.

### A.1.8 Foreign Functions

Swift is designed as a language for parallel coordination and scripting: the heavy computation work is typically outsourced to code written in other languages. Swift/T focuses primarily on integration with foreign functions written in programming languages including C, C++, Fortran, Python, Tcl. These functions can be called from Swift code by declaring a *foreign function* with Swift input and output argument types. Currently all foreign functions must be called via Tcl bindings [66]. Tcl bindings can be automatically generated for C, C++ and Fortran code using tools like SWIG [11].

Foreign Tcl functions must specify a Tcl package name and version to be loaded, and then provide a Tcl code template that is used to generate output Tcl code. The template is filled in with input variables as appropriate. The Swift/T compiler generates all necessary code to defer execution until input arguments are assigned and then marshal the arguments between Tcl data and Swift’s internal representations.
import io;

/*
 Declaration of log to arbitrary base (like log function in math module)
 */

@pure @dispatch=WORKER
(float o) my_log (float x, float base) "turbine" "0.7.0" [
"set <<o>> [ expr {log(<<x>>)/log(<<base>>)} ]"
];

printf("log10(100) = "+ my_log(100, 10));

Figure A.13: Swift/T Example - declaration of a foreign function.

The @pure function annotation is used to assert that the function is deterministic and has no side-effects. This allows Swift/T’s optimizer to reuse results of the function instead of recomputing them if needed. Annotations will be described in more detail in a later section. The @dispatch=WORKER function annotation tells Swift/T that the function may take a little while to run and should always be executed as an independent task on a worker (otherwise functions are sometimes executed serially under the assumption that there is no parallel speedup to be had).

A.1.9 For Loops

We introduced foreach loops earlier, which enable independent parallel iteration and do not allow one iteration to pass data (directly) to the next iteration. Sometimes the ability to chain together loop iterations by passing data is needed: if the number of iterations to execute is not known or if each iteration consumes the result of the previous.

The for loop enables this. The for loop’s syntax is based on the for loop in C-derived languages. It has three semicolon-separated clauses: the initializer, the condition, and the update. The initializer clause declares or initializes any iteration variables, the condition clause is evaluated before each iteration to check if it should continue, and the update clause updates values of iteration variables.

Swift/T’s for loop is fairly different from the for loop of imperative languages. Variables
import io;
import random;

int desired_heads = 10;

/* Simulate tossing a coin until we get the needed number of heads */
for (int nheads = 0; nheads < desired_heads; nheads = nheads + addtl_heads) {
  int addtl_heads;
  printf("Toss!");
  if (random() > 0.5) {
    addtl_heads = 1;
  } else {
    addtl_heads = 0;
  }
}

Figure A.14: Swift/T Example - for loop.

in Swift/T are single-assignment so cannot be mutated on each loop iteration. The loop needs to be constructed so that each variable is bound to a different storage location on each iteration. For this reason, Swift/T specially handles iteration variables that appear in the initializer clause: they are rebound to a new storage location on each iteration. The update clause allows assignment of the new iteration’s variables. The right hand side expressions of the update clause refer to the values of iteration variables in the previous iteration, while the condition clause refers to the values in the next iteration.

In the above example, we were not able to get any information out of the for loop. We can add additional iteration variables to act as accumulators. If an iteration variable is declared (but not assigned) outside of the loop, the final value can be accessed outside of the loop.

A.1.10 To write up

• Structs

• Files: app functions and file mapping

• Arrays - details and advanced examples

145
```python
import io;
import random;

int desired_heads = 10;
int ntosses; // Track actual number of tosses

/* Simulate tossing a coin until we get the needed number of heads */
for (int nheads = 0, ntosses = 0;
    nheads < desired_heads;
    nheads = nheads + addtl_heads, ntosses = ntosses + 1) {
    int addtl_heads;
    if (random() > 0.5) {
        addtl_heads = 1;
    } else {
        addtl_heads = 0;
    }
}

printf("Took %i tosses to get %i heads", ntosses, desired_heads);
```

Figure A.15: Swift/T Example - for loop with accumulator.

- Arrays: various ways of using them
- Command-line arguments.
- Updateable variables?
- Advanced: function definition annotations
- Advanced: function call annotations
- Advanced: work types and executors
- Advanced: keyword/optional arguments
- Advanced: type variables, function overloading
A.2 Grammar

Figure A.16: Grammar for Swift/T variant of Swift programming language in EBNF syntax [33] extended with ? for optional elements and + for one or more repetitions. Figures A.17, A.19, and A.18 contain additional grammar rules.
Figure A.17: Extended Backus-Naur Form grammar for Swift/T control-flow statements.

Figure A.18: Extended Backus-Naur Form grammar for Swift/T lexer tokens. ?* means non-greedy matching.
Figure A.19: Extended Backus-Naur Form grammar for Swift/T expressions.
APPENDIX B

TCL FOR HIGH PERFORMANCE COMPUTING

Using interpreted languages such as Tcl in High-performance computing environments has associated challenges resulting from typical design decisions made by the language implementers that require code to be dynamically loaded from multiple files at runtime. This code comes in three typical forms:

- Source code of the interpreted language: either the script being run or source code for libraries implemented in the interpreted language.

- Shared libraries loaded by the operating system’s linker at program startup.

- Shared libraries dynamically loaded by the language interpreter, typically modules that include some component of compiled code.

In the case of Python, running a realistic application will require thousands of filesystem metadata operations - mainly `open` and `stat` operations, which mainly attempt to open small library or source files [102]. Even a minimal Tcl script using our runtime libraries attempts to open 209 files at startup: 112 shared libraries, 82 Tcl source files, and miscellaneous configuration and system files. This less than ideal if the files are on the local file system of the machine running the interpreter: it will slow down startup of the application somewhat.

On typical high-performance computing systems with parallel file systems separate from and shared by the compute nodes, the performance implications of each compute node independently opening hundreds of files at application startup are much worse. HPC parallel file systems are typically optimized for high-throughput reads and writes on a relatively small number of files that are opened for extended periods of time. They are not optimized for supporting high throughput of metadata operations such as opens and closes. On some parallel file systems, this can mean that application startup takes many minutes or even hours for large application runs with several thousand processes or more [102].
Multiple approaches for more efficiently loading many shared libraries at application startup have been proposed in the literature [30, 36, 102]. These solutions all involve intercepting filesystem operations for shared libraries that would go the the shared file system, and instead serving the operations using an alternative mechanism that uses intermediate non-filesystem nodes in a more scalable fashion. Various approaches are possible: libraries can be cached on intermediate notes or distributed through a peer-to-peer or tree-like overlay network. Frings et. al. provide a good overview of the various approaches [36]. Many approaches have limitations on what patterns of loading they support: most require either all libraries to be specified ahead of time or assume that files will be loading synchronously on all processes. Neither assumption is generally true - scripting languages often load dependencies on-demand at runtime.

Such techniques are not widely deployed or have limitations, so the usual recommendation (or mandate!) in HPC environments is to deploy application as a single statically-linked executable containing all required code. This enables the executable to be efficiently shipped to compute nodes during the application startup process and avoids any need to load additional code from the file system.

To allow Swift/T applications to be deployed as a statically linked executable, I developed a wrapping utility that assists with building a self-contained executable including:

- The compiled Swift/T program (i.e. Tcl code)
- The Tcl interpreter and Tcl builtin libraries
- The Turbine runtime libraries
- Tcl packages with Tcl source code and compiled code, provided any compiled code is statically linkable
- Any additional compiled libraries used by the program, provided any compiled code is statically linkable
In the best case where all code can be linked into the executable, this allows the Swift/T application to be a fully self-contained executable. In the worst case, where no compiled libraries can be statically linked into the executable, it avoids loading many Tcl source files and Turbine/Tcl shared libraries.
C.1 Intermediate Representation Interpreter

C.2 Ordering of Optimization Passes

Ordering the optimization passes in a compiler is a difficult process that can have significant impacts on the speed and effectiveness and compiler optimization. The focus of our work has not been on this particular problem, so our implementation uses an ordering of optimization passes that works reasonably well in practice, but may be suboptimal, particularly for compilation time.

The optimization passes are divided into three phases: preprocessing, iterative optimization, and postprocessing.

The preprocessing phase transforms the IR to clean up the IR of unused functions and fix minor discrepancies between the IR emitted by the frontend and the IR processed by the optimizer. Preprocessing has three passes: Prune Functions → Unique Variable Names → Inline Nested Blocks. Prune Functions simply removes any unneeded functions with no call chain from the program entry point to the function. Unique Variable Names is required to transform the IR so that all variable names in each function are unique - the frontend may generate code where a variable with the same name is declared in different blocks of a function. Flatten Nested Blocks inlines any nested blocks not inside a continuation, which are generated by the frontend for nested blocks in the Swift/T code.

The bulk of the heavy lifting is done in the iterative phase of the optimization, where a sequence of passes is repeatedly applied, with some modifications. Repeatedly applying
Figure C.1: Pseudocode for parallel interpreter for STC IR-1 to illustrate IR-1 semantics. This figure provides pseudocode for the main interpreter loop. InitEnv and ChildEnv create empty and child environments, optionally with some mappings. Bind binds a variable in an existing environment. env[name] evaluates a variable or arg in an environment. A global task queue with data dependency support is assumed with PutTask and GetTask operations. ExecRun enqueues work with an executor and ExecComplete returns completed work from the executor.
**ExecInstruction**(env, instruction)

1 // Instructions can lookup and modify variables in env,
2 // access and modify shared datastore, and spawn tasks
3 **switch** instruction
4 **case** LOCALOP(builtin_opcode, out, in)
5 // execute local builtin op
6 **case** ASYNCOP(builtin_opcode, out, in)
7 // spawn task to execute async builtin op
8 **case** LOADINT(val, shared_var)
9 // Load value of shared_var
10 **case** STOREINT(shared_var, val)
11 // Store val into shared_var
12 **case** AINSERT(builtin_opcode, arr, i, var)
13 // Immediately assign arr[i] = var
14 // etc...
15 **case** LOOPCONTINUE loop args
16 // Start new loop iteration
17 deps = \{iv. var \mid iv \in loop.itervars \land iv. blocking\}
18 env' = CHILDENV(env.parent, itervars[i]. var \mapsto
19 \quad env[\text{args}[i]] \mid i \in [1, \text{loop.itervars.length}]\})
20 **PUTTask**(deps, env', loop.body)
21 **case** LOOPBREAK loop
22 // Marker for loop termination

Figure C.2: Pseudocode for parallel interpreter for STC IR-1 to illustrate IR-1 semantics. This figure provides pseudocode for IR instructions.
optimizations can enable optimizations in subsequent iterations to be more successful because of opportunities enabled by transformations of other passes in earlier iterations. It also allows opportunities to apply transformations such as loop unrolling or function inlining partway through the optimization process, so that optimization passes can attempt to optimize both before and after versions.

We iterate through the passes 10 times by default. Note that this could almost certainly be reduced without major modification by more careful ordering of passes and tweaking, but this has not been a focus of research. Some optimizations are not run in each iteration because there is limited benefit to repeated application of the pass. The sequence of optimization passes applied in iteration \( i \) is: Prune Functions \((i = 5) \rightarrow\) Inline Functions \((i \in \{0, 3, 8\}) \rightarrow\) Array and Struct Build \((i \in \{2, 5, 8\}) \rightarrow\) Unroll loops \((i \in \{2, 5, 8\}) \rightarrow\) Hoist Loops \((i \in [0, 8]) \rightarrow\) Reorder Instructions \((i \in \{1, 3, 5, 7\}) \rightarrow\) Propagate Aliases \((i \in \{0, 3, 6, 9\}) \rightarrow\) Value Number \((\text{reorder if } i \geq 8) \rightarrow\) Loop Simplify \(\rightarrow\) Dead Code Elimination \(\rightarrow\) DemoteGlobals \((i \in \{0, 3, 6, 9\}) \rightarrow\) Function Signature \(\rightarrow\) Control Flow Fusion \(\rightarrow\) Pipeline \((i = 7) \rightarrow\) Dataflow Op Inline \((i = 5) \rightarrow\) Merge Waits \((i \in \{6, 8\}) \rightarrow\) Wait Coalesce.

The postprocessing phase does final cleanup of the IR then adds reference counting and variable passing instructions required for code generation. The passes in postprocessing are: Dead Code Elimination \(\rightarrow\) Prune Functions \(\rightarrow\) Add Passed Variables \(\rightarrow\) Add Reference Counting \(\rightarrow\) Fixup Passed Variables

The first two passes simply remove any unused code that is leftover from iterative optimization. The next pass add annotations about which variables are used by child tasks, and whether read or write references need to be passed. Then the reference counting pass adds reference counting operations and applies reference counting optimizations (in part using the passed variable annotations). Finally, in some special cases, passed variables need to be added if reference counting operations refer to a variable that was not already passed.
C.3 Value Numbering Pseudocode

```
ValueNumberPass(irProgram)
  1 // Entry point for value numbering pass
  2 globalCongr = (NewUF(), NewUF())
  3 foreach func ∈ prog.funcs
  4     congrMap = FindCongr(func, [], globalCongr)
  5 PropCongr(func, block, congrMap, 0)
  6 inflineControl(func, block, congrMap)

FindCongr(block, congrMap, parentCongr)
  1 // Build congruence info for this block and descendants
  2 blockCongr = NewCongrScope(parentCongr)
  3 congrMap[block] = blockCongr
  4 foreach stmt ∈ block.stmts
  5     if IsInstruction(stmt)
  6         AddInstructionCongr(blockCongr, stmt)
  7     if IsConditional(stmt)
  8         foreach branch ∈ stmt.blocks
  9             FindCongr(branch, congr, blockCongr)
 10     if BranchesAreExhaustive(stmt)
 11         branchCongr = {congrMap[branch] ∣ branch ∈ stmt.blocks}
 12     UnifyBranches(blockCongr, branchCongr)
 13     foreach cont ∈ block.continuations
 14         // Add any info for continuation
 15         contCongr = NewCongrScope(blockCongr)
 16     AddContinuationCongr(contCongr, cont)
 17     foreach contBlock ∈ cont.blocks
 18         FindCongr(contBlock, congr, blockCongr)

NewCongrScope(congr)
  1 // Creates a new nested congruence scope
  2 (val, alias) = congr
  3 return (NewUFScope(val), NewUFScope(alias))

AddInstructionCongr(congr, inst)
  1 // TODO: see Table for instruction values
  2 foreach value ∈ InstructionValues(inst)
  3 UpdateCongruences(congr, value)

AddContinuationCongr(congr, continuation)
  1 foreach value ∈ ContinuationValues(inst)
  2 UpdateCongruences(congr, value)

UpdateCongruences(congr, value)
  1 // TODO: add to congruences with Union

UnifyBranches(congr, branchCongr)
  1 // TODO: unify values
```

```
PropCongr(block, congrMap, parentInit)
  1 // TODO: track location in source
  2 // Propagate congruence info and replace variables
  3 (vals, aliases) = congrMap[block]
  4 init = NewSetScope(parentInit)
  5 foreach stmt ∈ block.stmts
  6     if IsInstruction(stmt)
  7         // Propagate to instruction inputs and outputs
  8         stmt. inputs = (Prop(arg, vals, init, R) | arg ∈ stmt.inputs)
  9         stmt. outputs = (Prop(var, aliases, init, Write) | var ∈ stmt.outputs)
 10     AddToSet(init, InstInit(stmt))
 11     if IsConditional(stmt)
 12         stmt. condition = Prop(stmt. condition, vals, init, R)
 13     AddToSet(init, InstInit(stmt))
 14     foreach branch ∈ stmt.blocks
 15         PropCongr(branch, congrMap, init)
 16     if continuation ∈ block.continuations
 17         // TODO: replace continuation info
 18     if continuation ∈ block.continuations
 19         foreach contBlock ∈ continuation.blocks
 20         PropCongr(contBlock, congrMap, init)

Prop(arg, congr, init, rw)
  1 // Possibly replace with congruent value
  2 // TODO: doesn’t check if passable
  3 canonical = Find(congr, arg)
  4 if IsConst(canonical) ∨ (IsVar(canonical) ∧ (~ RequiresInit(arg, rw) ∨ SetContains(init, arg)))
  5 return canonical
  6 return arg
```

```
InlineControl(block, congr)
  1 // Attempt to inline conditionals and wait statements
  2 foreach structure ∈ block.controlFlowStructures
  3     // First recurse on child blocks
  4     foreach block ∈ structure.blocks
  5         InlineControl(block, congrMap)
  6     if IsConditional(structure)
  7         if IsConstant(structure.condition)
  8             // Inline the conditional (defn. not shown)
  9             InlineTakenBranch(block, structure)
 10     // TODO: IR position?
 11     if IsWait(structure)
 12     if WaitVarsFrozen(structure, congrMap[block])
 13         //Inline the wait body (defn. not shown)
 14         InlineWait(block, structure)
```

Figure C.3: Value numbering algorithm. The union find algorithms are shown in Figure C.4.

157
NewUF()
1 // Create empty scoped union find

NewUFScope(parent)
1 // Create new scope for union find

Union(uf, winner, loser)
1 // Merge disjoin sets. Winner is representative of merged set.

Find(uf, member)
1 // Find representative of set member is in

Figure C.4: Scoped union find algorithms used to track disjoint sets.
REFERENCES


