

# Compiler optimization for distributed dynamic data flow programs

Timothy G. Armstrong\*, Justin M. Wozniak<sup>†‡</sup>, Michael Wilde<sup>†‡</sup>, Ian T. Foster\*<sup>†‡</sup>

\* Dept. of Computer Science, University of Chicago, Chicago, IL, USA

<sup>†</sup> Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

<sup>‡</sup> Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

**Abstract**—Distributed, dynamic data flow is an execution model well-suited for many large-scale parallel applications, particularly scientific simulations and analysis pipelines running on large, distributed-memory clusters. In this paper we describe compiler optimization techniques and an intermediate representation for distributed dynamic data flow programs. These techniques are applied to Swift/T, a high-level declarative language that allows flexible data flow composition of functions written in other programming languages such as C or Fortran. We show that compiler optimization can reduce communication overhead by 70-93% on distributed memory systems, making the high-level language competitive with hand-coded coordination logic for certain common application styles.

## I. INTRODUCTION

The Swift/T programming system allows parallel composition of functions and external programs into highly parallel, distributed data flow applications for systems ranging from multi-core workstations to distributed-memory supercomputers with tens of thousands of cores [32]. The Swift/T implementation compiles a high-level script to lower-level statements that are executed by many nodes, which coordinate through a distributed data store and task queue.

The goal of the Swift language is to make implicitly parallel scripting as easy and intuitive as sequential scripting in, for example, shell scripts or Python, both of which have been heavily adopted by computational scientists. The language is declarative and provides determinism guarantees, while also offering standard niceties such as high-level control flow statements, mathematical functions, and string manipulation that make it possible to write the higher-level “glue code” required to compose library functions into complete applications.

This programming paradigm presents challenges for a language implementer. The Swift/T language has high-level, declarative semantics and asks little of a programmer beyond expressing data dependencies between functions through normal composition and variable passing. Thus, data movement, parallel task management, and memory management are left entirely to the implementation. Relieving application programmers of such concerns opens the door to rapid development of scalable parallel applications. However, this flexibility comes at the cost of requiring the language implementer to provide robust performance for typical application patterns.

Our experience implementing applications with Swift/T has shown that relying exclusively on runtime approaches for task and data management would result in unacceptable

overhead in many situations, particularly in the case of large-scale applications running on thousands of cores, which must dispatch hundreds of thousands of tasks per second to fully utilize the machine. For this reason, we have developed and implemented a range of compiler techniques that enable more efficient execution of high-level scripts on large clusters and supercomputers. Using compiler optimization, runtime coordination overhead can be reduced by an order of magnitude, bringing performance to a level that makes the programming model viable for many realistic applications. The contributions of this paper are:

- characterization of the novel compiler optimization problems that arise in a distributed data flow execution model;
- an intermediate representation for this execution model;
- application of both standard and novel compiler optimizations to reduce coordination cost by an order of magnitude; and
- characterization of the challenges imposed by memory management and compiler techniques to address them.

## II. MOTIVATION

We illustrate and motivate our work by showing how it applies to a simple, commonly occurring style of scientific application: the parameter sweep. The parameter sweep is a rather simple application pattern that can be expressed compactly in a high-level parallel programming language. Nonetheless it requires an efficient and scalable implementation.

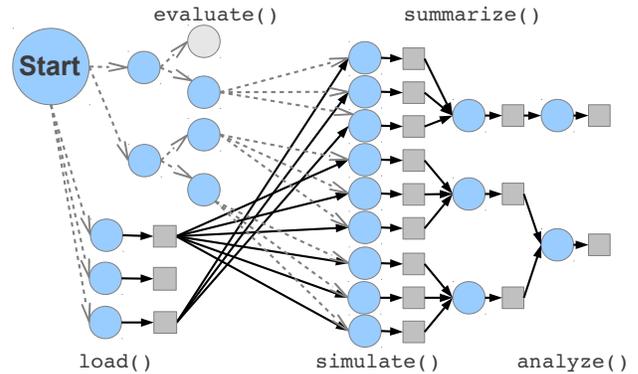
A parameter sweep generally involves running a simulation or evaluating a function for a large range of input parameters. The simplest examples can be implemented with nested loops, for example `foreach i in [1:N] { foreach j in [1:M] { f(i, j); } }`, where  $f$  could be a simple function call or an invocation of a command line application. Realistic examples involve further complications, such as conditional execution or manipulation of input parameters, e.g. `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, that is followed by further analysis only in regions of interest. We may want to overlap phases to improve computer utilization and reduce time to solution, an approach which is trivial in a data flow programming model. Figure 1 illustrates a number of these features.

```

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  $M = 2$   $N = 2$   $S = 3$

Fig. 1: An application - an amalgam of several real scientific applications - that runs an ensemble of simulations for many parameter combinations. All statements in the above 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.

Our experience indicates that even such seemingly trivial applications often require a large degree of language flexibility. A high level language is perhaps the most intuitive and powerful way to support this flexibility. 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 single-node 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. In the above example, a Swift/T implementation can take care of assigning function calls such as `simulate` and `analyze` to different available processors for execution.

Executing even fairly simple applications efficiently is challenging in a distributed environment. For example, in some representative science applications, the `simulate` function in Figure 1 might be implemented in C and have a long-tailed runtime distribution with mean  $0.1$ s. One instance of the application might be a massively parallel run with  $M * N * S \approx 10^9$ , while another might have  $M * N * S \approx 10^6$ , with the above code inside another sequential loop that uses analysis results to decide on a new round of simulations to run. In these scenarios, irregular parallelism and unpredictable task runtimes require dynamic, high performance load balancing. The need to rapidly dispatch short-running tasks places high demands on runtime systems. Large input, output, and

intermediate data sets require intelligent management of data movement and locality.

Intelligent algorithms and engineering of runtime systems can help deal with these challenges, but ahead-of-time compiler optimization, we believe, is essential for this high-level programming model to be viable for applications that demand high performance.

### III. SWIFT PROGRAMMING LANGUAGE

We work with a variant of the Swift programming language [30]. Swift was originally designed for expressing workflows of command-line applications producing and consuming file data. The language is easily generalized to support direct calling of functions written in other languages with in-memory data. These external functions or command-line applications are treated in the language 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.

The 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 loop structures are available: *foreach* loops, which express iteration over integral ranges or arrays with independent iterations, and *for* loops, where iterations are ordered and each iteration can pass data to subsequent iterations. The Swift/T implementation also supports unbounded recursion.

A Swift implementation can execute language statements sequentially when no speedup is likely to be gained from

parallelism, for example in the case of builtin arithmetic and string operations and simple data store operations.

### A. Data structures in Swift

Swift 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. It is possible to construct a rich variety of monotonic data types [8], [13]. The simplest in Swift is a single-assignment I-var [16], which starts off empty and is finalized upon the first assignment. All basic scalar primitives in Swift 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 not overwritten.

Use of monotonic variables allows Swift programs to produce deterministic results, despite the non-deterministic order of statement execution. The language provides *referential transparency* for monotonic variables in R-value expressions (expressions not on the left hand side of an assignment). This means that if a deterministic function  $f$  is applied to a monotonic variable,  $x$ , then the expression  $f(x)$  always evaluates to the same value anywhere that  $x$  is in scope. Given an assignment  $y = f(x)$ ,  $y$  and  $f(x)$  have the same value and are interchangeable.

Arrays in Swift are dynamically sized monotonic variables that may be sparse. The value 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;`). Referential transparency means that any operation that queries the array state must always return the same value. The array lookup operation  $A[i]$  is defined to either return the single value inserted into the array  $A$  at index  $i$ , or eventually fail if nothing is ever inserted at  $A[i]$ . A pending array lookup does not stall execution of the program, as other statements in a block can execute concurrently with the pending lookup. Functions of the whole array are defined based on the *final* value of the array. For example, `size(A)` is the final size of the array once no more elements can be added.

Such semantics allow programmers to express intricate data dependency patterns without any risk of non-determinism or need to manually implement synchronization logic. However, a consequence is that the language implementation must handle a range of complicated synchronization cases automatically. The implementation must automatically detect when an array is finalized, i.e. when it is no longer possible that new data will be inserted into it. The implementation is also responsible for memory management.

## IV. COMPILER IMPLEMENTATION

The rest of the paper describes the implementation of STC, an optimizing compiler for Swift. The compiler translates high-level implicitly parallel Swift code into a lower-level

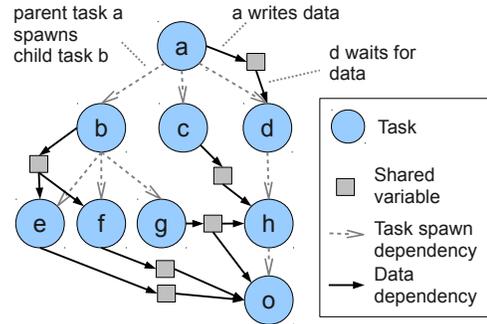


Fig. 2: Task and data dependencies in dynamic data flow. The tasks, together with spawn dependencies, form a spawn tree rooted at task  $a$ . Data dependencies resulting from shared variables defer execution of tasks until shared data is available.

execution model, (Section IV-A). An intermediate representation is used to capture the execution model program (Section IV-C), to which optimization techniques for synchronization, shared data and reference counting are applied (Sections IV-D, IV-E, IV-F) that reduce communication without loss of useful parallelism (Section IV-B).

STC currently generates code in the Tcl scripting language that calls runtime libraries that are implemented in C. MPI is used for interprocess communication. Using Tcl allows rapid development, and easy implementation of extension functions. For current applications, interpreting Tcl has not been a major bottleneck, but this may change in future with finer-grained parallelism. For that reason, STC is retargetable by design.

### A. Distributed Dynamic Data flow

Swift/T's runtime implements a *distributed dynamic data flow* execution model that supports data-driven task parallelism. As a program executes, the runtime maintains a set of tasks running on computational resources, a set of queued tasks that are ready to execute, and a set of tasks that are waiting for input data. This model of task-parallel computation, can expose more parallelism for many applications than less flexible models such as fork-join [27]. In our specific version of the model, tasks are not preemptible by other tasks and, once running, cannot block waiting for data or other tasks.

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, such as references to shared data, or small values such as numbers or short strings.

A *shared data store* provides a global address space in which *shared variables* can be read or written by any task. This feature allows flexible coordination patterns: 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. Thus, *data dependencies* are the only way to manage inter-task dependencies. Once a task is spawned, it can only execute after all data dependencies are finalized. The data dependencies of a pending task must be

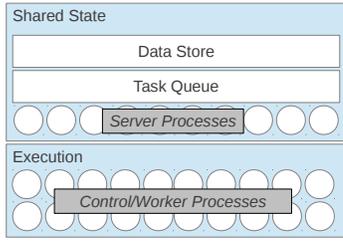


Fig. 3: Runtime architecture, illustrating how different services are distributed across many distributed processes.

fulfilled by other tasks writing the corresponding 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.

In contrast to the high-level Swift/T language, the dynamic data flow execution model provides no protections against accessing non-finalized state (e.g., the size of a non-finalized array), so determinism is not guaranteed. Thus, Swift/T must compile to a deterministic subset of possible programs in this execution model. 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 at high runtime cost. In many cases program variables need not be implemented as shared variables, runtime data dependency checks can be safely elided, and operations can execute synchronously.

Figure 3 illustrates a scalable, distributed implementation of the execution model. An arbitrary number of server processes implement a distributed data store and task queue with dynamic load balancing. Two different classes of processes execute worker and control tasks. The control processes also have local task queues [15], [31].

### B. Optimization for Distributed Data flow

The distributed data flow execution model presents distinct challenges for an optimizing compiler. The goal is to compile highly parallel coordination code so as to: *a)* preserve parallelism in the script where task granularity is sufficient to allow parallel speedup; *b)* optimize for scalability, e.g., by partitioning parallel loops to assist with load balancing; and *c)* optimize for efficiency and minimize runtime overhead.

In this paper we focus primarily on the first and third objectives: maintaining parallelism while reducing runtime overhead. Most inefficiency and coordination overhead is due to synchronization and communication. Reads and writes of shared data, along with task management, generally require interprocess communication, and the latency of sending a message and receiving a response in a distributed environment is orders of magnitude greater than primitive operations within a task. Therefore, our compiler optimizations are largely targeted at reducing the number of non-local task and data operations in the generated code.

### C. Intermediate Representation

The STC compiler uses a single intermediate representation (IR) that captures the distributed dynamic data flow

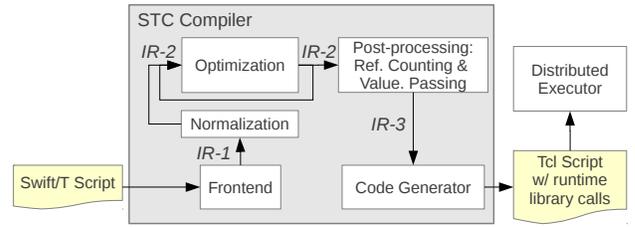


Fig. 4: STC Compiler Architecture showing frontend, intermediate representations, and code generation. IR-1 is first normalized to produce IR-2. Optimization passes are applied to produce successively more optimized IR-2 trees. Finally, post-processing adds inter-task data passing and read/write reference counting information to produce IR-3, which is directly used by the code generator.

```

1  () @main ()#waiton[] {
2  vars: { int n, $int v_n, int f }
3  CallExtLocal argv [ v_n ] [ "n" ] // get argument
4  StoreInt n v_n
5  Call fib [ f ] [ n ] closed=[true]
6  wait (f) { // print result once available
7  vars: { $int v_f }
8  LoadInt v_f f
9  CallExtLocal printf [ ] [ "fib(%i)=%i" v_n v_f ]
10 }
11 }
12
13 (int o) @fib (int i)#waiton[i] { // wait until i final
14 vars: { $int v_i, $boolean t0 }
15 LoadInt v_i i
16 LocalOp <eq_int> t0 v_i 0
17 if (t0) {
18 StoreInt o 0
19 } else {
20 vars: { $boolean t2 }
21 LocalOp <eq_int> t2 v_i 1
22 if (t2) {
23 StoreInt o 1
24 } else {
25 vars: { $int v_i1, $int v_i2, int i1, int i2,
26         int f1, int f2 }
27 LocalOp <minus_int> v_i1 v_i 1
28 StoreInt i1 v_i1
29 Call fib [ f1 ] [ i1 ] closed=[true]
30 LocalOp <minus_int> v_i2 v_i 2
31 StoreInt i2 v_i2
32 Call fib [ f2 ] [ i2 ] closed=[true]
33 AsyncOp <plus_int> o f1 f2
34 }
35 }
36 }

```

Fig. 5: Optimized IR-2 for recursive Fibonacci calculation

execution model just described. The IR makes it straightforward to optimize distributed code where execution crosses process boundaries. Three variants are used in different stages of compilation: *IR-1*, *IR-2*, and *IR-3*. IR-1 is generated by the compiler frontend. IR-2, used by the optimizer, is a normalized version of IR-1 in which all variables within a function are assigned unique names. IR-3 augments IR-2 with information required for code generation: explicit annotations for reference count manipulation and for variable passing from parent to child tasks. Figure 4 puts these different forms in context.

Figure 5 provides an illustrative example of intermediate representation for a parallel, recursive Fibonacci calculation. Figure 6 present partial pseudocode for an IR-2 interpreter, in order to illustrate IR structure and semantics, particularly how block instructions are executed in sequence while tasks are

```

INTERPRET(main_func, worker_rank)
1 if worker_rank == 0
2   SPAWNTASK(0, 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)
3 foreach inst ∈ block.instructions
4   EXECINSTRUCTION(env, inst)
5 foreach cont ∈ block.continuations
6   EXECCONTINUATION(env, cont)
7 foreach cleanup ∈ block.cleanup
8   EXECINSTRUCTION(cleanup.env, instruction)
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
6     ALLOCATESHAREDDATA(x, var.type)
7   BIND(env, var.name, x)
EXECINSTRUCTION(env, inst)
1 // Instructions can lookup and modify variables in env,
2 // access and modify shared datastore, and spawn tasks
3 switch (inst)
4 case LOCALOP(builtin_opcode, out, in)
5   // execute local builtin op
6 case ASYNCOPI(builtin_opcode, out, in)
7   // spawn task to execute async builtin op
8 case LOADINT(val, shared_var)
9   // Load value of shared_var
10 case STOREINT(shared_var, val)
11   // Store val into shared_var
12 case AINSERT(builtin_opcode, arr, i, var)
13   // Immediately assign arr[i] = var
14 // etc...
EXECCONTINUATION(env, continuation)
1 switch (continuation)
2 case WAIT(wait_vars, target, block)
3   SPAWNTASK(wait_vars, block, CHILDENV(env))
4 case IF(condition, then_block, else_block)
5   if (condition) EXECBLOCK(CHILDENV(env), then_block)
6   else EXECBLOCK(CHILDENV(env), else_block)
7 case FOREACH(array, mem_var, block)
8   foreach x ∈ array
9     SPAWNTASK(0, block, CHILDENV(env, mem_var = x))
10 case RANGELOOP(start, end, step, ix_var, block)
11   for i = start to end by step
12     SPAWNTASK(0, block, CHILDENV(env, ix_var = i))

```

Fig. 6: Pseudocode for simple parallel interpreter for STC IR-2. Support for data-dependent execution of tasks is assumed. SPAWNTASK( $wv, b, env$ ) spawns a task dependent on the variable set  $wv$ .

spawned off for asynchronous, data-driven execution. Table I lists primitive IR operations.

Figure 7 describes the type system used in the IR, with distinct types to represent task-local variables and data store shared variables. Reference types add an extra level of indirection to shared variables. For example, `string *x` is a shared variable storing a reference to a `string` variable.

#### D. Adaption of traditional optimizations

We first adapted a standard suite of compiler optimizations for our intermediate representation.

TABLE I: Opcodes for IR instructions. Some opcodes are omitted that support struct and file data types, mutable variables and memory management within tasks.

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, CallLocal	Swift function calls, distinguished by execution mode
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 shared vars, which may not be finalized, for indices. <i>AGetImm</i> performs the lookup immediately, and fails if the index is not present. All other operations can 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

$\langle \text{type} \rangle \models \langle \text{I-var} \rangle \mid \langle \text{local-val} \rangle \mid \langle \text{ref} \rangle \mid \langle \text{array} \rangle$   
 $\langle \text{I-var} \rangle \models \langle \text{prim-type} \rangle$   
 $\langle \text{local-val} \rangle \models \$ \langle \text{prim-type} \rangle$   
 $\langle \text{ref} \rangle \models * \langle \text{type} \rangle$   
 $\langle \text{array} \rangle \models \langle \text{type} \rangle []$   
 $\langle \text{prim-type} \rangle \models \text{int} \mid \text{bool} \mid \text{float} \mid \text{string} \mid \text{blob} \mid \text{file}$

Fig. 7: BNF grammar for IR type system. All types but  $\langle \text{local-val} \rangle$  are shared variables. Omitted are struct and mutable types. The Swift type system is distinct but overlaps.

*Constant folding/propagation.* Compile-time evaluation of many built-in operations including arithmetic and string operations such as concatenation is supported.

*A forward data flow analysis* propagates values and information about variable finalization forward in a block and into descendant blocks. A *global value numbering* scheme is used that assigns unique identifiers to values within a block. This method is effective at eliminating redundant computations, and particularly for eliminating data store reads and writes: in many cases I-vars can be bypassed using local temporary variables. *Finalized variable analysis* detects I-vars, monotonic arrays, etc., that are finalized at each instruction within the code. A variable is clearly finalized if preceding instructions have finalized it directly, or if it is within a *wait* statement for that variable. Basic inference is also performed based on variable dependences. For example, given the Swift conditional `if(x == 1) { ... }`, the code within the block cannot execute until the value of `x == 1` is final, which

also implies that  $x$  is final. This analysis also eliminates unnecessary wait statements and allows strength reduction, whereby expensive operations using runtime data dependence resolution are replaced with ones that execute immediately, or have less runtime overhead.

*Dead code elimination* is performed by building a variable dependence graph for the function. Monotonic variables simplify this process, as 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 since it can require unnecessary data store loads and stores. Small functions are common, either written by users or generated by the compiler to wrap foreign function calls, so function call overhead is an issue. Function inlining allows other optimization passes to eliminate unnecessary loads and stores, and to relocate instructions within inlined function bodies to reduce use of dependency resolution. Typical Swift programs can often be inlined entirely into the main function, allowing aggressive interprocedural optimization. *Asynchronous op expansion* is a variant of inlining where an asynchronous operation is expanded to a Wait statement plus local operation.

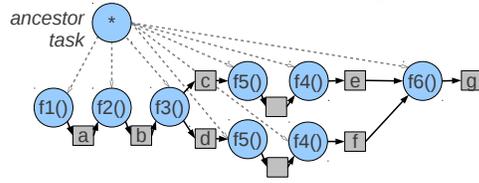
Several loop optimizations are implemented. *Loop invariant hoisting* is important for typical Swift scripts, in which large parallel nested foreach loops often include redundant computations such as nested array lookups in the innermost loop. *Loop fusion* fuses foreach loops with identical bounds, reducing runtime loop management overhead and allow optimization across loop bodies. *Loop unrolling* is also performed. Loops with known, small iteration counts are completely expanded. Loops with high or unknown iteration counts are unrolled at high optimization levels, subject to certain heuristics that take into account code size. Loop unrolling has different benefits in a parallel data flow language to a sequential compiled language. The main benefit of unrolling is that it allows optimization across loop iterations. Iterations of parallel foreach loops are not sequentially dependent, so unrolling loops is a straightforward way to do inter-iteration optimization.

There is further room for improvement in these algorithms. Most optimizations are implemented as single passes over the IR tree, with only information from direct ancestor blocks and continuations used to perform optimization in a given block. This approach is effective at identifying most redundancy in typical Swift scripts, but misses opportunities that would be detected by more sophisticated control flow analysis.

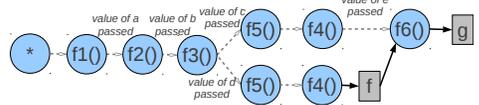
```

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 code fragment

```



(b) Unoptimized version, relying on shared data flow variables to pass data and runtime data dependency tracking



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



(d) After pipeline fusion merges tasks

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

### E. Data flow-specific optimizations

A number of further transformations are performed that are specific to task-parallel data flow execution. These transformations aim to restructure the task graph of the program to be more efficient, without reducing worthwhile parallelism: i.e. 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*: e.g., 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.

One optimization is called *wait coalescing*, as it performs a variety of transformations that relocate, coalesce, and otherwise transform wait statements in the IR. Data dependencies between tasks can be eliminated by pushing wait statements down in the IR to the location where the needed variable is assigned. The effect, shown in Figure 8c, is to convert data dependency edges to task spawn dependency edges. Some control tasks can also be merged without detriment if no progress is made in an enclosing wait, or if the sets of variables waited on by wait statements overlap.

Another optimization is *pipeline fusion*, illustrated in Figure 8d. A commonly occurring pattern is a sequentially dependent

dent 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.”

### F. Finalization and memory management

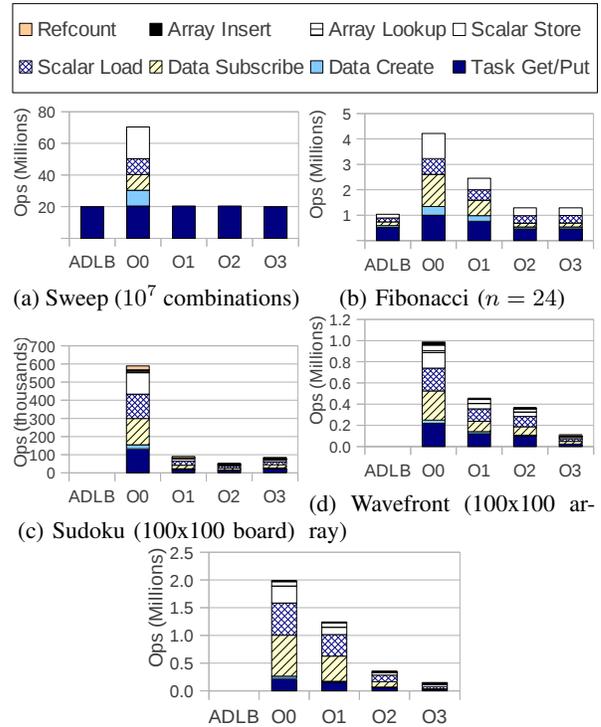
As already mentioned in Section III-A, the Swift language implementation is responsible for both memory management (automatically reclaiming memory allocated to variables) and variable finalization (detecting when a variable will no longer be written). These two problems are related, and we address them 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, and when both drop to zero, then the variable can be deleted. This design is quite flexible: for example, a fresh I-var has a write reference count of one, which is decremented upon assignment to finalize the variable. In the case of read references and write references for arrays, the compiler must determine which statements may read or write each variable.

The addition of reference counts is implemented as two post-optimization passes over the IR. 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 uses this information to insert 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 performance overhead: it could easily double the number of data store operations, and therefore messages.

The second pass applies several optimizations to reduce the number of operations:

- *Canceling and merging* reference count operations, for example if a reference is handed to a single child task while going out of scope in the parent.
- *Pulling up* reference increments from child blocks, allowing them to be merged.
- *Batching* reference counts for parallel foreach loops, exploiting chunked execution of loops.
- *Piggybacking* reference count increments or decrements on other data operations, such as variable creation or variable reads. In a distributed environment, the piggybacked reference count is almost free, as no additional messages need be sent.

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



(a) Sweep ( $10^7$  combinations) (b) Fibonacci ( $n = 24$ )

(c) Sudoku (100x100 board) (d) Wavefront (100x100 array)

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

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

## V. 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** is a more complex divide-and-conquer Sudoku solver that recursively prunes and divides the solution space, and terminates early when a solution is found; **Wavefront** has more complex data dependencies, where a two-dimensional array is filled in with each cell dependent on three adjacent cells; and **Simulated Annealing** is an iterative optimization algorithm with a parallelized objective function.

We ran benchmarks of these applications when they are 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, wait coalescing, hoisting, and small loop expansion.

**O3:** All optimizations: function inlining, pipeline fusion, loop unrolling, intra-block instruction reordering, and simple algebra.

For the two simplest applications, we also implemented hand-coded versions using the same runtime library,

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

ADLB [15], as a baseline.

### A. Impact of Individual Optimizations

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

Figure 9 shows the cumulative impact of each optimization level on the number of runtime operations, while Table II 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 can 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, Swift 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.

### B. Reference Counting

We also examined the impact of reference counting for garbage collection in isolation 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 number 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.

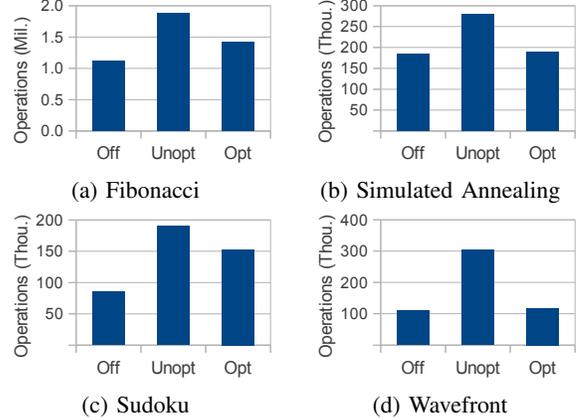


Fig. 10: Impact of unoptimized and optimized reference counting for memory management on runtime operations

### C. 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 processes, which execute user code, and control processes, which execute coordination code and serve data store and task queue requests. The ratio for Sweep was 192 : 48, for Fibonacci 204 : 36, and for Wavefront 128 : 112. Simulated Annealing had a variable number of workers and 48 control processes.

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.

The wide variance between tasks dispatched per second in different benchmarks is primarily due to different complexity of data flow. In some cases, such as for example for O0-O2

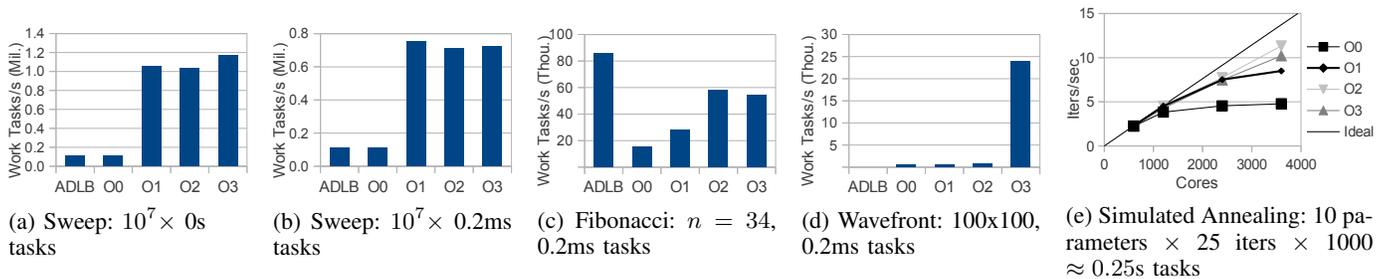


Fig. 11: Throughput at different optimization levels measured in application terms: tasks/sec, or annealing iterations/sec.

in Wavefront, the unoptimized code interacts poorly with the runtime system, causing further slowdown.

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, while code compiled at higher optimization levels scales better.

## VI. RELATED WORK

Many previous 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 [19], [25], [26], [33], 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 case for the importance of such applications [22] and the value of combining a low-level computation language and a high-level scripting language [18].

Hardware data flow-based languages and execution models received significant attention in the past [2]. There has been a resurgence in interest in hardware-based [12], [17] and software-based [5], [6], [7], [21], [27] data flow models due to their ability to expose parallelism, mask latency, and assist with fault tolerance. Previous work has sought to optimize data flow languages with arrays: SISAL [24] and Id [28]. Both languages have similarities to Swift, but both emphasize generating efficient machine code and lower level parallelism. Id targets data flow machines rather than a distributed runtime. The SISAL runtime used fork-join parallelism, meaning that the process of compilation eliminated 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 [34]. Our work differs by focusing on a restricted, data flow programming model that is suitable for parallel composition of lower-level code. Our restricted model allows aggressive optimization due to monotonic data-structures and loose rules on statement reordering. We also focus on a distributed execution context, in which communication overhead is a dominant concern.

Other authors have proposed related compiler techniques in different contexts. Task creation and management overhead is a known source of performance issues in task parallel programs. Zhao et al. describe optimizations that reduce task parallelism overhead by identifying opportunities to safely eliminate or reduce strength of synchronization operations [35]. Arandi et al. show benefits from compiler-assisted resolution of inter-task data dependencies with a shared memory runtime [1]. The communication-passing style transformation described by Jagannathan [10] is related to the STC wait coalescing optimization technique that relocates code to the point in the IR tree where required data is produced. Various optimizations has been proposed for reduction in reference counting overhead [11], [20], which have similar goals to STC’s reference counting optimization, such as cancelling or merging reference counts. The required analysis, however, is substantially different for sequential or explicitly parallel functional or imperative languages.

Other authors have reduced task parallelism overhead for data parallelism and fork-join task-parallelism through runtime techniques that defer task creation overhead [4], [9], [23]. However, these techniques do not easily apply to general dynamic task graphs.

## VII. FUTURE WORK

The STC optimizer suite comprises a range of optimizations, but many opportunities for further improvement remain. For example, standard analyses could be applied in several cases:

- Handling of control flow such as iterative loops and conditionals is currently simplistic: better data flow analyses would improve optimization.
- Certain well-known analyses of affine nested loops could be applied to applications with patterns such as the wavefront example [3], [14].
- Data structure representation could be optimized: there are unexploited opportunities, for example, to switch to a more efficient local representation for small arrays.

Further evolution of the language runtime and the relationship between compiler and runtime also present opportunities.

Past work [29] has identified opportunities for runtime systems to optimize data placement and movement for data-intensive applications given additional information about future workload. Or intermediate representation and other compiler infrastructure offers an opportunity to pass hints to runtime systems about patterns of data movement.

Our current intermediate representation and execution model has only synchronous operations for the data and task store. There is a clear opportunity to better mask communication latency by overlapping asynchronous operations. The current compiler infrastructure offers a good basis for such an extension, as it can easily support analysis of which operations can be overlapped.

## VIII. CONCLUSION

We have described a set of optimization techniques that can be applied to improving communication efficiency of distributed-memory data flow programs expressed in a high-level, deterministic programming language. Our performance results support two major claims: that a high-level data flow scripting language is a viable approach for building scalable applications with demanding performance requirements; and that applying a wide spectrum of compiler optimization techniques in conjunction with runtime and middleware techniques greatly helps with building scalable systems.

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 has been essential to meeting these scalability goals. 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 in part by the U.S. DOE Office of Science under contract DE-AC02-06CH11357, FWP-57810. This research 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. *Proc. DFM '11*, 0, 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. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. PACT '04*, 2004.
- [4] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. *J. Funct. Program.*, 22(4-5):382–438, Aug. 2012.
- [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] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşlılar. Concurrent collections. *Sci. Program.*, 18(3-4), Aug. 2010.

- [7] P. Cicotti and S. B. Baden. Latency hiding and performance tuning with graph-based execution. In *Proc. DFM '11*.
- [8] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proc. SoCC '12*.
- [9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.
- [10] S. Jagannathan. Continuation-based transformations for coordination languages. *Theoretical Computer Science*, 240(1):117 – 146, 2000.
- [11] P. G. Joisha. Compiler optimizations for nondeferred reference: counting garbage collection. In *Proc. ISMM '06*, pages 150–161, 2006.
- [12] R. Knauerhase, R. Cledat, and J. Teller. For extreme parallelism, your OS is soooooo last-millennium. In *Proc. HotPar '12*, 2012.
- [13] 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.
- [14] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proc. POPL '97*.
- [15] 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.
- [16] R. S. Nikhil. An overview of the parallel language Id. Technical report, DEC, Cambridge Research Lab., 1993.
- [17] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao. TIDEFlow: The time iterated dependency flow execution model. In *Proc. DFM '11*, pages 1–9, Oct. 2011.
- [18] J. K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3), Mar. 1998.
- [19] 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*.
- [20] Y. Park and B. Goldberg. Static analysis for optimizing reference counting. *Information Processing Letters*, 55(4):229 – 234, 1995.
- [21] 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.
- [22] 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.
- [23] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. *Proc. IPDPS '08*, 0.
- [24] V. Sarkar and D. Cann. POSC - a partitioning and optimizing SISAL compiler. *SIGARCH Comput. Archit. News*, 18(3b), June 1990.
- [25] G. Singh, C. Kesselman, and E. Deelman. Optimizing grid-based workflow execution. *J. Grid Comp.*, 3(3), 2005.
- [26] 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*, 2008.
- [27] S. Tasirlar and V. Sarkar. Data-Driven Tasks and their implementation. In *Proc. ICPP '11*.
- [28] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report MIT-LCS-TR-370, Cambridge, MA, USA, 1986.
- [29] 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.
- [30] 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, 2011.
- [31] 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 *Int'l Workshop Scalable Workflow Enactment Engines and Technologies (SWEET) 2012*, 2012.
- [32] 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*.
- [33] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. volume 146 of *Studies in Computational Intelligence*. 2008.
- [34] J. Zhao and V. Sarkar. Intermediate language extensions for parallelism. In *SPLASH '11 Workshops*.
- [35] 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.