Compiler optimization for massively parallel data flow

Tim Armstrong, Justin M. Wozniak, Michael Wilde, Ian T. Foster
TA, MW, IF: University of Chicago   JW, MW, IF: Argonne National Laboratory

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. Swift is a high-level declarative language that allows flexible data flow composition of functions written in other programming languages such a C or Fortran.

Swift/T is a high performance, scalable re-implementation of Swift for massive distributed memory clusters.

This poster focuses on one aspect of the implementation: compiler optimization to improve execution efficiency and scalability. We show that compiler optimization can reduce communication overhead by 70-93% on distributed memory systems at scales up to thousands of cores. With compiler optimization, the high-level Swift language becomes competitive with hand-coded coordination logic for certain common styles of computationally intensive applications.

Swift/T Language & Execution Model

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

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

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

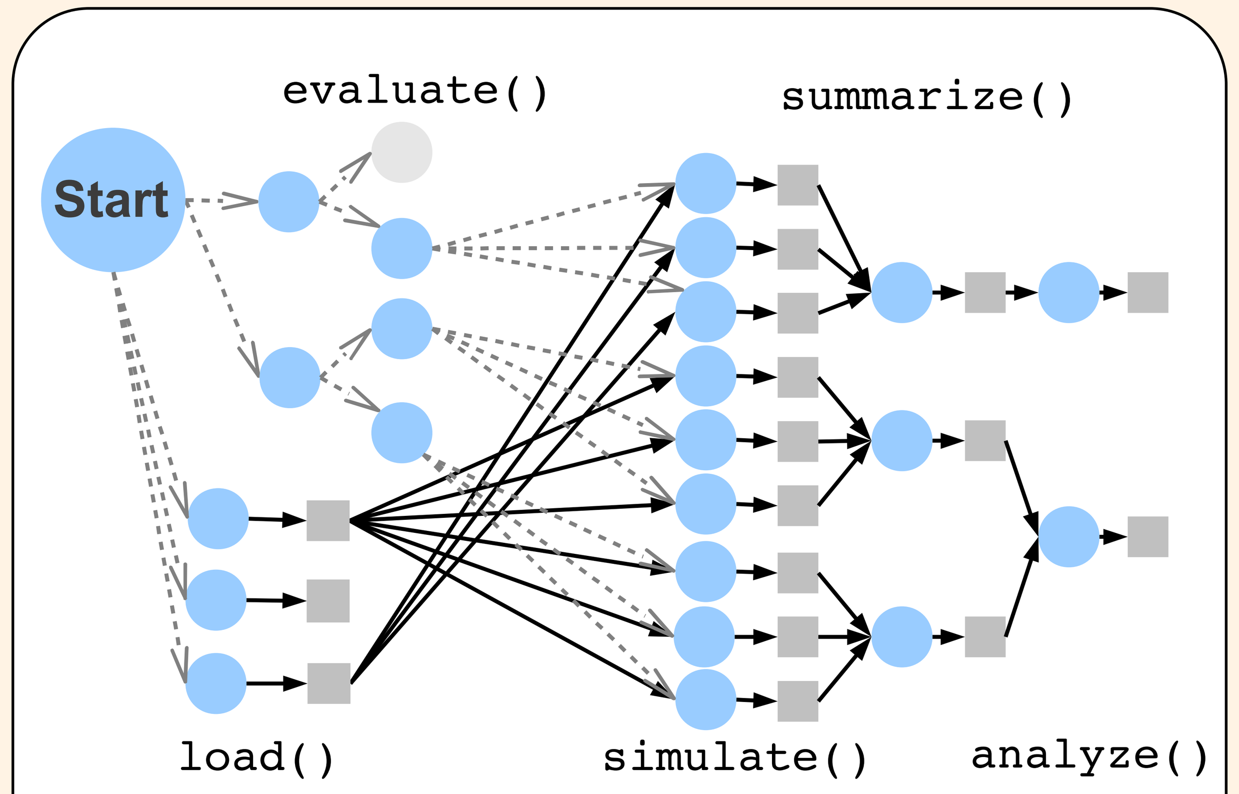


Figure 1: Visualization of parallel execution for M = 2, N = 2, S = 3, showing dynamic data dependencies. Blue circle: task, gray square: data, gray dashed line: task spawn, black line: data dependency

STC Optimizing Compiler

The STC compiler is an important part of the Swift/T toolchain. We depend on the compiler to translate high-level Swift/T scripts into efficient low-level code. This low-level code uses primitive operations for our distributed executor such as "get task", "put task", "fetch integer", "store integer", etc.

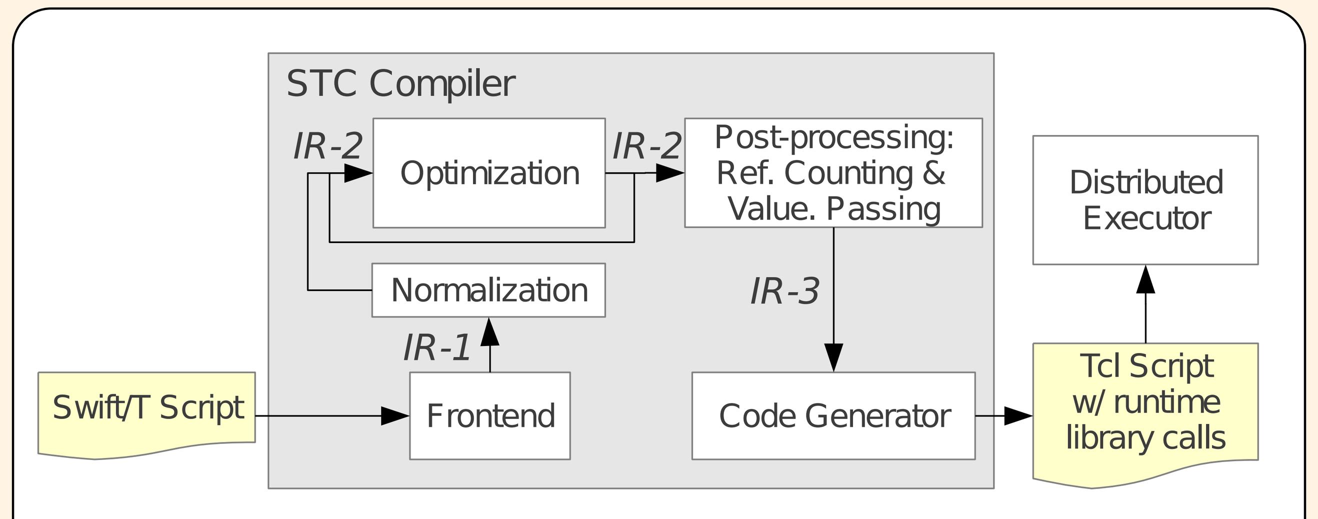
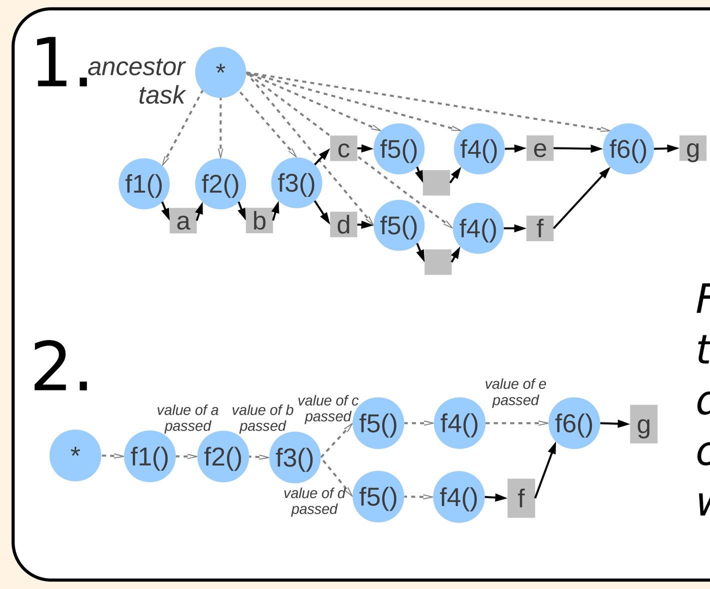


Figure 2: STC compiler architecture

Optimization Techniques

Standard compiler optimizations: constant folding, common subexpression elimination, dead code elimination, function inlining, loop invariant hoisting, etc.

Optimizations for asynchronous data flow: detection of finalized variables, task spawn deferral, task merging, etc.



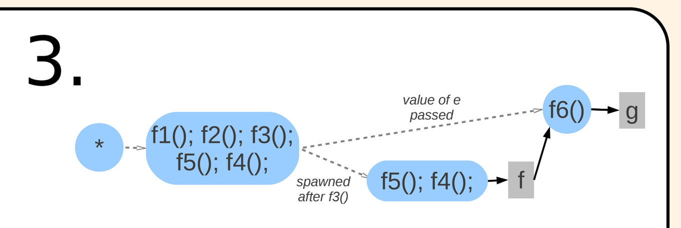


Figure 3: effect of compiler transformations. Runtime data dependencies and task spawn operations are eliminated without reduction in parallelism

Optimizations for reference counting: cancel and combine reference count operations; piggy-back reference counts on other messages; batch reference counts for loops. Reference counting is used for both memory management and for detecting when data structures are *finalized*. Optimization is critical to avoid high overhead.

Experimental Setup

Five benchmark applications were used:

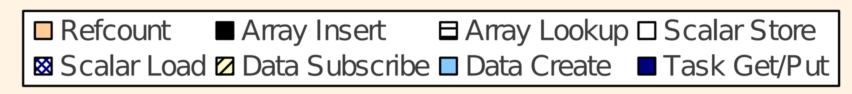
Sweep: a parameter sweep with two nested loops and completely independent tasks
Fibonacci: a synthetic divide-and-conquer application with the same dependency graph as a recursive Fibonacci calculation
Sudoku: a divide-and-conquer Sudoku solver that recursively prunes and divides the solution space, and terminates when solution found
Wavefront: 2D array with each cell a function of three neighbors
Simulated Annealing: an iterative optimization algorithm with a parallelized objective function.

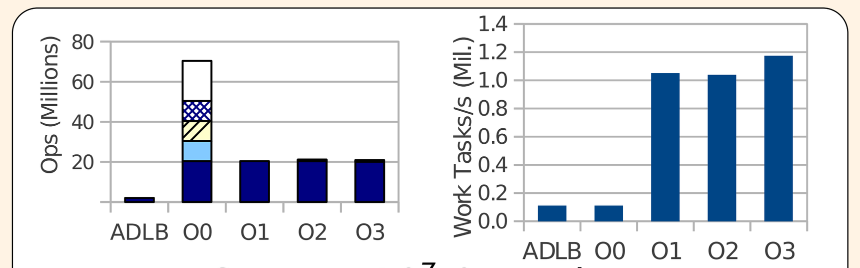
Four implementations were compared.
ADLB: hand-coded C using Swift's distributed runtime library directly
O0: Absolutely no optimizations: naïve compiled Swift/T
O1: Basic compiler optimizations: e.g. constant propagation
O2: More compiler optimizations: e.g loop invariant hoisting
O3: Aggressive optimization: e.g. function inlining, loop unrolling

We measure a) number of task and data operations (a good proxy for total communication) and b) application time-to-solution.

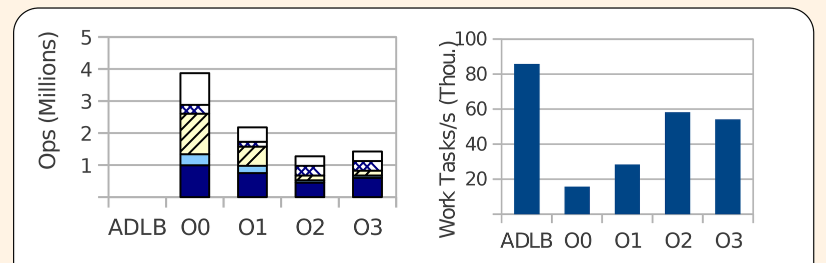
Benchmarks were run on the Beagle Cray XE6 supercomputer 2x12-core 2.1-GHz AMD Opteron 6100 per node. Ten nodes were used unless otherwise stated.

Results

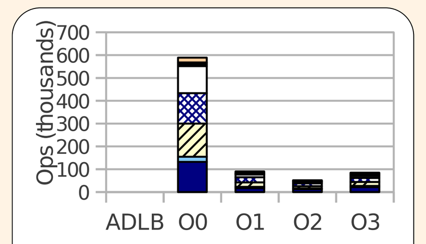




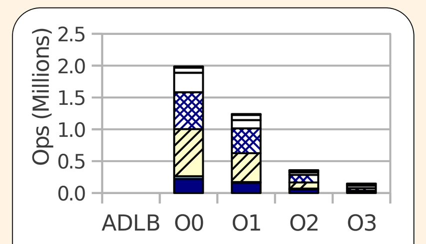
Sweep: 10^7 0s tasks



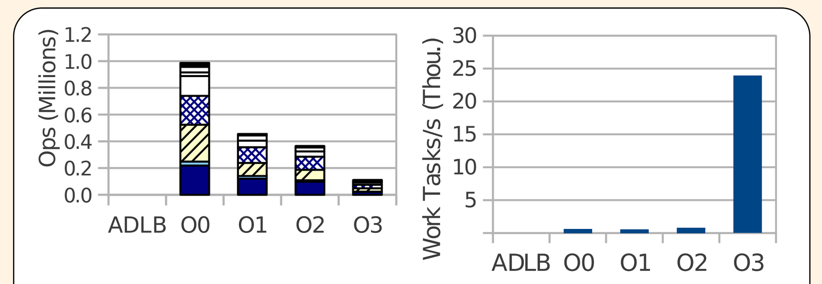
Fibonacci: n=24 0.2ms tasks



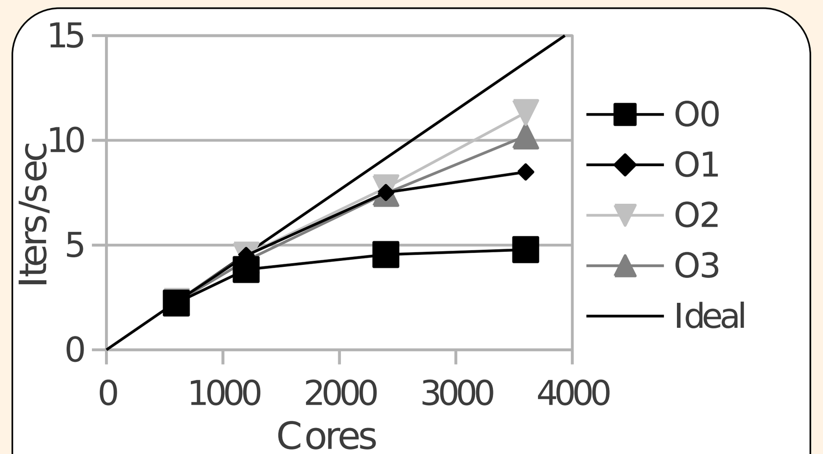
Sudoku: 100x100 board



Annealing: 125 iters, 100-way par.



Wavefront: 100x100, 0.2ms tasks



Annealing strong scaling: 5 params × 25 iters × 1000 ≈ 0.25s tasks

Compiler optimization reduces the number of runtime data and task operations and increases throughput. The benefit varies: basic optimizations always help, advanced optimizations help greatly with complex dataflow. Compiler optimization is essential for applications with short-running tasks and at large scale.

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

Further reading

[1] T. Armstrong, J. M. Wozniak, M. Wilde, I. T. Foster, "Compiler optimization for distributed dynamic data flow programs," full length paper in submission. Preprint available.
[2] J. M. Wozniak, T. Peterka, T. Armstrong, J. Dinan, E. Lusk, M. Wilde, I. T. Foster, "Dataflow Coordination of Data-Parallel Tasks via MPI 3.0," in submission.
[3] J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, E. Lusk, I. T. Foster, "Swift/T Large-scale Application Composition via Distributed-memory Data Flow Processing," CCGrid'13
[4] J. M. Wozniak, T. Armstrong, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed memory data flow engine for many-task applications," SWEET'12
[5] 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 Rev. 2010