Compiler Optimization for Extreme-Scale Scripting

Timothy G. Armstrong,* Justin M. Wozniak,†‡ Michael Wilde,†‡ Ian T. Foster,*†‡
* Dept. of Computer Science, University of Chicago, Chicago, IL, USA
† Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
‡ Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

***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 adaptions of standard techniques alongside novel techniques. These techniques are applied to Swift/T, a high-level scripting language for flexible data flow composition of functions, which may be serial or use lower-level parallel programming models such as MPI and OpenMP. This paper presents preliminary results, indicating that our compiler optimizations reduce communication overhead by 70% to 93% on distributed-memory systems.**

I. Introduction

Recent years have seen large-scale computationally intensive applications adopted in many fields, including disciplines that have not traditionally used high performance computing. These applications may harness a variety of distributed-memory resources including clouds, grids, and supercomputers. Traditionally, expert knowledge and laborious development have been required in order to use these resources. Issues such as data distribution, load balancing, and correct synchronization are major obstacles. New parallel programming models and languages could ameliorate these challenges for many applications, allowing development of large-scale parallel applications with greater ease.

Swift is an expressive high-level scripting language with a significant user base. Swift's high-level declarative semantics expose implicit parallelism, allowing use of generic approaches for data distribution, load balancing, and synchronization. Swift's users build applications with patterns of parallelism ranging from embarrassingly parallel to intricate dataflow. Even with trivially simple (yet highly-parallel) applications, Swift benefits users with automated load balancing and data movement. It comes into its own, however, for applications with complex, dynamic, data dependencies, where the Swift language can express applications with more generality than more restrictive models such as static task graphs.

Swift/T is a new implementation of Swift that aims for extreme scalability up to the largest supercomputers [9]. In pursuit of this goal, this paper explores two questions: what execution model and runtime primitive operations are needed to support extremely scalable data-driven task parallel applications, and how can a high-level declarative language be compiled to make efficient use of these lower-level primitives.

```
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]);
}
```

Fig. 1: Example Swift code for an ensemble of simulations. Execution is ordered purely by data dependencies. This example is not expressible with a static task graph because simulations are conditional on runtime values.

The contributions of this paper are:

- Description of a task-parallel execution model for distributed memory systems
- A compiler intermediate representation for the model
- Novel compiler optimizations that greatly reduce communication cost, greatly improving Swift/T's scalability
- Compiler techniques that achieve efficient automatic memory management in a distributed language runtime

II. Data-driven Task Parallelism

The execution model for Swift/T is *data-driven task parallelism*. Figures 1 and 2 illustrate how a high-level Swift/T script could be efficiently realized as parallel tasks and data. In the model, tasks and data are dynamically created as execution progresses, with tasks scheduled based on data availability. This model is more general and can expose more parallelism than can other models such as fork-join or static task graphs [7]. It is attractive for programming heterogenous and distributed-memory systems because it makes transparent data movement between devices and automatic placement of tasks on resources possible. Recent work has shown that the performance of applications using this execution model can match or exceed applications coded using lower-level interfaces, for example message passing or threads [1], [2].

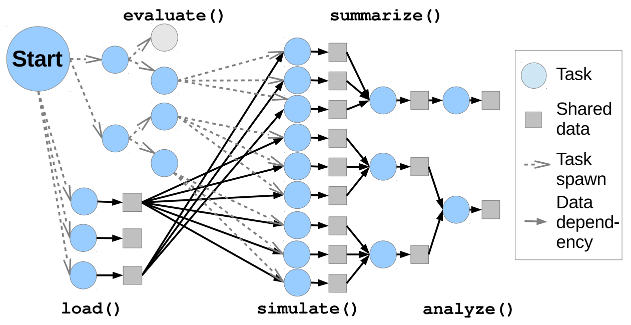


Fig. 2: Visualization of parallel execution of the script from Figure 1 with parameters $M = 2$ $N = 2$ $S = 3$. We show an optimized version that efficiently uses runtime tasks and data.

The reason is that sophisticated algorithms, such as work stealing for task distribution, or data-aware task placement can be implemented in an application-independent manner. Moreover, the asynchronous execution model effectively masks latency by dynamic assigning resources when task runtimes or parallelism patterns are irregular.

III. Swift/T Runtime System

Swift/T uses a scalable, distributed runtime system [8], as shown in Figure 3. The runtime has no central bottleneck and can be scaled up arbitrarily by increasing the number of processes in each role. All communication is through the distributed data store and task queue services provided by *server processes*. Requests to server processes are very low-latency, typically microseconds, minimizing delays to other processes. *Control processes* track data dependencies and release tasks for execution when ready. Control processes also execute short-running control tasks. *Worker processes* execute computationally or I/O-intensive functions. Worker processes can execute serial code, or form dynamic "teams" to execute parallel MPI code [10].

IV. Swift/T Compiler Architecture and Intermediate Representation

The STC compiler translates a high-level Swift/T program into optimized low-level code for the runtime. Figure 4 shows the stages of the compiler. The two middle stages that process an intermediate representation (IR) of the input program are of the most interest. The STC compiler uses a single IR throughout the compiler with two variants. The frontend produces IR-1, to which optimization passes are applied to produce successively more optimized programs. IR-2 augments IR-1 with memory management and data passing bookkeeping that is needed for code generation.

A major contribution of our work is development of an IR for the execution model of data-driven task parallelism. The importance of IR design for optimizing compilers is difficult to overstate [3]. A good IR has several attributes. First, simplicity and uniformity reduces the complexity of optimization passes. Second, the IR must be high level enough to hide irrelevant detail so that optimizations can "see the forest, not

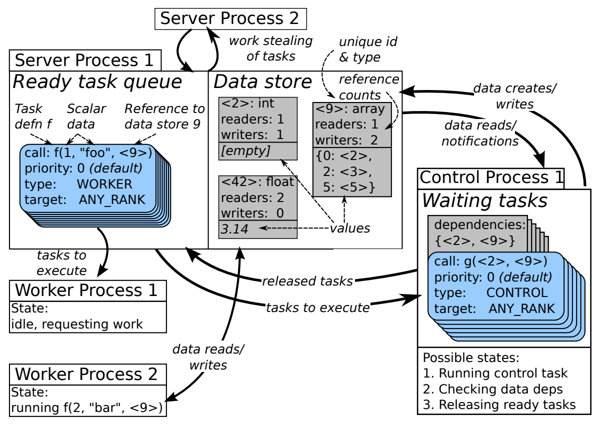
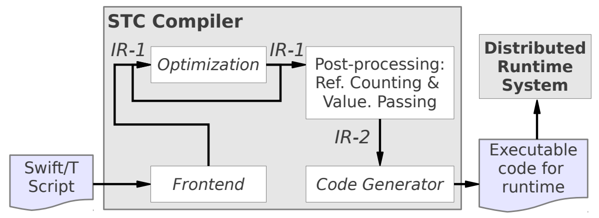


Fig. 3: Runtime architecture. Tasks and data flow across control/server/worker processes in a distributed memory system.



STC Compiler
IR-1
Optimization
IR-1
Post-processing: Ref. Counting & Value. Passing
Distributed Runtime System
IR-2
Swift/T Script
Frontend
Code Generator
Executable code for runtime

Fig. 4: STC compiler architecture showing frontend, intermediate representations, and code generation.

the trees." STC mainly aims to optimization communication so, for example, CPU registers and memory addresses are not exposed. Third, the IR must be low level enough to expose relevant details. STC needs to optimize communication, so the IR makes communication explicit, with explicit task creation and a distinction between local and nonlocal data.

V. Compiler Optimization

We have implemented a significant suite of optimizations in the STC compiler. Several novel optimizations have also been implemented that are specific to task parallel execution models. We have described and evaluated several approaches that, in essence, rearrange the relationship of tasks within the IR to reduce task creation and data operations without reduction in parallelism. The remainder of optimizations are adapted from techniques used for low-level imperative languages such as C or Fortran [6]. These optimizations translate naturally to the task parallel context, since they focus on identifying and eliminating redundant operations, which serves to reduce communication as well as machine instructions. Our contribution is in adapting these optimizations for the STC IR in particular and for data-driven task parallelism in general. Space does not permit description of each optimization. Grouped by optimization level, the optimizations are:

O0: Naïve compilation strategy with no optimization

O1: Basic optimizations: global value numbering, constant folding, dead code elimination, 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 algebra

VI. Distributed Reference Counting

Automatic memory management is a requirement for most high-level languages, and efficient implementation is challenging [4]. Swift/T is no exception. The Swift language does not require users to manually allocate and free memory. Therefore, it must automatically reclaim unused memory once all references to it are lost. References to data can be held by many distributed processes, requiring a distributed, automatic memory management approach. The Swift/T runtime system supports *distributed reference counting*, with reference counts stored for every item of data.

Naïve approaches to reference counting in a distributed setting can have high overhead, with reference count manipulation operations more than doubling the number of messages exchanged. Thus, one focus of our research has been to reduce memory management overhead in Swift/T through compiler optimization. STC's IR supports explicit annotation of data lifetime (e.g., if ownership is passed to a child task). It also includes standalone operations for manipulating reference counts, plus support for "piggybacked" reference counts on data operations. Several techniques have been implemented for reducing reference counting overhead, including merging, canceling, and batching the operations.

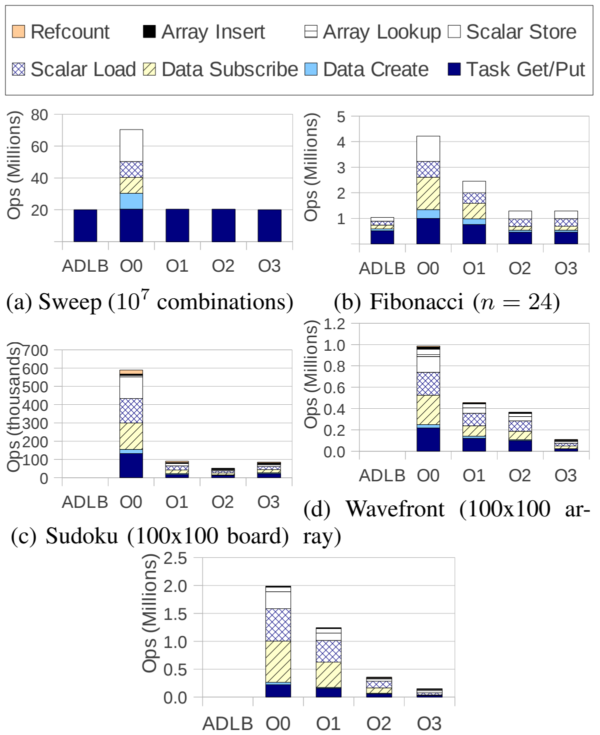
VII. Evaluation

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 divide-and-conquer application based on a recursive Fibonacci calculation with a custom calculation at each node. **Sudoku** is a divide-and-conquer Sudoku solver comprising of ~50 lines of Swift/T and ~800 lines of C that recursively divides and prunes the solution space and terminates early when a solution is found. **Wavefront** is a synthetic application in which 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 lines of C++ that implements an iterative optimization algorithm.

These applications were compiled at each optimization level. For two applications, hand-coded versions using the underlying **ADLB** runtime library [5] were used as a baseline.

We measure the impact of compiler optimization in two ways. First, we measure communication by counting runtime operations, each of which has approximately the same cost: a message round-trip to a server process. For most applications, reduced communication should directly lead to improved scalability and performance, since communication and synchronization between tasks is the typical bottleneck for scaling. Second, we measure application speedup at scale to confirm that reduction in communication translated to improved performance.

We emphasize that these are preliminary results, and do not reflect ongoing minor and major improvements to Swift/T.



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

A. *Communication Reduction from Compiler Optimization*

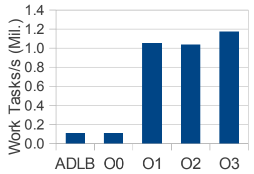
Figure 5 shows the cumulative impact of each optimization level on the number of runtime operations. Overall we see that all applications benefit markedly from basic optimization, while more complex applications benefit greatly from each additional optimization level. With all optimizations enabled, optimized Swift code is comparable to hand-coded ADLB.

B. *Application Speedup from Compiler Optimization*

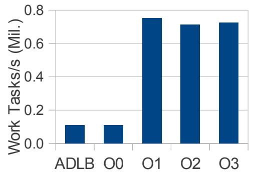
Application speedup benchmarks were run on a Cray XE6 supercomputer with 24 cores per node. Unless otherwise indicated, 10 nodes were used for benchmarks. We measure throughput in tasks per second dispatched to worker processes; this metric captures how efficiently the Swift/T system is able to distribute work and hand control to user code.

Figure 6 shows application speedup. 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 most challenging problem completed in $< 2s$: too short for reliable timings.

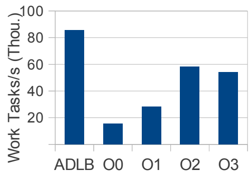
With each benchmark, operation count reductions in Figure 5 gave a roughly proportional increase in throughput. In Wavefront, the speedup was more than proportional to the reduction in runtime operations: the unoptimized code excessively taxed the data-dependency tracking at runtime, causing bottlenecks to form around some data. This result supports the



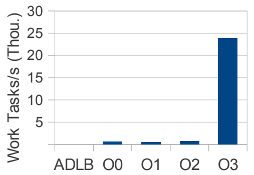
(a) Sweep: $10^7 \times$ 0s tasks



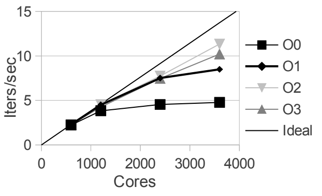
(b) Sweep: $10^7 \times$ 0.2ms tasks



(c) Fibonacci: $n = 34$, 0.2ms tasks



(d) Wavefront: 100x100, 0.2ms tasks



(e) Simulated Annealing: 10 parameters $\times$ 25 iters $\times$ 1000 $\approx$ 0.25s tasks

Fig. 6: Throughput at different optimization levels measured in application terms: tasks/sec, or annealing iterations/sec.

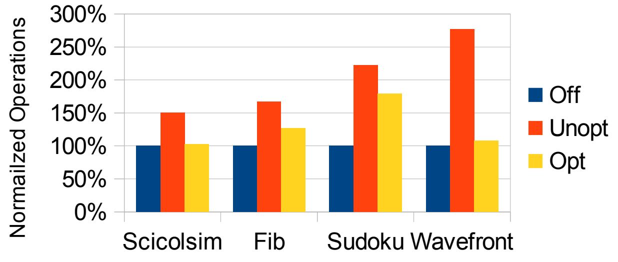


Fig. 7: Impact of automatic memory management on runtime operations. Values are normalized relative to the baseline **Off**.

hypothesis 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 differences in program complexity leading to varying ratios of runtime data/task operations to work tasks.

In comparing hand-coded with Swift/T versions we note some anomalies. Scaling of hand-coded ADLB on Sweep was poor, since it relied on a single process to create all work tasks. The Swift/T version, in comparison, scaled well because it was able to automatically parallelizing work task creation between all control processes. In contrast, the hand-coded Fibonacci program outperformed the Swift/T version because it spread creation and execution of tasks between *all* worker and control processes. Figure 6e shows strong scaling for the Simulated Annealing benchmark. The O2 and O3 optimization levels demonstrate greatly improved scalability from the O0 baseline.

C. Reference Counting

We also performed experiments to determine the performance overhead of automatic memory management. We ran the same benchmarks under three configurations, with all other optimizations enabled: **Off**, where memory management is disabled; **Unopt**, with reference counting optimizations disabled; and **Opt**, with those optimizations enabled. Figure 7 shows the results. The Sweep benchmark is omitted since at O3 no shared data was allocated. The results demonstrate the effectiveness of the reference counting optimizations. The overhead imposed by memory management is reduced to 2.5%-25% for three out of four benchmarks. Sudoku is an exception because reference counting optimizations do not yet handle struct data types.

VIII. Conclusion and Future Work

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 in achieving this aim.

Swift/T offers a powerful solution for rapid application development and scaling. It has been used for science applications running on up to 8,000 cores in production and over 100,000 cores in testing. The work presented in this paper was essential to reaching this scale.

Future work will further improve on the results reported here. We have identified opportunities for significant further communication reduction with efficient data representations and more intelligent optimization of iterative loops. Limitations of reference counting optimizations for struct data types have also been addressed following the experiments presented.

Acknowledgments

This research is supported in part by the U.S. DOE Office of Science under contract DE-AC02-06CH11357, NSF grant ACI-1148443, and by NIH grant S10 RR029030-01 through computing resources provided by the Computation Institute and the Biological Sciences Division of the University of Chicago and Argonne National Laboratory.

References

[1] 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*.

[2] 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*.

[3] F. Chow. Intermediate representation. *ACM Queue*, 11(10), Oct. 2013.

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

[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 Review*, 17:30–37, Jan. 2010.

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

[7] S. Taşirlar and V. Sarkar. Data-Driven Tasks and their implementation. In *Proc. ICPP '11*.

[8] 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*.

[9] 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*.

[10] 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*.