

Compiler optimization for data-driven task parallelism on distributed memory systems

Timothy G. Armstrong

University of Chicago
tga@uchicago.edu

Justin M. Wozniak Michael Wilde Ian T. Foster

Argonne National Laboratory & University of Chicago
{wozniak,wilde,foster}@mcs.anl.gov

Abstract

The data-driven task parallelism execution model can support parallel programming models that are well suited for large-scale distributed-memory parallel computing, for example, simulations and analysis pipelines running on clusters and clouds. We describe a novel compiler intermediate representation and optimizations for this execution model, including adaptations of standard techniques alongside novel techniques. These techniques are applied to Swift/T, a high-level declarative language for flexible data flow composition of functions, which may be serial or use lower-level parallel programming models such as MPI and OpenMP. We show that our compiler optimizations reduce communication overhead by 70 to 93% on distributed memory systems. This makes Swift/T competitive in performance with lower-level message passing-based coordination logic for many applications, while offering developers a gentler learning curve and higher productivity.

1. Introduction

Recent years have seen large-scale computationally intensive applications become an increasingly indispensable tool in many fields, including disciplines that have not traditionally used high performance computing. These vary from commercial applications of machine learning to scientific data crunching and high fidelity simulations. They may harness a variety of distributed-memory resources including clouds, grids, and supercomputers. The traditional development model for high-performance computing requires close cooperation between domain experts and parallel computing experts to build applications that make efficient use of distributed-memory systems. In SPMD programming models, for example, careful attention must be given to distribution of data, load balancing, and correct synchronization.

Such parallel programming expertise is a major limiting factor for potential adopters of large-scale computing. Both scarceness of expertise and budget constraints often limit the development of parallel applications. Development is more rapid, agile and thus pervasive if domain experts have greater control over their applications.

We believe that many real-world applications are amenable to generic approaches to data distribution, load balancing, and synchronization. Given a high-level programming model in which these concerns are automatically handled, domain experts could develop large-scale parallel applications without significant parallel programming expertise. Particularly, many applications are naturally implemented with data-driven task parallelism, where large numbers of tasks execute in parallel, and where synchronization is based on implicit intertask data dependencies. Variants of this execution model for distributed-memory and heterogeneous systems have received significant attention, because of the attractive confluence of high performance with ease of development for many applications on otherwise difficult-to-program systems [3, 8, 9].

One implementation of this concept is the Swift/T programming system, which aims to make implicitly parallel scripting as easy and intuitive as sequential scripting in, for example, Python, which has been widely adopted in science. The Swift/T language is declarative and provides determinism guarantees. It offers familiar control flow statements, mathematical functions, and rich libraries for writing high-level “glue code” that composes library functions, including parallel libraries, into application programs that can scale from multi-core workstations to supercomputers [39].

The Swift/T compiler “STC” compiles a high-level script to lower-level executable code that is executed in parallel by many nodes, which coordinate through two distributed, scalable components: a data store and a task queue. This paradigm is challenging to implement efficiently for realistic applications because the programmer specifies little beyond data dependencies through function composition or reads and writes to variables and data structures such as associative arrays. Thus, inter-node data movement, parallel task management, and memory management are left entirely to the implementation. Large-scale applications require execution rates of hundreds of tasks per second on thousands of cores, and naïve compilation of Swift/T for distributed execution imposes high coordination overhead, hindering scalability. Our experience with Swift/T applications has shown that novel compiler optimizations are necessary in order to achieve efficiency and scalability beyond what has been previously reported for implicitly parallel languages.

For this reason, we have developed, adapted, and implemented a range of compiler techniques for data-driven task parallelism, presented herein. By optimizing use of the distributed runtime system, communication and synchronization overhead has been reduced by an order of magnitude. This makes the high-level Swift/T language viable and performant for many important science, engineering and analytics applications. The contributions of this paper are as follows.

- Description of a task-parallel execution model for large-scale distributed memory systems, based on monotonic variables
- Characterization of the novel compiler optimization problems that arise with distributed data-driven implicit task parallelism
- An intermediate code representation for this execution model
- Novel application of both standard and novel compiler optimizations to reduce coordination cost by an order of magnitude
- Application of compiler techniques to achieve efficient automatic memory management in a distributed language runtime.

2. Motivation and Background

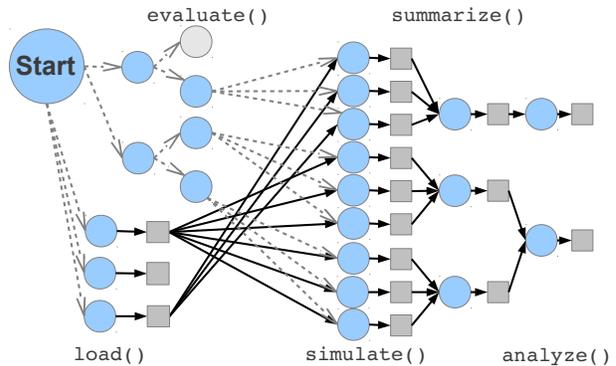
We illustrate and motivate our work by showing how it applies to a commonly occurring style of scientific application: the parameter

```

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);
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) Declarative Swift/T code



(b) Visualization of parallel execution for $M = 2$ $N = 2$ $S = 3$.

Figure 1: An application – an amalgam of several real scientific applications – that runs an ensemble of simulations for many parameter combinations. All statements in the code execute concurrently subject to data dependencies. This application cannot be directly expressed with a static task graph because simulations are conditional on runtime values. The diagram shows an optimized translation to runtime tasks and shared variables. Circles are tasks, squares are data and lines are dependencies.

sweep. A parameter sweep generally involves running a simulation or evaluating a function for a large range of input parameters. The simplest examples can be implemented with nested parallel loops, for example:

```
foreach i in [1:N] {foreach j in [1:M] {f(i, j);}}
```

Here f can be a simple function call or an invocation of a command line application. Although the parameter sweep can be expressed simply and compactly, it nonetheless requires an efficient and scalable implementation.

Realistic examples involve further complications, such as conditional execution or manipulation of input parameters, for example, `if (check(i, j)) { f(i**2, g(j)) }`. A parameter sweep may also simply be a prelude to further processing. For example, a parameter sweep may perform a coarse grid search, followed by further analysis only in regions of interest. Overlapping of phases may also be necessary to improve utilization and reduce time to solution. Figure 1 illustrates a number of these features.

Our experience indicates that even such seemingly trivial applications can require significant language expressiveness. A high-level language is perhaps the most intuitive and powerful way to express this kind of application logic. Ultimately, what many users want is a scripting language that lets them quickly develop scripts that compose high performance functions implemented in a compiled language such as C or Fortran. For sequential execution, dynamic languages such as shell scripts, Perl, or Python address this need. However, this paradigm breaks down when parallel computation is desired. With current sequential scripting languages, the logic must be rewritten and restructured to fit in a paradigm such as message passing, threading, or MapReduce. In contrast, Swift/T natively supports parallel and distributed execution while retaining the intuitive nature of sequential scripting, in which program logic is expressed directly with loops and conditionals.

This style of parallel applications maps naturally to a lower-level execution model: *data-driven task parallelism*, a model of parallelism where tasks are dynamically assigned to resources, scheduled based on data availability. This model of task-parallel computation can expose more parallelism for many applications than less flexible models such as fork-join [33]. It is attractive

for orchestration of tasks on heterogenous and distributed-memory systems, as transparent data movement between devices and data-aware task scheduling fit naturally. Recent work has explored implementing this execution model with libraries and conservative language extensions to C for distributed-memory and heterogenous systems [3, 8, 9, 32]. This work has shown that application performance can match or exceed performance of applications coded directly against the underlying interfaces (e.g. message passing or threads). One reason is that sophisticated algorithms for scheduling (e.g. work stealing) or data movement, usually impractical to reimplement for each application, can be implemented in an application-independent manner. Another reason is that the asynchronous execution model is effective at hiding latency and exploiting available resources in applications with irregular parallelism or unpredictable task runtimes.

Swift/T provides an even higher-level programming model for this execution model. It allows programmers to seamlessly and safely mix application logic with asynchronous task parallelism, using high-level data structures such as associative arrays and avoiding low-level concerns such as memory management.

3. Overview of Swift/T Programming Language

The Swift/T language’s syntax and semantics are derived from Swift [37]. Swift focuses on expressing workflows of command-line applications producing and consuming file data. Swift/T extends this programming model by supporting direct calling of foreign functions (including C and Fortran) with in-memory data, suitable for high performance execution on distributed-memory clusters.

These foreign functions or command-line applications are treated as typed *leaf functions*. The programming model assumes that fine-grained parallelism and computationally intensive code are contained in leaf functions, leaving coarser-grained parallelism for Swift/T. The Swift/T language is implicitly parallel. There is no sequential dependency between consecutive statements, so the order of execution of statements is constrained only by data flow and, when necessary, by control structures including conditionals

and explicit *wait* statements that execute code only once input data is ready. Two types of loop are available: *foreach* loops, for parallel iteration over integral ranges or arrays; and *for* loops, where iterations are ordered and each iteration can pass data to subsequent iterations. Swift/T also supports unbounded recursion. The implementation can execute language statements sequentially when no speedup is likely to be gained from parallelism, for example in the case of built-in arithmetic and string operations and simple data store operations. We avoid a semantic distinction between sequential and implicitly parallel statements because this would complicate the language, steepen the learning curve, and increase cognitive load for developers.

3.1 Data Structures in Swift/T

Swift/T provides several primitive data types. Most standard data types are *monotonic*; that is, they cannot be mutated in such a way that values are overwritten. A monotonic variable starts off containing no information, then incrementally accumulates information until it is *finalized*, whereupon it cannot be further modified. One can construct a rich variety of monotonic data types [11, 17]. The simplest in Swift/T is a single-assignment I-var [21], which starts off empty and is finalized upon the first assignment. All basic scalar primitives in Swift/T are semantically I-vars: ints, floats, booleans, and strings. Files can also be treated as I-vars. More complex monotonic data types can be incrementally assigned in parts but can not be overwritten. Swift/T programs using monotonic variables are deterministic by construction, up to order of side-effects such as I/O. Non-determinism is only introduced by non-Swift/T code, library functions such as `rand()`, or by use of special non-monotonic variables. This simplifies language semantics and lets execution be reordered based on data availability.

The sparse *array*, a dynamically sized monotonic variable, is the main composite data type in Swift/T. 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 imperative style by assigning individual array elements (e.g., `int A[]; A[i] = a; A[j] = b;`). Maintaining determinism requires that any operation based on the array’s state always return the same value. The array lookup operation `A[i]` either returns the single value inserted into the array `A` at index `i` or eventually fails if nothing is ever inserted at `A[i]`. An unfinished array lookup does not prevent progress; other statements can execute concurrently. Functions of the whole array are based on the *final* value of the array. E.g. `size(A)` is the final size of `A` once no more elements can be added, so will stall until `A` is finalized.

Such semantics allow programmers to express intricate data dependency patterns without any risk of nondeterminism or need to manually implement synchronization logic. The implementation is therefore responsible for correct synchronization. The implementation must automatically detect when an array is finalized, that is when new data will no longer be inserted into it according to language semantics. The implementation is also responsible for memory management.

4. Compiler Implementation

The rest of the paper describes the implementation of STC, an optimizing compiler for Swift/T. The compiler translates high-level implicitly parallel Swift/T code into a lower-level execution model (Section 4.1) implemented by Swift/T’s runtime (Section 4.2). An intermediate stage of compilation performs optimizations that reduce communication and synchronization without loss of useful parallelism (Section 4.3). An intermediate representation is used to capture the execution model program (Section 4.4), to which optimization techniques for synchronization, shared data, and reference counting are applied (Sections 4.5, 4.6, 4.7).

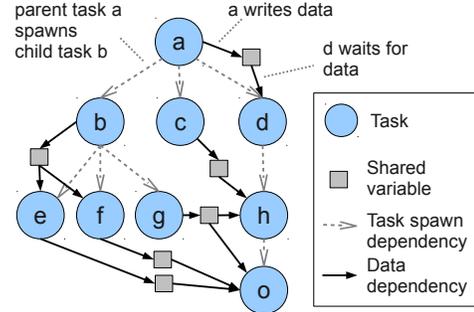


Figure 2: Task and data dependencies in data-driven task parallelism. The tasks, together with spawn dependencies, form a spawn tree rooted at task *a*. Data dependencies on shared variables defer execution of tasks until shared data is finalized by other tasks.

4.1 Data-driven Task Parallelism for Distributed Memory

STC compiles the high-level Swift/T language into executable code for the data-driven task-parallel execution model. STC emits an SPMD program that can be executed on the Turbine runtime system describe in the next section. Turbine implements an execution model based on *data-driven task parallelism*. The emitted code is organized into *task definitions*, which are procedures with explicit inputs containing code that does arbitrary computation and performs runtime operations such as spawning tasks, or reading/writing global data. A *task* is a runtime instantiation of a task definition with inputs bound to specific data. Once executing, tasks cannot be preempted by the runtime, and cannot suspend waiting for long-running operations to complete (in contrast with many thread-based models of parallelism). Thus, well-behaved tasks should generally not run indefinitely.

Each task can spawn *child tasks* that execute asynchronously, so a *spawn tree* of tasks is formed, as shown in Figure 2. Parent tasks can pass data to their child tasks at spawn time. This includes scalar values such as numbers or short strings, along with references to global data store items containing arbitrary data.

The execution model also includes *shared variables* that exist outside of the context of a single task, providing a means for coordination between multiple tasks: for example, a task can spawn two tasks, passing both a reference to a shared variable, which one task reads and the other writes. Unlike a fork-join model of task-parallel computation, parent tasks do not wait for child tasks to finish. *Data dependencies*, which defer the execution of tasks, are the primary synchronization mechanism provided by the runtime. Once a task is spawned, it can execute only after all data dependencies are finalized, which occurs when tasks finish writing those variables. Tasks are free to write, or not write, any data they hold a reference to, so the identity of the writer task may be undetermined until runtime.

If we compare the semantics of the runtime execution model with the semantics of the Swift/T language, they are lower-level in multiple aspects. Aside from the lack of high-level syntax, there are fewer semantic guarantees. For example, no absolute protection against race conditions (e.g., accessing non-finalized state) are provided, so determinism is not guaranteed. Explicit bookkeeping is also needed for both memory management and correct finalization of variables. Bookkeeping errors could result in memory leaks, prematurely freed data, or deadlocks. Thus, Swift/T can relieve a programmer of many burdens, and must correctly compile to a subset of valid programs in this execution model.

4.2 Runtime System Architecture

Figure 3 illustrates a scalable, distributed implementation of the execution model using the ADLB [18] and Turbine [38] runtime

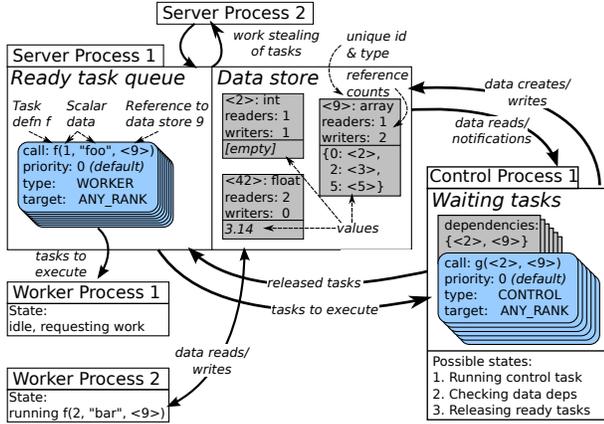


Figure 3: Runtime architecture, illustrating how tasks and data flow across distributed processes. All communication is via message passing. Scaling involves adding processes of each type in a ratio appropriate for the demands of the application. Data is partitioned using a hash function. For clarity, some possible actions are omitted, e.g. a worker can create a task.

libraries for task scheduling, globally shared data storage, and tracking of data dependencies. Turbine/ADLB programs can run on any environment implementing the MPI-2 or MPI-3 standard [34].

Each process is single-threaded and is assigned a distinct role. The system is designed to be scaled up arbitrarily by increasing the number of processes of each type. All communication is through the distributed data store and task queue services provided by *server processes*. Requests to server processes are low-latency by design, to minimize delays to other processes. Most requests take microseconds to process. *Control processes* are responsible for tracking data dependencies and releasing tasks for execution when ready. Control processes also execute control tasks, which contain arbitrary code emitted by the compiler. Control tasks must be of short duration to avoid delays in releasing tasks and the overall progress of the program. *Worker processes* execute arbitrary code, and are intended for longer-running tasks, e.g. those that execute computationally or I/O intensive leaf functions. Worker processes can form dynamic “teams” with an MPI communicator to execute parallel MPI leaf functions [40].

4.3 Optimization for Data-driven Task Parallelism

Compiling for this execution model presents distinct challenges for an optimizing compiler. The goal is to compile highly parallel coordination code so as to: 1) preserve parallelism in the script where task granularity is sufficient to allow parallel speedup; 2) optimize for efficiency and minimize runtime overhead.

A naïve compilation strategy would directly translate each program variable to a runtime shared variable, and each function call or operation to an asynchronous task, with runtime dependencies on all data read by a task. This approach guarantees correctness, but requires many runtime operations.

The primary source of inefficiency and overhead is from the synchronization and interprocess communication required by runtime task and shared data operations, which is much greater than overhead from process-local operations. Total operation latency comprises the latency of message passing across the network, plus queuing delays when server processes are busy with other work. Excessive communication and synchronization can also impair scalability, by causing bottlenecks and queues to form around frequently-accessed shared data or task queues.

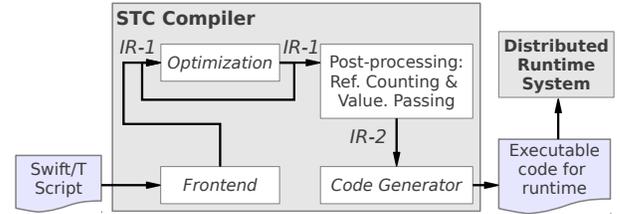


Figure 4: STC compiler architecture showing frontend, intermediate representations, and code generation. The frontend produces IR-1, to which optimization passes are applied to produce successively more optimized IR-1 trees. Postprocessing adds intertask data passing and read/write reference counting information to produce IR-2, which is directly used by the code generator.

$\langle \text{prim-type} \rangle \models \text{int} \mid \text{bool} \mid \text{float} \mid \text{string} \mid \text{blob} \mid \text{file}$
 $\langle \text{I-var} \rangle \models \langle \text{prim-type} \rangle \text{ e.g. } \text{int}, \text{ an I-var}$
 $\langle \text{local-val} \rangle \models \$ \langle \text{prim-type} \rangle \text{ e.g. } \$\text{string}, \text{ a task-local string}$
 $\langle \text{ref} \rangle \models * \langle \text{type} \rangle \text{ e.g. } * \text{int}, \text{ an I-var referring to an int I-var}$
 $\langle \text{array} \rangle \models \langle \text{type} \rangle [] \text{ e.g. } \text{file} [], \text{ an array of files}$
 $\langle \text{type} \rangle \models \langle \text{I-var} \rangle \mid \langle \text{local-val} \rangle \mid \langle \text{ref} \rangle \mid \langle \text{array} \rangle$

Figure 5: BNF grammar for IR type system. Omitted are struct and non-monotonic types.

Therefore, our compiler optimizations are first and foremost targeted at reducing runtime operations, such as task creation and shared data reads, so as to reduce interprocess communication and synchronization. For example, program variables often need not be implemented as shared variables; subscription to variables can often be safely elided; and task creation overhead avoided if data dependencies serialize execution or task granularity is too small.

Swift/T avoids the worst of the task granularity problem inherent to many high-level parallel languages, where a task size must be selected that balances runtime overhead with parallelism [19]. Computationally intensive logic in Swift/T is encapsulated in leaf functions implemented in other languages, which sets a minimum granularity usually large enough to avoid excessive overhead.

4.4 Intermediate Representation

The STC compiler uses a medium-level intermediate representation (IR) that captures the execution model of data-driven task parallelism. Each IR procedure 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 representation that is simple to analyze. Control flow is represented with high-level structures mirroring those of the Swift/T language: *if* statements, *foreach* loops, *do/while* loops, etc. Each block executes sequentially, but child blocks for some control-flow structures execute concurrently. Data-dependent execution can be implied by an IR instruction, e.g. “*async*” operations, or made explicit with *wait* statements that execute a block after data is finalized. 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 4 shows how two variants are used at different compiler stages: *IR-1* and *IR-2*. *IR-1* is generated by the compiler frontend and used by the optimizer. *IR-2* augments *IR-1* with information that is needed for code generation: explicit bookkeeping for reference count manipulation and for passing data to child tasks. This is omitted from *IR-1* because it would cause complications for other optimization passes to maintain.

Variables are either single-assignment scalar values (similar to SSA variables [20]), or handles for shared variables. These handles

```

1  () @main ()#waiton[] {
2  vars: { int n, $int v_n, int f } // declare block vars
3  CallExtLocal argv [ v_n ] [ "n" ] // get argument
4  StoreInt n v_n // store argument value into shared var
5  Call fib [ f ] [ n ] closed=[true] // fib runs async.
6  wait (f) { // print result once computed
7  vars: { $int v_f }
8  LoadInt v_f f // Load value of f (now finalized) to v_f
9  CallExtLocal printf [ ] [ "fib(%i)=%i" v_n v_f ]
10 }
11 }
12
13 // Compute o := fibonacci(i)
14 (int o) @fib (int i)#waiton[i] { // wait until i final
15 vars: { $int v_i, $boolean t0 }
16 LoadInt v_i i
17 LocalOp <eq_int> t0 v_i 0 // t0 := (v_i == 0)
18 if (t0) {
19   StoreInt o 0 // fibonacci(0) == 0
20 } else {
21   vars: { $boolean t2 }
22   LocalOp <eq_int> t2 v_i 1 // t2 := v_i + 1
23   if (t2) {
24     StoreInt o 1 // fibonacci(1) == 1
25   } else {
26     vars: { $int v_i1, $int v_i2, int i1, int i2,
27             int f1, int f2 }
28     // Compute fib(i-1) and fib(i-2) concurrently
29     LocalOp <minus_int> v_i1 v_i 1 // v_i1 := v_i + 1
30     StoreInt i1 v_i1
31     Call fib [ f1 ] [ i1 ] closed=[true]
32     LocalOp <minus_int> v_i2 v_i 2 // v_i2 := v_i + 2
33     StoreInt i2 v_i2
34     Call fib [ f2 ] [ i2 ] closed=[true]
35     // Compute sum once f1, f2 assigned
36     AsyncOp <plus_int> o f1 f2 // o := f1 + f2
37   }
38 }
39 }

```

Figure 6: IR-1 for recursive Fibonacci calculation optimized at -O2.

are either the initial handle for a variable allocated in the block, or an alias. Representing Swift/T monotonic variables directly like this lets us exploit their high-level semantics. Figure 5 summarizes the IR type system.

Figure 6 provides an example of IR-1 for a parallel, recursive Fibonacci calculation. Figure 7 presents partial pseudocode for an IR-1 interpreter, in order to illustrate IR structure and semantics, particularly how block instructions are executed in sequence while tasks are spawned off for asynchronous, data-driven execution. Table 1 lists primitive IR operations.

4.5 Adaption of Traditional Optimizations

We first adapted standard optimization techniques [20] for our intermediate representation.

Constant folding/propagation supports compile-time evaluation of many built-in operations including arithmetic and string operations. STC also allows binding of key-value command-line arguments to compile-time constants, allowing users to compile customized versions of an application. The algorithm uses a preorder walk over the IR tree with a scoped hash table. I-var semantics allow reads (by asynchronous operations) to precede writes in program order, so each block is iterated over until no more constants are found,

A *forward data flow analysis* propagates values and information about variable finalization down the IR tree in a preorder walk. Several optimizations are done in this pass. A *value numbering* scheme is used, similar to the hash-based approach described by Briggs for extended basic blocks [6] with extensions to handle monotonic variables, commutative operations, and limited analysis of branches. This effectively reduces much redundancy. It is particularly effective for eliminating redundant global data store reads and

```

INTERPRET(main_func, rank)
1  if rank == 0 // Rank 0 runs main function
2  SPAWNTASK(∅, main_func.block, INITENV())
3  while (task = GETREADYTASK())
4  EXECBLOCK(task.env, task.block)

EXECBLOCK(env, block)
1  foreach var ∈ block.vars
2  INITVAR(env, var) // Allocate all variables for block
3  foreach stmt ∈ block.statements
4  EXECSTATEMENT(env, stmt) // Execute sequential statements
5  foreach cont ∈ block.continuations
6  EXECCONTINUATION(env, cont) // Execute async. tasks
7  foreach clean ∈ block.cleanups
8  EXECSTATEMENT(clean.env, clean.statement) // Free variables

INITVAR(env, var)
1  if var.storage == LOCAL
2  x = ALLOCATELOCAL(var.type)
3  if var.storage ∈ {SHARED, SHAREDALIAS}
4  x = ALLOCATELOCALREFTO(var.type)
5  if var.storage == SHARED // Allocated in this block
6  ALLOCATESHARED(x, var.type)
7  BIND(env, var.name, x) // Add variable to environment

EXECSTATEMENT(env, statement)
1  // Statements can lookup and modify variables in env,
2  // access and modify shared datastore, and spawn tasks
3  switch (statement)
4  case IF(condition, then_block, else_block)
5  if GETVAR(env, condition)
6  EXECBLOCK(CHILDENV(env), then_block)
7  else EXECBLOCK(CHILDENV(env), else_block)
8  case LOCALOP(builtin_opcode, out, in)
9  // execute local builtin op
10 case ASYNCOPI(builtin_opcode, out, in)
11 // spawn task to execute async builtin op
12 case LOADINT(val, shared_var)
13 // Load value of shared_var
14 case STOREINT(shared_var, val)
15 // Store val into shared_var
16 case AINSERT(builtin_opcode, arr, i, var)
17 // Immediately assign arr[i] = var
18 // etc...

EXECCONTINUATION(env, continuation)
1  // Create new tasks for concurrency or data dependent execution
2  switch (continuation)
3  case WAIT(GETVARS(env, wait_vars), target, block)
4  SPAWNTASK(wait_vars, block, CHILDENV(env))
5  case FOREACH(array, mem_var, block)
6  foreach x ∈ GETVAR(env, array)
7  SPAWNTASK(∅, block, CHILDENV(env, mem_var = x))
8  case RANGELOOP(start, end, ix_var, block)
9  for i = GETVAR(env, start) to GETVAR(env, end)
10 SPAWNTASK(∅, block, CHILDENV(env, ix_var = i))

```

Figure 7: Pseudocode for parallel interpreter for STC IR-1 to illustrate IR-1 semantics. SPAWNTASK(*wv*, *b*, *env*) spawns a task dependent on the variable set *wv*. This to illustrate semantics only: our implementation *a*) compiles the IR to executable code, and *b*) has optimizations such as recursive splitting of loops.

writes where I-vars are replaceable with local temporary values. *Finalized variable analysis* detects I-vars, monotonic arrays, and so forth that are finalized at each statement. A variable is finalized if an instruction finalizes it directly (e.g. writing an I-var) or within a *wait* statement for that variable. Variable dependencies allow finalization to be inferred in further situations. E.g. if I-var *x* contains the value of *a + b*, we can infer that *a* and *b* are finalized inside

Table 1: Opcodes for IR instructions. Opcodes that support struct and file data types, mutable variables, and memory management within tasks are omitted.

Opcodes	Description
LocalOp, AsyncOp	Execute builtin operations, e.g. arithmetic. The local variant operates on local values and executes immediately in the current task context. The async. variant operates on shared variables and spawns a task.
CallExt, CallExtLocal	Foreign function calls, with async. and local versions analogous to above.
Call, CallSync	Sync. and async. Swift function calls
Load(prim-type), Store(prim-type)	Load/store values of shared vars
LoadRef, StoreRef	Load and store reference variables
CopyRef	Copy shared var handle to create alias
Deref(prim-type)	Spawn async. task to dereference e.g. *int to int
{Incr Decr} {ReadRef WriteRef}	Reference counting operations for shared vars
AGet, AGetImm, AGetFuture, ARGet, ARGetFuture	Array lookups. <i>A</i> and <i>AR</i> variants operate on arrays/references to arrays respectively. <i>Future</i> variants take I-var index arguments. <i>AGetImm</i> performs the lookup immediately, and fails if element is not present. All others execute asynchronously.
AInsert, AInsertFuture, ARInsert, ARInsertFuture	Array inserts, following same convention as before
ANestedImm, ANestedFuture, ARNested, ARNestedFuture	Create nested array at index if not present. Required to support automatic creation of nested arrays

`wait(z) { ... }`. The finalized variable analysis allows inlining of *wait* continuations and *strength reduction*, whereby statements using expensive runtime operations are replaced with ones that use fewer or no runtime operations. E.g. for some operations, one instruction version wraps an operation in a data-dependent task to wait for the value of an input variable, while another version executes immediately without synchronization.

Dead code elimination uses a tree walk to build a variable dependence graph for an entire function. Monotonic variables simplify this process, since we need not consider the scope of overwritten values of a variable. Live variables are identified by finding the transitive closure from variables that are either function outputs or input/outputs of side-effecting instructions. The analysis accounts for variables aliasing parts of data structures, with another graph capturing the *is a part of* relationship. Untaken branches of conditionals and any empty control flow structures are also eliminated.

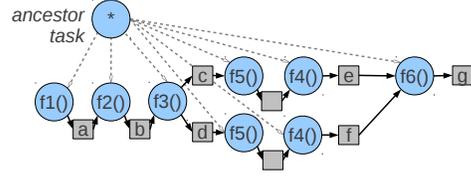
Function inlining is an important optimization. STC’s default function calling convention uses shared variables (e.g., I-vars) to pass arguments for generality. This method is often expensive because it can require unnecessary data store loads and stores. Function call overhead can be an issue due to the use of many small functions, either user-written, or compiler-generated to wrap foreign function calls. Function inlining allows other optimization passes to eliminate unnecessary loads and stores and more generally to create optimization opportunities for later passes. Functions with a single call site are always inlined. Otherwise, a simple heuristic is used: $function\ instruction\ count \times \#\ call\ sites < 500$. Directly or mutually recursive calls are identified to avoid infinite cycles of inlining. Typical Swift/T programs can often be inlined entirely into the main function, allowing aggressive interprocedural optimization. *Asynchronous op expansion*, a variant of inlining where an asynchronous instruction is expanded to a *wait* statement plus non-asynchronous instruction, is also used.

Several loop optimizations are implemented. *Loop invariant hoisting* is important for typical Swift/T scripts, in which large parallel nested foreach loops often include redundant computations

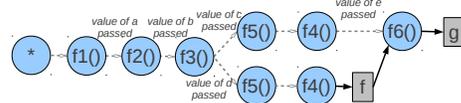
```

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

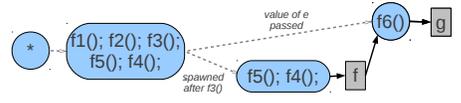
```



(b) Unoptimized version, relying on shared variables to pass data and perform synchronization



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



(d) After pipeline fusion merges tasks

Figure 8: Traces of execution showing optimization of task and data dependencies in a Swift/T code fragment.

such as nested array lookups in the innermost loop. *Loop fusion* fuses foreach loops with identical bounds, reducing runtime loop management overhead and allows optimization across loop bodies. *Loop unrolling* is also performed. Loops with < 16 iterations are completely expanded. Loops with high or unknown iteration counts are unrolled by a factor of 8x at high optimization levels. A simple heuristic caps the unroll factor to limit code size increase to at most 256 instructions per unrolled loop, thereby avoiding excessive code-size expansion. The main benefit of unrolling in STC is to allow optimization across multiple iterations by other passes.

4.6 Optimization for Data-driven Task Parallelism

A number of further transformations are performed that are specific to data-driven task parallelism. These transformations aim to restructure the task graph of the program to be more efficient, without reducing worthwhile parallelism: namely, any parallelism of sufficient granularity to justify incurring task creation overhead.

Two related concepts are used to determine whether transformations may reduce worthwhile parallelism. The first is whether an intermediate code instruction is *long running*: whether the operation will block execution of the current task for a long or unbounded time. Our optimization passes avoid serializing execution of long-running instructions that could run in parallel. The second is whether an instruction is *progress enabling*: for example, a store to a shared variable that could enable dependent tasks to execute. The optimizer avoids deferring execution of potentially progress-enabling instructions by a significant amount. For example, it avoids adding direct or indirect dependencies from a long-running instruction to a progress-enabling instruction. To categorize operations, whitelists of short-running instructions and blacklists of progress-enabling instructions are used. Annotations on library functions provide additional information: we assume that tasks to be executed on control processes are short-running.

One optimization pass is called *task coalescing*, because it relocate tasks, coalesces task, and generally reconfigures the IR task structure. One effective technique, which we call *task pushdown*, is to resolve data dependencies between tasks by pushing statements down in the IR tree to the block where an input variable is assigned. Wait statements are prime candidates for relocation, as they cannot

execute until a variable is assigned. Instructions that wait for input variables before executing are also relocated. This often enables further optimization by later passes. One effect that can result is conversion of data dependency edges to task spawn edges, as shown in Figure 8c. Task coalescing also tries to merge tasks where possible without impeding progress. It merges nested wait statements where no progress is made in the outer wait, and also merges together waits for overlapping sets of variables attached to the same block, which reduces data dependencies that must be resolved at runtime.

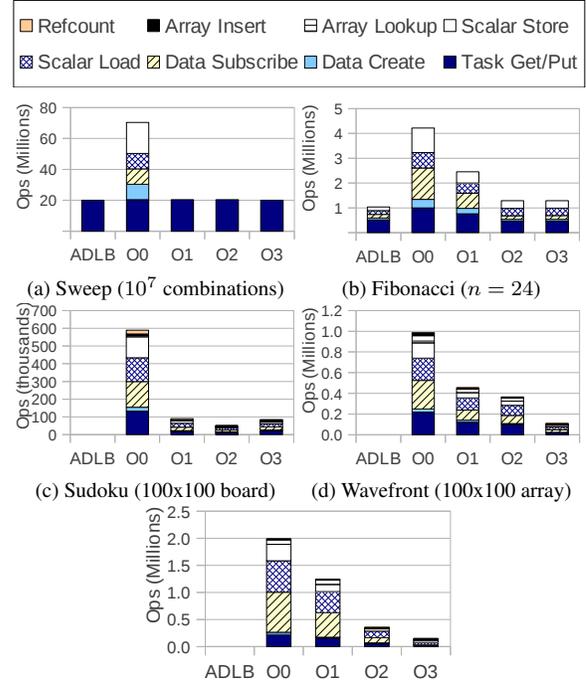
Another optimization is *pipeline fusion*, illustrated in Figure 8d. A commonly occurring pattern is a sequentially dependent set of function calls: a “pipeline.” We can avoid runtime task dispatch overhead and data transfer without any reduction in parallelism by fusing a pipeline into a single task. For short tasks or for tasks with large amounts of input/output data, this method saves much overhead. As a generalization, a fused task will spawn dependent tasks if a pipeline “branches.” This was inspired by pipeline fusion in streaming languages [13], which is similar in concept but essentially a different optimization. Streaming languages have static task graphs with dynamic flows of data, while data-driven task parallelism has dynamically created task graphs operating on 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 means that compiler analysis is required to identify valid opportunities.

4.7 Finalization and memory management

The Swift/T language implementation is responsible for both memory management (automatically reclaiming memory allocated to variables) and variable finalization (detecting when a variable will no longer be written). These two problems are related; and we address them with *automatic distributed reference counting*. Read and write reference counts are defined for each shared variable. When the write reference count drops to zero, the variable is finalized and cannot be written; when both reference counts drop to zero, the variable can be deleted. This design is multipurpose: for example, an I-var starts with one write reference, which is decremented upon assignment to finalize the variable. In the case of arrays, the compiler must determine which statements may read or write each variable, and write reference counts are incremented and decremented depending on the number of active tasks that could modify the array.

Two postoptimization passes over the IR add all necessary reference count operations. The first pass identifies where read and write references are passed from parent to child tasks. For example, if the array A is declared in a parent block and written within a wait statement, a passed write reference is noted. The second pass performs a post-order walk over the IR tree to add reference counting operations. A naïve reference counting strategy would be to increment or decrement the reference count of a shared variable every time a reference is copied or lost. However, this strategy would impose an unacceptable overhead: it could easily double the number of data store operations and therefore messages.

Instead, STC uses a more sophisticated approach. For each block, two integers, both initialized to zero, are maintained for each shared variable in scope to track read and write increments/decrements. A pass over the block increments the appropriate counter for a copied reference to a variable (e.g. passed to an instruction, or into an annotated child block), and decrements for each reference that goes out of scope at the end of the block. Once the counts are accumulated, reference count operations are added to the block. The fallback strategy is to place increments at the start of the block (or in the case of an alias, after it is initialized) and



(e) Simulated Annealing (125 iterations, 100-way objective function parallelism)

Figure 9: Impact of optimization levels on number of runtime operations that involve message passing or synchronization.

decrements at the end, which ensures that reference counts do not incorrectly drop to zero too early during execution.

This approach allows several optimizations. *Cancelling and merging* reference count operations is enabled by the use of counters. E.g. 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. Reference increments from child blocks can be *pulled up* for merging/cancelling. Reference counts for parallel foreach loops can be *batched*, exploiting chunked execution of loops. Reference count increments or decrements can be *piggybacked* on other data operations, such as variable creation or variable reads. With a distributed runtime, the piggy-backed reference count is almost free, since no additional messages need to be sent.

In combination, these techniques allow reference counting overhead to be reduced greatly. Separate reference count operations can be eliminated entirely in cases where the number of readers can be determined statically. In the case of large parallel loops, reference counting costs can often be amortized over the entire loop.

5. Evaluation

To characterize the impact of different optimization levels, we chose five benchmarks that capture commonly occurring patterns. **Sweep** is a parameter sweep with two nested loops and completely independent tasks. **Fibonacci** is a synthetic application with the same task graph as a recursive Fibonacci calculation with a custom calculation at each node that represents a simple divide-and-conquer application. **Sudoku** a divide-and-conquer Sudoku solver that recursively prunes and divides the solution space and terminates early when a solution is found. Sudoku is a non-trivial benchmark with ~ 50 lines of Swift/T and ~ 800 lines of C. **Wavefront** is a synthetic application with more complex data dependencies, where a two-dimensional array is filled in with each cell dependent on three adjacent cells. **Simulated Annealing** is a production science application comprising ~ 500 lines of Swift/T and ~ 2000

Table 2: Runtime operation counts, measured in thousands of operations, in simulated annealing run, showing impact of each optimization pass. Each row includes prior optimizations.

	Task	Create	Sub.	Load	Store	Lookup	Insert	RefCount	Total
O0	221.3	41.3	740.5	616.9	305.9	79.8	14.8	3.5	2024.1
+Constant fold +DC elim.	165.3	15.4	658.2	575.8	198.1	79.8	14.8	3.8	1711.2
+Forward dataflow	157.8	13.8	453.4	427.5	129.5	79.7	14.8	0.6	1277.3
O1: +Loop fusion	157.7	13.8	453.3	427.4	129.5	79.6	14.8	0.6	1276.8
+Expand async. ops	157.9	13.8	453.4	427.5	129.5	79.7	14.8	0.6	1277.3
+Expand small loops	157.8	13.8	453.4	427.5	129.5	79.7	14.8	0.6	1277.2
+Hoisting	58.6	13.8	354.5	414.7	67.2	18.2	14.8	0.6	942.3
O2: +Task coalesce	56.3	13.7	96.2	157.8	39.6	18.1	14.8	0.6	397.0
+Inline +Pipeline	28.8	13.3	5.4	78.4	39.1	16.9	14.8	0.6	197.4
+Reorder +Algebra	28.5	13.3	5.3	78.4	39.1	16.9	14.8	0.6	196.9
O3: +Full unroll	28.3	2.7	5.0	78.3	39.1	16.6	14.8	0.7	185.6

of C++ that implements an iterative optimization algorithm with a parallelized objective function.

We ran benchmarks of these applications compiled at different optimization levels. These levels each include the optimizations from previous levels: **O0**: Only optimize write reference counts

O1: Basic optimizations: constant folding, dead code elimination, forward data flow, and loop fusion

O2: More aggressive optimizations: asynchronous op expansion, task coalescing, hoisting, and small loop expansion

O3: All optimizations: function inlining, pipeline fusion, loop unrolling, intrablock instruction reordering, and simple algebra

For the two simplest applications, we also implemented hand-coded versions using the same runtime library, **ADLB** [18], as a baseline.

5.1 Impact of Individual Optimizations

We first measured how optimization affects communication by logging synchronization/communication operations during a benchmark run. Communication operations are a reasonable proxy for compiler effectiveness that is independent of runtime implementation. For most applications, reduced communication and synchronization will directly improve scalability and performance.

Figure 9 shows the cumulative impact of each optimization level on the number of runtime operations, while Table 2 shows a more granular breakdown of the effect of individual optimization passes on the simulated annealing application, the most complex benchmark. Garbage collection was disabled while running these benchmarks so that we could examine its impact separately. Overall we see that all applications benefit markedly from basic optimization, while more complex applications benefit greatly from each additional optimization level. Compared with hand-coded **ADLB**, **STC** at **O3** uses only fractionally more runtime synchronization and communication. More complex applications would present more opportunities to implement optimizations in a hand-coded version, so this gap may widen somewhat. However, more complex applications are also exactly when the higher-level programming model is most valuable.

5.2 Reference Counting

We also examined the impact of reference counting for garbage collection in isolation in order to understand the overhead imposed by automatic memory management and the impact of optimizations designed to reduce it. We ran the same benchmarks under three different configurations, based on the **O3** configuration: **Off**, where read reference counts are not tracked and memory is never freed; **Unopt**, where all reference counting optimizations are disabled; and **Opt**, with reference counting optimizations enabled. Figure 10 shows the results. The Sweep benchmark is omitted since at **O3** no shared variables were allocated. The results show that the reference counting optimizations are effective, reducing the additional num-

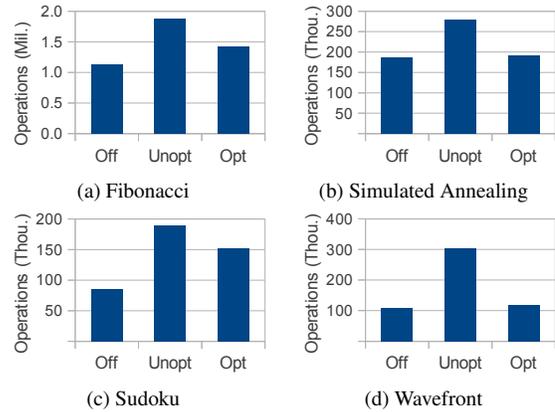


Figure 10: Impact of unoptimized and optimized reference counting for memory management on runtime operations

ber of operations required for memory management to 2.5%-25% for three benchmarks. The optimizations were less effective for Sudoku, which heavily uses struct data types that are not yet handled well by reference counting optimizations.

5.3 Application Speedup

The second part of the optimization evaluation is to examine the impact on runtime of different optimizations. We first ran the previously introduced benchmarks on a Cray XE6 supercomputer. Each node has 24 cores. Except otherwise indicated, 10 nodes were used for benchmarks. We measure throughput in tasks/sec dispatched to worker processes; this metric captures how efficiently the Swift/T system is able to distribute work and hand control to user code.

Different cluster configurations were chosen based on initial tuning, with different splits between worker, which execute the actual user code in the applications and control/server processes. These are different for different applications because some applications have more synchronization compared to computation. We performed a coarse search in increments of 8 for the best ratio for each application. The ratio for Sweep was 192 : 48, for Fibonacci 204 : 36, and for Wavefront 128 : 112. With Simulated Annealing, we used $n : 48$, where n is a variable number of workers.

Figure 11 shows the results of these experiments. For the **O0** and **ADLB** Sweep experiment runs and the **O1** Wavefront run, the 30-minute cluster allocation expired before completion. Since these were the baseline runs, we report figures based on a runtime of 30 minutes, to be conservative. We omitted Sudoku because the runtime was too short to obtain accurate timings: the most challenging Sudoku problem was solved at all optimization levels in 1.25-1.9 seconds, a 40-65x speedup.

With each benchmark, we can see that reduction in operation count in Figure 9 gave a roughly proportional increase in through-

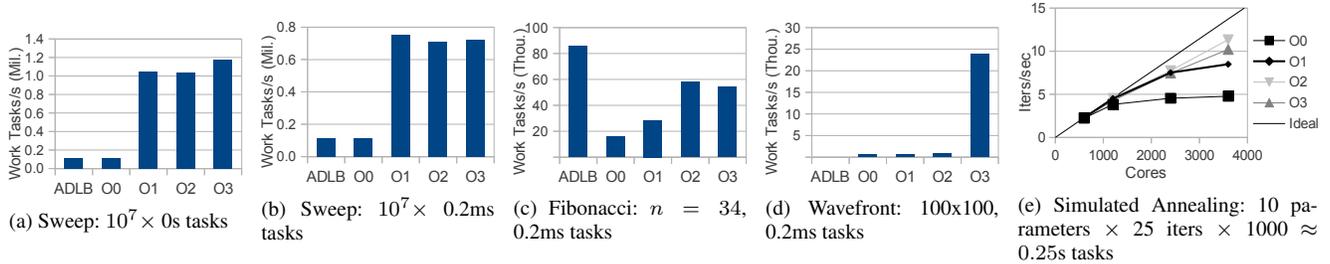


Figure 11: Throughput at different optimization levels measured in application terms: tasks/sec, or annealing iterations/sec.

put. In some cases, for example in Wavefront, speedup was more than proportional to the reduction in runtime operations: the unoptimized code excessively taxed the data-dependency tracking at runtime, causing bottlenecks to form around some data. This supports our claim in Section 4.6 that runtime operations are the primary bottleneck for Swift/T. The wide variance between tasks dispatched per second in different benchmarks is due primarily to the difference in runtime operation intensity. Performance of hand-coded ADLB on Sweep was bottlenecked by a single process generating work tasks, while the Swift/T version automatically parallelized work generation. With some effort, the ADLB issue could be fixed. In contrast, the hand-coded Fib program performed substantially better mainly because in the hand-coded version we avoided having two separate classes of worker and control processes and thus achieved better utilization. Figure 11e shows strong scaling for the simulated annealing benchmark. At lower optimization levels, task dispatch limits scaling, whereas code compiled at higher optimization levels scales better.

6. Related Work

Many authors have addressed the problem of improving performance of distributed *workflows* created through data flow composition, often with explicit task graphs. None have treated the problem as a compiler optimization problem. Rather, the problems addressed have been scheduling problems where resource availability and movement of large data are the major limitations. Thus, that work focused on computing efficient schedules for task execution and data movement [24, 30, 31, 41], generally assuming that a static task graph is available. We focus on applications with finer-grained parallelism in conjunction with a high-level programming model, in which runtime overhead is, in contrast, a dominant concern. Previous authors have made a case for the importance of such applications [27] and the value of combining a low-level computation language and a high-level scripting language [23].

Hardware data-flow-based languages and execution models received significant attention in the past [2]. There has been a resurgence of interest in hardware-based [16, 22] and software-based [5, 7, 10, 26, 33] data flow models because of their ability to expose parallelism, mask latency, and assist with fault tolerance. Previous work has sought to optimize data flow languages with arrays: SISAL [29] and Id [35]. Both languages have similarities to Swift/T, but both emphasize generating efficient machine code and lower-level parallelism. Id targets data flow hardware rather than a distributed software runtime. The SISAL runtime used fork-join parallelism, but its compilation process eliminated much potential parallelism. In STC, task-graph-based transformations and the more involved reference counting required for fully dynamic task graphs also necessitated new techniques.

Other authors have described intermediate representations for parallel programs, typically extending sequential imperative representations with parallel constructs [42]. Our work differs by focusing on a restricted data flow programming model that is suitable for parallel composition of lower-level codes. Our restricted model al-

lows aggressive optimization because of monotonic data structures and loose rules on statement reordering.

Related compiler techniques have been proposed in other contexts. Task creation and management overhead is a known source of performance issues in task-parallel programs. Zhao et al. reduce task parallelism overhead through safely eliminating or reducing strength of synchronization operations [43]. Arandi et al. show benefits from compiler-assisted resolution of intertask data dependencies with a shared-memory runtime [1]. The communication-passing transformation described by Jagannathan [14] is related to the STC task coalescing optimization technique that relocates code to the point in the IR tree where required data is produced. Optimizations have been proposed to reduce reference counting overhead [15, 25], similar in spirit to STC’s reference counting optimization. Such techniques, designed for sequential or explicitly parallel functional/imperative languages are, however, substantially different.

Other authors have reduced the cost of data parallelism and fork-join task parallelism through runtime techniques that defer task creation overhead [12, 19, 28]. However, these techniques do not easily apply to data-driven task parallelism.

7. Future Work

The STC optimizer comprises a flexible intermediate representation and a substantial suite of optimizations but opportunities for improvement naturally remain.

We had success from adapting optimizations from the imperative language optimization literature. We expect that adoption of proven techniques, including representations like SSA and more sophisticated control/data flow analyses would result in further improvement. Additionally, certain more specialized techniques could bring substantial benefits. Techniques for affine nested loops (e.g [4]) could be applied to applications with patterns such as the wavefront example. Data structure representation could be optimized: there are unexploited opportunities, for example, to use lighter-weight representations for small arrays.

Further evolution of the language runtime also present opportunities. Past work [36] has identified opportunities for runtime systems to optimize data placement and movement for data-intensive applications given hints about future workload. Our compiler infrastructure could be used to perform analysis and pass hints to the runtime about patterns of data movement. The current system also uses only synchronous operations for the data and task store. Communication latency could be better masked through use of overlapping asynchronous operations. The compiler infrastructure could support analysis of which operations can be safely overlapped.

8. Conclusion

We have described a set of optimization techniques that can be applied to improving efficiency of distributed-memory task-parallel programs expressed in a high-level programming language. Our performance results support two major claims: that a high-level

scripting language is a viable model programming model for scalable applications with demanding performance needs and that applying a wide spectrum of compiler optimization techniques in conjunction with runtime techniques greatly helps towards this aim.

The system described in this paper is in production use for science applications running on up to 8,000 cores in production and over 100,000 cores in testing. Application of compiler techniques to communication reduction 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.

Acknowledgments

This research is supported by the U.S. DOE Office of Science under contract DE-AC02-06CH11357. This research also was supported in part by NIH through computing resources provided by the Computation Institute and the Biological Sciences Division of the University of Chicago and Argonne National Laboratory, under grant S10 RR029030-01

References

- [1] 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.
- [2] K. Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3), Mar. 1990.
- [3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Comput.: Pract. Exper.*, 23(2):187–198, 2011.
- [4] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. PACT '04*, 2004.
- [5] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. In *Proc. IPDPS '11*.
- [6] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Softw. Pract. Exper.*, 27(6):701–724, June 1997.
- [7] Z. Budimlic, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent collections. *Sci. Program.*, 18(3–4), Aug. 2010.
- [8] 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*, Los Alamitos, CA, USA. IEEE Computer Society.
- [9] S. Chatterjee, S. Taşirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with MPI. In *Proc. IPDPS '13*, Los Alamitos, CA, USA. IEEE Computer Society.
- [10] P. Cicotti and S. B. Baden. Latency hiding and performance tuning with graph-based execution. In *Proc. DFM '11*.
- [11] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proc. SoCC '12*.
- [12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.
- [13] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162, Oct. 2006.
- [14] S. Jagannathan. Continuation-based transformations for coordination languages. *Theoretical Computer Science*, 240(1):117 – 146, 2000.
- [15] P. G. Joisha. Compiler optimizations for nondeferred reference: counting garbage collection. In *Proc. ISMM '06*, pages 150–161, 2006.
- [16] R. Knauerhase, R. Cledat, and J. Teller. For extreme parallelism, your OS is soooooo last-millennium. In *Proc. HotPar '12*, 2012.
- [17] L. Kuper and R. Newton. A lattice-theoretical approach to deterministic parallelism with shared state. Technical Report TR702, Indiana Univ., Dept. Comp. Sci., Oct 2012.
- [18] 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*, 17:30–37, Jan. 2010.
- [19] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proc. LFP '90*.
- [20] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
- [21] R. S. Nikhil. An overview of the parallel language Id. Technical report, DEC, Cambridge Research Lab., 1993.
- [22] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao. TIDeFlow: The time iterated dependency flow execution model. In *Proc. DFM '11*.
- [23] J. K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3), March 1998.
- [24] S. Pandey, L. Wu, S. Guru, and R. Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *Proc. AINA '10*.
- [25] Y. Park and B. Goldberg. Static analysis for optimizing reference counting. *Information Processing Letters*, 55(4):229 – 234, 1995.
- [26] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *Int. J. High Perform. Comput. Appl.*, 23(3), Aug. 2009.
- [27] 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.
- [28] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *Proc. IPDPS '08*.
- [29] V. Sarkar and D. Cann. POSC - a partitioning and optimizing SISAL compiler. *SIGARCH Comput. Archit. News*, 18(3b):148–164, June 1990.
- [30] G. Singh, C. Kesselman, and E. Deelman. Optimizing grid-based workflow execution. *J. Grid Comp.*, 3(3), 2005.
- [31] G. Singh, M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, and G. Mehta. Workflow task clustering for best effort systems with Pegasus. In *Proc. MG '08*.
- [32] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proc. SC '09*.
- [33] S. Taşirlar and V. Sarkar. Data-Driven Tasks and their implementation. In *Proc. ICPP '11*.
- [34] The MPI Forum. MPI: A message-passing interface standard, 2012.
- [35] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report MIT-LCS-TR-370, Cambridge, MA, 1986.
- [36] 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.
- [37] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Par. Comp.*, 37:633–652, 2011.
- [38] 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*.
- [39] 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*.
- [40] 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*.
- [41] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. *Studies in Computational Intelligence*, 146:173–214, 2008.

- [42] J. Zhao and V. Sarkar. Intermediate language extensions for parallelism. In *SPLASH '11 Workshops*.
- [43] 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.

(The following paragraph will be removed from the final version)

This manuscript was created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.