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

For Ruth

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thanks to my advisor, Ian Foster, for his encouragement, guidance, and support over the past six years as I have developed as a researcher. In particular, his ability to look at the big picture and identify important problems faced by computational scientists that nobody is tackling has been a big inspiration and influence on my own approach to research: it does not matter how smart you are if you are asking the wrong questions!

This work would not have been possible without the groundwork laid by past and present members of the Swift team: Dan Katz, David Kelly, Ketan Maheshwari, Yadu Nand, Michael Wilde, Justin Wozniak, and Zhao Zhang; or the motivation and inspiration provided by users brave enough to try out new and unproven programming languages and features.

Particular thanks go to Justin, who laid the technical foundation for Swift/T and with whom I have had a very productive collaboration over the last few years building out Swift/T. The journey from initial checkins to production science has been incredible. It is almost impossible to separate out our contributions – many of the final ideas resulted from several rounds of back-and-forth improvements. I hope I have given adequate credit for his ideas and work on which this thesis builds!

Thanks to John Reppy for volunteering to help define semantics and for his ability to quickly understand what I was trying to do and guide me to better approaches. Some ideas he gave after minutes of conversation would have taken an unknown amount of time to arrive at otherwise.

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 lattice data types and parallel tasks can be subjected to rigorous analysis and enhanced by a range of compiler optimization techniques, and thus offer high efficiency and scalability for a broad range of large-scale computing applications on massively parallel distributed-memory computer systems.*

ABSTRACT

In recent years, large-scale computation has become an indispensable tool in many fields. Programming models and languages play a central enabling role by abstracting the computational capabilities of a network of computers to enable programmers to construct applications without dealing with the full complexity of a distributed system of computers. This dissertation is motivated by the limitations of current programming models for high-performance computing in addressing emerging problems, including programmability for non-expert parallel programmers, abstraction of heterogeneous compute resources, composition of heterogeneous task types into unified applications, and fault tolerance.

We demonstrate that a high-level programming language built on top of a data-driven task parallelism execution model can feasibly address many of these problems for many applications while remaining expressive and efficient. We make several technical contributions towards this goal: formal semantics for a deterministic execution model based on lattice data types and task parallelism, a scalable distributed runtime system implementing the execution model, and an optimizing compiler for the Swift programming language that targets this runtime system. In combination, these contributions enable applications with common patterns of parallelism to be simply and concisely expressed in Swift, yet run efficiently on distributed-memory clusters with many thousands of compute cores.

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 fields 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 [28], which abstracts a distributed compution as map and reduce functions applied to streams of data tuples to enable automatic scaling and fault tolerance, and MPI [109], which abstracts network communication as standardized point-to-point and collective messaging operations.

This dissertation is motivated three major problems that are not addressed well with current programming models (and/or the current implementations of those programming models). I demonstrate that a high-level programming language built on top of the *data-driven task parallelism* execution model can feasibly address 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.

The first motivating problem for my work is ease of programming for irregular, task-parallel applications running at large scale on high-performance computing (HPC) systems with hundreds or thousands of cores. This style of HPC application is sometimes called *many-task computing* [90]. The traditional development model for HPC 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. Automation of these concerns, however, is viable for many real-world applications, through load balancing algorithms like work-stealing, through randomized or adaptive data placement, and through higher-level synchronization constructs such as task dependencies instead of locks or message passing. A programming system that provides robust scalable algorithms for these concerns and a simple and accessible parallel programming model could enable these applications to be constructed and scaled up rapidly [56, 119].

Another motivating problem is the increasing prevalence of heterogeneous 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 [37]. 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, OpenMP [78] for intranode CPU parallelism, and CUDA [76] or OpenCL [47] 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 using heterogeneous, distributed-memory resources with transparent data movement between devices and dynamic data-aware task scheduling. Recent work has explored implementing execution models of this style with libraries and conservative language extensions to C for distributed-memory and heterogeneous systems [11, 20, 24, 103]. Early, promising, results suggest that performance of applications in this higher-level execution model can match or exceed that of code written directly to lower-level programming interfaces such as message passing or threads. One reason for this success is that sophisticated algorithms for load balancing (e.g., work stealing) and data movement – that require significant effort to implement on a per-application basis – can be implemented in a generic form once and then used with minimal

additional effort for many applications. Another reason for these promising performance results is that the asynchronous execution model is effective at hiding latency and exploiting available resources in applications with irregular parallelism or unpredictable task runtimes: computation and communication can be overlapped so that individual processes can spend more time doing useful work and less time waiting for communication or synchronization with other processes.

A third and final set of problems comes from hardware constraints and limitations that are increasingly hard to ignore: energy consumption and hardware unreliability, particular in the case of future Exascale HPC systems [33]. 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 HPC 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 [23]. 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.

The work presented by this thesis was done in the context of a research effort, ExM (Exascale Many-task) [8], in which we explored the possibility of using implicitly parallel programming models to address the three aforementioned challenges. The concrete result of this research effort is Swift/T [119], a scalable and high performance implementation of a high-level implicitly parallel programming language. In short, Swift/T aims to make writing massively parallel code for data-driven task parallelism as easy and intuitive as sequential scripting in languages such as Python. The language design is based on the original Swift programming language implementation [114], which works well for executing scientific

workflows on wide-area distributed systems but cannot support extremely scalable and efficient execution because of its unscalable centralized design and relatively high-overhead task dispatch mechanisms.

Swift/T has great promise in addressing the challenges we identified with its the dataflow-oriented abstractions. However, the first obstacle to the practicality of Swift/T is efficiency and scalability: if performance is unacceptably poor, then it matters little whether it helps with fault tolerance or heterogeneity! 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 by 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

The goal of this dissertation is to lay the groundwork for Swift/T to address these challenges we have described. We present and evaluate abstractions and implementations for the high-level Swift/T programming model and demonstrate that high performance is achievable through a combination of runtime and compiler techniques.

This work touches on multiple levels of programming language design and 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 that are addressed in this work, 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

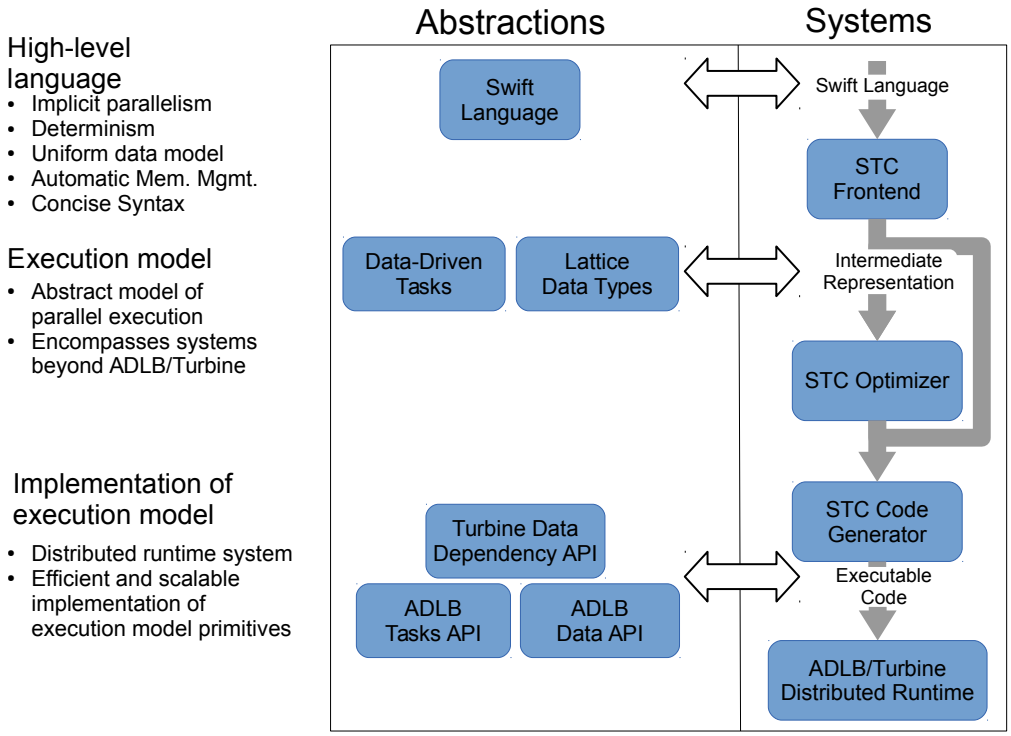


Figure 1.1: Conceptual overview of abstractions and systems under discussion, from high-level language to low-level parallel runtime, illustrating how abstractions of different levels correspond to interfaces between different components of the system.

the work to other contexts, particularly to 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 the Swift language, including task creation, data-dependent task execution, and lattice data types. Finally, the concrete APIs describe the actual programming language interface that can be used to construct task-parallel programs running on distributed 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 remainder of the dissertation is organized as follows. We first provide an introduction to the Swift language to familiarize readers with its syntax and semantics. (Chapter 2). We then present the main contributions of the dissertation, which are as follows:

- Abstract semantics for the data-driven tasks execution model with monotonic lattice data types (Chapter 3).
- 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 (Chapter 4).
- Techniques for efficiently compiling the Swift language for this style of runtime system, including compiler optimization techniques that allow applications developed in Swift language to execute efficiently on massively parallel distributed-memory systems (Chapter 5).

Finally, we describe additional elements of the work that are important to achieving the ultimate goal of an efficient and scalable programming language, but are not core contributions of the dissertation, which are:

- Aspects of the compiler implementation outside of the intermediate representation and optimization (i.e., semantic analysis, translation to intermediate representation, and code generation – also Chapter 5).
- Logistics of running Swift code in high-performance computing (HPC) environments (Appendix B).

CHAPTER 2

SWIFT/T LANGUAGE INTRODUCTION

The Swift programming language [114] has been successful in enabling scientific programmers without particular expertise in parallel programming to harness parallel computing resources. In this chapter, I give an introductory exposition of Swift's syntax and semantics to provide context for the more in-depth technical material that makes up the bulk of the thesis.

2.1 History – Swift/K and Swift/T

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 [48] or Falkon [91]. Even in favourable circumstances with a 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 parallellism 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) [68, 80]. 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. Swift/T focuses on enabling a *hierarchical* programming model for high-performance fine-grained task parallelism. Swift/T code orchestrates large-scale computations composed of foreign functions (including C and Fortran) with in-memory data, computational kernels on GPUs and other accelerators [58], and parallel functions implemented in lower-level parallel programming models - typically threads or message-passing. These functions and kernels 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.

2.2 Swift Language Semantics

Swift has been successfully used by many programmers to parallelize and speed up their applications [29, 36, 49, 50, 51, 82, 85, 86, 102, 117, 121]. We attribute this success to the fact that 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 complex reasoning about concurrency.

The main features that characterize Swift are:

- A hierarchical programming model where computationally intensive code is written in various other programming languages and parallel coordination is written in Swift.
- Implicit parallelism and relaxed execution ordering constraints: program statements can execute out of order whenever input data is available.
- 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.

- 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 feature is discussed in more depth in the next section.

2.2.1 Monotonicity and Lattice Data Types

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 modified further. Programs that attempt to overwrite data will fail at runtime (or compile time if the compiler determines that the write is definitely erroneous). If write operations that modify monotonic variables are *commutative*, then writes can be reordered without changing the final result. Combining these properties leads to a class of data types called *lattice data types*.

If reads to such data types are constrained so that transient states are not observable, then we can achieve deterministic computation even with non-deterministic ordering of operations. Swift programs using lattice data types with these restrictions on reads 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, e.g. the library function `rand()`.

Basic Swift variables are single-assignment I-vars [75] (sometimes alternatively called futures), 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 to which it is *mapped*. Assigning a mapped file variable in Swift then results in a file appearing at that path.

Composite lattice 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 [115], both of which are monotonic lattice data types. The *associative array* is the most complex and heavily used of the two. 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 assigned. 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 refer directly to properties such as the size of arrays and ensures that reads of non-existent array keys will eventually fail. The implementation of automatic freezing in both Swift/K and Swift/T requires both compiler analysis and runtime support. Semantics of freezing have been subtle and poorly documented in the past, particular for Swift/K. This undefined behavior 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.

Later in the thesis, in Chapter 3, I describe formal semantics for an execution model that captures lattice data types and execution of parallel tasks, then shows how Swift's high-level language constructs can be translated into it.

2.3 Semantics by Example

I introduce the Swift/T language in more detail 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/` [105].

2.3.1 Hello World

We begin with the Swift/T version of the classic hello world program in Figure 2.1, which needs two lines of code: the `import` statement that imports the builtin `io` module, then the call to the `printf` function from the `io` module to print a string.

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

Figure 2.1: Swift/T example: Hello World

Adding another `printf` in Figure 2.2 adds an interesting twist related to Swift's implicit parallelism. 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`! On my system, the order differs depending on the compiler optimization level – at `-O0`, `Goodbye World` is printed first, but at `-O2`, `Hello World` is printed first instead.

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

Figure 2.2: Swift/T example: Hello/Goodbye World

2.3.2 Variables and Scalar Data Types

Variables in Swift are strongly and statically typed: each variable's type is known at compile time and automatic conversion between types happens in 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 one

is declared, it can be assigned at most once. Assigning a variable twice leads to a runtime error. Figure 2.3 demonstrates various modes of declaration and assignment of variables in Swift.

```
1  //  Declaration then assignment
2  int x;
3  x = 0;
4  printf(x);
5
6  //  Combined declaration and assignment
7  float y = 2.0 + toFloat(x);
8
9  //  Use before assignment is valid
10 string z;
11 printf(z);
12 z = "The quick brown fox jumped over the lazy dog";
```

Figure 2.3: 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, in Figure 2.4, automatic declaration can be used for `x` and `condition`, but `y` requires an explicit declaration because the assignments are both in inner scopes.

```
1  import io;
2
3  //  x is automatically declared as a string variable
4  x = "Hello" + " " + " World";
5
6  //  x is automatically declared as a boolean
7  condition = true;
8
9  if (condition) {
10   y = x;
11 } else {
12   y = "";
13 }
14 //  Error! y is not defined in this scope
15 printf(y);
```

Figure 2.4: Swift/T example: Automatic Declaration

2.3.3 Dataflow Execution

As mentioned earlier, Swift is implicitly parallel, with program execution ordered by data dependencies. Thus, any two operators, function calls, or other parts of a Swift program can execute in parallel if there is no direct or indirect data dependency between them.

In Figure 2.5, 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 both of them.

```
1 x = f(0);
2 y = f(1);
3 z = g(x, y);
4
5 printf("%i %i %i", x, y, z);
```

Figure 2.5: Swift/T example: dataflow parallelism between statements

Different subexpressions of the same expression can also be evaluated in parallel. For example, Figure 2.6 implies the same pattern of parallelism as the previous example, despite the calls to `f` and `g` being embedded in the same expression.

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

Figure 2.6: Swift/T example: dataflow parallelism between expressions

2.3.4 Conditional Statements

Code in Swift can be conditionally executed using the `if` and `switch` statements. We omit discussion of `switch` statements here for the sake of brevity. The `if` statement's syntax is identical to that used in many imperative programming languages, such as C, but it executes in a data-dependent manner consistent with the rest of Swift. The condition of an 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 code in Figure 2.7, which 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 parallel 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.

```
1  import random;
2
3  float f1, f2;
4
5  f1 = simulationA();
6  f2 = simulationB();
7
8  if (f1 > f2) {
9    printf("Simulation A won!")
10 } else {
11   printf("Simulation B won!")
12 }
13
```

Figure 2.7: Swift/T example: conditional execution with if statement

2.3.5 Data-dependent Control Flow

Swift programmers can add explicit dependencies into their programs with two different constructs: the `wait` statement and the `=>` chaining operator. These constructs explicitly make statements depend on data, so that the statements execute only after the data is frozen, even if the statements do not actually depend on the value of the data. This is useful to add delays to a program or to print messages reporting progress. Most Swift functions have at least one output argument. Many functions that do not produce any output data have a `void` output argument that signals when the function has finished executing.

These two constructs differ subtly in several ways. `=>` waits on a statement, while `wait` waits on the expression supplied as its argument. Not all statements support chaining – the

```
1  import sys;
2
3  //  Chaining of multiple statements
4  printf("Going to sleep") =>
5    sleep(1) =>
6    printf("Woke up") =>
7    sleep(1) =>
8    printf("Woke up again");
9
10 x = compute_something();
11
12 //  The following forms are equivalent:
13 x => printf("Done!");
14
15 wait (x) {
16   printf("Done!");
17 }
```

Figure 2.8: Swift/T example: data-dependent control flow

statements must produce some kind of output variable. `=>` can have any statement on its right hand side, while `wait` must be followed by a block enclosed in curly braces.

2.3.6 Foreach Loops and Arrays

Foreach loops are tied closely with Swift arrays, so we introduce both constructs simultaneously.

Arrays in Swift are associative arrays: finite 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 with integer keys that do not need to be contiguous. There are multiple ways to declare and initialize arrays, as shown in Figure 2.9.

The workhorse control flow construct in most Swift programs is the `foreach` loop for parallel iteration over members of Swift data structures, including arrays. Iterations of a foreach loop are independent and execute in parallel, provided that data dependencies allow. Iteration over an array constructed with the `[begin:end:step?]` syntax is the idiomatic way to iterate over a range of integers in Swift. Literally, this instructs Swift to construct

```
1  // Two equivalent ways of declaring an array A mapping integers to strings
2  string A[int];
3  string A[];
4
5  // Declaration of an array mapping strings to integers
6  int A2[string];
7
8  // Equivalent statements that initialize an array with the numbers from 1 to 4
9  B = [1, 2, 3, 4]; // List of values (keys 0-3 are implied)
10 B = [1:4]; // Integer range (keys 0-3 are implied)
11 B = [1:4:1]; // Integer range with explicit step of 1
12 B = { 0: 1, 1: 2, 2: 3, 3: 4 }; // Explicit keys
13 // Assigning piece-by-piece
14 B[0] = 1;
15 B[1] = 2;
16 B[2] = 3;
17 B[3] = 4;
18
19 // Declaring two-dimensional nested array
20 string C[][];
21 C[0][0] = "top-left";
22 C[0][1] = "top-right";
23 C[1][0] = "bottom-left";
24 C[1][1] = "bottom-right";
```

Figure 2.9: Swift/T example: array declarations

an array and then iterate over it. However, Swift/T optimizes this idiom in such a way that construction of the intermediate array is always avoided, so the idiom comes with no performance penalty. To illustrate, Figure 2.10 shows code that builds an array by iterating over a range of integers and then iterating over the constructed array.

In the above example, the iterations of each loop will execute in an arbitrary order: the results will most likely not print in ascending order. Printing the loop in order can be achieved by combining ordered *for loops* (Section 2.3.10) with explicit data-dependent control flow (Section 2.3.5).

2.3.7 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

```
1  import io;
2  import stats;
3  import sys;
4
5  //  Get keyword command-line argument n, with default value of 100
6  int n = parseInt(argv("n", "100"));
7
8  float harmonic[];
9
10 //  Compute the harmonic series.
11 //  Note that this literally instructs Swift to construct an array containing
12 //  integers 1 to n, then iterate over the constructed array.  However, Swift/T
13 //  always optimizes this to iterate over the range without building the array.
14 foreach i in [1:n] {
15   harmonic[i] = 1 / toFloat(i);
16 }
17
18 //  Iterate over values and indices
19 foreach x, i in harmonic {
20   printf("H[%i] = %f", i, x);
21 }
22
23 printf("sum = %f", sum(harmonic));
```

Figure 2.10: Swift/T example: basic foreach loops

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. Figure 2.11 illustrates Swift functions through different implementations of the factorial function.

Function bodies can begin executing as soon as the function is called, regardless of the state of their input and output arguments. In Figure 2.12, assignment of each function input is delayed a different amount. The `printf` function calls in the function will execute at approximately one-second intervals once inputs are assigned.

2.3.8 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 heavily on

```
1  import io;
2  import sys;
3
4  x_val = parseInt(argv("x", "5"));
5
6  f1, f2 = fact2(x_val);
7
8  printf("fact(%i) = %i", x_val, f1);
9  printf("fact_tail(%i) = %i", x_val, f2);
10
11 //  Recursive implementation of factorial.
12 (int result) fact(int x) {
13   if (x == 0) {
14     result = 1;
15   } else {
16     result = x * fact(x - 1);
17   }
18 }
19
20 //  Tail-recursive implementation of factorial.
21 (int result) fact_tail(int x, int accum) {
22   if (x == 0) {
23     result = accum;
24   } else {
25     result = fact_tail(x - 1, accum * x);
26   }
27 }
28
29 //  Compute factorial in two ways, illustrating multiple output arguments
30 (int r1, int r2) fact2(int x) {
31   r1 = fact(x);
32   r2 = fact_tail(x, 1);
33 }
```

Figure 2.11: Swift/T example: basic Swift functions computing factorials

integration with foreign functions written in programming languages including C, C++, Fortran, Python, Tcl, alongside the command-line applications traditionally supported by Swift [115]. 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 [79], which can be automatically generated for C, C++ and Fortran code using tools like SWIG [14]. Foreign Tcl functions must specify a Tcl package name and version to be loaded, and then provide a Tcl code template that is filled in with input variables as appropriate. The Swift/T compiler automatically generates all necessary code to

```
1  import io;
2  import sys;
3
4  print_three(string x, string y, string z) {
5    printf("%s", x);
6    printf("%s", y);
7    printf("%s", z);
8  }
9
10 a = "Now";
11
12 sleep(1) =>
13   b = "Later" =>
14   sleep(1) =>
15   c = "Even later";
16
17 print_three(a, b, c);
```

Figure 2.12: Swift/T example: delayed assignment of Swift function arguments illustrating execution of Swift function body before arguments are all assigned.

defer execution until input arguments are assigned and then marshal the arguments between Tcl data and Swift's internal representations. Figure 2.13 shows a simple foreign function implemented in Tcl.

```
1  import io;
2
3  //  Declaration of log to arbitrary base (like log function in math module)
4  @pure @dispatch=WORKER
5  (float o) my_log (float x, float base) "turbine" "0.7.0" [
6    "set <<o>> [ expr {log(<<x>>)/log(<<base>>)} ]"
7  ];
8
9  printf("log10(100) = " + my_log(100, 10));
```

Figure 2.13: 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 annotation allows the Swift/T optimizer to reuse results of the function instead of recomputing them if needed. 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). Other annotations are documented

in the Swift/T user guide [105].

2.3.9 Advanced Usage of Arrays

Swift arrays are flexible data structures that can be used in many ways. One common pattern of usage is to use arrays to pass values from one iteration of a loop to the next. This implicitly adds data dependencies between statements in different loop iterations. For example, the code in Figure 2.14 refers to array values computed in other iterations of the same loop in order to compute the cumulative sum.

```
1  int n = 100;
2
3  //  Compute cumulative sum of some function f
4  float cumulative[];
5  cumulative[0] = f(0);
6
7  foreach i in [1:n] {
8    //  Each value of f can be computed independently , but data dependencies
9    //  force the sum to be computed by summing results sequentially
10   cumulative[i] = cumulative[i - 1] + f(i);
11 }
12
13 printf("Final sum: %f", cumulative[n]);
```

Figure 2.14: Swift/T example: cumulative sum in arrays

Other usage of arrays depends on the entire, final, contents of the array being known. For example, the size of the array or the full set of keys and values can change over time as different assignment statements finish executing. If a Swift expression refers to one of these properties, for example, by using the expression `size(A)` or by passing an array into a foreign function, evaluation of that expression is deferred until the array is automatically *frozen* [1]. The Swift implementation freezes an array once all evaluation of all Swift statements that could assign the array has completed. Figure 2.15 gives a simple example of statements that execute at different times based on when an array is frozen.

1. The term "closed" is sometimes used instead of "frozen" in Swift literature

```
1  string A[] = compute();
2
3  //  Prints once A[0] is assigned
4  printf("A[0] = %s", A[0]);
5
6  //  Prints once A is frozen
7  printf("size(A) = %i", size(A));
8
9  //  String representation of entire array
10 //  Prints once A is frozen
11 printf("A = %s", repr(A));
```

Figure 2.15: Swift/T example: automatic freezing of an array

2.3.10 For Loops

The `foreach` loops, introduced earlier, enables independent parallel iteration but does not allow one iteration to pass data (directly) to the next iteration (although it is possible to achieve this indirectly with creative use of arrays). Sometimes the ability to chain together loop iterations by passing data is needed: for example, if the number of iterations to execute is not known or if each iteration consumes the result of the previous.

The `for` loop construct enables this behavior. 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. A simple example is shown in Figure 2.16.

Swift/T's for loop is quite different from the for loop of imperative languages. Variables in Swift/T are single-assignment so cannot be mutated on each loop iteration. Instead, each variable is bound to a different storage location on each iteration. For this reason, Swift/T has special handling of *iteration variables* that appear in the initializer clause: they can refer to a different 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.

```
1  import io;
2  import random;
3
4  int desired_heads = 10;
5
6  //  Simulate tossing a coin until we get the needed number of heads
7  for (int nheads = 0; nheads < desired_heads; nheads = nheads + addtl_heads) {
8    int addtl_heads;
9    printf("Toss!");
10
11   if (random() > 0.5) {
12     addtl_heads = 1;
13   } else {
14     addtl_heads = 0;
15   }
16 }
```

Figure 2.16: Swift/T example: for loop.

In Figure 2.16, we were not able to refer to the loop variables outside of the for loop. Figure 2.17 shows howwe can add additional iteration variables to serve 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.

An astute reader may notice that a for loop is nearly identical in semantics and implementation to a tail-recursive function. This is true, and in many use cases, the for loop simply serves as syntactic sugar for tail recursion. However, the variable scoping rules in Swift/T have one substantial difference between tail-recursive functions and for loops: code in a for loop can partially or totally assign variables in the enclosing scope, while there is no way for functions in Swift/T to partially assign non-local variables outside of the scope of the function.

2.3.11 Structs

Swift supports *struct* types comprised of named fields – analogous to struct types in other programming languages. Struct fields can be assigned individually or a struct can be built all at once with a *constructor function* – a function with the same name as the struct type

```
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 2.17: Swift/T example: for loop with accumulator.

that is automatically defined. Figure 2.18 illustrates declaration and basic usage of structs.

```
type my_struct {
  string foo;
  float bar;
}

//  Assign elements individually
my_struct x;
x.foo = "baz";
x.bar = 0.0;

trace(x.foo, x.bar);

//  Initialise with constructor function
my_struct y = my_struct("qux", 44.0);
trace(y.foo, y.bar);

//  Copy struct
my_struct z = y;
trace(z.foo, z.bar);
```

Figure 2.18: Swift/T example: structs

2.3.12 *Files and App Functions*

Swift supports files as a first-class data type that can be treated similarly to a scalar value in the program. It also supports *app functions*: command-line programs that are wrapped as typed Swift functions. This means that scripts manipulating files and invoking command-line applications can be expressed with regular Swift variables and function calls, as shown below in Figure 2.19!

```
1 import files;
2
3 app (file out) cat (file inputs[]) {
4   "/bin/cat" inputs @stdout=out
5 }
6
7 file inputs[] = glob("*.txt");
8
9 file joined <"joined.txt"> = cat(inputs);
```

Figure 2.19: Swift/T example: files and app functions

2.4 Swift/T Grammar

The formal syntax for Swift/T is specified with EBNF in Figures 2.20, 2.21, 2.22, and 2.23 for reference purposes.

⟨*translation-unit*⟩ ::= ⟨*statement*⟩⋆ *(translation unit - Swift file)*

⟨*statement*⟩ ::= `';'` | ⟨*new-type-defn*⟩ | ⟨*global-const-defn*⟩ | ⟨*import-stmt*⟩ | ⟨*pragma-stmt*⟩ *(program statement)*
| ⟨*func-defn*⟩ | ⟨*block*⟩ | ⟨*if-stmt*⟩ | ⟨*switch-stmt*⟩ | ⟨*wait-stmt*⟩ | ⟨*foreach-loop*⟩ | ⟨*for-loop*⟩
| ⟨*iterate-loop*⟩ | ⟨*stmt-chain*⟩ | ⟨*update-stmt*⟩

⟨*new-type-defn*⟩ ::= `'type'` ⟨*type-name*⟩ `'{'` (⟨*var-decl*⟩ `';'`)⋆ `'}'` *(new struct or subtype)*
| `'type'` ⟨*type-name*⟩ ⟨*standalone-type*⟩ `';'` | `'typedef'` ⟨*type-name*⟩ ⟨*standalone-type*⟩ `';'`

⟨*global-const-defn*⟩ ::= `'global' 'const'` ⟨*var-decl*⟩ `';'` *(global constant)*

⟨*import-stmt*⟩ ::= `'import'` ⟨*module-path*⟩ `';'` | `'import'` ⟨*string*⟩ `';'` *(module import)*

⟨*module-path*⟩ ::= ⟨*id*⟩ (`'.'` ⟨*id*⟩)⋆ *(module path)*

⟨*pragma-stmt*⟩ ::= `'pragma'` ⟨*id*⟩ ⟨*expr*⟩⋆ `';'` *(pragma statement)*

⟨*func-defn*⟩ ::= ⟨*swift-func-defn*⟩ | ⟨*app-func-defn*⟩ | ⟨*foreign-func-defn*⟩ *(function definitions)*

⟨*func-hdr*⟩ ::= ⟨*type-params*⟩? ⟨*formal-arg-list*⟩? ⟨*func-name*⟩ ⟨*formal-arg-list*⟩? *(function header)*

⟨*type-params*⟩ ::= `'<'` ⟨*var-name*⟩ (`','` ⟨*var-name*⟩)⋆ `'>'` *(type variable parameters)*

⟨*formal-arg-list*⟩ ::= `'('` (⟨*formal-arg*⟩ (`','` ⟨*formal-arg*⟩)⋆)? `')'` *(formal argument list)*

⟨*formal-arg*⟩ ::= ⟨*type-prefix*⟩ `'...'`? ⟨*var-name*⟩ ⟨*type-suffix*⟩ (`'='` ⟨*expr*⟩)? *(formal argument)*

⟨*swift-func-defn*⟩ ::= ⟨*annotation*⟩⋆ ⟨*func-hdr*⟩ ⟨*block*⟩ *(Swift function definition)*

⟨*app-func-defn*⟩ ::= ⟨*annotation*⟩⋆ `'app'` ⟨*func-hdr*⟩ `'{'` ⟨*app-body*⟩ `'}'` *(app function definition)*

⟨*app-body*⟩ ::= ⟨*app-arg-expr*⟩+ (`'@'`(`'stdin'`|`'stdout'`|`'stderr'`) `'='` ⟨*expr*⟩)⋆ `';'`? *(app function body)*

⟨*foreign-func-defn*⟩ ::= ⟨*annotation*⟩⋆ ⟨*func-hdr*⟩ ⟨*foreign-func-body*⟩ *(foreign function definition)*

⟨*foreign-func-body*⟩ ::= ⟨*string*⟩ ⟨*string*⟩ ⟨*string*⟩? (`'['` ⟨*string*⟩|⟨*multiline-string*⟩`']'`)? *(foreign function body)*

⟨*var-decl*⟩ ::= ⟨*type-prefix*⟩ ⟨*var-decl-rest*⟩ (`','` ⟨*var-decl-rest*⟩)⋆ *(variable declaration)*

⟨*var-decl-rest*⟩ ::= ⟨*var-name*⟩ ⟨*type-suffix*⟩ ⟨*var-mapping*⟩? (`'='` ⟨*expr*⟩)? *(rest of variable declaration)*

⟨*type-prefix*⟩ ::= ⟨*type-name*⟩ | ⟨*param-type*⟩ *(type declaration prefix)*

⟨*param-type*⟩ ::= ⟨*type-name*⟩ `'<'` ⟨*standalone-type*⟩ `'>'` *(parameterized type)*

⟨*type-suffix*⟩ ::= (`'['` ⟨*standalone-type*⟩? `']'`)⋆ *(type declaration suffix)*

⟨*standalone-type*⟩ ::= ⟨*type-prefix*⟩ ⟨*type-suffix*⟩ *(standalone type)*

⟨*var-mapping*⟩ ::= `'<'` ⟨*expr*⟩ `'>'` *(variable mapping declaration)*

⟨*block*⟩ ::= `'{'` ⟨*statement*⟩⋆ `'}'` *(code block)*

⟨*stmt-chain*⟩ ::= ⟨*chainable-stmt*⟩ (`';'`|`'='` `'>'`⟨*stmt*⟩) *(statement chain)*

⟨*chainable-stmt*⟩ ::= ⟨*var-name*⟩ | ⟨*func-call*⟩ | ⟨*var-decl*⟩ | ⟨*assignment*⟩ *(statement that supports chaining)*

⟨*assignment*⟩ ::= (⟨*lval-list*⟩|`'('` ⟨*lval-list*⟩ `')'`) (`'='`|`'+='`) ⟨*expr-list*⟩ *(assignment)*

⟨*update-stmt*⟩ ::= ⟨*var-name*⟩ `'<'` ⟨*id*⟩ `'>'` `':='` ⟨*expr*⟩ `';'` *(update statement)*

Figure 2.20: Grammar for Swift/T variant of Swift programming language in EBNF syntax [39] extended with ? for optional elements and + for one or more repetitions. Figures 2.21, 2.23, and 2.22 contain additional grammar rules.

⟨*if-stmt*⟩ ::= **'if' '(' ⟨*expr*⟩ ')' ⟨*block*⟩ ('else' ⟨*block*⟩)?** *(if statement)*

⟨*switch-stmt*⟩ ::= **'switch' '(' ⟨*expr*⟩ ')' '{' ⟨*case*⟩⋆ ⟨*default*⟩? '}'** *(switch statement)*

⟨*case*⟩ ::= **'case' ⟨*int*⟩ ':' ⟨*stmt*⟩⋆** *(switch case)*

⟨*default*⟩ ::= **'default' ':' ⟨*stmt*⟩⋆** *(switch default case)*

⟨*wait-stmt*⟩ ::= **'wait' 'deep'? '(' ⟨*expr-list*⟩ ')' ⟨*block*⟩** *(wait statement)*

⟨*foreach-loop*⟩ ::= ⟨*annotation*⟩⋆ **'foreach' ⟨*var-name*⟩ (',' ⟨*var-name*⟩)? 'in' ⟨*expr*⟩ ⟨*block*⟩** *(foreach loop)*

⟨*for-loop*⟩ ::= ⟨*annotation*⟩⋆ **'for' '(' ⟨*for-init-list*⟩ ';' ⟨*expr*⟩ ';' ⟨*for-update-list*⟩ ')' ⟨*block*⟩** *(for loop)*

⟨*for-init-list*⟩ ::= ⟨*for-init*⟩ (',' ⟨*for-init*⟩)⋆ *(for loop initializer list)*

⟨*for-init*⟩ ::= ⟨*for-assignment*⟩ | ⟨*type-prefix*⟩ ⟨*var-name*⟩ ⟨*type-suffix*⟩ '=' ⟨*expr*⟩ *(for loop initializer)*

⟨*for-update-list*⟩ ::= ⟨*for-assignment*⟩ (',' ⟨*for-assignment*⟩)⋆ *(for loop update list)*

⟨*for-assignment*⟩ ::= ⟨*var-name*⟩ '=' ⟨*expr*⟩ *(for loop assignment)*

⟨*iterate-loop*⟩ ::= **'iterate' ⟨*var-name*⟩ ⟨*block*⟩ 'until' '(' ⟨*expr*⟩ ')'** *(iterate loop)*

Figure 2.21: Extended Backus-Naur Form grammar for Swift/T control-flow statements.

⟨*id*⟩ ::= (⟨*alpha*⟩|'_') (⟨*alpha*⟩|'_'|⟨*digit*⟩)⋆ *(identifier)*

⟨*string*⟩ ::= '"' ('\' .|¬'"')⋆ '"' *(string literal)*

⟨*multiline-string*⟩ ::= '----\n' .?⋆ '----' | '"""\n' .?⋆ '"""' *(multi-line string literal)*

⟨*int*⟩ ::= ⟨*digit*⟩⋆ | '0x'(⟨*digit*⟩|['a'..'f']|['A'..'F'])⋆ *(integer literal)*

⟨*decimal*⟩ ::= ⟨*digit*⟩+ '.' ⟨*digit*⟩+ *(decimal floating point literal)*

⟨*sci-decimal*⟩ ::= ⟨*digit*⟩+ ('.' ⟨*digit*⟩+)? ('e'|'E') '-'? ⟨*digit*⟩+ *(decimal scientific notation literal)*

⟨*digit*⟩ ::= ['0'..'9'] *(decimal digit)*

⟨*alpha*⟩ ::= ['a'..'z'] | ['A'..'Z'] *(alphabet character)*

⟨*line-comment*⟩ ::= ('//'|'#') (¬'\n')⋆ '\n' *(single-line comment)*

⟨*multi-line-comment*⟩ ::= '/*' .?⋆ '*/' *(multi-line comment)*

Figure 2.22: Extended Backus-Naur Form grammar for Swift/T lexer tokens. ?⋆ means non-greedy matching.

⟨*expr*⟩ ::= ⟨*or-expr*⟩ *(RValue expression)*

⟨*or-expr*⟩ ::= ⟨*and-expr*⟩ | ⟨*or-expr*⟩ `'||'` ⟨*and-expr*⟩ *(or expression)*

⟨*and-expr*⟩ ::= ⟨*eq-expr*⟩ | ⟨*and-expr*⟩ `'&&'` ⟨*eq-expr*⟩ *(and expression)*

⟨*eq-expr*⟩ ::= ⟨*cmp-expr*⟩ | ⟨*eq-expr*⟩ (`'=='`|`'!='`) ⟨*eq-expr*⟩ *(equality expression)*

⟨*cmp-expr*⟩ ::= ⟨*add-expr*⟩ | ⟨*cmp-expr*⟩ (`'<'`|`'<='`|`'>'`|`'>='`) ⟨*add-expr*⟩ *(comparison expression)*

⟨*add-expr*⟩ ::= ⟨*mult-expr*⟩ | ⟨*add-expr*⟩ (`'+'`|`'-'`) ⟨*mult-expr*⟩ *(addition precedence expression)*

⟨*mult-expr*⟩ ::= ⟨*pow-expr*⟩ *(multiplication precedence expression)*

| ⟨*mult-expr*⟩ (`'*'`|`'/'`|`'%/'`|`'%%'`|`'%'`) ⟨*pow-expr*⟩

⟨*unary-expr*⟩ ::= ⟨*unary-expr*⟩ | ⟨*pow-expr*⟩ `'**'` ⟨*unary-expr*⟩ *(power expression)*

⟨*unary-expr*⟩ ::= ⟨*postfix-expr*⟩ | (`'-'`|`'!'`) ⟨*postfix-expr*⟩ *(unary expression)*

⟨*postfix-expr*⟩ ::= ⟨*base-expr*⟩ | ⟨*postfix-expr*⟩(⟨*array-subscript*⟩|⟨*struct-subscript*⟩) *(postfix expression)*

⟨*array-subscript*⟩ ::= `'['` ⟨*expr*⟩ `']'` *(array key subscript)*

⟨*struct-subscript*⟩ ::= `'.'` ⟨*id*⟩ *(struct member subscript)*

⟨*base-expr*⟩ ::= ⟨*literal*⟩ | ⟨*func-call*⟩ | ⟨*var-name*⟩ | `'('` ⟨*expr*⟩ `')'` | ⟨*tuple-constructor*⟩ *(base expressions)*

| ⟨*array-constructor*⟩

⟨*func-call*⟩ ::= ⟨*annotation*⟩ ⋆ ⟨*func-name*⟩ `'('` ⟨*func-call-arg-list*⟩ `')'` *(function call)*

⟨*func-call-arg-list*⟩ ::= (⟨*expr*⟩|⟨*kw-expr*⟩) (`','` (⟨*expr*⟩|⟨*kw-expr*⟩))⋆ *(function call argument list)*

⟨*tuple-constructor*⟩ ::= `'('` ⟨*expr*⟩ (`','` ⟨*expr*⟩) + `')'` *(tuple constructor)*

⟨*array-constructor*⟩ ::= ⟨*array-list-constructor*⟩ | ⟨*array-range-constructor*⟩ *(array constructor)*

| ⟨*array-kv-constructor*⟩

⟨*array-list-constructor*⟩ ::= `'['` ⟨*expr-list*⟩? `']'` *(array list constructor)*

⟨*array-range-constructor*⟩ ::= `'['` ⟨*expr*⟩ `':'` ⟨*expr*⟩ (`':'` ⟨*expr*⟩)? `']'` *(array range constructor)*

⟨*array-kv-constructor*⟩ ::= `'{'` (⟨*array-kv-elem*⟩ (`','` ⟨*array-kv-elem*⟩)⋆)? `'}'` *(array key-value constructor)*

⟨*array-kv-elem*⟩ ::= ⟨*expr*⟩ `':'` ⟨*expr*⟩ *(array key-value)*

⟨*annotation*⟩ ::= `'@'` ⟨*id*⟩ | `'@'` ⟨*kw-expr*⟩ *(annotation)*

⟨*kw-expr*⟩ ::= ⟨*id*⟩ `'='` ⟨*expr*⟩ *(keyword argument)*

⟨*literal*⟩ ::= ⟨*string-literal*⟩ | ⟨*multiline-string-literal*⟩ | ⟨*int-literal*⟩ | ⟨*float-literal*⟩ *(literal value)*

| ⟨*bool-literal*⟩

⟨*float-literal*⟩ ::= ⟨*decimal*⟩ | ⟨*sci-decimal*⟩ | `'inf'` | `'NaN'` *(floating point literal)*

⟨*bool-literal*⟩ ::= `'true'` | `'false'` *(boolean literal)*

⟨*expr-list*⟩ ::= ⟨*expr*⟩ (`','` ⟨*expr*⟩)⋆ *(comma-separated expression list)*

⟨*type-name*⟩ ::= ⟨*id*⟩ *(type name)*

⟨*var-name*⟩ ::= ⟨*id*⟩ *(variable name)*

⟨*func-name*⟩ ::= ⟨*id*⟩ *(function name)*

⟨*lval-list*⟩ ::= ⟨*lval-expr*⟩ (`','`⟨*lval-expr*⟩)⋆ *(assignment LValue expression list)*

⟨*lval-expr*⟩ ::= ⟨*var-name*⟩ (⟨*array-subscript*⟩|⟨*struct-subscript*⟩)⋆ *(assignment LValue expression)*

⟨*app-arg-expr*⟩ ::= `'@'`?⟨*var-name*⟩ | ⟨*literal*⟩ | ⟨*array-constructor*⟩ *(app function argument expression)*

| `'('` ⟨*expr*⟩ `')'`

Figure 2.23: Extended Backus-Naur Form grammar for Swift/T expressions.

CHAPTER 3

THE SWIFT EXECUTION MODEL

In this chapter we introduce an execution model for *data-driven task parallelism* and formalize its semantics. The execution model provides a foundation for the semantics of the Swift programming language. 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 are executed in a nondeterministic order or in a concurrent manner with interleaved reads and writes, while accessing a shared data store.

Figure 3.1 illustrates how the execution model relates to previous work. Many of the basic ideas in the execution model, including the data types and automatic freezing, have existed in a prototypical and informally specified form since the initial versions of Swift [115]. 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 situation 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.

Our execution model formalizes and generalizes a core execution model that is based on Swift's semantics and proves determinism of the execution model. The lattice data types used in our execution model are closely related to LVars, which recent work has proposed as a solution for for deterministic parallel programming [59]. Related work on concurrent logic

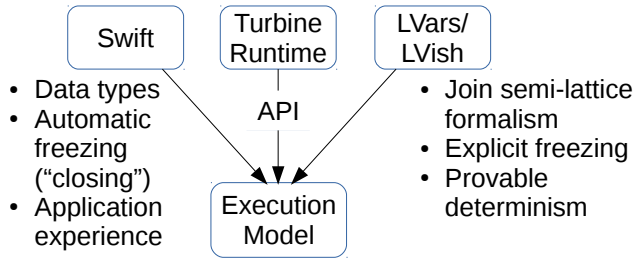


Figure 3.1: Lineage of ideas in the execution model.

and constraint programming [99, 97] dates back even further. We leave comparison of our work and LVars until Section 3.9.1 at the end of the chapter.

The execution model's constructs correspond fairly closely to the programming interface of the Turbine distributed runtime system [118] and is well suited to practical implementation on massively parallel distributed systems and allows a great deal of freedom to transform and optimize programs in the execution model: it provides a theoretical foundation for our practical work on our massively parallel runtime (Chapter 4) and for our optimizing compiler (Chapter 5).

3.1 Overview

In data-driven task parallelism, all computation is performed by ***tasks***, which are abstracted as mathematical functions that take input values and compute outputs of various kinds. Once executing, tasks run to completion and are not preempted. Tasks communicate by reading and writing ***shared data*** that resides in a data store. A task declares a set of shared data items that it will read and the computed output of a task includes a set of write operations on shared data items. 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 that data is available.

To visualize the execution model, we will use a graphical notation for ***trace graphs*** that show the tasks, shared data, and dependencies that arise during execution. A trace graph such as Figure 3.2 illustrates a single runtime execution of a program. Note that a single static graph cannot always serve as a specification of the data-driven tasks program because dependencies emerge dynamically at runtime. Tasks may selectively read, write, or spawn based on values computed at runtime: the tasks and relationships between tasks may vary between different executions of the same program, e.g. if the input data is varied. A trace graph never has cycles: in the case of deadlocks, deadlocked tasks will have fewer in-edges

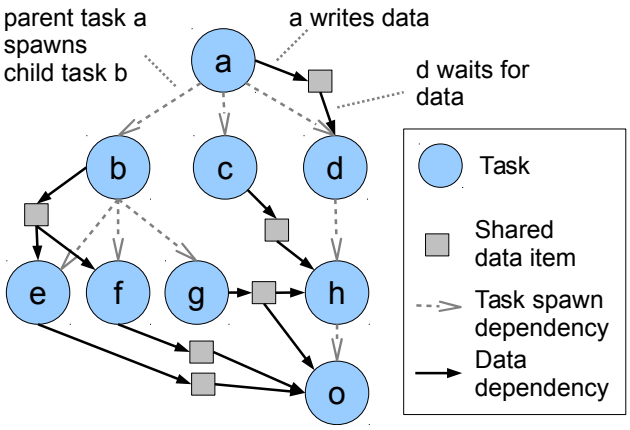


Figure 3.2: Trace graph showing task and data dependencies at runtime in data-driven task parallelism, forming a spawn tree rooted at task a. Data dependencies on shared data defer execution of tasks until the variables in question are frozen. Thus, for example, task h cannot execute until a data item is written by task c.

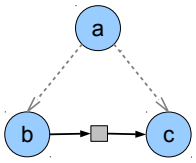


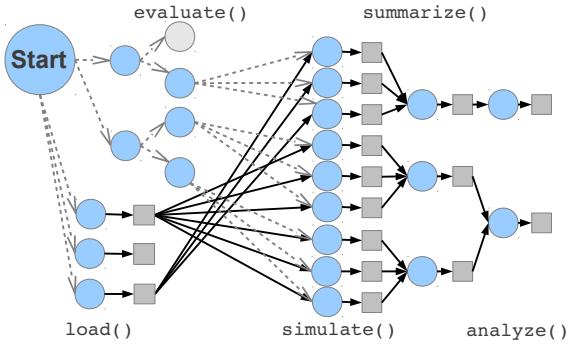
Figure 3.3: Task spawning two children that synchronize on an item of data.

than data dependencies.

Each task can spawn asynchronous ***child tasks***, resulting in a ***spawn tree*** of tasks as in Figure 3.2. In practice, tasks can be implemented through a set of parameterized ***task definitions*** that make up a program: at spawn time a task definition's parameters are bound to specific values by the parent to produce a child task. This allows parent tasks to pass data directly to their child tasks, for example small data such as numbers or short strings, along with references to arbitrary shared data. Shared data items can be read or written by any task that obtains a ***reference*** to the data. Shared data items provide a means for coordination between multiple tasks. For example, a task A can spawn two tasks, B and C, passing both a reference to a shared data item, which B writes and C reads, as shown in Figure 3.3. ***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

```
1  blob models[], res[][];
2  foreach m in [1:N_models] {
3    models[m] = load(sprintf("model%i.data", m));
4  }
5
6  foreach i in [1:M] {
7    foreach j in [1:N] {
8      //  initial quick evaluation of parameters
9      p, m = evaluate(i, j);
10     if (p > 0) {
11       //  run ensemble of simulations
12       blob res2[];
13       foreach k in [1:S] {
14         res2[k] = simulate(models[m], i, j, k);
15       }
16       res[i][j] = summarize(res2);
17     }
18   }
19 }
20
21 //  Summarize results to file
22 foreach i in [1:M] {
23   file out<sprintf("output%i.txt", i)>;
24   out = analyze(res[i]);
25 }
```

(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 runtime.

Figure 3.4: Swift source code and data-driven task trace graph of optimized code for a simple application.

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 into the execution model is shown in Figure 3.4. The illustrated application, an amalgam of several real scientific applications, runs an ensemble of simulations for many parameter combinations. The code (on the left) 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., `models` and `res`). Swift/T semantics allow functions (e.g., `load`, `evaluate`, and `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 task graph (on the right) shows an optimized translation to data-driven

task parallelism. An unoptimized version would comprise more variables and tasks.

3.2 Notation

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

To express logical formulas, we use standard notation TRUE, FALSE, $\wedge$ (and), $\vee$ (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), ($intersection$) $\subset$ (strict subset), $\subseteq$ (subset), $\{x \mid p(x)\}$ (set-builder notation: all x for which predicate $p(x)$ is TRUE). 2^X is the set of all subsets of X: all possible combinations of elements of X. $\langle x \mid p(x)\rangle$ is the equivalent notations for ordered sequences.

To denote a ***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., $f : X \times Y \to A \times B$

$\langle x_1, ..., x_n\rangle$ denotes an ordered ***tuple*** or ***sequence*** with n elements. The $\cdot$ operator denotes concatenation of sequences: $\langle x_1, ..., x_n\rangle \cdot \langle y_1, ..., y_n\rangle = \langle x_1, ..., x_n, y_1, ..., y_n\rangle$. X^* is the set of all sequences of members of set X: $X^* = \{\langle x_1, ..., x_n\rangle \mid n \in [0, n] \wedge (\forall i \in [1, n], x_i \in X)\}$. The ***Perms*** function Perms : $X^* \to 2^{X^*}$ enumerates all permutations of a sequence.

Multisets are sets that allow duplicate elements. Notation $x_{|n|} \in X$ means that element x has multiplicity n in multiset X. If the multiplicity is omitted, 1 is assumed. $\uplus$ is the union operation for multisets, which adds the multiplicities in the two sets. For example, $X \uplus x$ increases the multiplicity of x in X by one. Multiset subtraction $-$ similarly subtracts the multiplicities, with a floor of 0.

Finite maps are functions mapping of keys to values that are specified by a finite number of key to value mappings. E.g., a finite map M from a key type K to a value type V is denoted as $M : K \xrightarrow{fin} V \subseteq K \to V$. 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 $dom(M)$.

A labelled transition system is a tuple $\langle S, T, \{\xrightarrow{t} | \ t \in T\}\rangle$, where S is a set of states, T is a set of transition labels, and $\xrightarrow{t} \subseteq S \times S$ is a labelled transition relation [70].

3.3 Data Types and Data Structures

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

Definition 3.3.1 (Values)**.** Let $\mathcal{V}$ be the set of values used in our execution model, which includes sequences, sets, finite maps, integers, rational numbers, and characters:

$$\mathcal{V} = \mathcal{V}^* \cup 2^{\mathcal{V}} \cup (\mathcal{V} \xrightarrow{fin} \mathcal{V}) \cup \mathbb{Z} \cup \mathbb{Q} \cup \Sigma \cup \{\bot, \top\}$$

This set of values is sufficient to represent a wide range of data.

In keeping with standard mathematical notation for lattices and the notation used to describe LVars [59], we use top and bottom values with the following intuitive meanings:

Definition 3.3.2 (Bottom Value)**.** The bottom value $\bot \in \mathcal{V}$ represents a value that is not yet present.

Definition 3.3.3 (Top Value)**.** The top value $\top \in \mathcal{V}$ represents an invalid value.

We now introduce the data structures used in the execution model. Each data structure is represented as its current state, plus a set of operations that can be applied to the state, either to extract information or to transition to a new state. The set of operations, plus the set of valid states, is referred to as a ***data type implementation***.

We choose to model data in this way because it is directly analogous to how data types are implemented in our runtime system: as an opaque state plus methods that operate on that state. We treat each data structure as a "black box" and propose reasonable properties on observable behavior that will allow us to achieve desirable system-level properties such as determinism.

Definition 3.3.4 (Data Type Implementations)**.** $\mathcal{T}$ is the set of data type implementations. Each $t \in \mathcal{T}$ is composed of a set of valid states along with four operations on the state: $t = \langle S, \mathit{read}, \mathit{frozen}, \mathit{update}, \mathit{freeze} \rangle$. For convenience, we use subscript notation to refer to the components of t: S_t, read_t, frozen_t, update_t, and freeze_t.

Definition 3.3.5 (States)**.** S_t is the set of valid states for $t \in \mathcal{T}$. S_t must contain both $\top$ and $\bot$ to represent its initial state and any invalid state respectively.

Definition 3.3.6 (Data Structures)**.** A data structure is a state along with the implementation of the data type. $\mathcal{D} = \{\langle s, t \rangle \mid t \in \mathcal{T} \ \wedge \ s \in S_t\}$ is the set of all data structures.

Definition 3.3.7 (Paths)**.** A path $p \in \mathcal{P} = \mathcal{V}^*$ is a sequences of values that refers to some part or property of a data structure.

Definition 3.3.8 (Null Path)**.** The null path $\langle\rangle \in \mathcal{P}$ identifies the entire data structure.

All of a type's functions take a path as an argument. Each function is free to interpret a path in any way desired by the implementer of the data type. The most straightforward use is for paths to denote components of a data structure, e.g., subscripts $A[0]..A[n]$ for an array A of size n. The *read* function can also interpret paths in such a way that reading a path results in computation of an arbitrary computable property of a data structure. This is useful for implementing properties of data structures that are not stored directly in state representation, but can be computed from it, such as the length of an array. For example, e.g., ⟨"length"⟩ could refer to the length of the arrays. It is also useful for exposing

information in such a way that the value of the path becomes immutable. For example, if we have an increasing counter data type, then the actual value of the counter is mutable, but the path $\langle \text{“}\geq 5\text{”} \rangle$ will be frozen once the counter reaches five or more.

Definition 3.3.9 (Read Function)**.** The read function allows information to be extracted from a data structure. It takes two arguments: a state and a path value that identifies that information to be extracted from the data structure and returns a value. The function type is $\mathit{read}_t : S_t \times \mathcal{P} \to \mathcal{V}$. The read function must be computable and total.

Definition 3.3.10 (Frozen Function)**.** The frozen function allows checking whether a path of a data structure is frozen, i.e., the value that results from reading the path will not change after applying any sequence of updates. The function type is $\mathit{frozen}_t : S_t \times \mathcal{P} \to \{\textsc{True}, \textsc{False}\}$. The frozen function must be computable and total.

Note that a path p can become frozen in one of two ways: either the entire data structure is *explicitly* frozen with the freeze_t function or p becomes *semantically* frozen, if the combined properties of the update_t and read_t functions guarantee that any reads to p in subsequent states have the same result as reads to p in the present state. We precisely define this concept later in Property 3.3.8.

Definition 3.3.11 (Update Function)**.** The update function adds information to a data structure, producing a new state. It takes three arguments: a data structure state, a path, and a value that the path should be updated to. It returns a new data structure state. The function type is $\mathit{update}_t : S_t \times \mathcal{P} \times \mathcal{V} \to S_t$. If update_t returns $\top$, this indicates an error has occurred: the update is in some sense invalid or not permitted. The update function must be computable and total.

Definition 3.3.12 (Freeze Function)**.** The freeze function freezes a data structure, which prevents any future updates to the data structure. The function type is $\mathit{freeze}_t : S_t \to S_t$. The freeze function must be computable and total.

3.3.1 Equivalence of Data Structures

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

Definition 3.3.13 (Update Transition System). $\langle S_t, \mathcal{P} \times \mathcal{V}, \{\overset{t,p,v}{\underset{up}{\rightarrow}} | \ p \in \mathcal{P} \ \wedge \ v \in \mathcal{V}\}\rangle$ is the labelled transition system induced by the $update_t$ function of $t \in \mathcal{T}$. $\forall s, s' \in S_t$, $\forall p \in \mathcal{P}$, $\forall v \in \mathcal{V}$, $s \overset{t,p,v}{\underset{up}{\rightarrow}} s'$ iff $update_t(s, p, v) = s'$ Note that, by Property 3.3.1, once the update system for any valid type has transitioned into the $\top$ state, it cannot transition out of it.

Definition 3.3.14 (Iterated Update Transition System). $\langle S_t \cup \top, \mathcal{P} \times \mathcal{V}^*, \{\overset{t,u}{\underset{up}{\Rightarrow}} | \ u \in (\mathcal{P} \times \mathcal{V})^*\}\rangle$ is the transition system induced by iterated application of $update_t$. Let $u = \langle\langle p_1, v_1\rangle, ..., \langle p_n, v_n\rangle\rangle$. Then, $s \overset{t,u}{\underset{up}{\Rightarrow}} s'$ $(s, s' \in S_t)$ iff $update_t(...(update_t(s, p_1, v_1)...), p_n, v_n) = s'$.

Now that we have defined the transition relations, we define an observational equivalence relation for data structures. There are cases where two distinct states of a data structure may be, for all intents and purpose, equivalent. For example, a set data type that supports adding items and checking if items are in the set may use different states for the same set of items, depending on the order in which they were added. If the state is implemented as a hash table, for example, 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 3.3.15 (Data Structure Equivalence). $\overset{t}{\sim}$ is an equivalence relation for data structures with type t based on observational equivalence. Two data structures are equivalent iff the *read* and *frozen* functions have the same values for all paths after applying the same sequence of updates to both states. Formally, $\forall t \in \mathcal{T}, \forall s_1, s_2 \in S_t$, $s_1 \overset{t}{\sim} s_2$ iff $\forall u \in (\mathcal{P} \times \mathcal{V})^*$, if $s_1 \overset{t,u}{\underset{up}{\Rightarrow}} s_1' \ \wedge \ s_2 \overset{t,u}{\underset{up}{\Rightarrow}} s_2'$ then $\forall p \in \mathcal{P}$:

$$read_t(s_1', p) = read_t(s_2', p) \ \wedge \ frozen_t(s_1', p) \Leftrightarrow frozen_t(s_2', p)$$

3.3.2 Valid Data Type Implementations

The preceding definitions place minimal restrictions on the behavior of a data type implementation. We now present a set of reasonable restrictions on these implementations so that they are well-behaved – or *valid*.

Definition 3.3.16 (Valid Data Type Implementations)**.** $\mathcal{T}_{valid} \subset \mathcal{T}$ is the set of valid data type implementations.

Property 3.3.1 (Update $\top$)**.** Updating $\top$ must always result in $\top$. $\forall t \in \mathcal{T}_{valid}$, $\forall p \in \mathcal{P}$, $\forall v \in \mathcal{V}$, $update_t(\top, p, v) = \top$.

Property 3.3.2 (Freeze $\top$)**.** Freezing $\top$ must always result in $\top$. $\forall t \in \mathcal{T}_{valid}$, $\forall p \in \mathcal{P}$, $\forall v \in \mathcal{V}$, $freeze_t(\top) = \top$.

Property 3.3.3 (Reads of $\top$)**.** Reads of any path in $\top$ must return $\top$: $\forall t \in \mathcal{T}_{valid}$, $\forall p \in \mathcal{P}$, $read_t(\top, p) = \top$.

Property 3.3.4 (Frozen Path Prefix)**.** If path is frozen, then all paths that have that path as a prefix must also be frozen: $\forall t \in \mathcal{T}_{valid}$, $\forall s \in S_t$, $\forall p, p' \in \mathcal{P}$, $frozen_t(s, p) \Rightarrow frozen_t(s, p \cdot p')$

Property 3.3.5 (Read to Root Gives Entire State)**.** Valid types must handle the null path so that the *read* returns the complete information about the data structure, i.e., enough information to reconstruct an equivalent state. Formally, $\forall t \in \mathcal{T}_{valid}$, $\exists f : \mathcal{V} \to S_t$ s.t. $\forall r \in S_t$, $r \overset{t}{\sim} f(read_t(r, \langle\rangle))$.

Note that several of the prior properties are not strictly necessary in the sense that our theorems about the execution model hold independent of the properties. However, these properties rule out potentially counter-intuitive behavior without limiting the generality of the data structures that can be defined in our framework.

Property 3.3.6 (Path Remains $\top$)**.** Any path with read value of $\top$ must permanently remain in that state and is frozen. $\forall t \in \mathcal{T}_{valid}$, $\forall s \in S_t$, $\forall p \in \mathcal{P}$, $read_t(s, p) = \top \Rightarrow frozen_t(s, p)$

Property 3.3.7 (Freezing)**.** Once *freeze* is called on a data structure with a valid type, it must become frozen and not updatable. Reads should always return the same values as immediately before it was frozen. Formally, $\forall t \in \mathcal{T}_{valid}$, $\forall s, s' \in S_t$, $\forall p \in \mathcal{P}$, $\forall v \in \mathcal{V}$, if $s' = freeze_t(s)$ then:

$$frozen_t(s', p) \ \wedge \ update_t(s', p, v) = \top \ \wedge \ read_t(s', p) = read_t(s, p)$$

Property 3.3.8 (Reads of Frozen Paths)**.** After *frozen* has returned TRUE, the path must remain frozen and all reads must return the same values, *unless* the entire data structure is in the invalid state $\top$, in which case, by Property 3.3.3, *read* must return $\top$. Formally, $\forall t \in \mathcal{T}_{valid}$, $\forall p \in \mathcal{P}$, $\forall s, s' \in S_t$, $\forall u \in (\mathcal{P} \times \mathcal{V})^*$, if $frozen_t(s, p) \ \wedge \ s \overset{t,u}{\underset{up}{\Rightarrow}} s'$ then:

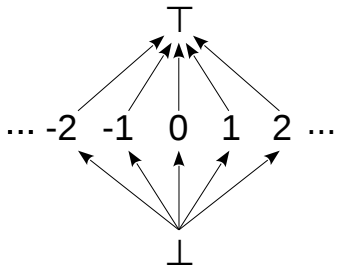
$$frozen_t(s', p) \ \wedge \ (s' = \top \vee read_t(s', p) = read_t(s, p))$$

We defer specifying the final restriction – Property 3.4.1 – until a later section because it is only relevant once further concepts are introduced.

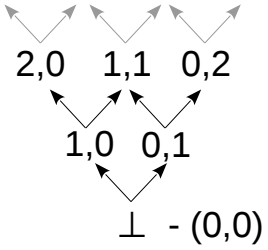
3.3.3 Lattice Data Types

In addition to the basic properties of valid data type implementations that we have already described, it is possible further restrict data types in useful ways. Previous work has shown that lattice-based data structures provide a framework that enables deterministic parallelism with data structures shared between threads [59]. A wide range of useful data structures can fit into the lattice framework [61, 100], including those shown in in Figure 3.5.

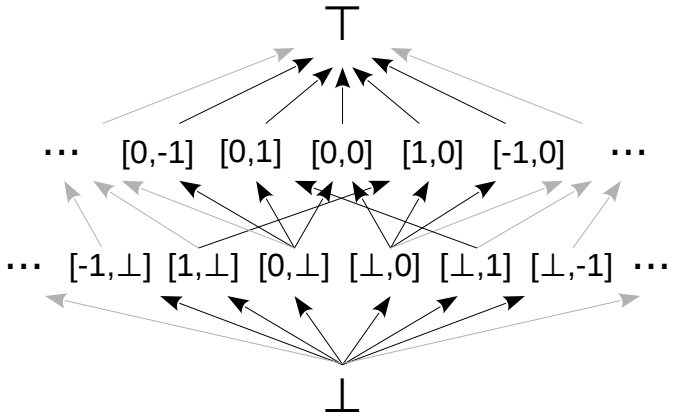
In this section we describe restrictions on a data type's functions that are necessary but not sufficient for a data type implementation to be a valid mathematical lattice. For brevity we refer to these data structures as lattice data structures even though not all such data types meet the requirements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



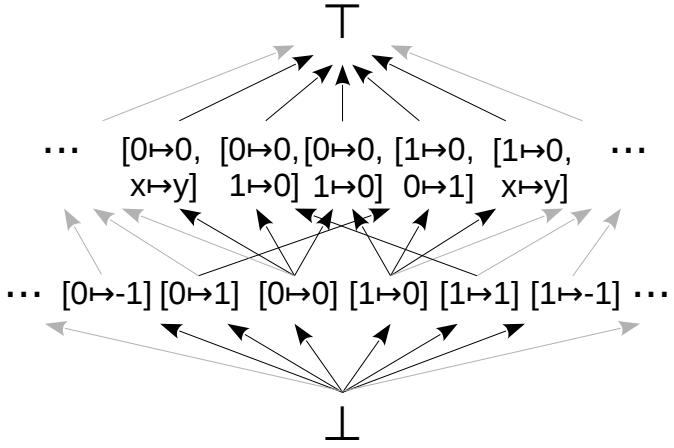
(a) Integer I-var (single-assignment variable)



(b) Counter pair with increment operation



(c) Integer I-var sequence of length two



(d) Finite map from integers to integers

Figure 3.5: Visualization of lattice data types. Arrows represent valid transitions. ⊤ is the top of the lattice – an invalid state. ⊥ is the bottom of the lattice – the initial value. Only a small part of the state space is shown for each type.

Definition 3.3.17 (Lattice Data Type Implementations)**.** t is a lattice data type implementation $t \in \mathcal{T}_{lat} \subset \mathcal{T}_{valid}$ if the $update_t$ function is commutative. Formally, $\forall t \in \mathcal{T}_{lat}$, $\forall s, s' \in S_t$, $\forall u \in (\mathcal{V} \times \mathcal{V})^*$, $\forall u' \in \text{Perms}(u)$, if $s \overset{t,u}{\underset{up}{\Rightarrow}} s' \ \wedge \ s \overset{t,u'}{\underset{up}{\Rightarrow}} s''$ then $s' \overset{t}{\sim} s''$

Informally, this means that a set of updates can be applied in any order to obtain an equivalent final result. Given the previous general restrictions on data structures, we can now prove a useful property of lattice data types: reads of frozen paths always return a consistent value, regardless of ordering of *read* and *update* operations. Equivalently, if a sequence of operations and a permutation thereof are applied to the same initial state, and a read of the same frozen path of any two intermediate states gives a different result, the final result of the operations is necessarily $\top$.

Theorem 3.3.1 (Consistent Frozen Reads under Permutation)**.** *Consider $t \in \mathcal{T}_{lat}$. $\forall s \in S_t$, $\forall u \in (\mathcal{P} \times \mathcal{V})^*$, $\forall u' \in Perms(u)$, $\exists s_{final}, s'_{final} \in S_t$ such that $s \overset{t,u}{\underset{up}{\Rightarrow}} s_{final}$ and $s \overset{t,u'}{\underset{up}{\Rightarrow}} s'_{final}$. Consider any way of splitting u and u' into prefix and suffix: $u = v_p \cdot v_s$ and $u' = v'_p \cdot v'_s$. Then, $\exists s_{prev}, s'_{prev} \in S_t$ such that $s \overset{t,v_p}{\underset{up}{\Rightarrow}} s_{prev}$ and $s \overset{t,v'_p}{\underset{up}{\Rightarrow}} s'_{prev}$. Reads of any path that is frozen in both s_{prev} and s_{prev} result in the same value, unless the complete sequence operations results in the invalid $\top$ state. I.e., reads will be consistent unless an error is inevitable, in which case the read values may be different between the intermediate states of the two traces. That is, $\forall p \in \mathcal{P}$, $frozen_t(s_{prev}, p) \wedge frozen_t(s'_{prev}, p) \wedge s_{final} \neq \top \Rightarrow read_t(s_{prev}, p) = read_t(s'_{prev}, p)$*

Proof. For the left hand side of the implication to be true, $frozen_t(s_{prev}, p)$ and $frozen_t(s'_{prev}, p)$. Also, $s_{prev} \overset{t,v_s}{\underset{up}{\Rightarrow}} s_{final}$ and $s'_{prev} \overset{t,v'_s}{\underset{up}{\Rightarrow}} s'_{final}$. Therefore, by Property 3.3.8, both final states must be frozen: $frozen_t(s_{final}, p)$ and $frozen_t(s'_{final}, p)$.

$t \in \mathcal{T}_{lat}$, so Definition 3.3.17 applies and $s_{final} \overset{t}{\sim} s'_{final}$. Furthermore, $s_{final} \neq \top$, so $s'_{final} \neq \top$, and therefore $read_t(s_{final}, p) = read_t(s'_{final}, p)$. By Property 3.3.8, $read_t(s_{final}, p) = read_t(s_{prev}, p)$ and $read_t(s'_{final}, p) = read_t(s'_{prev}, p)$. Therefore $read_t(s_{prev}, p) = read_t(s'_{prev}, p)$, which satisfies the right hand side of the implication in the theorem. □

3.3.4 Lattice Data Structure Examples

For the purposes of illustration, we will define some lattice data types in the framework described so far. We omit per-type definitions of *freeze* for the sake of brevity: it can be universally implemented with the technique shown later in Definition 3.3.25.

Definition 3.3.18 (Bottom Data Structure)**.** $T_\bot$ represents an uninitialized structure:

$$S_\bot = \{\bot, \top\}$$
$$read_\bot(s, p) = \bot$$
$$update_\bot(s, p, v) = \bot$$
$$frozen_\bot(s, p) = \textsc{False}$$

Definition 3.3.19 (I-Vars)**.** A $T_{Ivar,V}$ implements the I-Var data structure, which contains a single value that can be assigned at most once[1]. $V \subseteq \mathcal{V}$ is the set of valid values accepted by the I-var.

$$S_{Ivar:V} = \{\bot, \top\} \cup V$$
$$read_{Ivar:V}(s, p) = s$$
$$update_{Ivar:V}(s, p, v) = \begin{cases} v \text{ if } s = \bot \ \wedge \ v \in V \\ \top \text{ otherwise} \end{cases}$$
$$frozen_{Ivar:V}(s, p) = \begin{cases} \textsc{False} \text{ if } s = \bot \\ \textsc{True} \text{ otherwise} \end{cases}$$

1. I-vars are alternatively referred to as futures in Swift

Definition 3.3.20 (Tuples of Data Structures)**.** $T_{\langle t_1,...,t_n\rangle}$ implements tuples of data structures. We represent the state as a tuple of states. I.e., $s = \langle s_1, ..., s_n\rangle$. If $s = \bot$ then for convenience we assume $s_i = \bot$.

$$S_{\langle t_1,...,t_n\rangle} = \{\bot, \top\} \cup \{\langle s_1, ..., s_n\rangle \mid s_1 \in S_{t_1} \wedge ... \wedge s_n \in S_{t_n}\}$$

$$read_{\langle t_1,...,t_n\rangle}(s,p) = \begin{cases} \langle read_{t_i}(s_i, \langle\rangle) \mid i \in [1,n]\rangle \text{ if } p = \langle\rangle \\ read_{t_i}(s_i, p') \text{ if } p = \langle i\rangle \cdot p' \ \wedge \ i \in [1,n] \\ \top \text{ otherwise} \end{cases}$$

$$update_{\langle t_1,...,t_n\rangle}(s,p,v) = \begin{cases} \langle update_{t_i}(s_i, \langle\rangle, v_i) \mid i \in [1,n]\rangle \text{ if } p = \langle\rangle \ \wedge \ v = \langle v_1, ..., v_n\rangle \wedge \\ \qquad \bigwedge_{i=1}^{n} update_{t_i}(s_i, \langle\rangle, v_i) \neq \top \\ \langle update_{t_j}(s_j, p', v) \text{ if } j = i, s_j \text{ otherwise} \mid j \in [1,n]\rangle \\ \qquad \text{if } p = i \cdot p' \ \wedge \ i \in [1,n] \\ \top \text{ otherwise} \end{cases}$$

$$frozen_{\langle t_1,...,t_n\rangle}(s,p) = \begin{cases} \bigwedge_{i=1}^{n} frozen_{t_i}(s_i, \langle\rangle) \text{ if } p = \langle\rangle \\ frozen_{t_i}(s_i, p') \text{ if } p = \langle i\rangle \cdot p' \ \wedge \ i \in [1,n] \\ \textsc{True} \text{ otherwise} \end{cases}$$

Definition 3.3.21 (Arrays of Data Structures)**.** $T_{t[n]}$ implements fixed-size arrays, which 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 n times, e.g., $T_{t[3]} = T_{\langle t,t,t\rangle}$.

Definition 3.3.22 (Associative Arrays)**.** $T_{t[K]}$ implements associative arrays: finite maps of keys in set $K \subseteq \mathcal{V}$ to structures of lattice data type t. These are represented as finite maps from keys to states. I.e., $s = [k_1 \mapsto s_1, ..., k_n \mapsto s_n]$. For brevity, we assume that $s = \bot$ behaves as an empty map.

$$S_{t[K]} = \{\bot, \top\} \ \cup \ K \to S_t$$

$$read_{t[K]}(s,p) = \begin{cases} [k \mapsto read_t(s', \langle\rangle) \mid k \mapsto s' \in s] \text{ if } p = \langle\rangle \\ read_t(s(k), p') \text{ if } k \in dom(s) \\ \bot \text{ otherwise} \end{cases}$$

$$\text{where } p = k \cdot p'$$

$$update_{t[K]}(s,p,v) = \begin{cases} s[k \mapsto update_t(s(k), \langle\rangle, v') \mid k \mapsto v' \in v] \text{ if } p = \langle\rangle \ \wedge \\ \qquad\qquad v = [k_1 \mapsto v_1, ..., k_n \mapsto v_n] \ \wedge \\ \qquad\qquad \forall k \mapsto v' \in v, update_t(s(k), \langle\rangle, v') \neq \top \\ s[k \mapsto update_t(s(k), p', v)] \text{ if } p = k \cdot p' \ \wedge \ k \in K \\ \qquad\qquad \wedge \ update_t(s(k), p', v) \neq \top \\ \top \text{ otherwise} \end{cases}$$

$$frozen_{t[K]}(s,p) = \begin{cases} \textsc{False} \text{ if } p = \langle\rangle \\ k \in dom(s) \ \wedge \ frozen_t(s(k), p') \text{ if } p = k \cdot p' \ \wedge \ k \in K \\ \textsc{True} \text{ otherwise} \end{cases}$$

Additional collection types can be defined, including unordered ***sets*** and ***multisets*** with similar techniques just-defined collection types. For brevity we omit the definitions.

Definition 3.3.23 (Counters)**.** $T_{counter}$ implements counters, which support incrementing by a non-negative number.

$$S_{counter} = \{\bot, \top\} \cup [1, \infty)$$

$$read_{counter}(s, p) = \begin{cases} s \text{ if } p = \langle\rangle \\ s \notin \{\bot, \top\} \wedge s \geq x \text{ if } p = \langle \text{`} \geq x\text{'} \rangle \wedge x \in [1, \infty) \\ \top \text{ otherwise} \end{cases}$$

$$update_{counter}(s, p, v) = \begin{cases} \top \text{ if } v \notin [0, \infty) \vee p \neq \langle\rangle \\ v \text{ if } s = \bot \\ s + v \text{ otherwise} \end{cases}$$

$$frozen_{counter}(s, p) = \begin{cases} \textsc{False} \text{ if } p = \langle\rangle \\ s \neq \bot \wedge s \geq x \text{ if } p = \langle \text{`} \geq x\text{'} \rangle \wedge x \in [1, \infty) \\ \textsc{True} \text{ otherwise} \end{cases}$$

The above counter type uses the commutative positive increment operation combined with threshold reads. Many useful variations on counters are possible:

- A counter that tracks the maximum of all values assigned.
- A counter that uses the commutative multiplication operation with positive numbers.
- A counter that allows additions and subtractions, but not the $\geq$ threshold operation.

Definition 3.3.24 (File I-vars)**.** T_{File} implements file I-vars, which can mirror the state of a file in the file system and allow 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. First the file is assigned a path in the file system. Then, once the actual file is present at that path, the status can be set to PRESENT. Tasks that write a file wait on the file I-var's $\langle$`$filepath$'$\rangle$

path so that they execute once they know which file to write. Once the file is written, they assign the $\langle$'*state*'$\rangle$ path to signal that the file is present. Tasks that read a file wait on the $\langle$'*state*'$\rangle$ path so that they execute once the file is actually present in the file system.

$$S_{File} = \{\bot, \top\} \cup (\{\textsc{Present}, \textsc{Absent}\} \times \Sigma^*)$$

$$read_{File}(s,p) = \begin{cases} \bot \text{ if } s = \bot \\ \textit{filepath} \text{ if } p = \langle\text{'}\textit{filepath}\text{'}\rangle \wedge s = \langle \textit{status}, \textit{filepath}\rangle \\ \textit{status} \text{ if } p = \langle\text{'}\textit{status}\text{'}\rangle \wedge s = \langle \textit{status}, \textit{filepath}\rangle \\ s \text{ if } p = \langle\rangle \\ \top \text{ otherwise} \end{cases}$$

$$update_{File}(s,p,v) = \begin{cases} (\textsc{Absent}, v) \text{ if } p = \langle\text{'}\textit{filepath}\text{'}\rangle \wedge s = \bot \\ (\textsc{Present}, \textit{filepath}) \text{ if } p = \langle\text{'}\textit{status}\text{'}\rangle \wedge s = (\textsc{Absent}, \textit{filepath}) \\ \top \text{ otherwise} \end{cases}$$

$$frozen_{File}(s,p) = \begin{cases} \textsc{False} \text{ if } s = \bot \\ s = (\textsc{Present}, \textit{filepath}) \text{ if } p = \langle\rangle \\ s = (\textit{status}, \textit{filepath}) \text{ if } p = \langle\text{'}\textit{filepath}\text{'}\rangle \\ \textsc{True} \text{ otherwise} \end{cases}$$

Definition 3.3.25 (Freeze Wrapper)**.** $T_{wrapped:t}$ is a wrapper for $t \in \mathcal{T}$ that implements the correct semantics for *freeze*.

$$S_{wrapped:t} = S_t \times \{\textsc{True}, \textsc{False}\}$$

$$read_{wrapped:t}(s, p) = read_t(s', p) \qquad \text{where } s = \langle s', f \rangle$$

$$update_{wrapped:t}(s, p, v) = \begin{cases} \top \text{ if } f \\ \langle update_t(s, p, v), \textsc{False} \rangle \text{ otherwise} \end{cases} \qquad \text{where } s = \langle s', f \rangle$$

$$frozen_{wrapped:t}(s, p) = \begin{cases} \textsc{True} \text{ if } f \\ frozen_t(s, p) \text{ otherwise} \end{cases} \qquad \text{where } s = \langle s', f \rangle$$

3.4 Sequential Semantics with Nondeterministic Task Order

This section provides operational semantics for a version of the execution model without concurrency. In this version, tasks execute in a sequential but nondeterministic order. We treat all state as global to avoid complications of specifying distributed execution, which does not illuminate the core intended semantics. The sequential semantics is relatively easy to reason about and, as we will later show, can be extended to include additional features such as concurrency.

Definition 3.4.1 (System State)**.** The system state is comprised of three components: $\langle D, W, W^{fin} \rangle$. D is the state of the data store, W is the set of all tasks in the system, and $W^{fin} \subseteq W$ is the set of tasks that have finished. We define these components over the following sections.

Definition 3.4.2 (Error State)**.** An error state, denoted with **error**, is used in place of a valid system state to indicate that an error has occurred.

3.4.1 Data Store

Definition 3.4.3 (Data Store State)**.** $D : \mathcal{V} \xrightarrow{fin} \mathcal{D}$ is the data store state: a finite map of values – or *keys* – to data structures. A path within each data structure in the store D can be identified by concatenating the key of the data structure with the path into the data structure.

First we need a helper function to apply a sequence of updates.

Definition 3.4.4 (Iterated Update Function)**.** $update_t^*$ is a function that applies a sequence of updates defined by a sequence of (path, value) pairs. The type is $update_t^* : S_t \times (\mathcal{P} \times \mathcal{V})^* \to S_t$ and the definition is:

$$update_t^*(s, u) = \begin{cases} s & \text{if } u = \langle\rangle \\ update_t^*(s', u') & \text{otherwise} \end{cases}$$

$$\text{where } \langle\langle p_1, v_1\rangle\rangle \cdot u' = u \text{ and } s' = update_t(s, p_1, v_1)$$

We now define functions that let us access and modify the data store state.

Definition 3.4.5 (*dscreate* Function)**.** The dscreate function creates a data structure in the data store. It has type $dscreate : \mathcal{S} \times \mathcal{V} \times \mathcal{T} \to \mathcal{S} \cup \{\top\}$. It updates the data store with a new key and type, returning the new data store state:

$$dscreate(D, k, t) = \begin{cases} D[k \mapsto \langle t, \bot\rangle] & \text{if } k \notin dom(D) \wedge t \neq t_\bot \\ D[k \mapsto \langle t, update_t^*(\bot, r_{prev})\rangle](^2) & \text{if } t_{prev} = t_\bot \wedge t \neq t_\bot \\ \top & \text{otherwise} \end{cases}$$

$$\text{where } \langle t_{prev}, r_{prev}\rangle = D(k)$$

A valid *dscreate* operation will set the type of the key and apply any accumulated updates that were stored while the type was $t_\bot$. Applying a *dscreate* operation for the same key twice

is invalid. Storing updates means that *dscreate* can commute with *dsupdate*. This feature ensures determinism in a wider range of scenarios and simplifies later proofs.

A more restricted execution model could avoid the need for commutativity of *dscreate* and *dsupdate* if programs were always 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 3.4.6 (*dsupdate* Function)**.** The dsupdate function allows a data structure in the data store to be updated. It has type $dsupdate : \mathcal{S} \times \mathcal{P} \times \mathcal{V} \to \mathcal{S} \cup \{\top\}$. The function handles four cases: storing updates to keys which were not yet created (the first and second cases), invalid updates (the third case), and valid updates to created keys (the final case):

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

$$\text{where } \langle k\rangle \cdot p' = p,\ \langle t, s\rangle = D(k) \text{ and } s' = update_t(s, p', v)$$

Definition 3.4.7 (*dsread* Function)**.** The dsread function reads from a data structure in the data store. It has type $dsread : \mathcal{S} \times \mathcal{P} \to \mathcal{V}$ and is defined as:

$$dsread(D, p) = \begin{cases} read_t(s, p') & \text{if } k \in dom(D) \\ \perp & \text{otherwise} \end{cases}$$

$$\text{where } \langle k\rangle \cdot p' = p \text{ and } \langle t, s\rangle = D(k)$$

Definition 3.4.8 (*dsfrozen* Function)**.** The dsfrozen function checks if a data store path is frozen. It has type $dsfrozen : \mathcal{S} \times \mathcal{P} \to \{\text{TRUE}, \text{FALSE}\}$ and is defined as:

$$dsfrozen(D, p) = \begin{cases} frozen_t(s, p') & \text{if } k \in dom(D) \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$$\text{where } \langle k \rangle \cdot p' = p \text{ and } \langle t, s \rangle = D(k)$$

3.4.2 Data Store Permissions

In this section we will introduce the idea of permissions that determine 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 a data structure can no longer be written and therefore can be automatically frozen. These permissions implement a *effect system*: a system for specifying and enforcing restrictions on what data each task can read and write. Our effect system is quite simple but has similarities to other, earlier, effect systems [31, 67, 94]. The restrictions specified by an effect system, in general, can both prevent errononeous behavior of programs and enable other features – in our execution model, automatic freezing and in past work, automatic parallelization.

Definition 3.4.9 (Data Store Handle)**.** 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 $h = \langle k, m \rangle$, where $k \in \mathcal{V}$ is the key of a data store item, and $m \in \{\text{READ}, \text{WRITE}\} = \mathcal{E}$. The READ permission allows use of *dsread* on the key and the WRITE permission allows use of *dsupdate* to the key.

In order to support references between data structure in the store, there needs to be some way to encode data store handles into values. We assume the existence of some scheme for doing so.

Definition 3.4.10 (Reference Extraction Function for Values)**.** The function $refs : \mathcal{V} \to 2^{\mathcal{H}}$ extracts the set of data store handles encoded in a value. We do not specify the exact encoding, but require that $refs(\bot) = refs(\top) = \emptyset$.

Definition 3.4.11 (Reference Extraction Function for Data Structures)**.** The function $all\text{-}refs : \mathcal{D} \to 2^{\mathcal{H}}$ extracts all data store handles that can be extracted from a data structure.

$$all\text{-}refs(\langle s, t \rangle) = \bigcup_{p \in \mathcal{V}} refs(read_t(s, p))$$

Property 3.4.1 (Update Handles)**.** We further constrain the behavior of valid types so that any handles added to a data structure must be explicitly added through an *update* operation. $\forall t \in \mathcal{T}_{valid}$, $\forall s, s' \in S_t$, $\forall p \in \mathcal{P}$, $\forall v \in \mathcal{V}$, if $s \overset{t,p,v}{\underset{up}{\to}} s'$ then $s' = \top$ or $all\text{-}refs(\langle t, s' \rangle) \subseteq all\text{-}refs(\langle t, s \rangle) \cup refs(v)$.

3.4.3 Tasks

Definition 3.4.12 (Tasks)**.** $\mathcal{W}$ is the set of all possible tasks. ($t/\mathcal{T}$ is already in use, so we instead use w to stand for "work.") Any given task $w \in \mathcal{W}$ is a tuple $\langle f_w, handles_w, wait\text{-}data_w, read\text{-}data_w \rangle$. f_w is a arbitrary computable total function. (For practical purposes, it that could be thought of executable code plus parameter values.) 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 $f : \mathcal{V}^* \to (\mathcal{V} \times \mathcal{T}^* \times \mathcal{P} \times \mathcal{V}^* \times 2^{\mathcal{W}}) \cup \textbf{error}$. *handles* $\subseteq \mathcal{H}$ is a set of permissions to access data store keys. *wait-data* $\subseteq \mathcal{P}$ and *read-data* $\subseteq \mathcal{P}$ are both sets of paths that specify the data store keys that are dependencies for the task (i.e., which must be frozen before the task executes) and the data store keys that are read by the task to serve as inputs to f. **error** is returned to indicate an error.

Definition 3.4.13 (Tasks Set)**.** Let $W \subseteq \mathcal{W}$ be the multiset of all tasks created so far in a program's execution, including those that both have and have not yet been executed.

Definition 3.4.14 (Finished Tasks Set)**.** Let $W^{fin} \subseteq W$ be the multiset of all finished tasks. Tasks are added to W^{fin} as execution of the program proceeds.

Note that both W and W^{fin} are both multisets: the same task can appear multiple times in a given set. Subtraction of multisets accounts correctly for the fact that some instances of these tasks may have completed, while others may not have completed. We use multisets instead of sets to ensure that, if multiple structurally identical tasks are created, then they are all executed. Alternatively, we could replace the multiset with a set and tag each task with a unique identifier using the algorithm described later in Section 3.9.1.

3.4.4 Sequential Operational Semantics

The initial state of the system is $(\emptyset, \{w_0\}, \emptyset)$: no data and only an initial task. The system evolves in a stepwise way with the operational semantics shown in Figure 3.6. In this basic operational semantics, one task executes at a time, with no concurrency. The nondetermistic task selection at each step of execution leads to multiple possible schedules as illustrated in Figure 3.7.

The rules implement the behavior in the following way. The SELECT-NONDET rule initiates each step of execution by nondeterministically selecting any runnable task w – a task with all paths in *wait-data*$_w$ frozen – from the pending task set. The remaining rules execute the task and update the system state with the results of the task. The READ-DATA rule reads all the *read-data*$_w$ paths from the current state of the store and stores the values as a sequence. The task's function, f_w, is then computed and the results checked with the *validate* function. If the function returns **error**, EXEC-TASK-ERROR terminates execution. The validate function checks that the function only read and wrote data store keys that it had permissions to, and that it did not pass along any permissions that it did not own. If validation fails, then the EXEC-TASK-INVALID rule matches the state and terminates execution with **error**. If validation succeeds, then the EXEC-TASK rule matches

Given system state $S = \langle D, W, W^{fin} \rangle$:

$$W^{pending} = W - W^{fin}$$

$$W^{runnable} = \{w_{|n|} \in W^{pending} \mid \forall p \in \mathit{wait\text{-}data}_w, dsfrozen(D, p)\}$$

$$\frac{w \in W^{runnable}}{S \to \langle S, w \rangle} \textsc{Select-Nondet} \qquad \frac{}{\langle S, w \rangle \to \langle S, w, \langle dsread(D, p) \mid p \in \mathit{read\text{-}data}_w \rangle \rangle} \textsc{Read-Data}$$

$$\frac{f_w(rvals) = \langle C, U, W^{new} \rangle \qquad validate(w, rvals, C, U, W^{new})}{\langle \langle D, W, W^{fin} \rangle, w, rvals \rangle \to \langle \langle D, W, W^{fin} \uplus \{w\} \rangle, C, U, W^{new} \rangle} \textsc{Exec-Task}$$

$$\frac{f_w(rvals) = \textbf{error}}{\langle \langle D, W, W^{fin} \rangle, w, rvals \rangle \to \textbf{error}} \textsc{Exec-Task-Error}$$

$$\frac{f_w(rvals) = \langle C, U, W^{new} \rangle \qquad \neg validate(w, rvals, C, U, W^{new})}{\langle \langle D, W, W^{fin} \rangle, w, rvals \rangle \to \textbf{error}} \textsc{Exec-Task-Invalid}$$

$$\textsc{Apply-Create}\ \frac{D \neq \top}{\langle \langle D, W, W^{fin} \rangle, \langle \langle k, t \rangle \rangle \cdot C, U, W^{new} \rangle \to \langle \langle dscreate(D, k, t), W, W^{fin} \rangle, C, U, W^{new} \rangle}$$

$$\textsc{Apply-Update}\ \frac{D \neq \top}{\langle \langle D, W, W^{fin} \rangle, \langle \rangle, \langle \langle k, p, v \rangle \rangle \cdot U, W^{new} \rangle \to \langle \langle dsupdate(D, k, p, v), W, W^{fin} \rangle, \langle \rangle, U, W^{new} \rangle}$$

$$\textsc{Apply-Tasks}\ \frac{D \neq \top}{\langle \langle D, W, W^{fin} \rangle, \langle \rangle, \langle \rangle, W^{new} \rangle \to \langle \langle D, W \uplus W^{new}, W^{fin} \rangle, \textsc{Freeze} \rangle}$$

$$\textsc{Apply-Data-Fail}\ \frac{D = \top}{\langle \langle D, W, W^{fin} \rangle, C, U, W^{new} \rangle \to \textbf{error}}$$

$$\frac{}{\langle \langle D, W, W^{fin} \rangle, \textsc{Freeze} \rangle \to \langle [k \mapsto freeze?(D, k, d) \mid k \mapsto d \in D], W, W^{fin} \rangle} \textsc{Freeze}$$

Figure 3.6: Small-step operational semantics for execution model. *validate* and *has-perm* are defined in Definitions 3.4.15 and 3.4.17 respectively. Each rule in the semantics has form $\frac{\text{Preconditions}}{\text{Rewrite Rule}}$ and has a LABEL for purposes of identification. The preconditions are expressions that must be true for the rewrite rule to be applicable. The rewrite rule transforms a state matching its left-hand side into state matching its right-hand side. In cases where multiple rewrites are applicable, the rewrite is chosen nondeterministically. The only nondeterminism in the semantics comes from rule SELECT-NONDET, which can choose any runnable task.

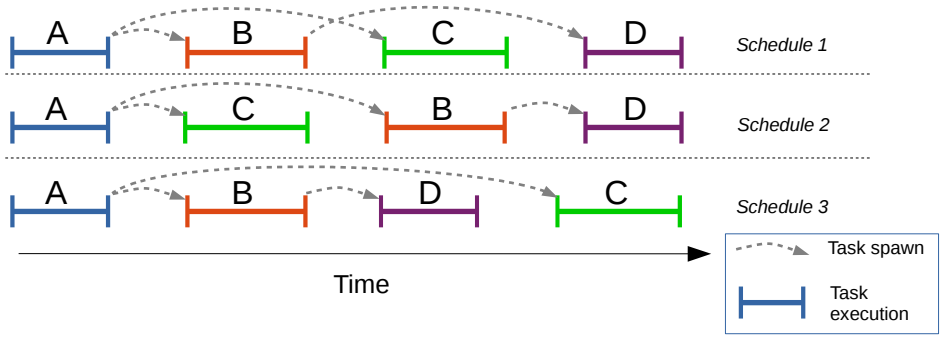


Figure 3.7: Nondeterministic task scheduling in sequential execution model. In this simple example involving four tasks, three different schedules are possible, based on A spawning B and C and B spawning D. The schedule that occurs on a given execution depends on which task is selected at each step of execution.

the state and the outputs of the function – C, U, and W^{new}: creates, updates, and new tasks, respectively – are stored for later rules to process. The Apply-Create, Apply-Update, and Apply-Tasks rules then update the state with the operations output by the task. If any of the data store operations are invalid, the data store enters the $\top$ state, and the Apply-Data-Fail rule terminates execution with **error**. Finally, if all state updates were successful, then Freeze rule freezes any data structures in the store that are not writable by any pending tasks, after which execution of the next step begins with Select-Nondet.

Checking the validity of a task's actions is somewhat complex. Invalid creates or updates result in an **error** state for the entire system. It is necessary to terminate execution because a previous task may have read a now-invalid value from the data structure and proceeded to further modify the global system state.

Permissions for data store keys are more subtle. Both Read and Write permissions are created when a task invokes *dscreate* for a new key and are can then be propagated to other tasks through task spawns and data store reads and writes. Only one task can successfully create a key and obtain the initial permissions because creating the same key twice results in **error**. 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. These violations can be divided into three cases: a task accessing keys for which it does not have permissions, a task

creating a child task that has permissions the parent does not have, and a task transferring permissions that it does not have to the data store.

Definition 3.4.15 (Task Validation Function)**.** The validate function determines whether a task's actions are valid according to the permissions that 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 3.4.1.

$$\begin{aligned} \mathit{validate}(w, \mathit{read\text{-}vals}, \mathit{creates}, \mathit{updates}, W^{new}) = \\ \mathit{read\text{-}keys} \times \{\textsc{Read}\} \subseteq \mathit{perms} \wedge \\ \mathit{update\text{-}keys} \times \{\textsc{Write}\} \subseteq \mathit{perms} \wedge \\ \forall (k, p, v) \in \mathit{updates}, \mathit{refs}(v) \subseteq \mathit{perms} \wedge \\ \forall w' \in W^{new},\ \mathit{handles}_{w'} \subseteq \mathit{perms} \end{aligned}$$

$$\begin{aligned} \text{where } \mathit{create\text{-}keys} &= \{k \mid (k, t) \in \mathit{creates}\} \\ \mathit{read\text{-}keys} &= \{k \mid (k, p) \in \mathit{read\text{-}data}_w\} \\ \mathit{update\text{-}keys} &= \{k \mid (k, p, v) \in \mathit{updates}\} \\ \mathit{perms} &= \{\langle k, \mathit{perm}\rangle \mid \langle k, t\rangle \in \mathit{create\text{-}keys}, \mathit{perm} \in \mathcal{E}\} \cup \\ & \quad \mathit{handles}_w \cup \bigcup_{v \in \mathit{read\text{-}vals}} \mathit{refs}(v) \end{aligned}$$

Theorem 3.4.1 (Task Outputs are Deterministic Function of Values)**.** *The outputs of a task:* $\langle \mathit{creates}, \mathit{updates}, W^{new}\rangle$ *and the result of the validate check is a deterministic function of the task* w *and read-vals.*

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 in turn lead to application of *dsupdates*, *dscreates*, and new tasks in an nondeterministic order to the system

state, which then can flow to task definitions w and *read-vals* in subsequent applications of EXEC-TASK.

Definition 3.4.16 (*freeze?* Function)**.** The *freeze?* function conditionally freezes a key in the data store if it has no remaining WRITE permissions.

$$\mathit{freeze?}(D, k, \langle t, s\rangle) = \begin{cases} \langle t, s\rangle \text{ if } \exists w \in W^{\mathit{pending}}, \mathit{has\text{-}perm}(w, \mathit{key}, \textsc{Write}) \\ \langle t, \mathit{freeze}_t(s)\rangle \text{ otherwise} \end{cases}$$

Definition 3.4.17 (*has-perm* Function)**.** The *has-perm* function determines whether a task can read or write a key directly or indirectly.

$$\mathit{has\text{-}perm}(w, \mathit{key}, \mathit{perm}) = \exists \langle \mathit{root}, \mathit{perm}\rangle \in \mathit{handles}_w, \mathit{has\text{-}perm\text{-}via}(\mathit{root}, \mathit{key})$$

$$\text{where } \mathit{has\text{-}perm\text{-}via}(\mathit{src}, \mathit{dst}) = (\mathit{src} = \mathit{dst}) \vee$$

$$\exists \langle \mathit{ref}, \mathit{perm}\rangle \in \mathit{all\text{-}refs}(D(\mathit{src})), \mathit{has\text{-}perm\text{-}via}(\mathit{ref}, \mathit{dst})$$

3.4.5 Selection of Data Store Keys

One issue that we have not yet addressed is how data store keys are chosen: most programs need a way to create and refer to unique storage locations. In most real-world systems, unique key generation in concurrent settings is nondeterministic, e.g., first-come-first-served. However, this approach complicates proof of determinism, because nondeterministic keys become part of the system state and can be read by tasks. To simplify our proofs, we assume that keys are chosen deterministically. This does not sacrifice generality because unique keys can be generated deterministically.

One possible scheme is to generate a unique task identifier based on the task's place in the spawn tree, e.g., the first task is 0, the first child task spawned by 0 is 0.0, the third child task spawned by 0.0 is 0.0.2, and so forth. Given a unique task identifier, a unique data key then

LVars [59]	Deterministic (non-termination/error interchangable)
LVIsh [62]	Quasi-deterministic (i.e. nondeterministic in certain cases)
Swift execution model	Deterministic (non-termination/error interchangable)
Swift execution model with bounded task duration and bounded task waiting	Deterministic

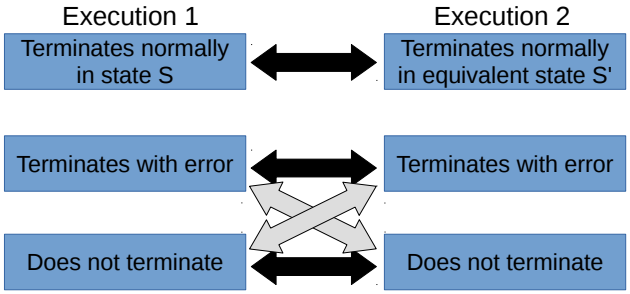


Figure 3.8: Summary of determinism results.

can be generated by concatenating the task's unique identifier with a unique value within the task. Leiserson et al. refer to this method of labelling tasks as ***spawn pedigree*** [64]. A similar technique is used in Swift/K for the purposes of uniquely identifying variables in restart logs [115].

It is possible to show determinism up to key values by defining an equivalence relation over system states that is not sensitive to key values and by constraining tasks to not directly inspect or modify key values. Intuitively, it is apparent that the exact keys do not matter as long as nothing in the program is dependent on the values of the keys. Moreover, Kuper et al. have proven determinism in the similar setting of LVars with nondeterministic location selection [59].

3.5 Determinism of Execution Model

Now that we have specified the operation semantics, we next want to define how the execution model behaves. 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** state, or it can not terminate and run for an unbounded number of steps. In the execution model, if a

program with the same initial state is executed multiple times, the intermediate states at each step can be nondeterministic because of nondeterministic task selection. We need to show that two executions starting from the same initial state will converge to the same result – a fact that is far from obvious. In this section we will prove that:

- a program that terminates in a valid state on one execution will terminate in equivalent valid states on all executions, i.e., deterministically.
- given arbitrary nondeterministic task selection, programs that terminate in **error** or do not terminate on one execution may have a different outcome (**error** or non-termination) on different executions;
- if the task selection policy guarantees that any runnable task executes within a bounded number of steps, then programs that terminate in **error** end in **error** on all executions and programs that do not terminate do not terminate on any execution.

We will prove these properties through *bisimulation* [70] – reasoning about the behavior of two different executions starting from the same state. Proving that any given pair of executions behave equivalently is sufficient to prove that *all* such executions behave equivalently, since the equivalence relation is transitive. We will formally state these theorems in this section, but first need some definitions.

Definition 3.5.1 (Execution Traces)**.** 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. I.e., it is a sequence of the elements of W_{i+1}^{fin}. 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 selected and executed. If s_{n+1} is **error**, then the trace terminated in an error. Any non-error states are a tuple $s_i = \langle D_i, W_i, W_i^{fin} \rangle$.

Definition 3.5.2 (Terminating Execution Traces)**.** A terminating execution trace is an execution trace of length n where either $s_{n+1} =$ **error** or $W_{n+1}^{runnable} = \emptyset$. If the trace terminates without error, then $\langle D_{n+1}, W_{n+1}, W_{n+1}^{fin} \rangle$ is the final state of the system.

Definition 3.5.3 (Read Paths Set)**.** Let $P_i^{read} \subseteq \mathcal{P}$ be the paths read up to step i of an execution trace: $P_i^{read} = P_{i-1}^{read} \cup \textit{read-data}_i$, where $\textit{read-data}_i$ is the *read-data* value of w_i.

We need to define a notion of equivalence for data store states, based on equivalence of the states of data structures.

Definition 3.5.4 (Data Store Equivalence)**.** Two data store states D and D' are equivalent: $D \overset{DS}{\sim} D'$ iff $\forall (k \mapsto \langle r, t\rangle) \in D,\ k \in dom(D') \ \wedge\ \exists r'\ D'(k) = \langle r', t\rangle \ \wedge\ r \overset{t}{\sim} r'$, and vice-versa, i.e. $\forall (k \mapsto \langle r, t\rangle) \in D',\ k \in dom(D) \ \wedge\ \exists r'\ D(k) = \langle r', t\rangle \ \wedge\ r \overset{t}{\sim} r'$, .

For the purposes of proof, we assume that all tasks explicitly wait on data before reading.

Property 3.5.1 (Task Wait before Reading)**.** Given system state $\langle D, W, W^{fin}\rangle$, $\forall w \in W$, $\textit{read-data}_w \subseteq \textit{wait-data}_w$.

Now we can state our main theorems, leaving proofs for Appendix A.

Definition 3.5.5 (Equivalence of System States)**.** $\langle D, W, W^{fin}\rangle \sim \langle D', W', W'^{fin}\rangle$ iff $D \overset{DS}{\sim} D'$, $W = W'$, and $W^{fin} = W'^{fin}$,

Theorem 3.5.1 (Execution is Deterministic – Normal Termination)**.** *Execution is deterministic if the program terminates in a non-**error** state, all data store types are lattice types, and tasks wait before reading. Given any two execution traces with the same initial state $\langle D_0, W_0, W_0^{fin}\rangle$, then if the first trace terminates with a valid state $\langle D_{n+1}, W_{n+1}, W_{n+1}^{fin}\rangle$, 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, $S_{n+1} \sim S'_{m+1}$ and $n = m$.*

Now we must consider the other possible outcomes of execution: non-termination and termination with **error**. Theorem 3.5.1 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**). This is true under some assumptions and not true under others.

Theorem 3.5.2 (Non-terminating Task Functions Cause Divergence on Error)**.** *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****.*

This is unsurprising and not entirely under the control of the execution model: in the absence of preemption or concurrent execution of tasks, a non-terminating task function can always hold up the entire program. Therefore, for the remaining theorems we will assume the task functions are well-behaved and always terminate.

Theorem 3.5.3 (Error Non-determinism Given Arbitrary Scheduling)**.** *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****.*

The proof of this theorem in Appendix A demonstrates that realistic task selection policies like last-in first-out (LIFO) can exhibit this divergent behavior for some programs. Other task selection policies can avoid this problem. To avoid this mixing up of **error** non-termination, we need a task selection policy where tasks are run within a bounded number of steps.

Definition 3.5.6 (Bounded Waiting Scheduling Policy)**.** A task selection policy for selecting $w \in W^{runnable}$ guarantees bounded waiting if, given a system state $\langle D, W, W^{fin} \rangle$, $\exists c$ such that $\forall w \in W^{runnable}$, 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. As execution progresses, c can grow over time in an unbounded manner, but, crucially, no specific task can remain stuck in a runnable state for an unbounded amount of time.

Theorem 3.5.4 (Error Determinism Given Bounded Waiting)**.** *If a trace* $\langle w_0, w_1, ..., w_n \rangle$ *terminates in* ***error****, then another trace* $\langle w_0, w_1', ... \rangle$ *with the same initial state and bounded waiting also terminates in* ***error****.*

Corollary 3.5.1 (Non-termination Determinism Given Bounded Waiting)**.** *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 model is superficially attractive and can avoid non-termination in some circumstances. In the above example in Theorem 3.5.3 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 $i > 1$ is at least 3^i, then a loose upper bound on the probability of eventually executing Task A is $\frac{1}{2} + \frac{1}{9} + \frac{1}{27} + ... < \frac{2}{3}$. This possible non-termination is 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 **error** the model does not capture!

3.6 Extensions to Semantics

In this section we will describe a number of extensions or modifications to the basic sequential semantics that are useful in modelling more realistic implementations of the execution model or reasoning about transformations of programs in the execution model.

3.6.1 Concurrent Execution of Tasks

An execution model that only allows tasks to execute sequentially is of limited applicability. However, the desirable properties of the sequential execution model are preserved when it is extended with concurrent execution of tasks. We consider two extensions and argue

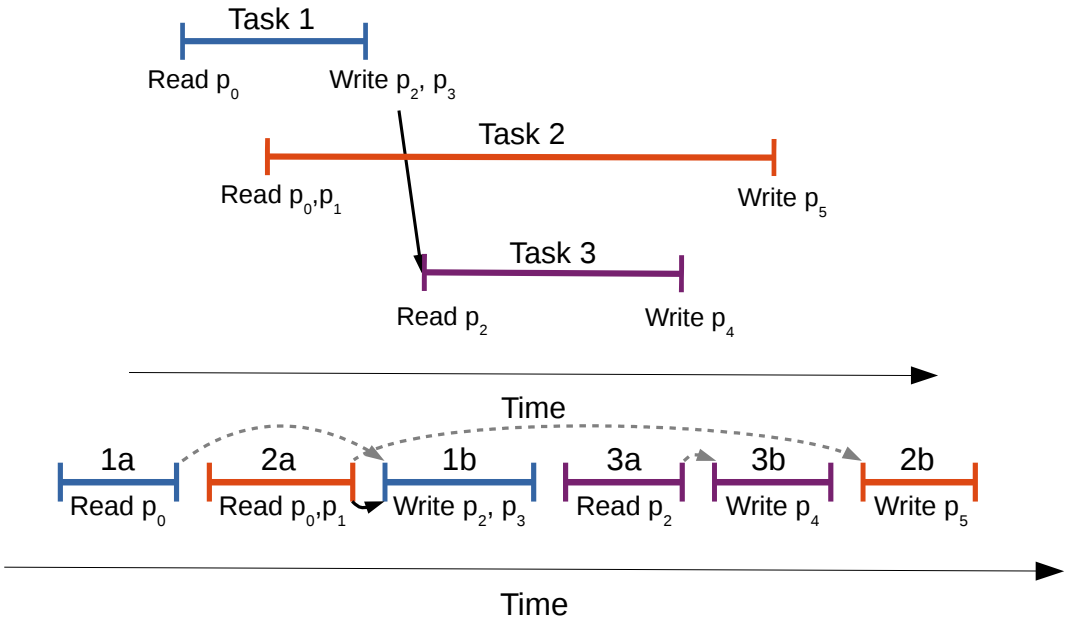


Figure 3.9: Diagram showing how concurrent execution of a set of tasks is equivalent to interleaved execution of the same tasks broken into separate read and write tasks.

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 and 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 sequential execution model, if each task is split into two tasks. The initial task reads inputs and spawns the second contination task, passing along the values that it read. The continuation task then computes the function and produces outputs. Figure 3.9 illustrates how concurrent execution can be mapped to an equivalent sequential execution. Any valid concurrent trace of tasks can be mapped to a sequential trace of these split tasks. The sequence of reads and writes, and therefore the end state is the same in both cases.

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 wait before reading and not read any non-frozen data. A similar argument to to that presented in the preceding para-

graph can be applied to enable concurrent execution of tasks applies – a task that performs 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. Interleaved execution of these tasks in the sequential model is equivalent to concurrent execution because the order in which reads and writes is applied is the same. 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. This extension also allows a single task to follow chains of references, i.e., issue a read for something for which it has a handle, then issue another read to a newly obtained reference.

3.6.2 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 3.5.1 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 path $p \in \mathcal{P}$ can be removed from a task's *wait-data* without altering program behavior if, at the time the task is created, $dsfrozen(D, p)$ is 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 on which the current task is dependent via data, or
- in general, any chain of spawn and data dependencies leads to a task with $\langle key, path \rangle$ in *wait-data* or a task that wrote the key in such a way that it is frozen.

3.6.3 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. As no information is removed from the data store, its size can grow boundlessly. Furthermore, a naïve implementation of the algorithm to detect freezing could require scanning 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 [54] is needed. The problem is superficially straightforward: if no tasks have permissions for a key, then the associated data structure can be garbage collected. This process can be formulated in a similar way to freezing: if no pending tasks have READ or WRITE permissions to access a key (directly or via another data structure), then garbage collection has no effect on the program's execution. However, *dscreate* present a problem: *dscreate* returns $\top$ if the key was previously created. To faithfully implement this restriction, the system needs to track the set of keys that have been used at any point: a smaller but still unbounded amount of information.

Suppose that we do not attempt to preserve this information: all information for a key is removed when it becomes inaccessible, i.e., $D(k)$ is reset to $\langle T_\bot, \langle\rangle\rangle$. Then race conditions are possible if a second *dscreate* races with the garbage collection of the key.

This problem can be worked around in at least two ways. First, programs can be constructed so that this kind of data race is not possible. For example, the program could be constructed so that *dscreate* is invoked at most once for any key, for example by using a unique key generation algorithm like the one described in Section 3.4.5. Second, programs 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 successor tasks of the task that invokes *dscreate* to create a new version of the key. The second approach is used

in Swift/T- data store keys are not exposed to user code and the generated code will not attempt to recreate a key until all references to the previous key are out of scope.

The efficiency problem can be solved with one of the many known garbage collection algorithms that are more efficient than a naïve algorithm of scanning all references at each step [54]. One approach is to use reference counting to track the number of active read/write references to each key. This method 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 cannot reclaim cycles of references. In Swift/T this is not a problem because the type system prevents construction of reference cycles.

3.6.4 Event Handlers and Iteration over Collections

One pattern that is difficult to implement directly in the proposed execution model is iteration over a collection of values concurrently with the addition of those elements. For example, a programmer may wish to process elements added to an array concurrently with the processes that are adding them. We propose ***event handlers*** to address this and related problems, similar to the event handlers proposed for LVish [62].

Let $E \subseteq V$ be an ***event space*** associated with a data structure $\langle r, t \rangle$. 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, such that a task $f(e)$ should be created whenever a path $e \in E$ becomes frozen. When the event handler is initialized, tasks are 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 $f(e)$ for each. E is an infinite set in many cases, so

an infinite number of tasks will remain pending 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 $\mathcal{V}$ and that a successor function $succ(v)$ is defined. Then we can simulate an event handler for event space E in data structure $\langle r, t \rangle$ with the following process:

- Modify t so that the following paths implementing boolean predicates are defined: "$x \in E$", "$\exists e \in E$ s.t. $e > x$", and "$\exists e \in E$ s.t. $e < x$."
- When the event handler is created, choose an arbitrary event $e_0 \in E$, and create three tasks that will run when the following subscripts are frozen: A) "$e_0 \in E$", B) "$\exists e \in E$ s.t. $e > e_0$", and C) "$\exists e \in E$ 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 "$\exists e \in E$ 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 the same as task B, except it progresses in the opposite direction - $<$ replaces $>$ and $succ^{-1}$ replaces $succ$.

3.7 Mapping Swift to Execution Model

In Section 3.1 we outline the relationship between high-level Swift code and the lower-level execution model and illustrated in Figure 3.4 how code might be translated into the model in an optimized way. 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 sketch the strategy that 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 Swift and the execution model, The abstract execution model is much lower level than 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 and I/O. A real implementation also requires explicit bookkeeping for memory management and correct freezing of variables: erroneous programs in the execution model can 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 strategy 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. The compiler optimizations that we describe later in Chapter 5 improve on the inefficient naïve 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 declared variable and temporary intermediate variable, *dscreate* is invoked to

create a data structure in the global data store. For variables local to a block, this occurs when the block begins executing.

- Each statement in a block separated by `;` is spawned as a task.
- 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. The main task execution loop in the execution model will then later execute the function. This "trampoline" [43] allows tail recursion without stack growth.
- For each subexpression in a statement, a task is spawned to compute it and store the result to a temporary data structure in the store.
- A conditional if or switch statements is implemented as a task that first waits, then reads the condition, and finally executes the appropriate branch.
- A wait statements is implemented as a task that executes the block of code with the variables waited added to the task's *wait-data*.
- A parallel foreach loop is implemented by spawning a task per iteration. If the loop has many iterations, task spawning is parallelized by recursively dividing the range.
- A foreach loops over containers is implemented with an event handler.
- An ordered for loop 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.

3.8 Limitations and Future Work

3.8.1 Limitations of Permissions

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. For example, Swift/K originally implemented automated freezing of arrays by injecting code that froze arrays when a stack frame in the interpreter exited. This caused deadlocks with certain patterns of parallelism. Swift/K has since switched to a more general reference-counting-based approach to freezing. However, there are some limitations arising from automatic freezing working at key granularity that are problematic 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 3.10 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. `A` is never frozen because assignment of `A[0]` only happens after `size(A)` returns.

```
1 import io;
2 int A[];
3 A[0] = size(A);
4 printf("%i", A[0]);
```

Figure 3.10: Example Swift code that deadlocks in Swift/T but could potentially execute successfully.

However, basic inspection reveals that it would be entirely consistent if the program printed "1" because it can be inferred statically that the statement will assign exactly one subscript, 0, of the array. Exploiting this inference in the execution model would require extensions to support a *partially frozen* state in which 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 behavior would require more granular tracking of the possible variables and paths that unexecuted tasks like `A[0] = size(A)` may write.

More subtle issues occur with multi-dimensional arrays, which in Swift/T, are implemented as nested arrays with outer arrays holding references to inner arrays. Inner arrays are not frozen until the outer arrays are frozen [3], so as long as a write reference to the outer arrays is held, no inner arrays can be frozen.

```
1  int A[];
2  int N = 10;
3
4  foreach i in [0:N] {
5    foreach j in [0:N] {
6      if (i == 0) {
7        // Random integer from 1 to 100
8        A[i][j] = randint(1,100);
9      } else {
10       // Depends on entire previous inner array
11       int prev_sum = sum(A[i-1]);
12       A[i][j] = f(A[i-1][j], prev_sum);
13     }
14   }
15 }
```

Figure 3.11: Example Swift code that deadlocks at some optimization levels in Swift/T but executes successfully at high optimization levels.

In the code shown in Figure 3.11, data dependencies flow in one direction: from iteration $i-1$ to iteration i. However, whether the code is guaranteed to make progress after translation to the execution model depends on subtleties of how write permissions are handled. Each subarray `A[i]` is not frozen until the root array `A` is frozen, which cannot happen so long as a task has a WRITE handle for `A`. At lower optimization levels, a WRITE handle to `A` is retained for the entire time the loop is execution so that `A[i][j]` can be written in the innermost loops. This handle prevents subarrays `A[i]` with $i > 0$ from being filled in because `sum(A[i-1])` cannot be computed until `A[i-1]` is frozen. I.e., at lower optimization levels, the program will fail to make progress. At higher optimization levels, the contrary happens: it is able to run to completion because writes to the outer array `A` are hoisted out of the inner loop body and only write handles for subarrays `A[0]`, `A[1]`, etc are retained while the loop

3. a simplification – if the inner array is assigned in whole, then the inner array may be frozen

is executing. This allows outer array to be frozen while inner arrays are still being written.

Having progress guarantees depend on code optimizations that are best-effort is undesirable and a significant shortcoming of our current translation strategy and permissions model: the result is that language semantics are implementation-dependent in an unnecessary way. Unfortunately it is non-trivial to address this problem – we would need to devise both semantics and an implementation that guarantee exactly when a program will or will not run to completion. Solving this problem would likely require an extension to the execution model for more granular tracking of which paths of a data structure are writable by each task. We leave this task 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 [16, 31, 67, 94].

3.8.2 In-place Updates and Linear Types

A practical limitation of the execution model is that lattice types do not allow in-place destructive updates. In the context of high-performance distributed systems working with large data such as matrices, this limitation 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 [19]. The essence of linear type systems is that the value of linearly typed variables is only allowed to be read once. Thus 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 behavior could be implemented through restrictions on duplication and transfer of READ handles to linearly typed keys.

3.9 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 thread interleavings. 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, program behavior is nondeterministic and can vary from one execution of the same program to another.

Previous work has dealt with this problem by imposing restrictions on how shared state can be read and written. Restricting writes to shared state can disallow some state transitions and greatly reduce the space of possible states that can be reached. 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 behavior is guaranteed, constructing deterministic programs is greatly simplified. This is the approach that we took with our execution model.

Single-assignment data is a well-known way of achieving determinism in parallel programs that trivially fulfils 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 behavior is generally implemented by blocking threads of execution until the location is written. Single-assignment data has been used over at least three decades, for example I-vars and arrays of I-vars [10].

Concurrent logic programming and concurrent constraint programming [99, 97] have monotonicity as a cornerstone of the programming model: as the program executes, the set of true derivations monotonically grows. Logic variables in these languages are also lattices.

In recent years, multiple research groups [27, 62, 100] have explored use of data types that are also *monotonic* but are more general than single-assignment data. Constraining

data types to be a lattice (or to have some properties of a lattice) is a natural way to achieve monotonicity and thus to guarantee that writes are be commutative, i.e., applying a set of writes results in the same final state regardless of order. This property 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 lattice-based data types for distributed systems, where they can enable eventual consistency in distributed systems with replicated state. Conflict-free replicated data types [100] (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 approach allows a scalable, high-performance implementations of these data types in which 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 eventual convergence to a consistent state. Kuper [60] compares CFRDTs and LVars in detail.

3.9.1 Comparison with LVars

Our semantics has many similarities, but also several differences from, the LVars/LVish semantics of Kuper et al. [62]. This section describes the major differences in operational semantics and between their LVars and our lattice data types.

The two semantics were developed concurrently: our work is a formalization and extension of the behavior of Swift [115, 119] and Turbine [118]. Previous work on CFRDT and LVars was 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 focus reflects its history as a formalization of the

runtime system described later in the thesis and makes it simpler for us to draw connections between the execution model and the runtime system API described in the next chapter.

Idempotence of Updates

One significant difference is that LVars are required to be a join semi-lattice, while our lattice data types only meet some of those requirements. The main 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, a feature that can be leveraged in some implementations of work-stealing [69]. However, idempotence is disallows some useful behaviors. For example, if I-var V is updated first with $V \leftarrow x$ then with $V \leftarrow y$, then 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. Kuper et al. recognized in later work that idempotence was not always desirable and proposed extensions to LVars with non-idempotent operations [61].

Even with the idempotence requirement, however, LVars are equivalently powerful: it is possible to simulate non-idempotent operations in LVars by tagging every *put* value with a unique identifier, for example, using a unique key generation algorithm (Section 3.4.5). The unique identifier can simply be ignored by the implementation, but this approach gets around the idempotence requirement by guaranteeing that two put arguments are never identical.

Join Operation

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 $u_1, u_2 \in \mathcal{V} \times \mathcal{V}^*$. 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 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 [100]. We have not explored the use of join operations yet in practice, but it is almost certainly applicable to implementing distributed lattice data structures.

Non-blocking Reads

Another difference of LVars is that the *get* operation is a blocking read operation – termed a "threshold read" – that only returns once the read has crossed a threshold in the lattice above which the read always returns the same value (unless it goes into the $\top$ state). The criteria for a threshold read to return a value are the same to the criteria our semantics require for *frozen* to return TRUE for a path (Property 3.3.8). In contrast we treat all reads as non-blocking, but constrain behavior enough to prove determinism with the assumption that tasks wait before reading (Property 3.5.1); 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.

Store Key Generation

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 the stores are handled in the two systems. The LVars *new* operation generates unique keys nondeterministically. This approach complicates proof of determinism of LVars because the proofs must reason about the equivalence of configurations with permuted locations. In contrast, our proofs make the simplifying assumption that keys are generated by tasks and argue that this leads to no loss of generality because one can deterministically generate unique keys based on the task lineage. 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. In the LVars semantics, which are based on a concurrent lambda calculus, there are no black-box operations so the behavioral constraints are easier to specify formally.

Automatic Freezing

The treatment of freezing and quiescence in LVish is quite different to our handling of freezing. The central difference is that our permissions system allows exact determination of when no more updates to data structures are possible, so they can be frozen automatically. LVish relies on the programmer to freeze variables and only guarantees "quasi-determinism" because races between *put* and *freeze* operations are possible.

CHAPTER 4

A MASSIVELY PARALLEL RUNTIME SYSTEM

Swift/T's runtime system, Turbine, provides the required runtime support for massively scalable data-driven task parallelism. Turbine uses the Message Passing Interface (MPI) [109] for process management and communication. MPI is a standard programming interface that is universally supported by current high-performance computing (HPC) clusters and supercomputers. MPI provides process management functionality – setting up many processes on the system, each with a unique numeric *rank* – and communication functionality – providing point-to-point messaging operations to send a message to another process and collective operations to exchange messages efficiently between many processes.

The Turbine runtime system is an extensively modified derivative of the ADLB [68] load-balancing library, as shown in Figure 4.1. ADLB implements a distributed task queue that allows MPI processes to enqueue (*put*) and dequeue (*get*) tasks. Tasks in ADLB hav an associated priority and can optionally be targeted to a specific MPI process (identified by an integer rank). ADLB task payloads are arbitrary binary data. Using targeted tasks and encoding of data into task payloads, a creative programmer can implement a range of parallel applications with ADLB's simple primitives. ADLB splits its task queue between multiple ADLB servers, each of which is responsible for a set of worker processes. Load balancing in ADLB is 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 prototype versions of Turbine extended ADLB with required functionality, such as a distributed data store and data-dependent task release [118]. Since then, we have further extended and enhanced Turbine to produce a complete and scalable distributed language runtime for Swift. This work includes task queue performance and scalability enhancements, work stealing to rebalance work between servers, richer data functionality, and support for garbage collection through reference counting.

The work described in this thesis contributes several important extensions and optimizations that we developed that that greatly improved the runtime's efficiency and scalability:

- Techniques to improve scalability of server-to-server operations
- Algorithms and data structures for efficient matching of put and get requests with support for priorities and location-awareness
- Specialized work stealing algorithms with asynchronous probes to improve scalability of the distributed task queue
- Extensions to the Turbine data model to support lattice data types
- Reference counting support to enable freezing and garbage collection of data

4.1 Runtime Architecture

The Turbine/ADLB runtime is a distributed system that allows many *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 tracks data dependencies of tasks. Figure 4.2 illustrates the interactions between these services.

The remainder of the section describes the primitive operations that are supported by

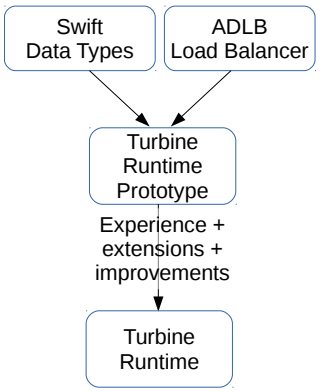


Figure 4.1: Lineage of runtime system.

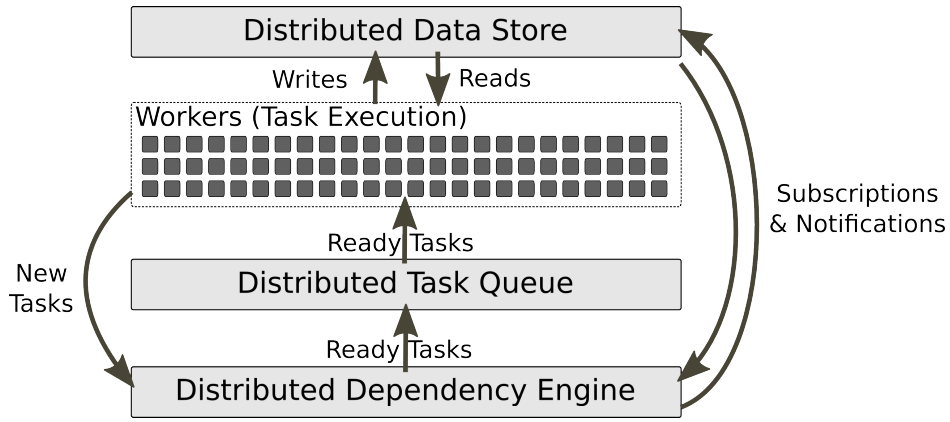


Figure 4.2: A view of the Swift/T distributed runtime (Turbine/ADLB) as distributed services enabling coordination between workers. In brief, new tasks created by code running on workers are passed to the dependency engine. The dependency engine holds these tasks until required input data are available, and then passes the tasks to the task queue. Tasks are then sent from the task queue to workers to be executed. While executing, tasks can read and write to the distributed data store. Writes to the distributed data store can trigger the store to send notifications to the dependency engine if the dependency engine has subscribed to that data.

the services (Section 4.1.1) and discuss division of processes into servers and worker (Sections 4.1.2 and 4.1.3). This material then leads into the implementation of the data store, task queue, and dependency engine (Sections 4.2, 4.3, and 4.4 respectively).

4.1.1 Runtime Operations

The runtime services provide operations that support distributed execution of Swift (or more generally, any task-parallel program that uses the API). This section lists the main operations that the runtime application programming interface (API) provides to set the stage for later discussion about implementation of the distributed runtime system.

Runtime Task Operations

Table 4.1 lists task operations that support adding and removing tasks from the distributed task 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

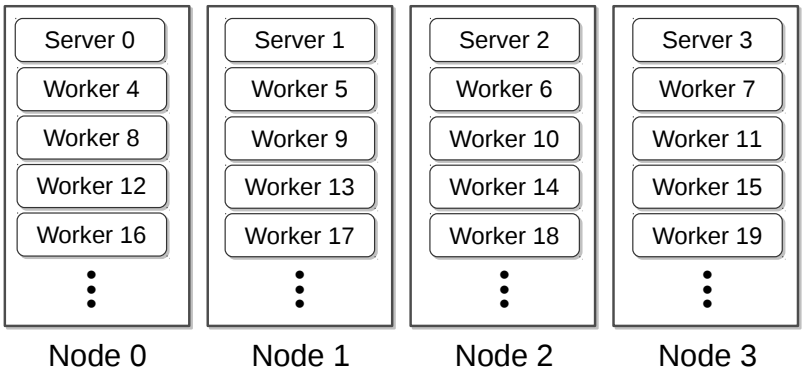


Figure 4.3: Runtime process layout on a distributed-memory system. Processes are divided into workers and servers, which are then mapped onto the processes of multi-core systems.

Table 4.1: Runtime task operations. Some advanced options and operations are omitted.

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

task in the dependency engine and PUT enqueues a task for immediate execution, bypassing the dependency engine. PUT is equivalent to DPUT with *dependencies* = $\emptyset$. GET is the basic way to get a task from the queue of the desired *type*. The AMGET, AGET_TEST, and AGET_WAIT operations support non-blocking gets of tasks: an advanced feature that is useful when a worker can execute multiple tasks in parallel or when a programmer wants to overlap task execution with task gets. AMGET initiates a request for a number of tasks of the specified type, then returns immediately before a matching task is received. A worker can then check for completion with AGET_TEST or AGET_WAIT, which are non-blocking and blocking respectively. Currently this feature is limited to supporting only multiple concurrent gets of the same type. The relationship between the blocking and non-blocking GET and AMGET operations is analogous to the relationship between RECV and IRECV in MPI.

A task has several attributes, specified when the task is added to the task queue with a *put* operation. These attributes influence which tasks are matched to a worker that issues a *get* operation. The *type* is one of a fixed set of task types specified at startup time that is used to segregate tasks into distinct types, e.g., CPU and GPU tasks. The *parallelism* indicates the number of workers that the task requires to execute – tasks with $parallelism > 1$ require assembling a team of multiple workers. Tasks with a higher integer *priority* are matched to workers preferentially, although global priority order is not guaranteed because it would require prohibitively expensive global synchronization. The task location specifies the workers to which the task should be sent. 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 a specific MPI process (each process is identified by a numeric *rank* in MPI parlance), or NODE if the task is targeted to a specific node. There are one or more ranks per node (see Figure 4.3), so specifying one of

the ranks contained in a node is sufficient to identify a node.

These features are useful and important in practice. Task types have been used to support GeMTC GPU tasks [58], task dispatch to multiple worker types when integrated with the NAMD molecular dynamics software [85], and integration with Coasters for execution of remote command-line applications [48]. Parallel tasks have many applications, such as running ensembles of the OSUFlow particle tracing application [120]. The use of task priorities to prioritize critical tasks (e.g., tasks that are longer-running or on the critical path of the application) can significantly improve system utilization and reduce time-to-solution [9, 7]. Rank and node-level targeting – both HARD and SOFT – have found applications in data-intensive applications where data is stored locally on compute or the cost of remotely reading data is significant [35, 121].

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 4.2 lists the 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 3.

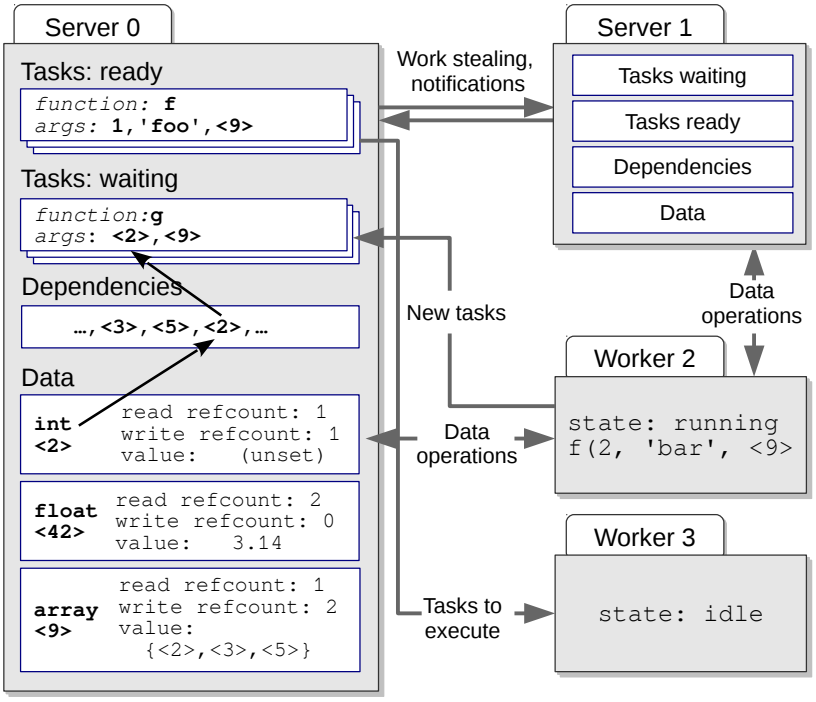


Figure 4.4: Runtime architecture showing distributed worker processes coordinating through task and data operations. Ready/waiting tasks and shared data items are stored on servers, with each server storing a subset of tasks and data. Servers must communicate to redistribute tasks through work stealing, and to request/receive notifications about data availability on other servers.

Table 4.2: Runtime data operations. Some advanced options and operations are omitted.

Operation	Arguments	Returns	Notes
CREATE	type, read_refcount, write_refcount	id	Create a typed data store item
MULTICREATE	create_params	ids	Create multiple data store items
RETRIEVE	id, path, refcount_decr, referand_acquire	value, type	Retrieve value of path of data store item. Caller can release refcounts or acquire refcounts if the value refers to another data store item
STORE	id, path, type, value, refcount_decr, referand_transfer	-	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.
INSERTATOMIC	id, path	success, value	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
REFCOUNTINCR	id, refcount_change	-	Increment or decrement refcount
SUBSCRIBE	id, path	subscribed	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.
COPY	from_id, from_path, to_id, to_path	-	Copy data from one path to another once frozen. Data types must match.

4.1.2 Server and Worker Processes

In order to implement the three distributed services provided by the runtime system, MPI processes are divided into two roles: *workers* and *servers*. Figure 4.3 shows a common way of distributing servers: one server per node. The system can be scaled up arbitrarily by proportionally adding processes of both types. Worker processes execute any program logic, coordinating with each other using the data and task operations provided by distributed services. The distributed services are implemented by the server processes and accessed with remote procedure calls (RPCs). An RPC is initiated with a worker sending a *request* message to the appropriate server – either the worker's local server or the server that owns the data to which the operation is being applied. The RPC 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 repeatedly checks for a incoming request message, processes the request, then responds to the request in a *server loop*. In some cases, e.g., a *get* request for which there is no matching task, the response is deferred until a later time. Any worker can communicate and synchronize with any server, but direct communication between workers is not supported, as shown in Figure 4.4.

Dedicated Server Processes

This design dedicates MPI processes (and therefore, in typical MPI configurations, CPU cores) to be servers. The obvious disadvantage of this approach 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, so as to avoid one thread waiting for an arbitrarily long period of time for another thread. Adding server processes solves the two-sided involvement 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.

All alternative solutions that we considered before settling on this design had downsides. One possible alternative is to use one-sided communication primitives, such as those provided in MPI. However, these primitives have a much narrower interface and still require a

dedicated thread to guarantee asynchronous progress in many cases [125]. Another alternative is to allocate more threads that cores and dedicate communication thread to processing incoming requests. 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, therefore, we believe that the server/worker model is the best solution for implementing distributed services in our runtime now and in the immediate future, despite the need to dedicate cores. Furthermore, 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 [104], the overhead is fairly low: 3–6%. We note that other systems [24, 125] have used dedicated communication threads for asynchronous handling of messages for essentially the same reasons.

Server Responsiveness

The server/worker architecture introduces challenges that must be addressed to achieve high performance and scalability. All communication is intermediated through the servers, so their responsiveness is critical to overall system throughput. 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 not block on operations that may take a significant amount of time, such as I/O or a response from another process. Sections 4.2 and 4.3 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 to achieve runtime performance and scalability, 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, in which 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 [30], data replication [45] and data migration [63]. We will not discuss solutions to this problem in detail here, leaving it to future work.

Inter-server synchronization occurs when a server has to 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. These issues are discussed in the next section.

4.1.3 Server-to-Server Synchronization

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

We 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.

Deadlocks

The possibility of deadlock arises when a server sends a message to invoke an operation on another server and enters a state in which 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 to the first server 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

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 become contagious. If all servers are lightly loaded and can quickly process any pending operations, invoking an operation on another server will cause only a slight delay in processing a server's own requests that can be quickly recovered from. However, if a server is heavily loaded, cascading dependent on the heavily loaded server and it is unable to delays can result when another server's progress becomes serve more of its own requests until the heavily loaded server responds. This phenomenon is rarely significant at small scale but can be catastrophic under certain circumstances at scales of 10,000+ cores. This behavior is further exacerbated by effects at the level of the MPI implementation and network stack: long unexpected message queues in MPI [111] and network contention [15] can both markedly reduce throughput.

Asynchronous Server to Server Operations

We can avoid both deadlocks and cascading delays by sending 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. Such *asynchronous RPCs* are 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 sync messages can be sent via low-overhead protocols such as the "short" and "eager" protocols used by BlueGene/Q [44] and Cray XE [88] 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.

Delegating Operations to Workers

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.

Deadlock Avoidance for Synchronous RPCs

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 alter-

nately, 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). Deadlock can occur when two servers (or a cycle of servers) are attempting to complete the other's request. To avoid this situation 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 method prevents a cycle forming.

4.2 Task Queue

Implementation of an efficient and scalable task queue hinges on two key features: efficient task matching algorithms and data structures to maximize throughput per server, and scalable work distribution algorithms to handle load imbalances between servers.

4.2.1 Task Matching

Our task matching algorithms and data structures handle the matching of tasks to workers on a single server, as illustrated in Figure 4.5. 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 4.6 provides pseudocode for the matching process for single-work tasks.[1] HandleGet and HandlePut are called

1. We omit detailed discussion of task matching for the case of tasks with *parallelism* > 1, which uses a somewhat different approach because it needs to assemble multiple workers into a team. Wozniak, Peterka,

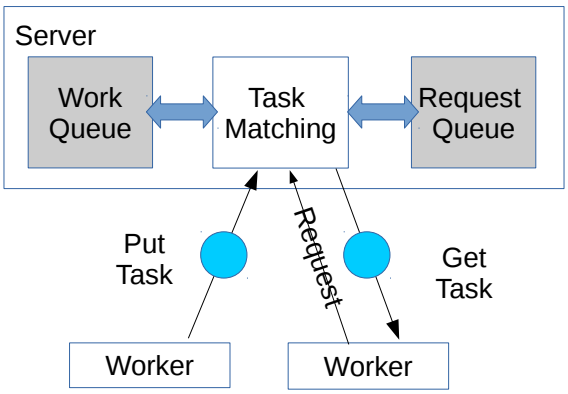


Figure 4.5: Task matching in the Swift/T runtime.

```
HandlePut(task, type, target, priority)
 1  if target.rank = Any
 2     match = FindRequestByType(type)
 3     if match ≠ Null
 4        SendWork(match.rank, task)
 5        return
 6  else
 7     match = FindRequestByTarget(target, type)
 8     if match ≠ Null
 9        SendWork(target, task)
10        return
11
12  EnqueueWork(task, type, target, priority)
```

```
HandleGet(rank, type)
 1  task = FindWorkByTarget(rank, type)
 2  if task ≠ Null
 3     SendWork(rank, task)
 4     return
 5
 6  task = FindWorkByType(type)
 7  if task ≠ Null
 8     SendWork(rank, task)
 9     return
10
11  EnqueueRequest(rank, type)
```

Figure 4.6: Matching algorithm for tasks in ADLB server. HandleGet and HandlePut are called from the main server loop when processing get and put operations.

from the server loop when get and put requests, respectively, are received from workers to match incoming requests and work to unmatched work and requests, respectively, stored on the server.

The main challenge of task matching in the runtime system is design of data structures that support efficient storage and lookup of unmatched requests and work. A single server manages 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 approach required linear search of the lists for most get and put

Armstrong, et al. [120] give a high-level description.

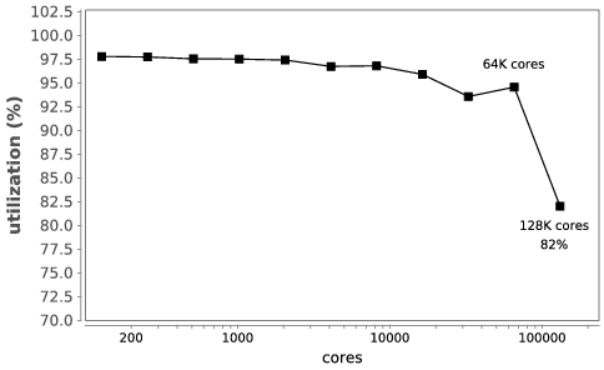


Figure 4.7: Scalability of previous version of Swift/T when using the original work queue design for 100–second tasks on Blue Gene/P Intrepid [119]

operations [22]. Thus, 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 become large in workloads with surplus parallelism. Thus linked lists perform acceptably only for applications with limited surplus parallelism. For this reason, early 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, relatively long, 100–second tasks, as shown in Figure 4.7 [119]. This behavior was due to limitations of early versions of both work queue data structures and work stealing algorithms.

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

Request Queue

The request queue tracks unmatched requests. It needs to support 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 4.8 shows a data structure that supports insertion and lookup of requests in $O(1)$ time for both targeted and untargeted

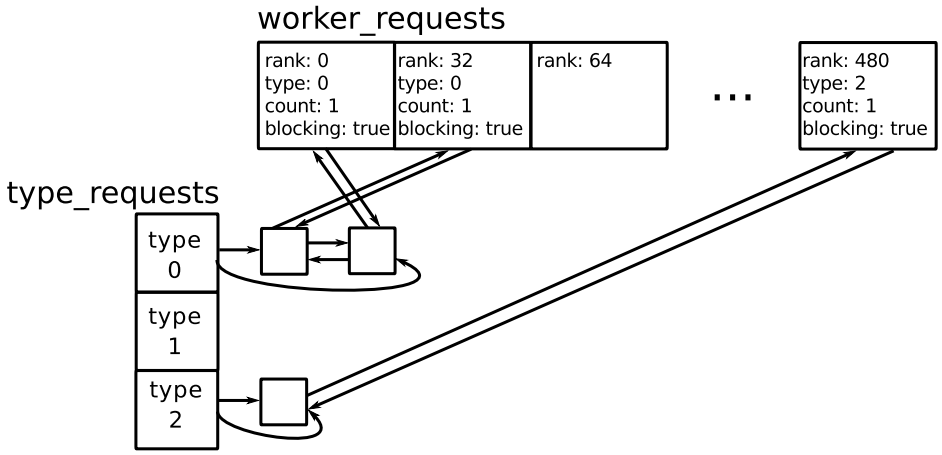


Figure 4.8: Request queue data structures for ADLB server 0 in a 16 node, 32 core per node system with 16 ADLB servers and 496 workers. Targeted and untargeted work is matched to requests by indexing into the `worker_requests` and `type_requests` arrays respectively. If the AMGET operation is used, each worker can have multiple concurrent requests of the same type, which are tracked by the *count* field. Matching incoming work to existing requests requires $O(1)$ time with this data structure.

work. In the runtime system architecture, each server manages a number of workers. A worker will only request request tasks via the server managing it. Therefore the requests for at most $\lceil \frac{num\_workers}{num\_servers} \rceil$ workers need to be tracked per server. A worker can also have multiple outstanding get requests of the same type from the AMGET operation.

Work Queue

The work queue tracks unmatched work. The work queue must support more complex matching rules, so the data structure is correspondingly more complex, as illustrated in Figure 4.9. 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-case amortized time to insert into or delete

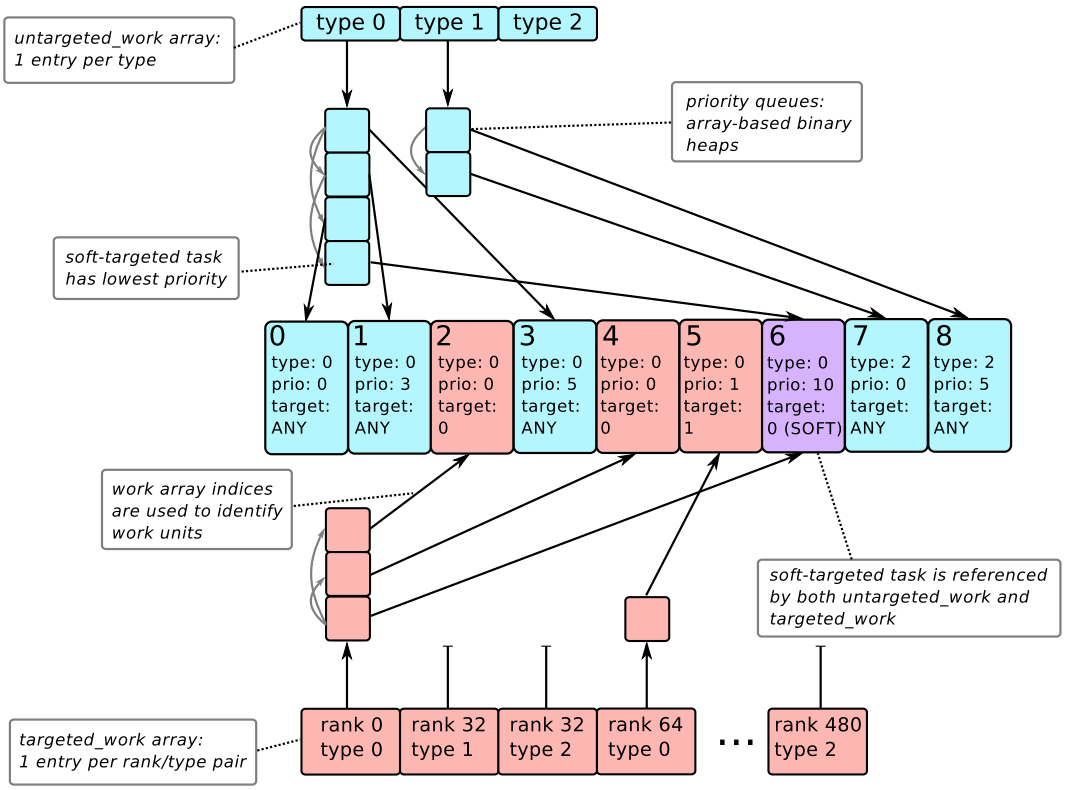


Figure 4.9: Work queue data structures for ADLB server 0 in a 16 node, 32 core per node system with 16 ADLB servers and 496 workers. These data structures support fast matching of requests to tasks with $O(log(n))$ time complexity.

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 (common) case where all priorities are equal, no heap entries need to be rearranged to maintain heap priority order. Therefore heap insertion/deletion and the overall complexity of put and get operations improves to $O(1)$ time if priorities are equal: the priority feature does not hurt the asymptotic complexity of work queue operations when it is not used. The array-based data structures also have good memory locality, so the constant factors associated with these operations are low on typical modern computer systems.

A significant advantage of this design – a main work array with indexes – is that additional indexes can easily be added For example, we implemented node-aware task targeting by adding and additional index `node_targeted_work` analogous to `targeted_work` that

Table 4.3: Comparison of work queue data structures. We assume multiple queues per type/rank are used. n is the number of enqueued tasks.

Operation	**Deque**	**Linked lists (ADLB)**	**Heaps (pointers)**	**Heaps (work array)**
Enqueue - no priorities	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Dequeue - no priorities	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Enqueue - priorities	$O(1)$	$O(1)$	$O(log(n))$	$O(log(n))$
Dequeue - priorities	$O(n)$	$O(n)$	$O(log(n))$	$O(log(n))$
Dequeue - soft targeting	$O(n)$	$O(n)$	$O(n)$	$O(log(n))$

supports lookup by a unique node index. For simplicity, this index is not shown in figures.

Futhermore, a task can be referenced by multiple indexes if needed. This is used to implement soft targeted tasks: the task is included in both the `targeted_work` and `untargeted_work` 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 and 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 the same type, target, and priority. In this case it in unproblematic if this task is returned despite being accessed through a stale entry.

Table 4.3 compares our data structure with alternatives. Deque is a traditional double-ended queue suitable for FIFO or LIFO policies. Finding the maximum priority task, however, requires scanning the entire deque. Operations of the linked list data structure from earlier versions of ADLB have similar complexity. The heap (pointers) structure was used in an earlier version of Turbine. In this approach, both targeted and untargeted heaps held pointers to soft targeted tasks. Removing a soft targeted task was a worst-case $O(n)$ operation because tasks found and removed via one heap had to be found in the other heaps via linear scans [2].

2. This could have been solved by adding a reference count to the task, but that approach would have deferred freeing the memory of the potentially large task payload.

Table 4.4: Comparison of related work on task matching.

	FIFO/LIFO Work Stealing	**Priority Scheduling**	**Grid Scheduling**	**Swift/T**
Task types	1	1	Many	Many
Scheduling	Online	Online	Ahead of time/ periodic	Online
Per-task overhead	ns/μs	ns/μs	ms/s	ns/μs
Prioritization	Heuristic (depth/breadth-first)	Numeric priority	Arbitrary policy	Numeric priority
Locality awareness	Heuristic (inertia)	Heuristic (inertia)	Arbitrary policy	Explicit (multiple levels and strictnesses)

Task Matching Related Work

Previous work on task scheduling and matching has involved devising task-matching algorithms to support a variety of policies and performance constraints that differ in various ways from those considered here. Table 4.4 summarizes the characteristics of existing solutions and our problem. LIFO and FIFO scheduling policies can be implemented efficiently with deque data structures in shared memory [1, 41] or distributed memory [32]. These deque-based approaches do not readily support priority scheduling or locality-aware scheduling beyond the heuristic of inertia, by which tasks stay in their original location until stolen. Priority scheduling can also be implemented very efficiently, although there is a somewhat higher overhead than when using priority queues [3, 74, 116]. However, again, pure priority scheduling only supports locality-aware scheduling to the same limited extent as deques. Grid scheduling systems have implemented 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 [66, 92, 108].

Outside of task scheduling, implementation of the MPI standard [109] 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 support for priorities or hierarchies of locations is provided. Popular MPI implementations

such as MPICH [71] and Open MPI [89] 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 [127]. 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.

4.2.2 Work Stealing

The task matching work described so far only solves the problem of matching work on an individual server to that server's own workers. If a server runs out of work, then it must somehow acquire more work from another server to prevent its workers from sitting idle. In practice it is common for uch load imbalances 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, 41] and distributed memory [32, 65, 96, 107] systems. Implementing work stealing in our runtime system presented challenges not directly addressed by previous work. All of the pre-existing literature on work stealing deals with a single uniform task type. However, to support heterogeneous task parallelism with hetereogenous workers, multiple task types need to be distributed. Thus we must generalize the work stealing algorithm to deal with per-type idleness. We also must factor the stealing message exchanges into asynchronous messages, as described in Section 4.1.3.

Figure 4.10 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 a response 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 [32] implements a distributed termination detection algorithm that uses a spanning tree 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 an older algorithm by Francez and Rodeh [40].

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 in which 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. We solve the multiple task type problem 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.

4.2.3 Task Queue Limitations and Future Work

The current task queue implementation provides an effective solution to 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

```
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 4.10: 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.

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 right workers. In the current implementation, work will find eventually find workers that can run 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" work stealing algorithm [96] offers an alternative approach: 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 and the idle thief can recommence work. Lifeline-based work stealing can decrease overhead and enable faster distribution of work in situations where work is scarce. This is a promising direction for future work, particularly since the request queue mechanism could be leveraged to implement lifelines.

4.3 Data Store

The runtime's data store implements a distributed data store with semantics based on the abstract data store described in Section 3.4.1.

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 to modify a data structure with no write references, will cause a runtime error. A path into a data structure is represented with a string of bytes that is interpreted in a manner 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 3.3.4.

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 a key/path pair becoming frozen. Release is implemented efficiently through a subscription mechanism: any process in the system can subscribe to a key/path pair. Subscriptions are tracked by the server to which the key maps, and when the frozen state is entered, a notification message is set to the subscriber.

4.3.1 Reference Counting

Efficient memory management is a challenging issue in a distributed context, especially in the highly dynamic execution model of data-driven task parallelism, because references to a data item may be held by many processes at any given time. 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 the executing program. As discussed in the previous chapter, the variable freezing problem can be formulated similarly: detecting when no WRITE references are held.

We tackle both problems with *automatic distributed reference counting*. We give each shared data item two reference counts (*refcount*), one for read references and one for write references. When a data item's write refcount drops to zero, it is frozen and cannot be written; when both refcounts drop to zero, the data can be deleted. Single-assignment

variables do not require special treatement, but for variables such as arrays, where multiple assignments are possible, refcounts must be correctly incremented and decremented to track the number of tasks and data structures with references to a key. 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, which avoids the problem in the case of Swift/T.

Correct reference counting depends on cooperation between the runtime and the compiler-generated application code. This code must correctly increment and decrement refcounts to correctly track the number of references in existence. Refcounts can be incremented or decremented explicitly with the RefcountIncr operation. Alternative, various runtime data operations (see Table 4.2) take additional arguments that specify refcount increments or decrements, which is is useful in 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 keep refcounts up to date. The data store must generate increment/decrement messages when owner references are copied and data structures holding references are freed. To support certain optimizations, a reference stored in a data store item can own multiple Read and Write refcounts for 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. The basic insight – that by owning multiple reference counts, a remote increment can be replaced with a local decrement – also motivates weighted reference counting schemes for distributed garbage collection [87].

4.4 Dependency Engine

The last service – the dependency engine – is perhaps the most straightforward of the three. Its sole responsibility is to accept data-dependent tasks and release them to the task queue when all their dependencies, specified in a list provided to the DPUT operation, are frozen.

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. When a new task is added, the engine subscribes to all of its input data paths. Once each path is frozen, the engine is notified. Once notifications are received for all input data, the task is released to the task queue. Subscribes and notifications are sent directly from the data store to the dependency engine within a server process when the subscribed-to data is in the same process or with server-to-server *sync* messages otherwise. Tasks are moved from the dependency engine to the task queue by copying a pointer from one data structure to another.

Two effective optimizations have been implemented in the engine:

- Multiple concurrent subscribes to the same data from a single engine are avoided with the help of a hash table that tracks current subscriptions for which a notification has not yet been received. When a notification is received, all of the tasks that subscribed to it are updated.
- Results of recent notifications about data on other servers is cached to avoid repeated subscribes to the same data. A least-recently-used policy is used to evict cache entries.

In the original architecture of the Turbine prototype [118], the dependency engine was implemented using a third dedicated class of "engine" processes. This architecture had some disadvantages that became clear as we gained experience with it. First and foremost, deciding how to allocate processes among three classes while taking into account various performance bottlenecks was considerably harder than allocating processes into two classes. There was also an efficiency disadvantage in some cases in which 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 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.

4.5 Evaluation

In this section we 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 the University of Illinois. Both systems have the same 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 [26].

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 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 [73].

Table 4.5: Task matching workload variants.

EQUAL	all priorities are equal
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

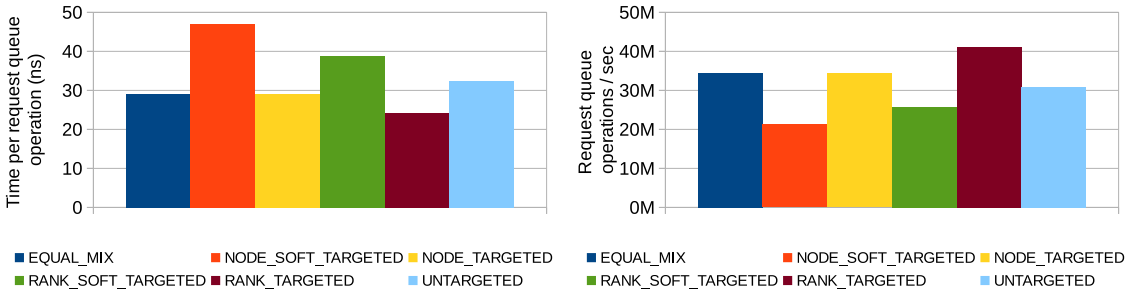


Figure 4.11: Efficiency of request queue data structure with task mixes of varying targeting parameters.

4.5.1 Task Matching Data Structure Evaluation

To evaluate the performance of the matching data structures, we ran a series of benchmarks that simulated incoming tasks and requests and measured the average time per request queue or work queue operation. In the experiments that involve a work queue, I varied the initial work queue size to understand how the work queues performed with thousands or millions of enqueued tasks. All workloads had equal numbers of adds and removes, so that the queue size for the duration of the experiment is closely correlated with the initial queue size – the queue size may drift slightly up or down but will on average not increase or decrease. I organized the experiments so as to test the data structures in isolation without any costs due to network communication and memory allocation for tasks. Specifically, before the timed portion of each experiment began, the experiment code generated and stored the tasks and sequence of operations in memory. This precomputation also avoided the measurable overhead of generating random numbers during the timed benchmark. Multiple workloads were tested, which are listed in Table 4.5.

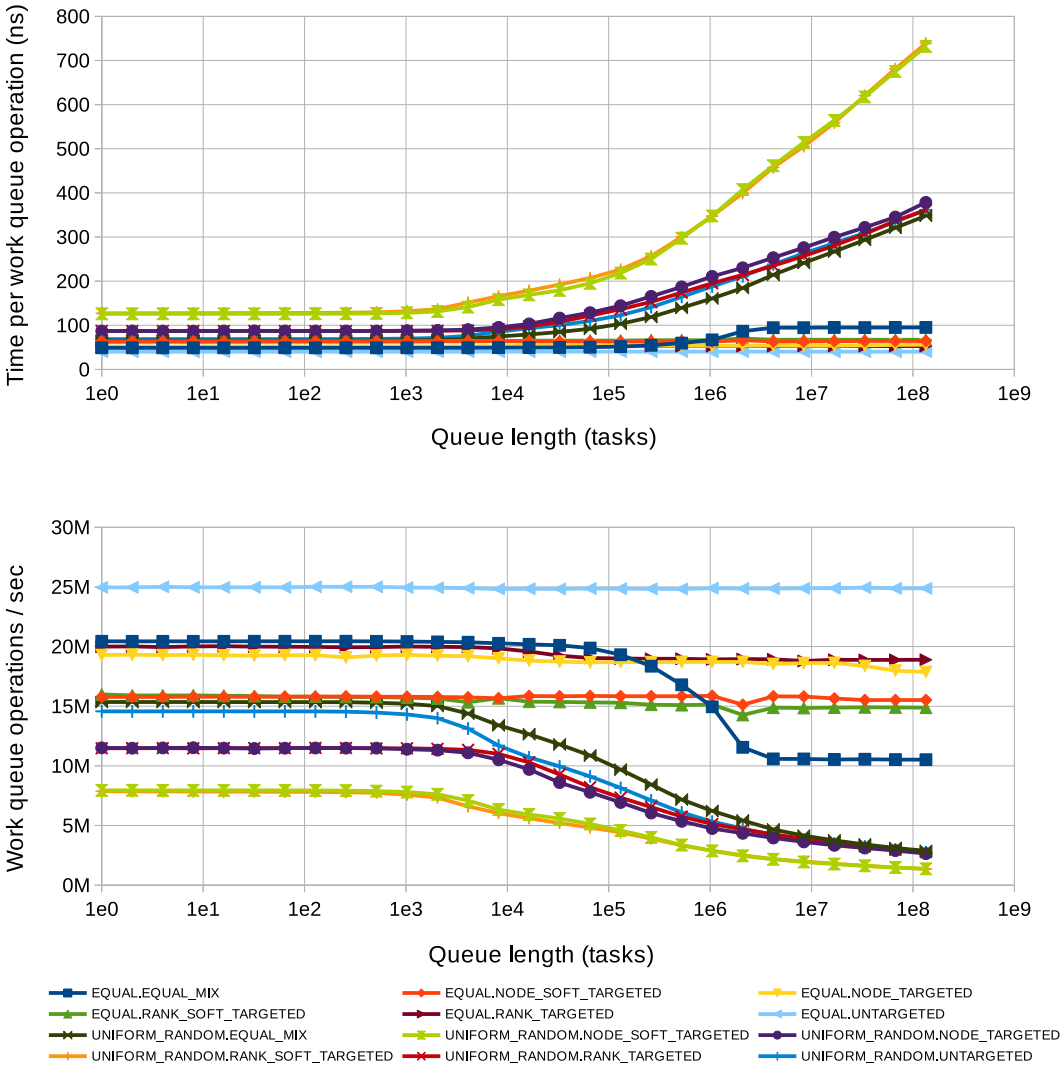


Figure 4.12: Efficiency and scalability of work queue data structure with task mixes of varying priority and targeting.

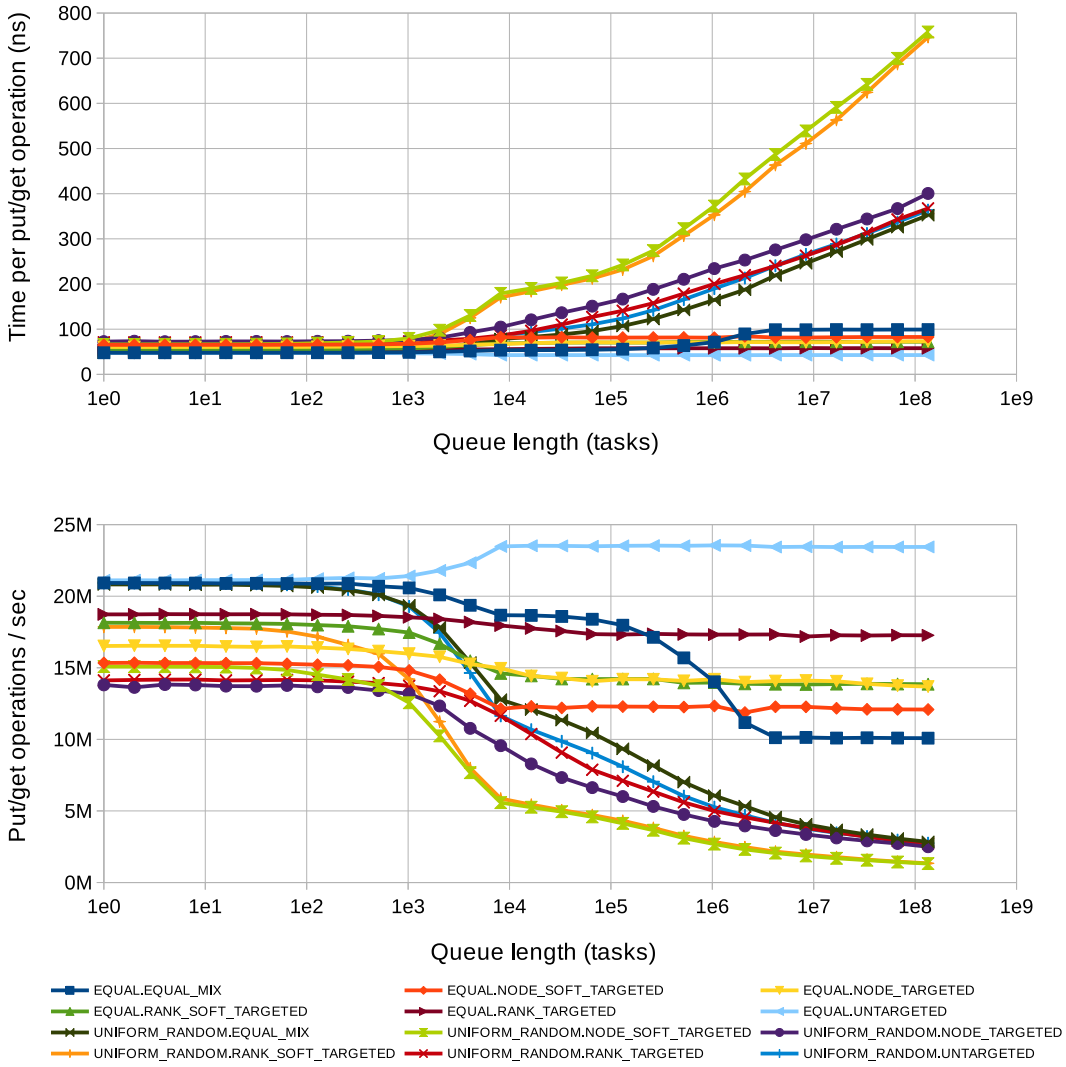


Figure 4.13: Efficiency and scalability of request and work queue data structures with task mixes of varying priority and targeting.

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

I first tested the request queue and work queue separately. Figure 4.11 shows performance results for the request queue. Matching strict rank targeted tasks is efficient. Matching node and soft targeted tasks is also efficient, with slightly higher overhead. Figure 4.12 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 4.6, with results shown in Figure 4.13. Here the numbers reflect the mean time per PUT or GET operation. Matching a task requires both a PUT and a GET operation, so we can double the numbers in Figure 4.13 to obtain the time taken to match a task.

The scalability curves for the work queue are similar to the curves for both queues (Figure 4.13). Overall everything is efficient – the worst workloads with 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 50–100 nanoseconds. With EQUAL priorities, performance is almost independent of queue length – $O(1)$ – with one exception: EQUAL-MIX, where performance drops between performances steeply markedly between 100,000 and 5,000,000 enqueued tasks before stabilizing. This unusual effect is likely caused by an interaction between the the growing working set size of the benchmark and the memory hierarchy. With UNIFORM-RANDOM priorities, the average time scales logarithmically – $O(log(n))$ – scaling to over 100 million enqueued tasks while maintaining good performance. Both NODE and SOFT targeting have lower performance – up to 50% slower that the equivalent workloads with other targeting modes. This is explained by the added complexity of the logic and data structures used to implement NODE and SOFT targeting.

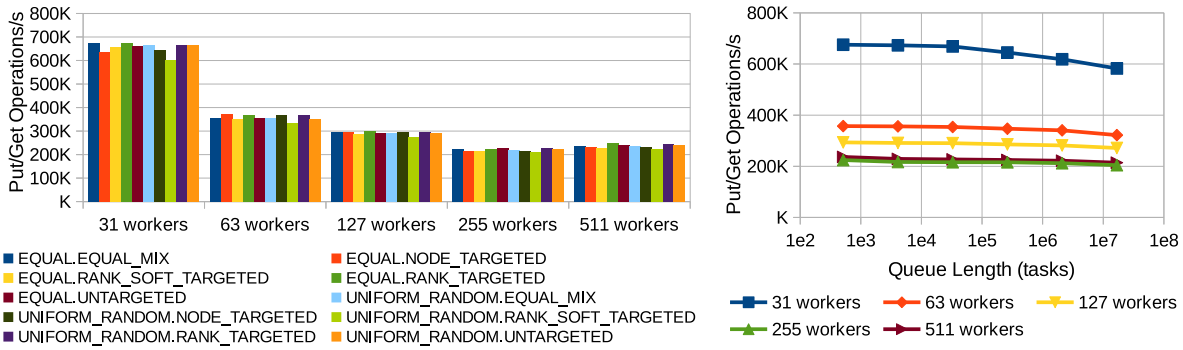


Figure 4.14: Single server task queue throughput measured in GET/PUT operations per second. The left figure shows performance for different task mixes with queue length of 512 tasks. The right figure shows performance by queue length for the UNIFORMRANDOM/EQUALMIX task mix.

The measured throughputs for the data structures 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 4.14 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 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: throughput drops markedly from one to two nodes when the network becomes involved and drops further as the number of nodes increases further.

We can draw two main conclusions from the single server experiment. First, the optimal configuration for per-server task throughput is to have at least one server per node so that all workers can *put* and *get* tasks to a server on the same node. Second, the task queue throughput is now limited by the MPI and network stack and synchronous RPC per task: further improvements would require batching tasks, bypassing MPI, or avoiding synchronous RPCs.

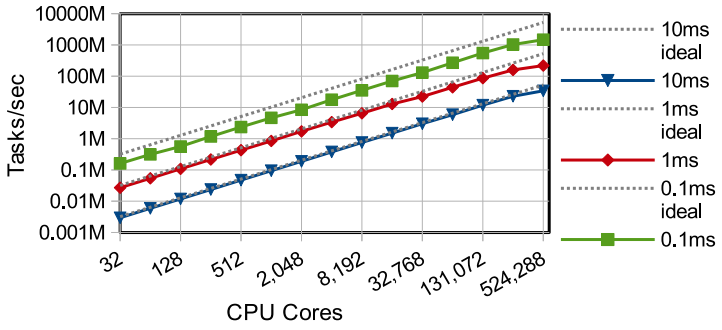


Figure 4.15: Throughput and scaling of runtime system for varying task durations.

4.5.2 Scalability Evaluation

I also performed large-scale scaling experiments using an earlier version of Swift/T on the Blue Waters supercomputer. This version of Swift/T did not incorporate all of the work stealing and task matching algorithm improvements described in this chapter. Unfortunately, at the time of writing, 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 run again today even better scaling and throughput would be achieved.

Figure 4.15 shows our scalability and task throughput results obtained by running an embarrassingly parallel Swift/T program that exercises task matching and work stealing. Swift/T achieved a peak throughput of 1.47 billion tasks/s on 524,288 cores running the Sweep benchmark described later in Section 5.8. Tasks of 1 ms or more achieve high efficiency when the servers are lightly loaded and queuing delays are minimal.

4.6 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 extension 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 [120].

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 [48] 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 [58].

4.7 Related Work

Execution models and runtime systems combining task parallelism with data dependencies for HPC applications have been explored by several groups [38]. Tarragon [25] and DaGuE [17] implement efficient parallel execution of explicit dataflow DAGs of tasks from within an MPI program. ParalleX [55] 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 [11] and OmPSS [20] 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 [106] and Habanero C [24] 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, although this falls short of providing transparent task migration between nodes. X10 supports asynchronous task parallelism, but synchronization is based on a finish statement and termination detection algorithms, instead of data-dependencies [107].

The Asynchronous Dynamic Load Balancer (ADLB) [68], 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 [32] 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 [80], CloudKon [95], and Apollo [18] 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 [113] has similar goals to our own work: implementing high-performance distributed task scheduling with policies such as data-aware scheduling. However, MATRIX has a very different design: it is built on a general-purpose key-value store. This design decision has its benefits, but it limits performance of task scheduling applications because the API and data structures are not well-suited to high-performance task scheduling. Reported task throughputs are limited to several thousand tasks per second. Our work shows that special-purpose data structures for task matching are required to achieve high performance given scheduling policies such as location-awareness and priorities.

CHAPTER 5

COMPILING SWIFT FOR MASSIVE PARALLELLISM

STC is a whole-program optimizing compiler for Swift/T that compiles high-level Swift code to run on the distributed runtime described in Chapter 4. Figure 5.1 illustrates how STC fits into the Swift/T toolchain. We discuss the overall design of STC in Section 5.1. Within STC we have implemented optimizations aimed at reducing communication and synchronization without loss of parallelism (Section 5.3). An intermediate representation captures the execution model (Section 5.4), allowing optimization to reduce synchronization and runtime task/data operations involving shared data (Section 5.5). It enables garbage collection by reference counting and optimization thereof (Section 5.6). STC generates code in the Tcl scripting language that can execute on the distributed runtime system described in the previous chapter (Section 5.7). Finally, we evaluate the effectiveness of the compiler optimizations on several benchmarks (Section 5.8). This chapter is an expanded version of a previously published paper [6].

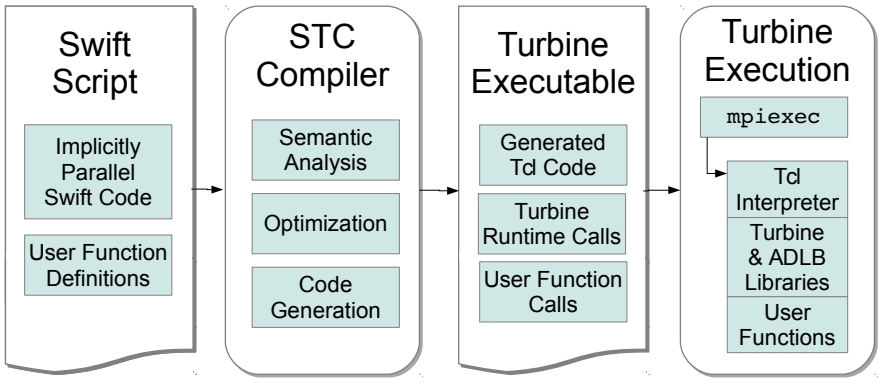


Figure 5.1: The STC compiler is in the middle of the Swift/T toolchain and translates high-level Swift code into execution code for the Turbine runtime.

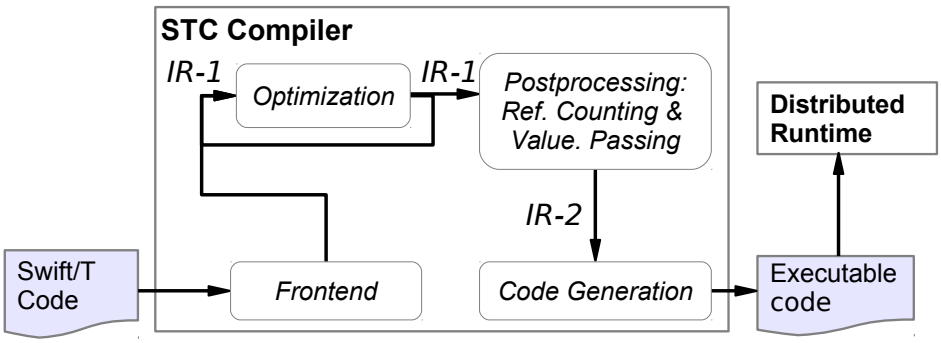


Figure 5.2: STC compiler architecture. The frontend produces IR-1, to which optimization passes are applied to produce successively more optimized IR-1 trees. Postprocessing adds intertask data passing and reference counting information to produce IR-2 for code generation.

5.1 Compiler Architecture

The STC compiler architecture 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 3.7, 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 postprocessing, which augments IR-1 with additional information required for code generation: reference counting and data transfer between tasks. STC's phases are illustrated in Figure 5.2. Almost all modern compilers follow similar designs.

5.2 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 [84] that translates the input text into an abstract syntax tree (AST). Typechecking is performed on the AST to ensure

```
1 | foreach i in [1:N] {
2 |   foreach j in [1:M] {
3 |     a, b, c = A[i-1][j-1], A[i-1][j], A[i][j-1];
4 |     A[i][j] = h(f(g(a)), f(g(b)), f(g(c)));
5 |   }
6 | }
```

Figure 5.3: Swift code fragment illustrating wavefront pattern.

the validity of the program. Some basic semantic analysis is also performed that generates warnings and errors for simple errors in 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 Section 3.7 works to translate Swift into the compiler's intermediate representation because the intermediate representation closely mirrors the abstract execution model.

A user-supplied 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.

5.3 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 in the data store or task queue services.

The implicitly parallel Swift/T code in Figure 5.3 illustrates the opportunities and challenges of optimization. The code specifies a dynamic, data-driven wavefront pattern of parallelism, in which 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: for example, 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 over-

head, on the other hand, we would merge these four tasks into a single task that executes the `f(g(...))` calls sequentially.

5.4 Intermediate Representation

The STC compiler uses a medium-level intermediate representation (IR) that captures the execution model of data-driven task parallelism. The tree structure of the intermediate representation can be mapped to the spawn tree of tasks, and dependencies through lattice data types are a first-class part of the IR.

Two IR variants are used in STC, as shown in Figure 5.2. 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.

5.4.1 Structure of Intermediate Representation

We now describe the structure of the basic STC intermediate representation, IR-1, which is formally specified by the formal grammar in Figure 5.4. Each IR program is comprised of a number of functions. The functions are either external functions, such as foreign functions or invocations of command-line 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 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

⟨*program*⟩	::=	⟨*ext-fn*⟩ ⋆ ⟨*fn*⟩⋆	*(program - global variables are omitted)*
⟨*ext-fn*⟩	::=	⟨*fn-hdr*⟩ ⟨*binding*⟩	*(external/foreign function)*
⟨*fn*⟩	::=	⟨*fn-hdr*⟩ ⟨*block*⟩	*(IR function definition)*
⟨*fn-hdr*⟩	::=	⟨*id*⟩ ⟨*exec-target*⟩ *Out*⟨*var*⟩ ⋆ *In*⟨*in-arg*⟩⋆	*(function header)*
⟨*binding*⟩			*(external/foreign function binding information)*
⟨*exec-target*⟩	::=	⟨*exec-context*⟩ ⟨*exec-mode*⟩	*(execution target - how task is run)*
⟨*exec-context*⟩	::=	*Any* | *Control* | *Work*⟨*work-type*⟩	*(required execution context - where)*
⟨*exec-mode*⟩	::=	*Sync* | *Async* | *LoadBalanced*	*(execution mode - how to dispatch)*
⟨*in-arg*⟩	::=	⟨*var*⟩ *WaitFor*⟨*bool-literal*⟩	*(input - function may wait until frozen)*
⟨*block*⟩	::=	⟨*var*⟩ ⋆ ⟨*stmt*⟩ ⋆ ⟨*continuation*⟩⋆	*(IR code block)*
⟨*stmt*⟩	::=	⟨*inst*⟩ | ⟨*if*⟩ | ⟨*switch*⟩	*(statement)*
⟨*inst*⟩	::=	⟨*opcode*⟩ *Out*⟨*var*⟩ ⋆ *In*⟨*arg*⟩⋆	*(instruction)*
⟨*if*⟩	::=	*If* ⟨*arg*⟩ *Then*⟨*block*⟩ *Else*⟨*block*⟩	*(if stmt.)*
⟨*switch*⟩	::=	*Switch* ⟨*arg*⟩ (*Case*⟨*int-literal*⟩⟨*block*⟩) ⋆ *Default*⟨*block*⟩	*(switch stmt.)*
⟨*continuation*⟩	::=	⟨*wait*⟩ | ⟨*foreach*⟩ | ⟨*range*⟩ | ⟨*loop*⟩ | ⟨*async-exec*⟩	*(continuation)*
⟨*wait*⟩	::=	*Wait* ⟨*wait-var*⟩ ⋆ ⟨*exec-target*⟩ ⟨*wait-mode*⟩ ⟨*block*⟩	*(wait statement)*
⟨*wait-var*⟩	::=	⟨*var*⟩ *Explicit*⟨*bool-literal*⟩	*(variable waited on)*
⟨*wait-mode*⟩	::=	*WaitOnly* | *TaskDispatch*	*(wait mode - controls allowable optimization)*
⟨*foreach*⟩	::=	*Foreach*⟨*var*⟩ ⟨*var*⟩ (*Key* ⟨*var*⟩)? ⟨*block*⟩	*(foreach loop)*
⟨*range*⟩	::=	*Foreach* ⟨*arg*⟩ *to* ⟨*arg*⟩ ⟨*var*⟩ ⟨*block*⟩	*(foreach integral range loop)*
⟨*loop*⟩	::=	*Loop*(*IterVar* ⟨*var*⟩ *Init* ⟨*arg*⟩ *Blocking* ⟨*bool-literal*⟩) ⋆ ⟨*block*⟩	*(tail-recursive loop)*
⟨*async-exec*⟩	::=	⟨*executor*⟩ ⟨*arg*⟩ ⋆ *Continuation*⟨*block*⟩ *ErrCont*⟨*block*⟩	*(async. execution stmt.)*
⟨*executor*⟩			*(external executor identifier)*

Figure 5.4: Grammar for structure of the IR-1 variant of the STC intermediate representation. We omit punctuation and some syntax used in the textual representation.

⟨*arg*⟩	::=	⟨*var*⟩ | ⟨*const*⟩	*(argument)*
⟨*var*⟩	::=	⟨*id*⟩ ⟨*type*⟩ ⟨*storage*⟩	*(variable declaration/reference)*
⟨*id*⟩	::=	⟨*string-literal*⟩	*(identifier)*
⟨*type*⟩	::=	⟨*prim-type*⟩ | ⟨*shared-type*⟩ | ⟨*array-type*⟩ | ⟨*bag-type*⟩ | ⟨*struct-type*⟩	*(types)*
⟨*prim-type*⟩	::=	'*int*' | '*bool*' | '*float*' | '*string*' | '*blob*' | '*file*'	*(primitive types)*
⟨*shared-type*⟩	::=	⟨*type*⟩ ' ∗ '	*(reference to shared type)*
⟨*array-type*⟩	::=	⟨*type*⟩[⟨*prim-type*⟩]	*(associative array type with primitive type key)*
⟨*bag-type*⟩	::=	*bag*⟨⟨*type*⟩⟩	*(bag type - unordered multiset)*
⟨*struct-type*⟩	::=	*Struct* (⟨*id*⟩ ⟨*type*⟩)⋆	*(structure type)*
⟨*storage*⟩	::=	*TaskLocal* | *Shared* | *SharedAlias*	*(variable storage)*
⟨*const*⟩	::=	⟨*int-literal*⟩ | ⟨*bool-literal*⟩ | ⟨*float-literal*⟩ | ⟨*string-literal*⟩	*(literal constant)*

Figure 5.5: Type system and variables used in STC intermediate representation.

```
1  import io;
2  import sys;
3
4  int n = parseInt(argv("n")); //  Get command line argument
5  int f = fib(n);
6
7  //  Print result once computation finishes
8  printf("fib(%i)=%i", n, f);
9
10 (int o) fib (int i) {
11    if (i == 0) {
12      o = 0;
13   } else  if (i == 1) {
14      o = 1;
15   } else {
16       //  Compute fib(i-1) and fib(i-2) concurrently
17      o = fib(i - 1) + fib(i - 2);
18   }
19 }
```

Figure 5.6: Swift/T code implementing the naïve recursive algorithm for computing Fibonacci numbers.

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.

Figure 5.7 provides a concrete example of IR-1 with unoptimized and optimized versions of the Swift Fibonacci program shown in Figure 5.6.

5.4.2 Detailed Description of IR

This section describes semantics of key aspects of the IR in more detail and additional properties of different IR structures that are used in the compiler for analyzing and transforming the intermediate representation. To clarify IR semantics, pseudocode for an IR interpreter is provided in Appendix C of the appendices.

Types, Variables, and Args

The type and variable system is shown in Figure 5.5. Variables are identified by name – $\langle id \rangle$ – and have associated type and storage information. Fields that can either be a variable or a constant – for example, instruction input arguments – are represented with $\langle arg \rangle$.

```
() @__entry () {
  // Declare variables for block
  declare *int f, *string t0, *string t1,
          *int t2, *string t3, *void t4
  // Store value to shared variable
  Store [ t0 ] [ "n" ]
  // Run functions as asynchronous tasks
  // Runtime manages data dependencies
  CallForeignAsync argv [ t1 ] [ t0 ]
  CallForeignAsync parseInt [ n ] [ t1 ]
  CallAsync fib [ f ] [ n ]
  Store [ t3 ] [ "fib(%i)=%i" ]
  CallForeignAsync printf [ t4 ] [ t3 n f ]
}

// Compute fibonacci(i) and store to o
// Inputs and outputs are shared variables
(int o) @fib (int i) {
  declare *boolean t0, *int t1
  // t0 = (i == 0)
  Store [ t1 ] [ 0 ]
  AsyncOp EqInt [ t0 ] [ i t1 ]
  wait (t0) {
    declare $bool v_t0
    Load [ v_t0 ] [ t0 ]
    if (v_t0) {
      Store [ o ] [ 0 ] // o = 0
    } else {
      declare *boolean t2, *int t3
      // t2 = ( i == 1 )
      Store [ t3 ] [ 1 ]
      AsyncOp EqInt [ t2 ] [ i t3 ]
      wait (t3) {
        declare $bool v_t3
        Load [ v_t3 ] [ t3 ]
        if (v_t3) {
          Store [ o ] [ 1 ]
        } else {
          declare *int t4, *int t5, *int t6,
                  *int t7, *int t8, *int t9
          // t4 = fib(i - 1)
          Store [ t6 ] [ 1 ]
          AsyncOp MinusInt [ t5 ] [ i t6 ]
          CallAsync fib [ t4 ] [ t5 ]
          // t7 = fib(i - 2)
          Store [ t9 ] [ 2 ]
          AsyncOp MinusInt [ t8 ] [ i t9 ]
          CallAsync fib [ t7 ] [ t8 ]
          // Compute sum o = t4 + t7
          AsyncOp PlusInt [ o ] [ t4 t7 ]
} } } } }
```

(a) IR-1 unoptimized.

```
() @__entry () {
  // Declare variables for block
  declare int v_n, string t0, *int f, void t1
  CallForeign argv [ t0 ] [ "n" ]
  CallForeign parseInt [ v_n ] [ t0 ]
  CallAsync fib [ f ] [ v_n ]
  wait (f) { // execute block once f is frozen
    declare int v_f
    Load [ v_f ] [ f ] // Load value of f to v_f
    CallForeign printf [ t1 ]
                       [ "fib(%i)=%i" v_n v_f ]
} }

// Compute fibonacci(i) and store to o
// input v_i is value, output o is shared var
(int o) @fib ($int v_i) {
  declare boolean t0
  // t0 = (v_i == 0)
  LocalOp EqInt [ t0 ] [ v_i 0 ]
  if (t0) {
    Store [ o ] [ 0 ] // o = 0
  } else {
    declare boolean t2
    // t2 = ( v_i == 1 )
    LocalOp EqInt [ t2 ] [ v_i 1 ]
    if (t2) {
      Store [ o ] [ 1 ] // o = 1
    } else {
      declare int v_t4, int v_t7, *int t4,
              *int t7
      // t4 = fib(v_i - 1)
      LocalOp MinusInt [ v_t4 ] [ v_i 1 ]
      CallAsync fib [ t4 ] [ v_t4 ]
      // t7 = fib(v_i - 2)
      LocalOp MinusInt [ v_t7 ] [ v_i 2 ]
      CallAsync fib [ t7 ] [ v_t7 ]
      // Compute sum o = t4 + t7
      AsyncOp PlusInt [ o ] [ t4 t7 ]
} } }
```

(b) IR-1 optimized at -O2.

Figure 5.7: Sample STC IR-1 for recursive Fibonacci algorithm, corresponding to the Swift code shown in Figure 5.6. The IR comprises two functions: `main` and `fib` functions. IR instructions include Swift function calls (e.g., `CallAsync fib`), foreign function calls (e.g., `CallForeign printf`), immediate arithmetic operations (e.g., `LocalOp EqInt`), data-dependent arithmetic operations (e.g., `AsyncOp MinusInt`), and reads and writes of shared data items (`Load` and `Store`, respectively). Control flow constructs used include conditional `if` statements and `wait` statements for data-dependent execution.

⟨*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 an integer stored in the data store, and *Shared Shared IntType* is a reference to a reference to an integer. The latter type – a reference to a reference – cannot be expressed directly in Swift code but sometimes the compiler frontend makes use of these types to resolve array lookups when the key value is not yet known. Numeric, string and boolean literal values are captured with the ⟨*const*⟩ production. Shared data types are based on the lattice data types described in Section 3.3.4, including I-vars, associative arrays, and tuples (used to implement structs).

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. Global variables 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 5.5: 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.

There are a number of rules that must be followed for reading and writing variables for the IR to be valid:

- Each TaskLocal variable can only be assigned once, either in the same block in which it is declared, or on disjoint branches of conditionals inside that block.
- Each TaskLocal variable can only be read after being written.
- Function input variables cannot be written inside the function.
- Alias variables must be *initialized* to refer to some shared data before being read or written. Aliases are initialized by instructions as described in Section 5.4.2.

IR Tree

Each IR function is essentially a tree of code blocks linked together with control-flow structures. The main block of each function is the root of each tree. Every block has zero or more child control flow structure (conditionals and continuations), and each of those has at least one child block. Each block, except for the main one, has a parent control-flow structure, which in turn has a parent block.

The intermediate representation tree can be mapped onto tasks. Given an IR block, all descendent statements and continuations with a synchronous ⟨*exec-target*⟩ execute in the same task. Continuations with an asynchronous ⟨*exec-target*⟩ execute in a child task: an asynchronous continuation can be thought of as a task boundary.

Task Execution Contexts

The intermediate representation has a notion of *execution context*, which describes the context that the task is executing in and is represented in the grammar with the ⟨*exec-context*⟩ production. Execution contexts are divided in two ways: by the locations the code executes

in and whether the code may execute long-running "work" instructions.

An instruction is *long running* if it executes synchronously (i.e., in the current task) for a long or unbounded duration. STC's optimization passes avoid serializing execution of long-running instructions that could run in parallel. Short-running tasks – where the overhead of parallel execution is large compared to parallel speedup – can be serialized. The exact division of short and long-running instructions is somewhat arbitrary, but in practice most instructions fall clearly in one category or the other: simple built-in functions such as arithmetic, string operations, task, and data operations versus computationally intensive user functions.

In the Control context, executing long-running work functions is not allowed: all operations must finish in a short, bounded, amount of time. Thus, the optimizer to is free to batch and group operations in a Control context without reducing (much) parallelism. The Control context can execute on any worker in the runtime system. In contrast, Worker contexts can be subdivided based on work type: the runtime system supports dividing workers into roles, e.g., CPU vs. GPU workers. The Worker execution context is optionally parameterized by work type. Long-running function calls are permitted in Work contexts, but the optimizer must take care to not eliminate parallelism that could occur between two long-running function calls in Work contexts. The Any context is used is certain places to indicate that code can run in any Control or Work context.

The execution context of a given block of code can be determined by traversing the containing IR function from its root. The main block of any IR function executes in the Control execution context. All child blocks of an IR function execute in the same execution context as their parent, unless a continuation changes the execution context.

We describe where code executes relative to the current task with *execution targets*, which are represented in the grammar with the ⟨*exec-target*⟩ production. Every IR statement and control-flow structure, including instructions, conditionals, and continuations, has an execu-

⟨*opcode*⟩ ::= *Comment* | ⟨*invoke-opcode*⟩ | ⟨*data-opcode*⟩ | ⟨*file-opcode*⟩ *(instruction operation code)*
| ⟨*checkpoint-opcode*⟩ | ⟨*loop-opcode*⟩

⟨*invoke-opcode*⟩ ::= | *Call* ⟨*id*⟩ | *CallAsync* ⟨*id*⟩ | | *CallForeign* ⟨*id*⟩ *(operation invocation operation codes)*
| *CallForeignAsync* ⟨*id*⟩ | *LocalOp* ⟨*builtin-opcode*⟩ | *AsyncOp* ⟨*builtin-opcode*⟩ | *Exec*

⟨*data-opcode*⟩ ::= *Store* | *Load* | *StoreRecursive* | *LoadRecursive* *(data store operation codes)*
| *FlattenRecursive* | *Copy* | *CopyAsync* | *Deref* | *CreateAlias* | *CreateNested*

⟨*file-opcode*⟩ ::= *CopyPhysicalFile* | *LoadFilename* | *StoreFilename* *(file-specific operation codes)*
| *CreateFilenameAlias* | *CopyInFileName* | *IsMapped* | *ChooseTmpFilename*

⟨*checkpoint-opcode*⟩ ::= *CheckpointWriteEnabled* | *CheckpointRestoreEnabled* *(checkpointing operation codes)*
| *WriteCheckpoint* | *LookupCheckpoint* | *PackBlob* | *UnpackBlob*

⟨*loop-opcode*⟩ ::= *LoopBreak* | *LoopContinue* *(loop iteration opcodes)*

⟨*builtin-opcode*⟩ ::= *PlusInt* | *MinusInt* | *MultInt* | *DivInt* | *ModInt* | *PlusFloat* *(builtin sub-operation codes)*
| *MinusFloat* | *MultFloat* | *DivFloat* | *Dircat* | *NegateInt* | *NegateFloat* | *PowInt* | *PowFloat*
| *MaxInt* | *MaxFloat* | *MinInt* | *MinFloat* | *AbsInt* | *AbsFloat* | *EqInt* | *NeqInt* | *GtInt*
| *LtInt* | *GteInt* | *LteInt* | *EqFloat* | *NeqFloat* | *GtFloat* | *LtFloat* | *GteFloat* | *LteFloat*
| *EqBool* | *NeqBool* | *EqString* | *NeqString* | *Not* | *And* | *Or* | *Xor* | *Strcat* | *Substring*
| *CopyInt* | *CopyFloat* | *CopyBool* | *CopyString* | *CopyBlob* | *CopyVoid* | *Floor* | *Ceil* | *Round*
| *IntToFloat* | *FloatToInt* | *ParseInt* | *IntToStr* | *ParseFloat* | *FloatToStr* | *Log* | *Exp* | *Sqrt*
| *IsNan* | *AssertEq* | *Assert* | *Sprintf*

Figure 5.8: Grammar describing operation codes for the IR-1 variant of the STC intermediate representation.

tion target. In the case of wait statements, the execution target is explicitly specified. For other constructs, it is implied and can be obtained with a function ExecTarget(*structure*). An ⟨*exec-target*⟩ specifies any ⟨*exec-context*⟩ plus some additional information: whether the code executes synchronously or asynchronously, and whether the code should be *dispatched* – eligible for load balancing and execution on a different process.

Instruction Properties

For the purposes of IR semantics and analysis, each instruction has a number of inherent and derived properties. Each instruction has an ⟨*opcode*⟩, which identifies the general kind of the instruction. These are listed in Figure 5.8. Each opcode will accept different numbers and types of arguments. Some even accept variable numbers of arguments.

An instruction's *inputs* are any program variables or literal constants that are read by

the instruction. They are not written by the instruction. *Blocking inputs* are defined for functions with asynchronous execution targets. These are inputs that must be frozen before the code executes and before instruction outputs are written. *Closed outputs* are defined for functions with synchronous exection targets. These are outputs that are always frozen once the instruction execution completes.

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 5.4.2.

Each instruction may or may not have *side effects*, where the instruction has some effect on something other than its 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 the instruction's side effect status is not known, the compiler must conservatively assume that the instruction does have side-effects. The process by why an instruction is determined to be side-effect free depends on the opcode. Some 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.

As mentioned before, instructions have an associated ⟨*exec-target*⟩ that describes how the instruction will execute (e.g., if the instruction spawns a task, or if it is a long-running work task, or if it must execute in a particular work context). Instructions are *progress enabling* if execution of the instruction may fulfill data dependencies of other tasks: for example, a shared data write. The STC optimizer will avoid 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 to classify whether instructions 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. Execution 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.

Control Flow Structure Properties

Each control-flow structure has a number of properties:

- Construct defined variables: variables that are newly defined in the blocks insides the structure.
- Required variables that are read or written as part of construct's execution.
- MustRunLast: a boolean flag that forces a continuation to run after any continuations without the flag.
- Whether it executes each enclosed block exactly once – e.g. loops may execute the loop body multiple times, but a wait statement will execute the block exactly once.
- Blocking variables: any variables that the continuation waits for befor executing any enclosed blocks.
- Frozen variables: a superset of blocking vars that also includes any construct defined variables that are closed.

Conditional Properties

Conditional statements – if and switch statements – have related properties that are used in optimizer analyses. All conditionals in the IR are *exhaustive*, meaning that 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.

5.4.3 Initialization Analysis

The initialization analysis allows determining the set of initialized aliases and assigned TaskLocal variables at each point in the intermediate representation. This information is useful in many passes for checking whether a transformation is allowed. The set of initialized variables can be updated incrementally as part of an in-order walk over the IR tree.

5.5 STC Optimizer

The STC optimizer is comprised of many *optimization passes*, each of which analyzes and transforms the IR into a form that (hopefully!) executes more efficiently on the runtime system. We categorize individual optimization passes into traditional optimizations (Section 5.5.1), shared data optimizations (Section 5.5.2) and task parallelism optimizations (Section 5.5.3). For reference, the ordering of optimization passes is documented in Section D.1 of the appendices.

5.5.1 Adaption of Traditional Optimizations

The foundation of our suite of optimizations is a range of traditional optimization techniques adapted from conventional compilers [72] to our intermediate representation and execution model in general. This work 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* [72] (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 has no side-effects. The VN pass identifies congruence relations for each IR block. Value congruence, for example, $retrieve(x) \cong^V y * 2 \cong^V 6$, 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 $A \cong^V store(1)$ and $1 \cong^V retrieve(A)$. These new relations are added to the known relations, perhaps merging existing congruence classes. For example, if $B \cong^V store(1)$, 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 $x \cong^V retrieve(A)$ and $y \cong^V retrieve(A)$ on the branches, a new variable t_1 is assigned x and y on the respective branches, so that $t_1 \cong^V retrieve(A)$ 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* [72], whereby expressions with constant arguments can be evaluated during the VN tree walk. All builtin operations with ⟨*builtin-op*⟩ operation codes listed in Figure 5.8, including common arithmetic, logical, and string operations, can be evaluated by the compiler. 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.

Dead Code Elimination

Dead code elimination (DCE) eliminates unneeded code that is never executed or that computes unneeded results. It can eliminate 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 [57]. 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 lattice data structures. Thus, any assignment to a variable may generally 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 limits the optimization's ability to eliminate unneeded data structures such as nested arrays or struct. For example, in the example in Figure 5.9, value numbering can replace the argument to `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`.

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

Figure 5.9: 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. We 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, $*$, which indicates that a reference must be traversed, e.g., $\langle x, *\rangle$ 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 $maybeAliases(G, v_1, c_1)$, and add an edge to the dependency graph from v_2 to v_1 for each $v_2 \in maybeAliases(G, v_1, c_1)$ 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 live variable, 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 can be inlined into a single function, allowing optimization across the whole program.

STC's inlining pass uses several simple heuristics 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: *function IR instruction count* $\times$ *# 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, in which redundant operations such as array accesses occur inside nested parallel foreach loops. The hoisting pass can also hoist operations out of Wait statements.

IR instructions can be hoisted out of an enclosing construct when they are deterministic, have no side-effects or only idempotent side-effects, and their inputs are declared and written outside of the construct. These conditions can be checked with a single walk over the IR tree, keeping track of the last enclosing scope in which each variable was assigned.

Loop Unrolling

Loop unrolling performs loop unrolling for range loops. The main benefit of this pass 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 pass 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 transformation allows optimization across the fused blocks. For foreach and range loops, it also can reduce runtime overhead associated with the loop.

This pass is implemented by visiting each block in the IR tree and pairwise checking each pair of conditionals and each pair of continuations of the same type in the block to see if the necessary conditions for fusing them hold.

Alias Propagation

The *alias propagation* pass tries to reduce the use of aliases created by operations such as CreateAlias or CreateNested by replacing the alias with the original at the point of use. This can entail concatenating paths if the alias is for a path of a shared data structure.

5.5.2 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) infers which I-vars, arrays, and other data structures are frozen at each statement.

There are several ways that frozenness can be inferred. Frozenness can be directly inferred after a single-assignment variable is assigned or within a *wait* statement for any variable. Alias congruence relations from VN adds an additional way of inferring frozenness: if a variable aliases a frozen variable, it too is frozen. Finaly, data dependency analysis also allows frozenness to be inferred in more situations. For example, if I-var `x` is the output of an operation `x = f(y)`, then `y` must be frozen inside `wait(x) { ... }`. This analysis uses a dependency graph with edges from operation outputs to inputs (e.g., from `x` to `y`) that can be traversed when checking if `x` is frozen.

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:

```
1  () @main () {
2    declare float* a, float* b, float* c, float r, void *t0
3
4    //  Print c once assigned
5    CallForeignAsync trace [ t0 ] [ c ]
6    AsyncOp PlusFloat [ c ] [ a b ]
7    Store [ a ] [ 1 ]
8    CallForeign rand [ r ] [ ]
9    Store [ b ] [ r ]
10 }
```

Figure 5.10: Intermediate representation with instructions in reverse dataflow order.

In this example, multiple instructions reading variables occur after the corresponding instructions that write them. For example, `AsyncOp PlusFloat` on line 6 takes `a` and `b` as inputs; these are produced on lines 7 and 9. 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 `PlusFloat` before it is assigned by `Store`.

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 the code shown in Figure 5.11.

```
1 () @main () {
2   declare float* a, float* b, float* c, float r, void *t0
3
4   Store [ a ] [ 1 ]
5   CallForeign rand [ r ] [ ]
6   Store [ b ] [ r ]
7   AsyncOp PlusFloat [ c ] [ a b ]
8   CallForeignAsync trace [ t0 ] [ c ]
9 }
```

Figure 5.11: Intermediate representation with instructions reordered into dataflow order.

The frozen variable analysis and value numbering passes can then optimize it to the simpler code in Figure 5.12 that executes in a single task with reduced runtime performance overhead.

```
1 () @main () {
2   declare float c, float r, void t0
3
4   CallForeign rand [ r ] [ ]
5   LocalOp PlusFloat [ c ] [ 1 r ]
6   CallExtLocal trace [ t0 ] [ c ]
7 }
```

Figure 5.12: 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 invalid reordering. Forward edges (for which 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, whereby 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 can be 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.

5.5.3 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 transformations must avoid reducing *worthwhile parallelism* that has granularity to justify task creation overhead. Two properties of each IR instruction described in Section 5.4.2 decide whether a transformation reduces worthwhile parallelism: whether it is *long running* (i.e., its ⟨*exec-target*⟩ is a work context) and whether it is *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. The execution target specified by the instruction is respected by copying it to the execution target field of the wait.

This optimization pass inlines all asynchronous instructions with synchronous equivalents at an intermediate stage in the optimization process, so that other optimizations can be applied to the more compact version with asynchronous instructions and the expanded version with wait statements containing synchronous instructions.

Task Coalescing

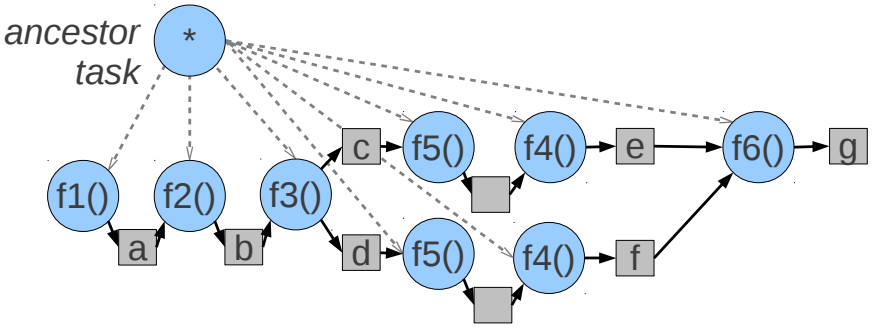
Task coalescing is a family of techniques that reconfigure the IR task structure. One effective technique, which we call *task pushdown*, is to resolve data dependencies between tasks by relocating statements, such as wait statements and data-dependent IR instructions, to descendant blocks in the IR tree where input variables are assigned. This can enable the sequence of transformations in Figure 5.13c, in which VN, FVA, and DCE eliminate shared data items, completing conversion of data dependencies 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

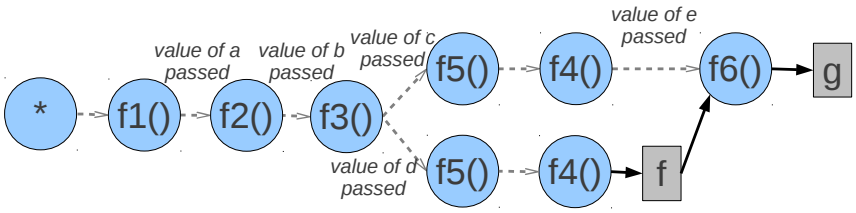
Another optimization is *pipeline fusion*, illustrated in Figure 5.13d. A commonly occurring pattern is a sequentially dependent set of function calls: a "pipeline." We can avoid runtime task dispatch overhead and data transfer without any reduction in parallelism by fusing a pipeline into a single task. For tasks with short duration or large input/output data, this

```
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);
```

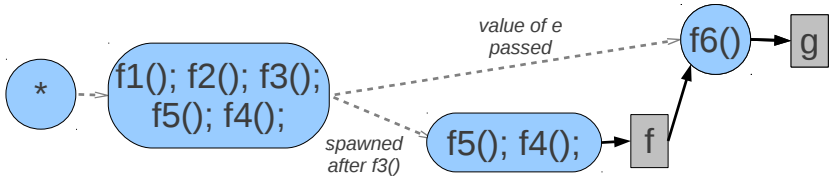
(a) Swift/T code fragment



(b) Unoptimized version, passing data as shared data and performing 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 5.13: Traces of execution showing optimization of task and data dependencies in a Swift/T code fragment. Circles are tasks and squares are data.

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 streaming languages [46], which is similar in concept but not 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.

5.6 Compiler Postprocessing

This section covers the postprocessing phase of the compiler in which STC augments IR-1 with the following additional information to form IR-2:

- Continuations where data must be passed from a parent to a child task.
- Places where reference counts must be incremented or decremented to correctly implement for memory management and freezing.

5.6.1 Memory Management and Freezing Optimizations

Two postoptimization passes over the IR add all necessary refcount operations to implement Swift/T with the reference counting support provided by the runtime (Section 4.3.1). 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.

A naïve reference counting strategy is to update reference counts every time a reference is passed into an instruction or goes out of scope. However, this strategy leads to very inefficient code: 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 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 increments and decrements at the start and end of the block, respectively, thus ensuring that refcounts are not reduced to zero too early during execution.

This framework supports several optimizations. *Merging* increments/decrements 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 increments/decrements can happen, for example, when an increment for a reference passed to a single child task cancels out a decrement for the variable going out of scope in the parent. This transformation is subject to a pass over the block to verify that the reference is not used after being handed to the child. Refcount increments or decrements 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 increments/decrements can be *hoisted* to the parent, subject to conditions: the increment/decrement is not in a loop; if in a conditional, the increment/decrement must occur on all branches; and if a decrement, 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 increments/decrements 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.

5.6.2 Extensions to IR for Variable Passing and Reference Counting

For implementation and optimization of reference counting and variable passing, addition information is added to IR-1 to produce IR-2. Specifically, the following additional properties

and fields are defined in the IR-2 tree:

- The variables for each block have a field that specifies the initial read and write refcount values that are set upon variable creation at runtime. The default value is one.
- Each instruction has *in* and *out* refcounts, if the instruction takes ownership of references or gives ownership of references to the caller respectively.
- Control flow structures have variable passing modes: synchronous control flow structures automatically have variables passed in, while asynchronous control flow structures require that the control flow structure have a list of variables used by inner blocks so that code can be generated to correctly pass the data from parent to child task and manage refcounts accordingly.
- Foreach loops can be annotated with variable READ/WRITE increments/decrements for before/after the loop executes so that reference counts can be incremented in bulk, typically by a multiple of the number of loop iterations.

5.7 Code Generation

STC generates code in the Tcl scripting language that calls into the C functions implementing the distributed runtime. 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 situation may change in future with finer-grained parallelism. For that reason, STC has a modular code generator and is retargetable.

5.8 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.

ReduceTree 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 [77]. 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 5.3 that executed a single function call to compute each grid cell, with runtime following a log-normal distribution with mean 5ms.

Finally, **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 the first four benchmarks as C programs that directly use the **ADLB** [68] 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: **O0**: naïve compilation strategy with no optimization; **O1**: basic redundancy-reducing optimizations, namely, value numbering, alias propagation, constant folding, dead code elimination, loop fusion, frozen variable analysis, and refcount optimizations; **O2**: more aggressive optimizations, namely, asynchronous op inlining, task coalescing, arg localization, and hoisting; and **O3**: the remaining optimizations, namely, function inlining, pipeline fusion, loop unrolling, and instruction reordering. Automatic memory management was enabled in all cases.

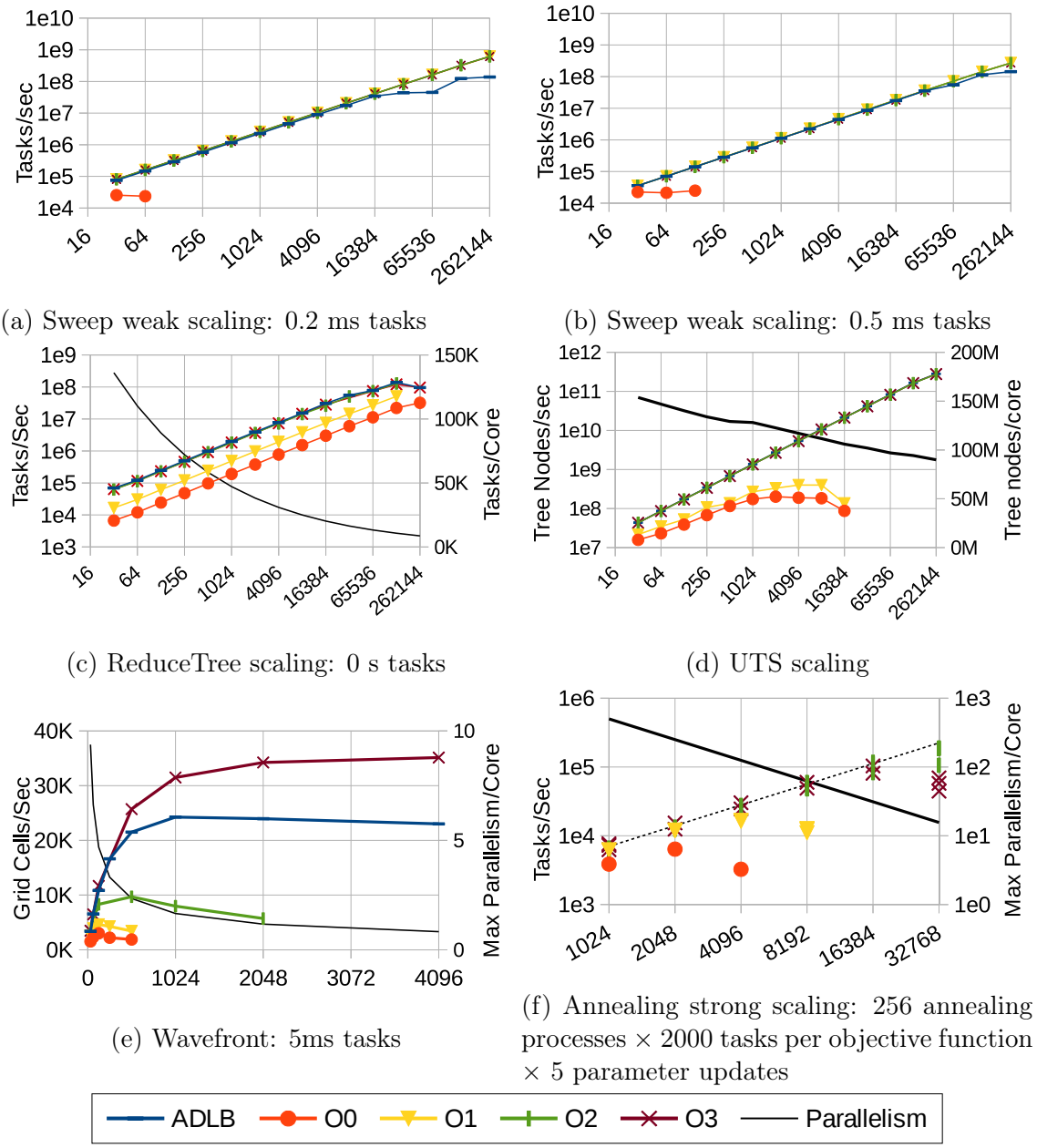
We expect that the O0 baseline configuration will perform poorly because of the nature of the Swift language, whereby language features such as implicit parallelism, transparent data movement, and monotonic data structures do not have general-purpose implementations that map efficiently to lower-level 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.

5.8.1 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[1] was used for the experiments. We have made the benchmark source code publically available[2], with the exception of our collaborators' annealing code. We ran the benchmarks on the Cray XE6 nodes of the Blue Waters supercomputer [73], 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 5.14 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 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 5.15 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.

1. `http://swift-lang.org/Swift-T`
2. `https://github.com/swift-lang/exm-stc/tree/master/bench/suite`



(a) Sweep weak scaling: 0.2 ms tasks

(b) Sweep weak scaling: 0.5 ms tasks

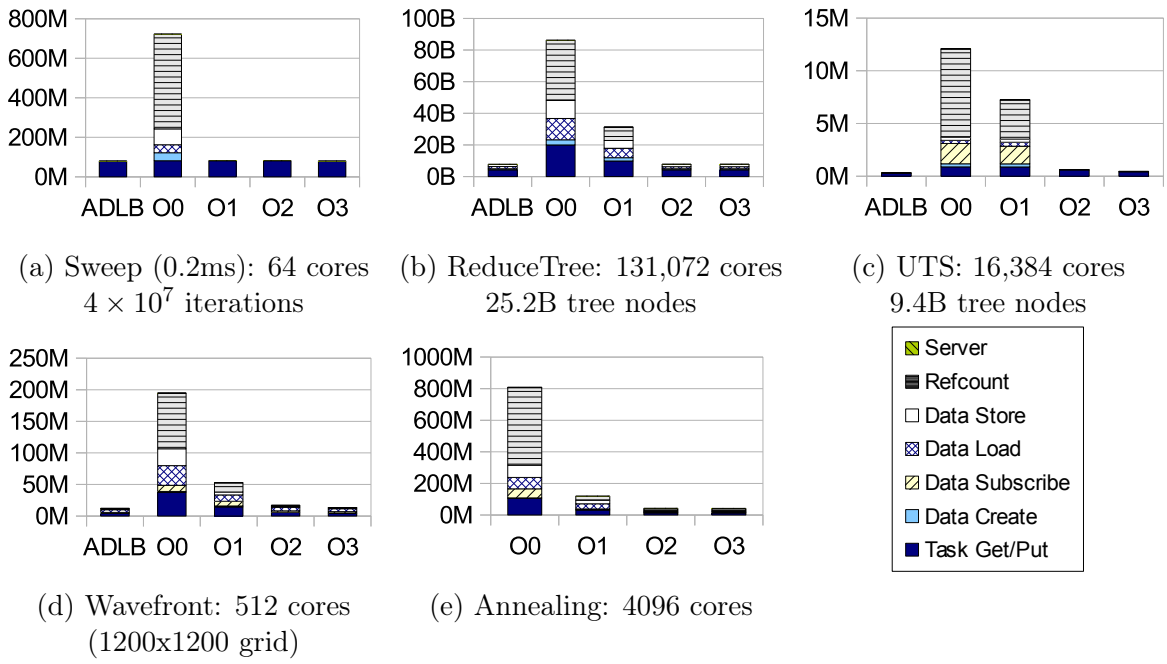
(c) ReduceTree scaling: 0 s tasks

(d) UTS scaling

(e) Wavefront: 5ms tasks

(f) Annealing strong scaling: 256 annealing processes × 2000 tasks per objective function × 5 parameter updates

Figure 5.14: 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.



(a) Sweep (0.2ms): 64 cores 4×10^7 iterations

(b) ReduceTree: 131,072 cores 25.2B tree nodes

(c) UTS: 16,384 cores 9.4B tree nodes

(d) Wavefront: 512 cores (1200x1200 grid)

(e) Annealing: 4096 cores

Figure 5.15: 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.

5.8.2 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 5.14 with Figure 5.15, we see that reduction in operation counts leads directly to application speedup. This result means that the 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

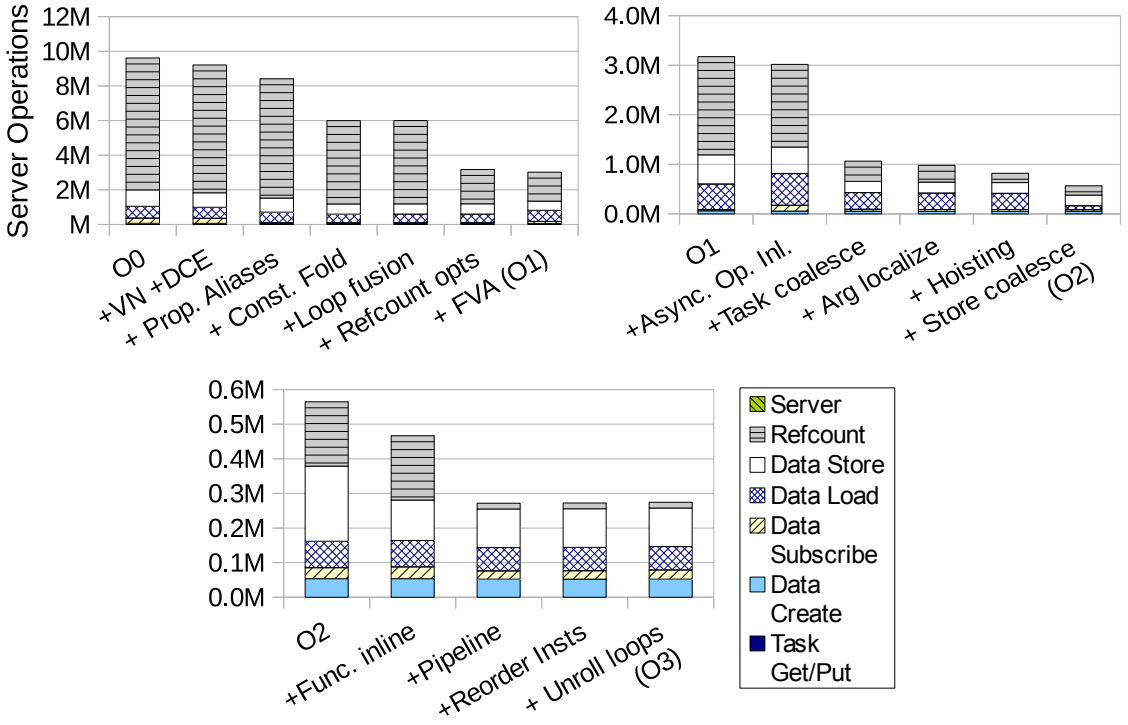


Figure 5.16: Runtime operation counts for Annealing, measured in thousands of operations. Each bar includes previous optimizations, so the difference between adjacent bars shows the incremental impact of each optimization pass.

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 [107].

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 5.14f 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.

5.8.3 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 individually disable optimizations to see how critical they are.

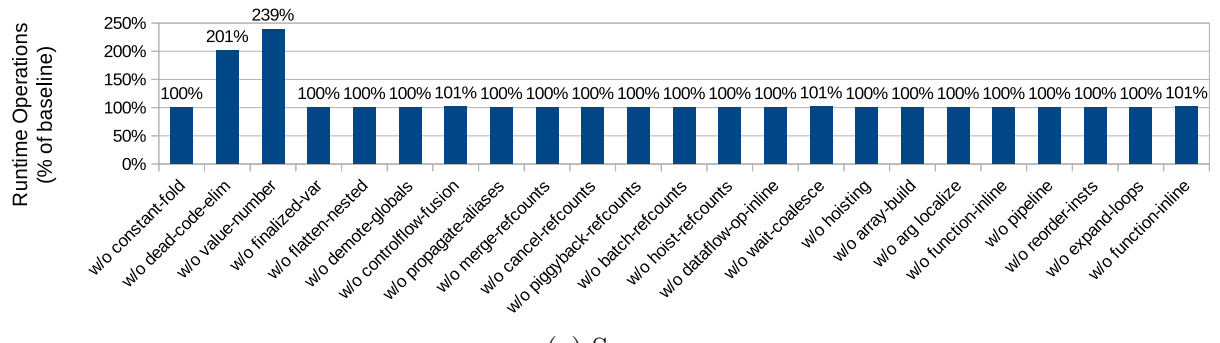
Annealing Incremental Contributions

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 5.16 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.

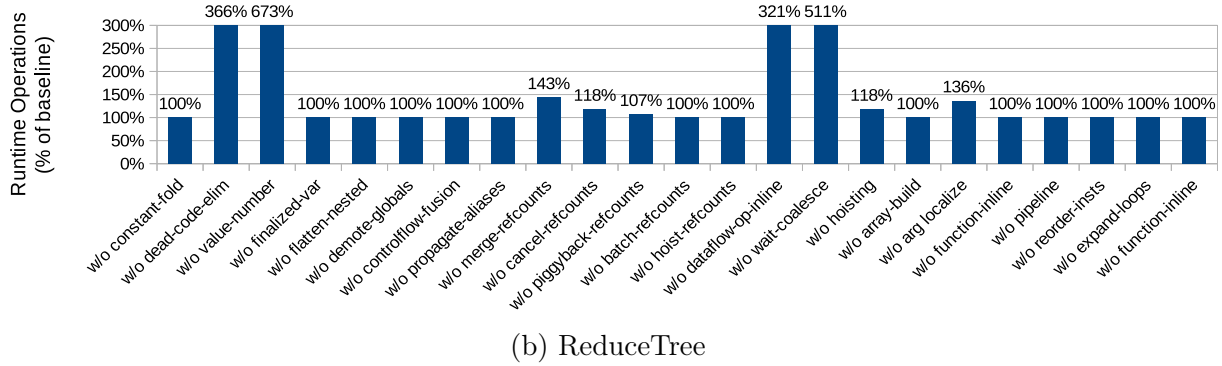
In other benchmarks, different optimizations proved to be critical. In UTS, the task coalescing optimization was able to transform the dataflow logic of the UTS tree search procedure into purely recursive function calls with all intermediate data passed directly to child tasks. Wavefront benefited greatly from loop unrolling, which allowed loads of neighboring array cells to be shared by multiple loop iterations.

Take-one Experiment

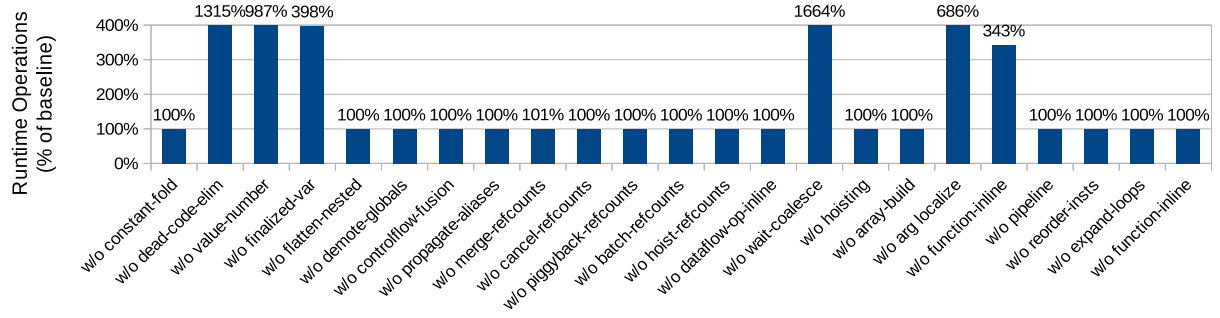
It is also helpful to understand which optimizations are most critical to the results – that is, which have the biggest impact if removed. In some cases, optimizations can have a big impact, but be somewhat redundant when used in combination with other optimizations. In other cases, optimizations are difficult to substitute. To understand such effects, we ran each benchmark with all optimizations enabled save one and compared operation counts to a baseline in which all optimizations were enabled. Figure 5.17 shows the results. We see that DCE and VN were both 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 made some contribution in at least one benchmark. The more



(a) Sweep



(b) ReduceTree

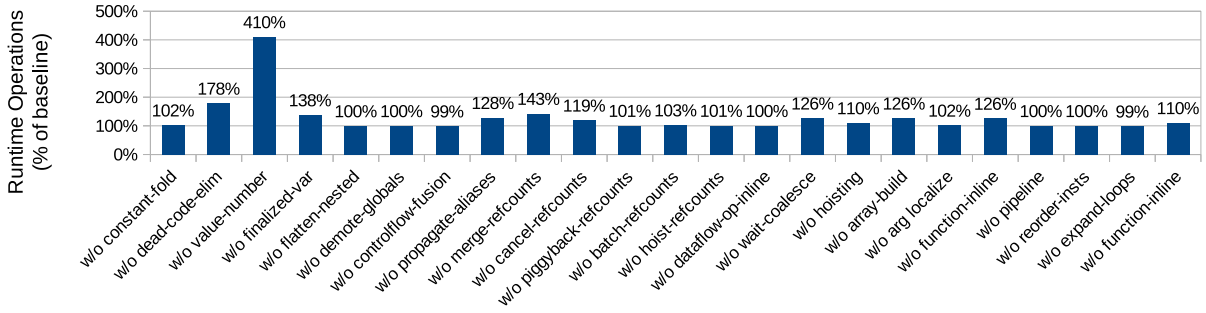


(c) UTS

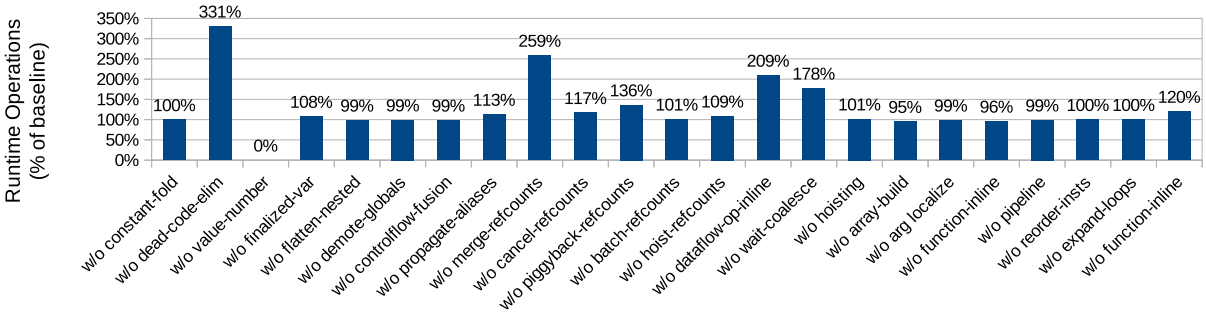
Continued on next page.

Figure 5.17: 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 led to a compile error. Note: the Wavefront benchmark without value-numbering failed because of a bug in the version of the compiler used that could not be fixed by the time of writing.

Figure 5.17: *Continued from previous page.*



(d) Annealing



(e) Wavefront

complex annealing benchmark benefited from the widest range of optimizations.

5.8.4 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 by our optimizations. We ran scaled-down instances of the benchmarks under multiple configurations: **Off**, where read reference counts are not tracked and memory is never freed; **Unopt**, where all reference counting optimizations are disabled; and different levels in which reference counting optimizations are incrementally enabled. All other optimizations were enabled; hence, for some benchmarks, shared data was optimized out in all cases.

Figure 5.18 shows that the reference counting optimizations are effective: the overhead

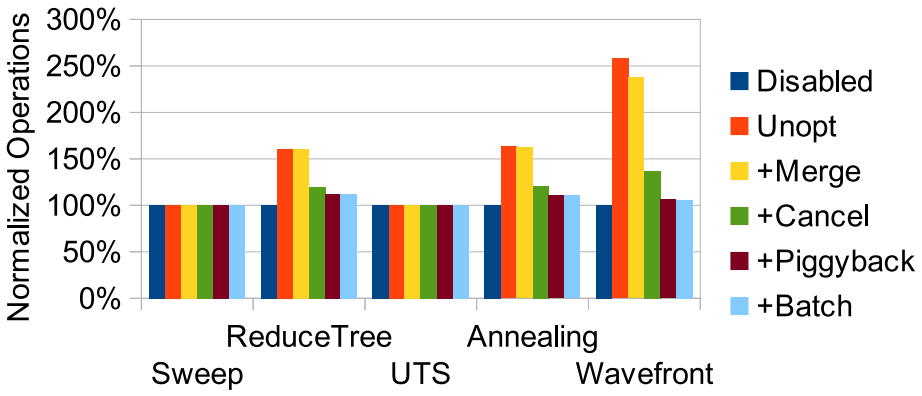


Figure 5.18: Impact of unoptimized and optimized reference counting for memory management, normalized to the total count of runtime operations without memory management. Each bar includes previous optimizations.

measured in operations due to automatic memory management is at most 12.2% after optimization. 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 for a given scalability level. This granularity increase is small enough that automatic memory management in Swift is viable for large-scale computing. This result 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.

5.8.5 Compilation Time

In order for an optimizing compiler to be useful in practice, the compilation process must be fast enough that programs can quickly 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

Table 5.1: Compile time in seconds for STC at different optimization levels.

	O0	O1	O2	O3
Sweep	0.8	1.1	0.9	2.3
ReduceTree	0.8	1.1	1.2	1.2
UTS	0.9	1.5	1.5	1.4
Annealing	2.1	3.6	3.9	5.5
Wavefront	0.9	1.4	1.4	3.3

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 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.

5.9 Related Work on Compiler Optimization

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

Hardware dataflow-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 dataflow languages with arrays: SISAL [98] and Id [110]. 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 [81]. Optimizations have been implemented that specialize communication operations using analysis of communication patterns in Concurrent ML [93]

Research on the PGAS family of programming languages [2, 12, 107] 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 [123].

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 [124]. Arandi et al. show benefits from compiler-assisted resolution of task data dependencies with a shared-memory runtime [5]. Jagannathan's communication-passing transformation [52] 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 [53, 83].

Distributed computing researchers have explored optimization of distributed *workflows* expressed as tasks with data dependencies. This work has focused on scheduling the workflows with constraints of resource availability and data movement cost [101, 122, 21], 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.

5.10 Future Work

There are many opportunities to improve the STC compiler and leverage the STC compiler to solve problems.

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, in both compiler optimization and combined runtime/compiler approaches. For example, opportunities exist to implement further techniques from the extensive compiler optimization literature. More sophisticated control and dataflow analyses could bring further incremental improvements to many applications, while more specialized techniques, such as for affine nested loops [13], 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 [112] 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.

CHAPTER 6

CONCLUSION

In the thesis I proposed a hierarchical programming model for massively-parallel computing that uses a high-level implicitly parallel language – Swift – to orchestrate computational tasks implemented in a range of other programming languages. This proposed approach is a promising direction for addressing future systems challenges of unreliability and heterogeneity while making it substantially easier for non-expert programmers to construct scalable parallel applications.

I defined a formal execution model to provide a firm foundation for the semantics of the Swift programming model, which, prior to this work, was only informally and imprecisely defined. I proved that the execution model was deterministic and showed how it was general and expressive enough to capture the control-flow constructs and data types previously offered by the Swift language, along with new extensions.

I demonstrated that execution model could be implemented in practice as a distributed runtime system – Turbine – for large-scale distributed memory systems. This implementation presents many challenges, some of which – scalable synchronization, task matching, and task distribution – I described and addressed with algorithms and data structures. These new approaches enabled great improvements in runtime performance and made the programming model performant and scalable enough to be attractive for many applications that fit the execution model.

Finally, I demonstrated that compiler optimization techniques were both necessary and sufficient to efficient translate the high-level Swift language to execute on the lower-level execution model – as implemented by the Turbine runtime system. I 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. These techniques were applied to Swift/T, but are not necessarily specific to Swift/T: I

expect that they could also be applied to other task-parallel programming models. The performance results we obtained support two major claims: that applying a wide spectrum of compiler optimization techniques can greatly improve performance.

Overall, this dissertation presents strong evidence that that a combination of runtime algorithms and compiler techniques can enable 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

It is impossible to enumerate all of the possible ways in which the systems presented in this thesis could be improved: Swift/T has enough features and good enough performance that implementing a range of applications in Swift/T is a viable and attractive option. We are still far, though, from exhausting all of the opportunities to improve performance, add features, and address the future challenges inherent in the next generation of Exascale computing. I already described future work pertaining directly to each chapter in the chapters themselves: the execution model can be extended, runtime performance can be improved, and the compiler infrastructure can be leveraged for further optimizations. There remain a

couple more research directions of particular interest to discuss.

Fault tolerance remains a major issue, particular with Exascale computer systems on the horizon. We believe that an execution model with discrete tasks and explicit data dependencies provides many opportunities to address fault tolerance by replaying tasks and regenerating lost data. Preliminary work has been conducted on logging-based approaches to checkpointing and recovery, but a huge spectrum of alternative approaches is possible, including replication, transactions, and so on.

Applying the work to alternative distributed computing infrastructures such as clouds also presents a range of challenges, yet would be enormously useful. The current runtime system has been engineered to run on systems where the networks have low and predictable latencies. As a result, the performance of the runtime system is quite sensitive to network latency, despite the fact that asynchronous task parallelism is excellent at hiding other kinds of latency. Reengineering the runtime system to be less sensitive to latency would be a promising direction.

Finally, applications work is of great interest: attempting to apply Swift/T to new and different problems will reveal further strengths, weaknesses, and opportunities.

Appendices

APPENDIX A

PROOFS OF EXECUTION MODEL DETERMINISM

In this appendix we will restate the main theorems and provide proofs along with the required lemmas.

Theorem 3.5.1 (Execution is Deterministic – Normal Termination)**.** *Execution is deterministic if the program terminates in a non-**error** state, all data store types are lattice types, and tasks wait before reading. Given any two execution traces with the same initial state* $\langle D_0, W_0, W_0^{fin}\rangle$*, then if the first trace terminates with a valid state* $\langle D_{n+1}, W_{n+1}, W_{n+1}^{fin}\rangle$*, 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,* $S_{n+1} \sim S'_{m+1}$ *and* $n = m$*.*

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: $D_{n+1} \overset{DS}{\approx} D'_{m+1}$, $W_{n+1}^{fin} \neq W'^{fin}_{m+1}$, or $W_{n+1} \neq W'_{m+1}$[1]. The first subcase in fact subsumes the remaining two cases. In the second subcase, $D_{n+1} \overset{DS}{\approx} D'_{m+1}$ follows from Lemma A.8. In the third subcase, we assume $W_{n+1}^{fin} = W'^{fin}_{m+1}$ (otherwise it is subsumed by second subcase). Therefore $W_{n+1}^{pending} \neq W'^{pending}_{m+1}$. By Lemma A.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 A.1,

1. Note that $n \neq m \Rightarrow W_{n+1}^{fin} \neq W'^{fin}_{m+1}$

they both have bounded waiting. Lemma A.9 therefore implies that the initial states of the two traces are different: a contradiction for case 1.

In case 2, by Lemma A.10, 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 A.8 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, Lemma A.9 implies that that the initial states of the traces are different: a contradiction for cases 2 and 3. □

Theorem 3.5.2 (Non-terminating Task Functions Cause Divergence on Error)**.** *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. □

Theorem 3.5.3 (Error Non-determinism Given Arbitrary Scheduling)**.** *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 $X \leftarrow 0$

Task B: assigns $X \leftarrow 1$

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 $\langle B, C, C, C, C, C, C, ... \rangle$. □

Theorem 3.5.4 (Error Determinism Given Bounded Waiting)**.** *If a trace* $\langle w_0, w_1, ..., w_n \rangle$ *terminates in* ***error****, then another trace* $\langle w_0, w'_1, ... \rangle$ *with the same initial state and bounded waiting also terminates in* ***error****.*

Proof. Theorem 3.5.1 implies that the second trace does not terminate successfully (otherwise the first trace would have terminated successfully). Therefore we only need to rule out the possibility that the second trace does not terminate. Let us suppose, for the sake of contradiction, that this is the case.

By Lemma A.10, there is a task w_i where the first trace is divergent from the second. The bounded waiting policy guarantees bounded waiting in the second trace by Corollary A.3, so by Lemma A.9 the initial states were different: a contradiction. □

Lemma A.1 (Store Creates and Updates Commute)**.** *Given that all data store types are lattice types, i.e., for every* t *provided to dscreate,* $t \in \mathcal{T}_{lat}$*, then dscreate and dsupdate operations commute: applying the operations in any order will give an equivalent 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 *dscreate* operations, then the final result will be $\top$. Either the argument to one *dscreate* is $T_\perp$, which results in $\top$, or the second *dscreate* fails, resulting in $\top$. Once the data store state is $\top$, both *dscreate* and *dsupdate* always result in $\top$. Therefore we only need consider sets of operations with a single *dscreate*.

Any *dsupdate*s operations beforethe first *dscreate* are saved, so a sequence with a single *dscreate* and zero or more *dsupdate*s will be reordered so that the create is applied first to the data structure. Therefore, the only differences between sequences is the order in which updates are applied to the data structure. The commutativity of *dsupdate* therefore follows from the commutativity of update (Definition 3.3.17).

□

Lemma A.2 (Reads of Frozen Paths in Data Store)**.** *Given that all data store types are valid types, i.e., for every t provided to dscreate, $t \in \mathcal{T}_{valid}$, then after dsfrozen has returned* TRUE *for a data store path, it remains frozen and all reads return the same value,* unless *an invalid update is applied, which makes the entire data store invalid. $\forall p \in \mathcal{P}$, $dsfrozen(D, p) \Rightarrow \forall D' \in dsupdatecreate_t^+(r)$:* [2]

$$dsfrozen_t(D', p) \ \wedge \ (D' = \top \vee dsread_t(D', p) = dsread_t(D, p))$$

Proof. This follows from the similar property (Property 3.3.8) that applies to standalone lattice data types and the construction of the data store functions. □

Lemma A.3 (Consistent Reads within Trace)**.** *Consider an execution trace $\langle w_0, w_1, ..., w_n \rangle$ in which Property 3.5.1 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, $\forall p \in P_n^{read}$, $\exists v$, $\forall 0 \leq i \leq n$, $p \notin P_i^{read} \vee v = dsread(D_i, p)$.*

Proof. Let us assume for the purpose of induction that for all paths $p \in P_{i-1}^{read}$,

- $dsfrozen(D_{i-1}, p)$ and
- $\exists v$, $\forall 0 \leq j \leq i-1$, $p \notin P_j^{read} \vee v = dsread(D_j, p)$

2. $dsupdatecreate^+$ is the transitive closure of *dscreate* and *dsupdate* operations

$i = 0$ is the base case for the induction. We only need consider paths read by w_0. All paths read are frozen by Property 3.5.1. There were no previous reads, so any values read by w_0 are consistent.

First, consider the first part of inductive assumption for i. All paths in P_{i-1}^{read} are frozen by the inductive assumption. By Property 3.5.1, all paths read by task w_i were frozen in state D_i. Therefore $\forall p \in P_i^{read}$, $dsfrozen(D_i, p)$.

Next, consider the second part of inductive assumption for i. For all paths read by w_i, either $p \notin P_{i-1}^{read}$, or $p \in P_{i-1}^{read}$. In the first case, $v = dsread(D_i, p)$ satisfies the condition because there were no previous values read. 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 A.2, $dsread(D_i, p) = v$ and the condition is satisfied. □

Definition A.1 (Consistent Value Map)**.** Let $V : \mathcal{P} \to \mathcal{V}$ 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 A.3, 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 A.2 (Frozen Path Set)**.** Let $F \subseteq \mathcal{P}$ be the frozen path set for a trace $\langle w_1, w_2, ..., w_n \rangle$: the set of all paths that were frozen at some point in the trace. Because of Lemma A.2, once a path is frozen it is always frozen. Therefore we can define it as:

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

Lemma A.4 (Common Ancestor Task)**.** *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. □

Lemma A.5 (First Differing Ancestor)**.** *Consider a task* w_i *that 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 exists a the first differing ancestor task* w_l *(*$k < l \leq i$*) that is a child of* w_k*, where* $i = l$ *or* w_l *is an ancestor of* w_i*.* w_l *is not in the second trace: either it is in* $W'^{pending}_m$ *or it was not 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 $W'^{pending}_m$. If $w_l \notin W'_m$, it was never spawned by w_k. □

Definition A.3 (Bounded Waiting)**.** A trace has bounded waiting if, given any intermediate state in the traces, there is a bound c on the number of steps until all the tasks that were runnable in that state are executed. Formally, $\exists c$ such that $w_{|n|} \in W_i^{runnable} \wedge w_{|m|} \in W_i^{fin}$ $\Rightarrow \exists j \leq i + c,\ w_{|n+m|} \in W_j^{fin}$.

The multiplicities n and m of elements in the multiset come up in this definition to correctly deal with the presence of duplicate tasks: we need to ensure that all $n - m$ unexecuted duplicates are executed within c steps.

Corollary A.1 (Normal Termination Implies Bounded Waiting)**.** *Any trace that terminates in a valid state after n steps has bounded waiting, because $W_{n+1}^{runnable} = \emptyset$ and therefore no task waited for more than n steps.*

Corollary A.2 (No Bounded Waiting on Error)**.** *Any trace that terminates in* ***error*** *does not have bounded waiting because the error task is never added to W^{fin}.*

Corollary A.3 (Bounded Waiting Policy Implies Bounded Waiting)**.** *A trace generated with a bounded waiting task selection policy (Definition 3.5.6) has bounded waiting (Definition A.3).*

Definition A.4 (Trace Divergence)**.** 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 value maps 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., $\exists p \in \textit{read-data}_i$ such that $V(p) \neq V'(p)$.

Corollary A.4 (Divergence on Differing Outputs)**.** *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 Theorem 3.4.1, w must have read different values in the two traces. □

Lemma A.6 (Differing Values Implies Earlier Divergence)**.** *Given two traces, $\langle w_0, w_1, ... \rangle$ and $\langle w_0, w'_1, ... \rangle$ 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. 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 A.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. By Property 3.5.1, the path p was frozen at steps i and j in the two traces, respectively. 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 A.2. Neither of these conditions is true: a contradiction, therefore $O \not\subset O'$.

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. □

Lemma A.7 (Different Frozen Sets Implies Earlier Divergence)**.** *Given two traces,* $\langle w_0, w_1, ...\rangle$ *and* $\langle w_0, w'_1, ...\rangle$ *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 the first trace.**
  - **Case 1a: k was not created by the same task in both traces.** We know that there is some $j < i$ for which w_j in the first trace output a *dscreate* call for k. If this task is not in the second trace, then the first trace is divergent at w_j. If it is in second 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 have bounded waiting.
  - **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 did not have in the first trace, or a data structure holds a write permission for some key that it did not have 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 propagate 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 the first trace.** The path k was frozen, but not with the dsfreeze operation. Therefore, then the path was frozen in the first trace as a result of the set of operations applied, O. *dsfrozen* has the same properties as *dsread* w.r.t. commutativity of *dsupdate* and *dscreate*, so the same argument used in Lemma A.6 can be applied to prove that the first trace is divergent at w_j with $j < i$.

□

Lemma A.8 (Missing Task Implies Earlier Divergence)**.** *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 A.4, there is a common ancestor task w_k of w_i and by Lemma A.5, a first differing ancestor w_l. This lemma gives us two cases to consider:

- w_l was spawned but never ran: $\exists n$, $\forall m \geq n$, $w_l \in W_m'^{pending}$. Because we assumed bounded waiting, the task cannot be in $W_m'^{runnable}$ indefinitely. Therefore there is a path $p \in \textit{wait-data}_l$ where $p \notin F'$. But $p \in F$ because w_l ran in the first trace, so by Lemma A.7, 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 Theorem 3.4.1, a read to path p by w_k yielded a different value in the two traces. Therefore the first trace is divergent from the second at w_k with $k < i$.

□

Lemma A.9 (Divergence Implies Different Initial States)**.** *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 result 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 A.8, 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 A.6, 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. □

Lemma A.10 (Error Termination and Divergence)**.** *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. We can divide the reasons why w_n failed into two cases:

- w_n did something that is invalid in isolation, e.g., wrote something for which it had no permission, read or wrote a bad path, updated with a bad value, or created a bad child task.
- w_n output a *dscreate*/*dsupdate* that conflicted with one or more *dscreates*/*dsupdates* output by another tasks (or set of other tasks). The order in which the *dscreate*/*dsupdate* operations were applied does not matter because of Lemma A.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 (the first case above).
- w_n produced the same outputs, but another task w_i $(0 \leq i < n)$ in the first trace was not in the second trace or produced different outputs (the second case above).

In all of these cases a task w_i produced different outputs, and therefore was divergent by Corollary A.4 or a task w_i was not present and therefore is divergent by definition. □

APPENDIX B

TCL FOR HIGH PERFORMANCE COMPUTING

Using interpreted languages such as Tcl and Python on High-performance computing systems has associated challenges that arise from the way interpreters for these languages dynamically load code at runtime. In many cases, the interpreters are implemented in a way that many files are opened at runtime even for the most minimal applications. The dynamically loaded code falls into three categories:

- 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 dynamic linker when the interpreter is launched.
- Shared libraries dynamically loaded by the language interpreter – typically programming language modules that are fully or partially implemented with compiled code in a language like C.

In the case of Python – one of the most popular interpreted languages used in HPC – running a typical application requires thousands of filesystem metadata operations – mainly `open` and `stat` operations for many small library or source files [126]. Even a minimal Tcl script using our runtime libraries attempts to open 209 files at startup: 112 shared libraries, 82 Tcl source files, and miscelleneous configuration and system files. This behavior is inefficient but even if the files are on the local file system of the machine running the interpreter: it will slow down startup of the application.

On typical HPC systems with parallel file systems separate from 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 coordinated reads and writes on a relatively small number of files. They are not optimized for supporting high throughput of metadata operations such as opens

and closes. On some parallel file systems, the consequence of this behavior can be that applications using interpreted languages take many minutes or even hours to start execution of large runs with several thousand processes or more [126].

Multiple approaches for more efficiently loading many shared libraries at application startup have been proposed in the literature [34, 42, 126]. These solutions all involve intercepting file system operations for shared libraries before they go the the shared parallel file system and instead serving them with an alternative mechanism that is more scalable. Various approaches are possible. For example, libraries can be cached on intermediate nodes or distributed through a peer-to-peer or tree-like overlay network. Frings et al. provide a good overview of the various approaches [42]. 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 without any coordination between processes.

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 approach enables the executable to be efficiently shipped to compute nodes during the application startup process and avoids the need to load any more 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 statically linkable compiled libraries used by the program

In the best case where all code can be linked into the executable, this approach allows the Swift/T application to be a fully self-contained executable. In the worst case, where compiled libraries cannot be statically linked, it avoids loading many Tcl source files and Turbine/Tcl shared libraries.

APPENDIX C

INTERMEDIATE REPRESENTATION

```
INTERPRET(program, threadnum)
 1  // Entry function is run only once
 2  if threadnum == 0
 3     entry = program.fns['__entry']
 4     EXECBLOCK(INITENV(), entry.block)
 5  // Worker loop: repeatedly execute tasks
 6  while HAVEWORK()
 7     task = GETTASK()
 8     if task ≠ NULL
 9        EXECBLOCK(task.env, task.block)
10     foreach c ∈ EXECCOMPLETED()
11        env' = CHILDENV(c.env, r.var ↦
                     r.val | r ∈ c.results)
12        EXECBLOCK(env', c.continuation)

EXECBLOCK(env, block)
 1  // Allocate all variables for block
 2  foreach var ∈ block.vars
 3     if var.storage == SHARED
 4        BIND(env, var ↦ ⟨DSCREATE(
              UNIQUEID(), var.type), ⟨⟩⟩)
 5  // Execute sequential statements
 6  foreach stmt ∈ block.statements
 7     EXECSTATEMENT(env, stmt)
 8  // Execute sync. or async. continuations
 9  foreach cont ∈ block.continuations
10     EXECCONTINUATION(env, cont)

EXECSTATEMENT(env, statement)
 1  switch stmt
 2  case IF condition then else
 3     if env[condition]
 4        EXECBLOCK(CHILDENV(env), then)
 5     else
 6        EXECBLOCK(CHILDENV(env), else)
 7  case SWITCH condition cases default
 8     if cases[env[condition]] ≠ NULL
 9        EXECBLOCK(CHILDENV(env),
                cases[env[condition]])
10     else
11        EXECBLOCK(CHILDENV(env), default)
12  default
13     EXECINSTRUCTION(env, statement)

EXECCONTINUATION(env, continuation)
 1  switch (continuation)
 2  case WAIT waitvars target waitmode block
 3     vars = {wv.var | wv ∈ waitvars}
 4     PUTTASK(vars, CHILDENV(env), body)
 5  case FOREACH collection val-var key-var body
 6     // Iterate over array keys and values
 7     foreach (k, v) ∈ env[collection]
 8        env' = CHILDENV(env, val-var ↦ v,
                     key-var ↦ k)
 9        PUTTASK(∅, env', block)
10  case FOREACH collection val-var body
11     // Iterate over collection values only
12     foreach v ∈ VALUES(env[collection])
13        env' = CHILDENV(env, val-var ↦ v)
14        PUTTASK(∅, env', block)
15  case FOREACH lo to hi var body
16     // Iterate from lo to hi inclusive
17     for i = env[lo] to env[hi]
18        env' = CHILDENV(env, var ↦ i)
19        PUTTASK(∅, env', body)
20  case LOOP itervars body
21     // Start first iteration of loop
22     // LOOPCONTINUE inst. starts next iter.
23     deps = {env[iv.var] | iv ∈ itervars∧
                     iv.blocking}
24     env' = CHILDENV(env, iv.var ↦ iv.init |
                iv ∈ itervars)
25     PUTTASK(deps, env', body)
26  case ASYNCEXEC executor args cont errcont
27     vals = ⟨env[arg] | arg ∈ args⟩
28     EXECRUN(executor, vals, env cont, errcont)
```

INITENV and CHILDENV create empty and child environments, optionally with some mappings
BIND binds a variable in an existing environment
env[*name*] evaluates a var or arg in an environment
PUTTASK and GETTASK operations add and remove tasks from the global task queue
EXECRUN enqueues work with an executor
EXECCOMPLETE returns completed executor work

Figure C.1: Pseudocode for a parallel interpreter for STC IR-1 to illustrate IR-1 semantics.

```
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     vals = EvalBuiltin(builtin-opcode, ⟨env[i] | i ∈ in⟩)
 7     BindAll(env, out, vals)
 8  case AsyncOp builtin-opcode out in
 9     // spawn task to execute async builtin op
10     PutTask({env[i] | i ∈ in}, env, {
11        invals = LoadAll(env, in)
12        outvals = EvalBuiltin(builtin-opcode, invals)
13        StoreAll(env, out, outvals)
14     })
15  case Call fnname out in
16     fn = GetFnDefn(fnname)
17     fnenv = MakeFnEnv(env, fn, out, in)
18     ExecBlock(fnenv, fn.block)
19  case CallAsync fnname out in
20     fn = GetFnDefn(fnname)
21     fnenv = MakeFnEnv(env, fn, out, in)
22     deps = {o.var | o ∈ fn.outargs ∧ o.waitfor}
23     SpawnTask(deps, fnenv, fn.block)
24  case CallForeign fnname out in
25     // Similar to LocalOp, except call foreign function
26  case CallForeignAsync fnname out in
27     // Similar to AsyncOp, except call foreign function
28  case CreateAlias alias var path
29     // Create alias for path within var
30     ⟨key, prefix⟩ = env[var]
31     Bind(env, alias ↦ ⟨key, prefix · env[path]⟩)
32  case Store out path in
33     Store(env[out], env[path], env[in])
34  case Load out in path
35     Bind(env, out ↦ Load(env[in], env[path]))
36  case StoreRecursive out in
37     // Recursively build data store structure
38     // with references according to out.type
39  case LoadRecursive out in
40     val = Load(env[in])
41     // ... then recursively follow references
42     while HasRefs(val)
43        val = LoadRefs(val)
44     return val
45  ... continued on next page ...
```

```
BindAll(env, vars, vals)
1  Bind(env, var ↦ val |
        ⟨var, val⟩ ∈ Zip(vars, vals))

Load(handle, path)
1  // Load value of variable in env
2  // Handles can reference a path
3  ⟨key, pathprefix⟩ = handle
4  return ⟨DSRead(key, prefix · path)⟩

Store(handle, path, val)
1  ⟨key, pathprefix⟩ = handle
2  DSUpdate(key, prefix · path, val)

LoadAll(env, vars)
1  return ⟨Load(env[var], ⟨⟩) |
        var ∈ vars⟩

StoreAll(env, vars, vals)
1  foreach ⟨var, val⟩ ∈ Zip(vars, vals)
2     Store(env[var], ⟨⟩, val)

MakeFnEnv(env, fn, out, in)
1  fnenv = InitEnv()
2  BindAll(fnenv, fn.outargs,
           ⟨env[i] | i ∈ out⟩)
3  BindAll(fnenv, fn.inargs,
           ⟨env[i.var] | i ∈ in⟩)
4  return fnenv
```

Figure C.2: Pseudocode for parallel interpreter for STC IR-1 to illustrate IR-1 semantics. This figure provides pseudocode for selected IR instructions.

Continuation of Figure C.2

```
45  case COPY out inpath in outpath
46    val = LOAD(env[in], env[inpath])
47    STORE(env[out], env[outpath], val)
48  case COPYASYNC out in
49    PUTTASK({env[in]}, env, {
50      val = LOAD(env[in], env[inpath])
51      STORE(env[out], env[outpath], val)
52    })
53  case DEREF out in
54    PUTTASK({env[in]}, env, {
55      derefed = LOAD(env[in])
56      PUTTASK({derefed}, env, {
57        val = LOAD(derefed)
58        STORE(env[out], val)
59      })
60    })
61  case CREATENESTED subarray array path
62    // Atomically set path to nested array,
63    // or return existing array if present
64  case LOOPCONTINUE loop args
65    // Start new loop iteration
66    deps = {env[iv.var] | iv ∈ loop.itervars ∧ iv.blocking}
67    env' = CHILDENV(env.parent, itervars[i].var ↦
          env[args[i]] | i ∈ [1, loop.itervars.length])
68    PUTTASK(deps, env', loop.body)
69  case LOOPBREAK loop
70    // Marker for loop termination
```

APPENDIX D

COMPILER OPTIMIZATION DETAILS

D.1 Ordering of Optimization Passes

Ordering the optimization passes in a compiler is a difficult problem that can significantly affect the compilation time and effectiveness of compiler optimization [72]. Our work has not focused on optimizing this ordering. Currently we order passes optimization passes in a way that works reasonably well in practice, but may be suboptimal, particularly for compilation time, because it runs many passes multiple times. The optimization passes in STC 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 discrepances between the IR emitted by the frontend and the IR processed by the optimizer. Preprocessing has three passes: Unused Function Elimination → Unique Variable Names → Inline Nested Blocks. Unused Function Elimination simply removes any unneeded functions with no call chain from the program entry point to the function. Unique Variable Names transforms 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, but the optimization passes assume that each variable's name is unique within the scope of the 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 heavy lifting of optimization is done in the iterative phase, where a sequence of passes is repeatedly applied, with some modifications. Repeatedly applying 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 major irreversible 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 the transformation.

Some optimizations are not run in each iteration because there is limited benefit to repeated application of the pass. We iterate through the passes 10 times by default. This number could almost certainly be reduced without loss of effectiveness by more careful analysis, ordering, and other tweaks. The sequence of optimization passes applied in iteration i is: Unused Function Elimination ($i = 5$) $\rightarrow$ Function Inlining ($i \in \{0, 3, 8\}$) $\rightarrow$ Store Coalescing ($i \in \{2, 5, 8\}$) $\rightarrow$ Loop Unrolling ($i \in \{2, 5, 8\}$) $\rightarrow$ Loop Invariant Hoisting ($i \in [0, 8]$) $\rightarrow$ Instruction Reordering ($i \in \{1, 3, 5, 7\}$) $\rightarrow$ Alias Propagation ($i \in \{0, 3, 6, 9\}$) $\rightarrow$ Value Number $\rightarrow$ Argument Localization $\rightarrow$ Dead Code Elimination $\rightarrow$ Control-flow Fusion $\rightarrow$ Pipeline Fusion ($i = 7$) $\rightarrow$ Asynchronous Op Inlining ($i = 5$) $\rightarrow$ Task Coalescing.

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 $\rightarrow$ Unused Function Elimination $\rightarrow$ Add Passed Variables $\rightarrow$ Add Reference Counting $\rightarrow$ Fixup Passed Variables

The first two passes simply remove any unused code that is left over from iterative optimization. The next pass add annotations to indicate to the code generator the variables that must be passed to child tasks and whether read or write handles 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, the final fixup pass needs to add additional passed variables needed as arguments to reference counting operations.

REFERENCES

[1] U. A. Acar, A. Chargueraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *Proc. PPoPP '13*, 2013.

[2] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proc. PPoPP '07*, 2007.

[3] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The SprayList: A scalable relaxed priority queue. In *Proc. PPoPP '07*, 2015.

[4] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *Proc. HOTI '10*, Aug 2010.

[5] S. Arandi, G. Michael, P. Evripidou, and C. Kyriacou. Combining compile and run-time dependency resolution in data-driven multithreading. In *Proc. DFM '11*, 2011.

[6] T. G. Armstrong, J. M. Wozniak, W. Michael, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proc. CCGrid '14*, Nov 2014.

[7] T. G. Armstrong, J. M. Wozniak, and M. Wilde. Exploring scientific discovery with large-scale parallel scripting. In *SCALE Challenge, CCGrid '13*, 2013.

[8] T. G. Armstrong, J. M. Wozniak, M. Wilde, K. Maheshwari, D. S. Katz, M. Ripeanu, E. L. Lusk, and I. T. Foster. ExM: High level dataflow programming for extreme-scale systems. In *Proc. HotPar '12*, 2012.

[9] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster. Scheduling many-task workloads on supercomputers: Dealing with trailing tasks. In *Proc. MTAGS '10*, 2010.

[10] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, October 1989.

[11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Conc. Comput.: Pract. Exper.*, 2011.

[12] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimizations for distributed-memory X10 programs. In *Proc. IPDPS '11*, May 2011.

[13] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. PACT '04*, 2004.

[14] D. M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proc. USENIX Tcl/Tk Workshop*, 1996.

[15] A. Bhatele and V. Laxmikant. An evaluative study on the effect of contention on message latencies in large supercomputers. In *Proc. IPDPS '09*, 2009.

[16] R. L. Bocchino. Deterministic Parallel Java. In D. Padua, editor, *Encyclopedia of Parallel Computing*. Springer US, 2011.

[17] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. Technical Report ICL-UT-10-01, U. Tennessee, 2010.

[18] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proc. OSDI '14*, Oct. 2014.

[19] M. Broy and C. Jones, editors. *Linear types can change the world!* North Holland, 1990.

[20] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *Proc. IPDPS '12*, 2012.

[21] P. Bui. *A Compiler Toolchain for Distributed Data-Intensive Scientific Workflows*. PhD thesis, Notre Dame University, 2012.

[22] R. Butler. ADLB: Asynchronous dynamic load balancing, 2014. `https://www.cs.mtsu.edu/~rbutler/adlb/`.

[23] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int'l J. High Performance Computing Applications*, 2009.

[24] S. Chatterjee, S. Taşırlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with MPI. In *Proc. IPDPS '13*, 2013.

[25] P. Cicotti. *Tarragon: a Programming Model for Latency-Hiding Scientific Computation*. PhD thesis, U. California, San Diego, 2011.

[26] U. o. C. Computation Institute. Beagle: the CI supercomputer for biomedical simulations and data analysis, 2015. `http://beagle.ci.uchicago.edu/technical-specification/`.

[27] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proc. SoCC '12*, 2012.

[28] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, Jan. 2008.

[29] J. DeBartolo, G. Hocky, M. Wilde, J. Xu, K. F. Freed, and T. R. Sosnick. Protein structure prediction enhanced with evolutionary diversity: Speed. *Protein Science*, 2010.

[30] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, Oct. 2007.

[31] J. DeSouza and L. V. Kalé. Msa: Multiphase specifically shared arrays. In R. Eigenmann, Z. Li, and S. P. Midkiff, editors, *Languages and Compilers for High Performance Computing*, volume 3602 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005.

[32] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. *Int'l Conf. on Parallel Processing*, 2008.

[33] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. The international exascale software project roadmap. *Int'l J. High Performance Computing Applications*, 2011.

[34] M. G. F. Dosanjh, P. G. Bridges, S. M. Kelly, and J. H. L. III. A peer-to-peer architecture for supporting dynamic shared libraries in large-scale systems. In *Proc. ICCPW '12*, 2012.

[35] F. R. Duro, J. G. Blas, F. Isaila, J. Carretero, J. M. Wozniak, and R. Ross. Exploiting data locality in swift/t workflows using hercules. In *Proc. Network for Sustainable Ultrascale Computing (NESUS) Workshop*, 2014.

[36] J. Elliott, D. Kelly, J. Chryssanthacopoulos, M. Glotter, K. Jhunjhnuwala, N. Best, M. Wilde, and I. Foster. The parallel system for integrating impact models and sectors (psims). *Environmental Modelling and Software*, 2014.

[37] J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. E. Stone, and J. C. Phillips. Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters. In *Proc. Int'l Green Computing Conf.*, Aug 2010.

[38] S. Evripidou, G. Gao, J.-L. Gaudiot, and V. Sarkar, editors. *1st Workshop on Data-Flow Execution Models for Extreme Scale Computing: DFM'11*. IEEE Computer Society, Oct. 2011.

[39] I. O. for Standardization. EBNF syntax specification standard: ISO/IEC 14977: 1996 (e), 1996.

[40] N. Francez and M. Rodeh. Achieving distributed termination without freezing. *IEEE Trans. Softw. Eng.*, May 1982.

[41] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 1998.

[42] W. Frings, D. H. Ahn, M. LeGendre, T. Gamblin, B. R. de Supinski, and F. Wolf. Massively parallel loading. In *Proc. ICS '13*, 2013.

[43] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proc. ICFP '99*, 1999.

[44] M. Gilge. *IBM System Blue Gene Solution Blue Gene/Q Application Development, Second Edition*. IBM Redbooks, 2013.

[45] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *Proc. Int'l Conf. Distributed Computing Sys.*, 2004.

[46] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, Oct. 2006.

[47] K. O. W. Group et al. The OpenCL specification, version 2.0, 2014. `http://khronos.org/registry/cl/specs/opencl-2.0`.

[48] M. Hategan, J. Wozniak, and K. Maheshwari. Coasters: uniform resource provisioning and access for scientific computing on clouds and grids. In *Proc. Utility and Cloud Computing*, 2011.

[49] O. Heinonen, D. Karpeyev, K. Maheshwari, X. Zhong, B. Narayanan, S. Sankaranarayanan, M. Wilde, P. Zapol, and I. Rungger. Coupled molecular-dynamics and first-principle transport calculations of metal/oxide/metal heterostructures. March 2015. APS Meeting March 2015.

[50] G. M. Hocky, L. Berthier, and D. R. Reichman. Equilibrium ultrastable glasses produced by random pinning. *J. Chemical Physics*, 2014.

[51] G. M. Hocky and D. R. Reichman. A small subset of normal modes mimics the properties of dynamical heterogeneity in a model supercooled liquid. *J. Chemical Physics*, 2013.

[52] S. Jagannathan. Continuation-based transformations for coordination languages. *Theoretical Computer Science*, 2000.

[53] P. G. Joisha. Compiler optimizations for nondeferred reference counting garbage collection. In *Proc. ISMM '06*, 2006.

[54] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[55] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *Int'l Conf. Parallel Processing Workshops (ICPPW) 2009*, Sep. 2009.

[56] D. S. Katz, T. G. Armstrong, Z. Zhang, M. Wilde, and J. Wozniak. Many task computing and Blue Waters. Technical Report CI-TR-13-0911, Computation Institute, University of Chicago, November 2011.

[57] K. Kennedy. Use-definition chains with applications. *Computer Languages*, 1978.

[58] S. J. Krieder, J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu. Design and evaluation of the GeMTC framework for GPU-enabled many task computing. In *Proc. HPDC '14*, 2014.

[59] L. Kuper and R. R. Newton. LVars: Lattice-based data structures for deterministic parallelism. In *Proc. Workshop Functional HPC: FHPC '13*, 2013.

[60] L. Kuper and R. R. Newton. Joining forces: Toward a unified account of LVars and Convergent Replicated Data Types. In *Proc. Workshop Determinism and Correctness in Parallel Programming*, 2014.

[61] L. Kuper, A. Todd, S. Tobin-Hochstadt, and R. R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[62] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *Proc. POPL '14*, 2014.

[63] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, Apr. 2010.

[64] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proc. PPoPP '12*, 2012.

[65] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proc. HPDC '12*, 2012.

[66] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso. A survey of data-intensive scientific workflow management. *J. Grid Computing*, 2015.

[67] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. POPL '88*, 1988.

[68] E. L. Lusk, S. C. Pieper, and R. M. Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, Jan. 2010.

[69] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *Proc. PPoPP '09*, 2009.

[70] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[71] MPICH. MPICH: High-performance portable mpi, 2015. `http://www.mpich.org/`.

[72] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[73] NCSA. Blue Waters user portal, 2015. https://bluewaters.ncsa.illinois.edu/hardware-summary.

[74] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proc. SOSP '13*, 2013.

[75] R. S. Nikhil. An overview of the parallel language Id. Technical report, DEC, Cambridge Research Lab., 1993.

[76] NVIDIA. NVIDIA CUDA programming guide, version 6.0, 2014. `http://docs.nvidia.com/cuda/index.html`.

[77] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007.

[78] OpenMP. OpenMP specification 4.0, 2013. `http://openmp.org/wp/openmp-specifications/`.

[79] J. K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, March 1998.

[80] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *Proc. HotOS '13*, 2013.

[81] C. Pareja, R. Pena, F. Rubio, and C. Segura. Optimizing eden by transformation. *Trends in Functional Programming*, 2000.

[82] M. Parisien, X. Wang, G. Perdrizet, C. Lamphear, C. A. Fierke, *Ketan Maheshwari*, M. J. Wilde, T. R. Sosnick, and T. Pan. Discovering RNA-protein interactome by using chemical context profiling of the RNA-protein interface. *Cell Reports*, May 2013.

[83] Y. Park and B. Goldberg. Static analysis for optimizing reference counting. *Information Processing Letters*, 1995.

[84] T. Parr and K. Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proc. PLDI '11*, 2011.

[85] J. C. Phillips, J. E. Stone, K. L. Vandivort, T. G. Armstrong, J. M. Wozniak, M. Wilde, and K. Schulten. Petascale Tcl with NAMD, VMD, and Swift/T. In *Proc. Workshop for High Performance Technical Computing in Dynamic Languages*, 2014.

[86] J. J. Pitt, L. L. Pesce, D. Fitzgerald, A. J. Grundstad, M. Murphy, A. Sokovic, I. T. Foster, R. L. Grossman, K. P. White, Y. Babuji, and M. Wilde. Robust scaling of next-generation sequencing analyses using the modular SwiftSeq workflow. In *Proc. Cray User Group*, 2015.

[87] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In H. G. Baler, editor, *Memory Management*, volume 986 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1995.

[88] H. Pritchard, I. Gorodetsky, and D. Buntinas. A uGNI-based MPICH2 Nemesis network module for the Cray XE. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

[89] T. O. M. Project. Open MPI: Open source high performance computing, 2015. `http://www.open-mpi.org/`.

[90] I. Raicu, I. T. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Proc. MTAGS '08*, Nov 2008.

[91] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *Proc. SC '08*, 2008.

[92] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay. Accelerating large-scale data exploration through data diffusion. In *Proc. Int'l Workshop Data-aware Distributed Computing*, 2008.

[93] J. Reppy and Y. Xiao. Specialization of CML message-passing primitives. *SIGPLAN Not.*, Jan. 2007.

[94] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, May 1998.

[95] I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belagodu, P. Purandare, K. Ramamurty, K. Wang, and I. Raicu. Achieving efficient distributed scheduling with message queues in the cloud for many-task computing and high-performance computing. In *Proc. CCGrid '14*, May 2014.

[96] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proc. PPoPP '11*, 2011.

[97] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1990.

[98] V. Sarkar and D. Cann. POSC - a partitioning and optimizing SISAL compiler. *SIGARCH Comput. Archit. News*, June 1990.

[99] E. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, Sept. 1989.

[100] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

[101] G. Singh, C. Kesselman, and E. Deelman. Optimizing grid-based workflow execution. *J. Grid Comp.*, 2005.

[102] S. L. Small, M. Wilde, S. Kenny, M. Andric, and U. Hasson. Database-managed grid-enabled analysis of neuroimaging data: the CNARI framework. *Int'l J. Psychophysiology*, 2009.

[103] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proc. SC '09*, 2009.

[104] E. Strohmaier, H. Simon, M. Meuer, , and J. Dongarra. Top500 Supercomputer Sites, November 2014, 2014. `http://top500.org/lists/2014/11/`.

[105] Swift team. Swift/T - high performance dataflow computing, 2015. `http://swift-lang.org/Swift-T/`.

[106] S. Taşırlar and V. Sarkar. Data-driven tasks and their implementation. In *Proc. ICPP '11*, 2011.

[107] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. In *Proc. PPoPP '14*, 2014.

[108] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 2005.

[109] The MPI Forum. MPI: A message-passing interface standard, 2012.

[110] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report MIT-LCS-TR-370, Massachusetts Institute of Technology, Cambridge, MA, 1986.

[111] K. D. Underwood and R. Brightwell. The impact of MPI queue usage on message latency. *Proc. ICPP '04*, 2004.

[112] E. Vairavanathan, S. Al-Kiswany, L. Costa, M. Ripeanu, Z. Zhang, D. Katz, and M. Wilde. Workflow-aware storage system: An opportunity study. In *Proc. CCGrid*, 2012.

[113] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu. Optimizing load balancing and data-locality with data-aware scheduling. In *Proc. Int'l Conf. Big Data*, Oct 2014.

[114] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 2009.

[115] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Par. Comp.*, 2011.

[116] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas. Data structures for task-based priority scheduling. In *Proc. PPoPP '14*, 2014.

[117] L. Wolf. Argonne leadership computing facility, "in the news" boosting beamline performance. December 2014.

[118] J. M. Wozniak, T. G. Armstrong, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster. Turbine: A distributed memory data flow engine for many-task applications. In *Proc. SWEET '12*, 2012.

[119] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Large-scale application composition via distributed-memory data flow processing. In *Proc. CCGrid '13*, 2013.

[120] J. M. Wozniak, T. Peterka, T. G. Armstrong, J. Dinan, E. Lusk, M. Wilde, and I. Foster. Dataflow coordination of data-parallel tasks via MPI 3.0. In *Proc. EuroMPI '13*.

[121] J. M. Wozniak, H. Sharma, T. G. Armstrong, M. Wilde, J. D. Almer, and I. Foster. Big data staging with MPI-IO for interactive x-ray science. In *Proc. Big Data Conf.*, 2014.

[122] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. *Studies in Computational Intelligence*, 2008.

[123] J. Zhao and V. Sarkar. Intermediate language extensions for parallelism. In *SPLASH '11 Workshops*, 2011.

[124] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar. Reducing task creation and termination overhead in explicitly parallel programs. In *Proc. PACT '10*, 2010.

[125] X. Zhao, D. Buntinas, J. Zounmevo, J. Dinan, D. Goodell, P. Balaji, R. Thakur, A. Afsahi, and W. Gropp. Toward asynchronous and MPI-interoperable active messages. In *Proc. CCGrid '13*, May 2013.

[126] Z. Zhao, M. Davis, K. Antypas, Y. Yao, R. Lee, and T. Butler. Shared library performance on Hopper. In *Proc. Cray User Group*, 2012.

[127] J. A. Zounmevo and A. Afsahi. An efficient MPI message queue mechanism for large-scale jobs. *Proc. ICPADS '12*, 2012.