Open access • Proceedings Article • DOI:10.1109/SC.2014.30

Compiler techniques for massively scalable implicit task parallelism — Source link

Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, Ian Foster

Institutions: University of Chicago, Argonne National Laboratory

Published on: 16 Nov 2014 - IEEE International Conference on High Performance Computing, Data, and Analytics

Topics: Implicit parallelism, Task parallelism, Data parallelism, Compiler and Explicit parallelism

Compiler Techniques for Massively Scalable Implicit Task Parallelism

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*—Swift/T is a high-level language for writing concise, deterministic scripts that compose serial or parallel codes implemented in lower-level programming models into large-scale parallel applications. It executes using a data-driven task parallel execution model that is capable of orchestrating millions of concurrently executing asynchronous tasks on homogeneous or heterogeneous resources. Producing code that executes efficiently at this scale requires sophisticated compiler transformations: poorly optimized code inhibits scaling with excessive synchronization and communication. We present a comprehensive set of compiler techniques for data-driven task parallelism, including novel compiler optimizations and intermediate representations. We report application benchmark studies, including unbalanced tree search and simulated annealing, and demonstrate that our techniques greatly reduce communication overhead and enable extreme scalability, distributing up to 612 million dynamically load balanced tasks per second at scales of up to 262,144 cores *without* explicit parallelism, synchronization, or load balancing in application code.**

I. Introduction

In recent years, large-scale computation has become an indispensable tool in many fields, including those that have not traditionally used high-performance computing. These include data-intensive applications such as machine learning and scientific data crunching and compute-intensive applications such as high-fidelity simulations.

The traditional development model for high-performance computing requires close cooperation between domain experts and parallel computing experts to build applications that efficiently run on distributed-memory systems, with careful attention given to low-level concerns such as distribution of data, load balancing, and synchronization. Many real-world applications, however, are amenable to generic approaches to these concerns. In particular, many applications are naturally expressed with *data-driven task parallelism*, in which massive numbers of concurrently executing tasks are dynamically assigned to execution resources, with synchronization and communication handled using 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. Data-driven task parallelism can expose more parallelism than can alternative models such as fork-join [29], and it addresses challenges of utilizing heterogenous, distributed-memory resources with transparent data movement between devices and dynamic data-aware task scheduling. 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], [28] and has shown that performance can match or exceed performance of code directly using the underlying interfaces (e.g., message passing or threads). One reason for this success is that sophisticated algorithms for load balancing (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 [36] is a high-level implicitly parallel programming language that aims to make writing massively parallel code for this execution model as easy and intuitive as sequential scripting in languages such as Python. Implementing a very high-level language such as Swift/T efficiently and scalably is challenging, however, because the programmer only specifies synchronization and communication implicitly through function composition or reads and writes to variables and data structures. Thus, internode data movement, parallel task management, and memory management are left entirely to the language's compiler and runtime system. Since large-scale applications may require execution rates of hundreds of millions of tasks per second on many thousands of cores, this complex coordination logic must be implemented both efficiently and scalably.

For this reason, we have developed, adapted, and implemented a range of compiler techniques for data-driven task parallelism, presented here. By optimizing the use of a distributed runtime system's operations, communication and synchronization overhead is reduced by an order of magnitude. In addition to this primary outcome of the work, we make the following technical contributions:

- Characterization of the novel compiler optimization problems arising in data-driven implicit task parallelism.
- Design of an intermediate representation for effective optimization of the execution model.
- Novel compiler optimizations that reduce coordination costs by an order of magnitude.
- Compiler techniques that achieve low-overhead distributed automatic memory management at massive scale.

SC14, November 16-21, 2014, New Orleans, Louisiana, USA
978-1-4799-5500-8/14/$31.00

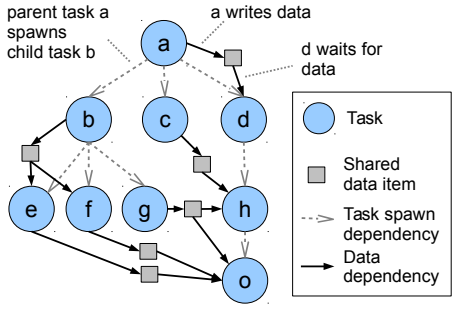


Fig. 1: Task and data dependencies in data-driven task parallelism, forming a spawn tree rooted at task a. Data dependencies on shared data defer execution of tasks until the variables are frozen.

II. Data-driven Task Parallelism and Swift/T

We introduce the *data-driven task parallelism* execution model (Section II-A), show how it is programmable with the high-level Swift/T language (Section II-B), and describe a massively scalable implementation (Section II-C).

A. Abstract Execution Model

In data-driven task parallelism, a program is organized into *task definitions* with explicit inputs. A *task* is a runtime instantiation of a task definition with inputs bound to specific data. Once executing, tasks run to completion and are not preempted.

Each task can spawn asynchronous *child tasks*, resulting in a *spawn tree* of tasks as in Figure 1. We assume support for *shared data*: data items that can be read or written by any task that obtains a reference to the data. Parent tasks can pass data to their child tasks at spawn time, for example small data such as numbers or short strings, along with references to arbitrary shared data. Shared data items provide a means for coordination between multiple tasks. For example, a task can spawn two tasks, passing both a reference to a shared data item, which one task reads and the other writes. *Data dependencies*, which defer the execution of tasks, are the primary synchronization mechanism. The execution model permits a task to write (or not write) any data it holds a reference to, allowing many runtime data dependency patterns beyond static task graphs.

The execution model is much lower level than high-level programming models such as the Swift/T language discussed in the next section. There is no high-level syntax, and safety guarantees are limited. A task can execute arbitrary code that performs arbitrary computation, I/O, and runtime operations such as spawning tasks or reading/writing data. This makes invalid, non-deterministic, or otherwise unsafe behavior possible. For example, race conditions are possible if shared data is read without synchronizing using data dependencies. Explicit bookkeeping is also needed for both memory management and correct freezing of variables. Programming errors could result in memory leaks, prematurely freed data, or deadlocks. Many other more restrictive task-parallel programming models, such as task graphs or fork-join parallelism, can be expressed with these basic constructs, so optimizations for this model are broadly applicable.

B. Overview of Swift/T Programming Language

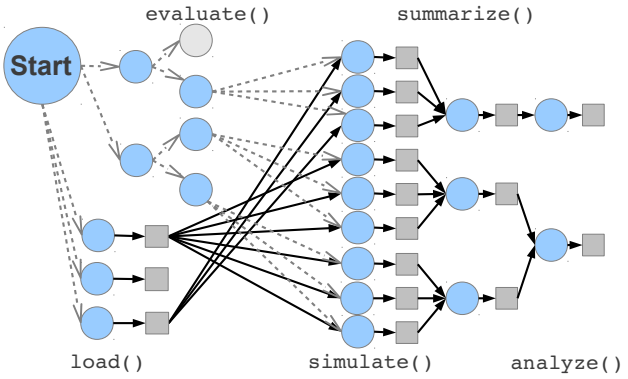
The overall Swift/T system has been described in previous work [36], so we focus here on language semantics relevant to compiler optimization. The Swift/T language's syntax and semantics are derived from the Swift language [34]. Swift/T focuses on high-performance fine-grained task parallelism, such as calling foreign functions (including C and Fortran) with in-memory data and launching kernels on GPUs and other accelerators [16]. These foreign functions are integrated into the Swift/T language as typed *leaf functions* that encapsulate computationally intensive code, leaving parallel coordination, task distribution, and data dependency management to the Swift/T dataflow programming model. Figure 2 illustrates how leaf functions can be composed into an application, with complexities such as data-dependent control flow expressible naturally in the language.

The Swift/T language is a global-view implicitly parallel language, meaning that, by default, execution order of statements is constrained only by data dependencies, and that execution location is left to language implementation, with program variables logically accessible to code regardless of where it executes. That is, program logic can be expressed without explicit concurrency, communication, or data partitioning. Certain control structures, including conditionals and explicit *wait* statements, add additional control flow dependencies to code, while annotations can provide hints or constraints for data or task placement. Two types of loops are available: *foreach* loops, for parallel iteration, and *for* loops, where iterations are pipelined, with data passed from one iteration to the next. Swift/T also supports unbounded recursion.

Swift/T can guarantee deterministic execution even with implicit parallelism because its standard data types are *monotonic*; that is, they cannot be mutated in such a way that information is lost or overwritten. A monotonic variable starts off empty, then incrementally accumulates information until it is *frozen*, whereupon it cannot be further modified. One can construct a wide variety of monotonic data types [11], [17]. Basic Swift/T variables are single-assignment I-vars [21], which are frozen when first assigned. Composite monotonic data types can be incrementally assigned in parts but cannot be overwritten. Programs that attempt to overwrite data will fail at runtime (or compile time if the compiler determines that the write is definitely erroneous). Swift/T programs using only monotonic variables are deterministic by construction, up to the order of side-effects such as I/O. For example, the output value of an arbitrarily complex function involving many data and control structures is deterministic, but the order in which debug print statements execute depends on the nondeterministic order in which tasks run. Further nondeterminism is introduced only by non-Swift/T code, library functions such as `rand()`, or by rarely-used nonmonotonic data types that are outside the scope of this paper.

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

(a) Implicitly parallel Swift/T code.



(b) Visualization of optimized parallel tasks and data dependencies for parameters $M = 2$ $N = 2$ $S = 3$. Tasks and data are mapped dynamically to compute resources at runtime.

Fig. 2: An application – an amalgam of several real scientific applications – that runs an ensemble of simulations for many parameter combinations. The code executes with implicit parallelism, ordered by data dependencies. Data dependencies are implied by reads and writes to scalar variables (e.g. `p` and `m`) and associative arrays (e.g. `models` and `res`). Swift/T semantics allow functions (e.g. `load`, `evaluate`, and `simulate`) to execute in parallel when execution resources are available and data dependencies are satisfied.

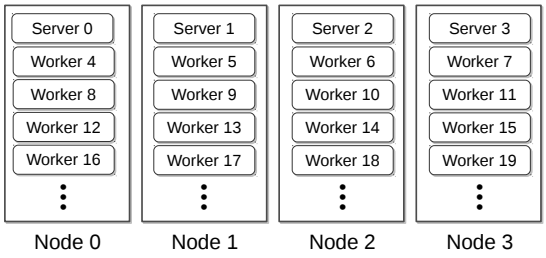


Fig. 3: Runtime process layout on distributed-memory system. Processes are divided into workers and servers, which are then mapped onto the processes of multi-core systems.

The sparse dynamically sized *array* 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 parts (e.g., `int A[]; A[i] = a; A[j] = b;`). The array lookup operation `A[i]` will return when `A[i]` is set. An incomplete array lookup does not prevent progress; other statements can execute concurrently.

C. Massively Scalable Data-Driven Task Parallelism

The ADLB [18] and Turbine [35] runtime libraries provide the runtime support for massively scalable data-driven task parallelism on a MPI-2 or MPI-3 communication layer [31].

In this runtime system, MPI processes are divided into two roles: *workers* and *servers*, which can be laid out in various ways, for example with one server process allocated to each shared-memory node, as shown in Figure 3. Worker processes execute any program logic, coordinating with each other through remote execution of data and task operations

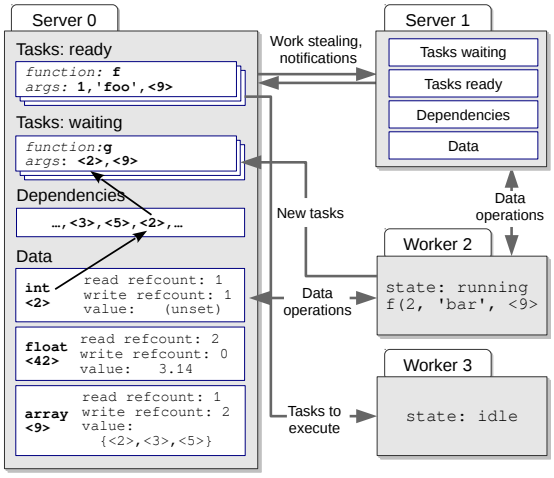


Fig. 4: Runtime architecture showing distributed worker processes coordinating through task and data operations. Ready/waiting tasks and shared data items are stored on servers, with each server storing a subset of tasks and data. Servers must communicate to redistribute tasks through work-stealing, and to request/receive notifications about data availability on other servers.

on servers, as shown in Figure 4. These operations are low-latency, typically taking microseconds to process, which minimizes delays to worker processes. If needed, parallel MPI functions can be executed by worker processes that are dynamically grouped into "teams" with a shared MPI communicator [37].

The data functionality includes rich data structures such as scalar values, strings, binary blobs, structs, and associative arrays, providing the primitives needed to implement

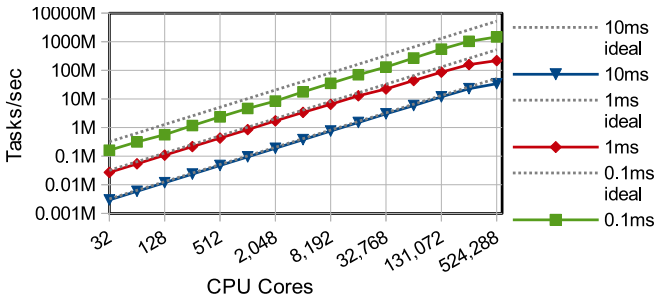


Fig. 5: Throughput and scaling of runtime system for varying task durations.

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

Fig. 6: Swift code fragment illustrating wavefront pattern.

Swift's monotonic data types as shared data items. Memory management of this data is supported using read and write reference counters for each data item, allowing unused data to be deleted and frozen data to be read-only. The task functionality implements a scalable distributed task queue, with load balancing using randomized work stealing between servers. Task data dependencies are supported, so that tasks can be released when data is frozen, at the granularity of an entire data structure or individual array subscripts, as shown in Figure 4. Figure 5 illustrates the scalability and task throughput of Swift/T programs using the runtime system on the Blue Waters supercomputer, where Swift/T achieved a peak throughput of 1.47 billion tasks/s on 524,288 cores running the Sweep benchmark described later in Section IV. Tasks of 1 ms or more achieve high efficiency the servers are lightly loaded and queuing delays are minimal.

III. Compiler Optimization

STC is a whole-program optimizing compiler for Swift/T that targets the distributed runtime described previously. Within STC we have implemented optimizations aimed at reducing communication and synchronization without loss of parallelism (Section III-A). An intermediate representation for the program captures the execution model (Section III-B), allowing optimization of synchronization, shared data, and reference counting (Sections III-C, III-E, III-F, respectively).

A. Optimization Goals for Data-driven Task Parallelism

To optimize a wide range of data-driven task parallelism patterns, we need compiler optimization techniques that can understand the semantics of task parallelism and monotonic variables in order to perform major transformations of the task structure of programs to reduce synchronization and communication at runtime, while preserving parallelism. Excessive runtime operations impair program efficiency because tasks waste time waiting for communication; they can also impair scalability by causing bottlenecks for data or task queues.

The implicitly parallel Swift/T code in Figure 6 illustrates the opportunities and challenges of optimization. The code

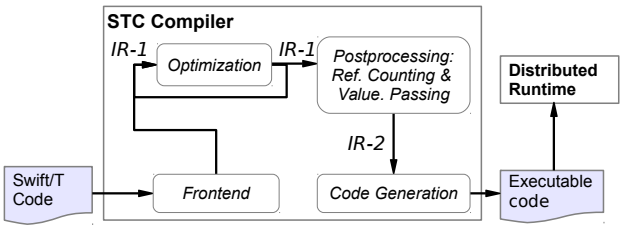


Fig. 7: STC compiler architecture. The frontend produces IR-1, to which optimization passes are applied to produce successively more optimized IR-1 trees. Postprocessing adds intertask data passing and reference counting information to produce IR-2 for code generation.

specifies a dynamic, data-driven wavefront pattern of parallelism, where evaluation of cell values is dynamically scheduled based on data availability at runtime, allowing execution to adapt to variable task latencies. Two straightforward transformations give immediate improvements: representing input parameters such as `i` and `j` as regular local variables rather than shared monotonic variables and hoisting the lookups of `A[i-1]` and `A[i]` out of the inner loop body. The real challenge, however, is in efficiently resolving implied data dependencies between loop iterations. The naïve approach uses three data dependencies per input cell; but with this strategy, synchronization can quickly become a bottleneck. Smarter approaches can identify common inputs of neighboring cells to avoid redundant data reads, or defer task spawns until input data is available: if the task for $(i-1, j)$ spawns the task for (i, j), only grid cell $A[i][j+1]$ must be resolved at runtime since both other inputs were available at $(i-1, j)$. The characteristics of the `f`, `g`, and `h` functions also affect performance of different parallelization schemes. Fusing `f` and `g` invocations is a clear improvement because no parallelism is lost; but, depending on function runtimes and other factors, the optimal parallel structure is not immediately obvious. To maximize parallelism, we would implement the loop body invocations as three independent `f(g(...))` tasks that produce the input data for a `h(...)` task. To minimize runtime overhead, on the other hand, we would merge these four tasks into a single task that executes the `f(g(...))` calls sequentially.

B. Intermediate Representation

The STC compiler uses a medium-level intermediate representation (IR) that captures the execution model of data-driven task parallelism. Two IR variants are used by stages of the compiler (Figure 7). IR-1 is generated by the compiler frontend and then optimized. IR-2 includes additional information for code generation: explicit bookkeeping for reference counts and data passing to child tasks. Sample IR-1 code for a parallel, recursive Fibonacci calculation is shown in Figure 8.

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, simple-to-analyze representation. Control flow is represented with high-level structures: *if* statements, *foreach* loops, *do/while* loops, and so forth. The statements in each block

execute sequentially, but blocks within some control-flow structures execute asynchronously and some IR instructions spawn asynchronous tasks. Data-dependent execution is implicit in some asynchronous IR instructions or explicit with *wait* statements that execute a code block after a set of variables is frozen.

Variables are either single-assignment locally stored values or references to shared data, that is, unique identifiers used to locate data on a remote process. A reference is either the initial reference to a variable allocated in the block or an alias obtained by duplicating a reference, by acquiring a reference stored in a data structure, or by subscripting a composite variable. Shared monotonic data is a first-class construct in the IR, so optimizations can exploit monotonic semantics.

C. Adaption of Traditional Optimizations

The foundation of our suite of optimizations is a range of traditional optimization techniques adapted from conventional compilers [19] to our intermediate representation and execution model in general. This required substantial changes to many of the techniques, particularly to generalize them to monotonic variables, and also to be able to effectively optimize across task boundaries and with concurrent semantics.

STC includes a powerful *value numbering* [19] (VN) analysis. that discovers *congruence relations* in an IR function between various expression types, including variables, array cells, constants, arithmetic expressions, and function calls. Annotations on functions, including standard library and user functions, assist this optimization. For example, the annotation `@pure` asserts that a function output is deterministic, and it has no side-effects. The VN pass identifies congruence relations for each IR block. Value congruence, for example, $retrieve(x) \cong^V y * 2 \cong^V 6$, means that multiple expressions have the same value. Alias congruence, for example $y \cong^A z \cong^A A[0]$, means that IR variables refer to the same runtime shared data. Alias congruence implies value congruence. A relation for a block B applies to B and all descendant blocks, because of the monotonicity of IR variables. A set of expressions congruent in B defines a *congruence class*.

STC's VN implementation visits all IR instructions in an IR function with a reverse postorder tree walk. Each IR instruction, for example, `StoreInt A 1`, can yield congruence relations: in this case $A \cong^V store(1)$ and $1 \cong^V retrieve(A)$. These new relations are added to the known relations, perhaps merging existing congruence classes. For example, if $B \cong^V store(1)$, then $A \cong^V B$. Erroneous user code that double-assigns a variable forces VN to abort, since the correctness of the analysis depends on each variable having a consistent value. Congruence relations in a block always apply to descendant blocks. We also propagate congruence relations upward to parent blocks in the case of conditional statements. For example, if $x \cong^V 1$ on both branches of an if statement, it is propagated to the parent. We create temporary variables if necessary to do this, for example, if $x \cong^V retrieve(A)$ and $y \cong^V retrieve(A)$ on the branches, a

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

(a) Swift/T code for recursive Fibonacci.

```
1  () @main () {
2    declare $int v_n, int f //  variables for block
3    CallExtLocal argv [ v_n ] [ "n" ]  //  call to argv
4    Call fib [ f ] [ v_n ] //  fib runs asynchronously
5    wait (f) { //  execute block once f is frozen
6      declare $int v_f
7      LoadScalar v_f f //  Load value of f to v_f
8      CallExtLocal printf [ ] [ "fib(%i)=%i" v_n v_f ]
9    }
10 }
11
12 //  Compute o := fibonacci(i)
13 //  input v_i is value, output o is shared var
14 (int o) @fib ($int v_i)
15   declare $boolean t0
16   LocalOp <eq_int> t0 v_i 0    //  t0 := (v_i == 0)
17   if (t0) {
18     StoreScalar o 0            //  o := 0
19   } else {
20     declare $boolean t2
21     LocalOp <eq_int> t2 v_i 1 //  t2 := v_i + 1
22     if (t2) {
23       StoreScalar o 1          //  o := 1
24     } else {
25       declare $int v_i1, $int v_i2, int f1, int f2
26       LocalOp <minus_int> v_i1 v_i 1 //  v_i1 := v_i - 1
27       //  Fib calls run asynchronously
28       Call fib [ f1 ] [ v_i1 ]
29       LocalOp <minus_int> v_i2 v_i 2 //  v_i2 := v_i - 2
30       Call fib [ f2 ] [ v_i2 ]
31       //  Compute sum once f1, f2 assigned
32       AsyncOp <plus_int> o f1 f2      //  o := f1 + f2
33     }
34   }
35 }
```

(b) IR-1 optimized at -O2.

Fig. 8: Sample Swift/T program and corresponding IR for recursive Fibonacci algorithm. The IR comprises two functions: `main` and `fib` functions. IR instructions include Swift function calls (e.g. `Call fib`), foreign function calls (e.g. `CallExtLocal printf`), immediate arithmetic operations (e.g. `LocalOp <eq_int>`), data-dependent arithmetic operations (e.g. `AsyncOp <minus_int>`), and reads and writes of shared data items (`LoadScalar` and `StoreScalar`, respectively). Control flow constructs used include conditional `if` statements, and `wait` statements for data-dependent execution.

new variable t_1 is assigned x and y on the respective branches, so that $t_1 \cong^V retrieve(A)$ in the parent.

After the initial VN analysis, IR transformations can use the congruence information. The basic VN optimization replaces variables with the canonical congruence class member: inputs using value congruence classes, and outputs using alias congruence classes. The canonical member is chosen based on the expression type (e.g., constants are preferred) and other factors (e.g., the first variable to be computed is preferred). Variables are thereby replaced with constants, and redundant computations or shared data loads can be avoided.

STC's VN analysis supports *constant folding* [19], where expressions with constant arguments can be evaluated during the VN tree walk. Constant results can then be propagated by using congruence relations, allowing constant folding of further expressions and merges of congruence classes. STC supports binding key-value command-line arguments to constants to compile specialized versions of a program.

Dead code elimination (DCE) eliminates unneeded code that is never executed or that computes unneeded results. This includes both unexecuted user code and dead code from earlier optimization passes. VN, for example, eliminates uses of redundant variables but depends on DCE to later eliminate any IR instructions that became redundant as a result.

Function inlining is an important optimization that creates interprocedural optimization opportunities for later passes and eliminates function call overhead. Functions with a single call site are always inlined. Otherwise, a simple heuristic is used: *function IR instruction count* $\times$ *# call sites* $<$ *500*. Cycles of recursive calls are identified in order to avoid infinite cycles of inlining. Entire programs can often be inlined into the main function, allowing full interprocedural optimization.

Several loop optimizations are implemented. *Loop invariant hoisting* is important for many Swift/T scripts, where redundant operations such as array accesses occur inside nested parallel foreach loops. *Loop fusion* fuses foreach loops with identical bounds to reduce runtime loop management overhead and allow optimization across loop bodies. *Loop unrolling* completely expands foreach loops with <16 iterations, and unrolls the rest by up to a a factor of 8x, limited by a simple heuristic that caps code-size growth at 256 IR instructions per unrolled loop. The main benefit of unrolling in parallel coordination code is to allow optimization across iterations.

Instruction reordering constructs a dataflow graph of IR instructions in a block and attempt to place readers of monotonic variables after writers to aid further optimization.

D. Shared Monotonic Data Optimizations

We devised further optimizations that exploit the properties of shared monotonic data in order to reduce runtime operations and to assist other optimizations by simplifying the IR.

Frozen variable analysis (FVA) detects which I-vars, monotonic arrays, and so forth are frozen at each statement. A variable is frozen if an IR instruction freezes it directly (e.g., writing an I-var) or within a *wait* statement for that variable. Alias congruence relations from VN are used to enhance the analysis. Data dependency analysis also allows freezing to be inferred in further situations. For example, if I-var `x` is the output of an operation with input `y`, then `y` must be frozen inside `wait(x) { ... }`. FVA allows inlining of *wait* continuations and *strength reduction*, whereby statements using expensive runtime data or task operations are replaced with ones that use fewer or no runtime operations, for example by skipping runtime data-dependency checks or executing an operation within the current task context.

Store coalescing combines writes to shared composite data types such as arrays and structs into single store operations. It is applied when a variable is written multiple times in a block, for example if multiple indices of an array are assigned.

Argument localization addresses inefficiencies in the default function calling convention, where arguments are passed as references to shared data that may not be frozen, which often leads to unnecessary data dependencies, reads, and writes to shared data. This overhead is significant, especially for short functions. The same essential problem exists with values passed between sequential loop iterations. We address this problem with an analysis that identifies when code cannot make progress without an input being frozen. The code is transformed so that the input is passed as a regular value, rather than a reference to shared data, and then add wait statements to function call sites where necessary.

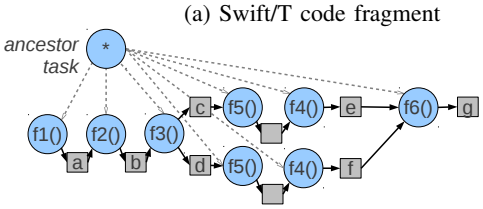
E. Task Parallelism Optimizations

We implemented a further set of transformations, specific to data-driven task parallelism, that rearrange the task structure of the program to reduce runtime operations and assist further optimization. These must avoid reducing *worthwhile parallelism* that has granularity to justify task creation overhead.

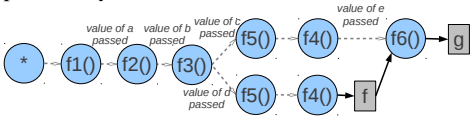
Two properties of IR instructions can decide whether a transformation reduces worthwhile parallelism. First, an instruction is *long running* if it executes synchronously in the current task for a long or unbounded time. STC's optimization passes avoid serializing execution of long-running instructions that could run in parallel, but attempt to merge short-running tasks where the overhead of parallel execution is not justified. The exact boundary between short and long-running instructions is difficult to define and somewhat arbitrary. In practice, however, most instructions fall clearly in one category or the other: simple built-in functions such as arithmetic, string operations, and runtime operations versus computationally intensive user tasks. Second, an instruction is *progress enabling* if execution of the instruction may fulfill data dependencies of other tasks: for example, a shared data write. The optimizer avoids deferring execution of progress enabling code by a long or unbounded amount of time. For example, it will not add direct or indirect dependencies from a long-running instruction to a progress-enabling instruction. Several criteria are used by the optimizer to determine if an intermediate representation instruction is long running or progress enabling. Any function call or built-in operations that is not explicitly specified as short running by hardcoded compiler rules or a user function annotation is assumed to be

```
1 | a = f1();          b = f2(a);
2 | c, d = f3(a, b);   e = f4(f5(c);
3 | f = f4(f5(d);      g = f6(e, f);
```

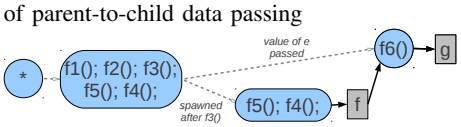
(a) Swift/T code fragment



(b) Unoptimized version, passing data as shared data and perform synchronization



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



(d) After pipeline fusion merges tasks

Fig. 9: Traces of execution showing optimization of task and data dependencies in a Swift/T code fragment.

long running. Execution of a long running instruction counts as progress, so all long running instructions are progress enabling. The optimizer also conservatively assumes that any operation that assigns a dataflow variable releases parallel work and is therefore progress-enabling.

Asynchronous op inlining is a variant of inlining where an asynchronous built-in operation (e.g., an arithmetic operation or array lookup) is expanded to a *wait* statement plus non-asynchronous IR instruction, allowing later optimizations to manipulate task structure.

Task coalescing is a family of techniques that reconfigure the IR task structure. One effective technique, which we call *task pushdown*, is to resolve data dependencies between tasks by relocating statements, such as wait statements and data-dependent IR instructions, to descendant blocks in the IR tree where input variables are assigned. This can enable the sequence of transformations in Figure 9c, where VN, FVA, and DCE eliminate shared data items, completing conversion of data dependency to task spawn edges. Task coalescing also merges tasks, for example nested or sibling wait statements, when it can determine that the transformation will not prevent progress at runtime.

Another optimization is *pipeline fusion*, illustrated in Figure 9d. A commonly occurring pattern is a sequentially dependent set of function calls: a "pipeline." We can avoid runtime task dispatch overhead and data transfer without any reduction in parallelism by fusing a pipeline into a single task. For tasks with short duration or large input/output data, this method saves much overhead. As a generalization, a fused task will spawn dependent tasks if a pipeline "branches." STC's pipeline fusion was inspired by the like-named technique from streaming languages [12], which is similar in concept but not similar in implementation. Streaming languages have static task graphs with dynamic flows of data, whereas data-driven task parallelism has dynamic task graphs with discrete data. In streaming languages, pipeline fusion trades off pipeline parallelism for lower overhead. In Swift/T, there is no pipeline parallelism to lose, but the more dynamic execution model requires more analysis to identify valid opportunities.

F. Memory Management and Freezing Optimizations

Efficient memory management is a challenging issue in a distributed, parallel context, especially in the highly dynamic execution model of data-driven task parallelism, because references to data may be spread across a large number of concurrent processes. The classic memory management problem is generally formulated as the problem of detecting when no direct or indirect references to a data item are held by executing program. In a programming model with monotonic data, such as Swift/T, the variable freezing problem can be formulated similarly as detection of when no remaining references to a data item will be used to write to it (i.e., are read-only references).

We tackle both problems with *automatic distributed reference counting*, by giving each shared data item read/write reference counts (*refcount*). When the write refcount drops to zero, the variable is frozen and cannot be written; when both refcounts 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 freeze it. In the case of arrays, the compiler must determine which statements may read or write each variable, and write refcounts are incremented and decremented as active tasks that could modify the array are spawned and complete. A well-known weakness of reference counting is that it cannot handle cycles of references. The Swift/T data model does not permit such reference cycles, so this problem is avoided.

Two postoptimization passes over the IR add all necessary refcount 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 postorder walk over the IR tree to add reference counting operations.

Naïve reference counting strategies that update reference counts every time a reference is passed into an instruction or goes out of scope are problematic: even in seemingly straightforward code, these reference count updates can more than double the number of data operations executed.

STC uses a range of techniques to address this problem. The basic analysis computes the number of read/write increments/decrements (*incrs/decrs*) in each IR block (i.e. four integers per block). Increments accrue from copied references to a variable (e.g., if a reference is passed to an instruction or into an annotated child block), and decrements accrue from references that is acquired from newly initialized variables, variables passed from the the parent block, or from instructions that return references. Refcount operations are placed in the block only after this counting process completes. The basic

placement strategy puts incrs and decrs at the start and end of the block, respectively, thus ensuring that refcounts are not dropped to zero too early during execution.

This framework supports several optimizations. *Merging* incrs/decrs is achieved by the use of counters and an alias analysis that detects when two program variables refer to the same reference counted data item. *Cancellation* of incrs/decrs can happen, for example, when an incr for a reference passed to a single child task cancels out a decr for the variable going out of scope in the parent. This is subject to a pass over the block to verify that the reference is not used after being handed to the child. Refcount incrs or decrs can be *piggybacked* on other data operations, such as variable creation or reads. With a distributed runtime, the piggybacked operation is essentially free because it requires no additional synchronization and minimal additional communication (a few bytes). Unplaced incrs/decrs can be *hoisted* to the parent, subject to conditions: the incr/decr is not in a loop. If in a conditional, the incr/decr must occur on all branches; and if a decr, the child block must be executed synchronously within the parent. In combination, these techniques allow reference counting overhead to be reduced greatly. In cases where the number of readers/writers is determined statically, such as static task graphs, all incrs/decrs are merged, cancelled, or piggybacked, which eliminates the need for reference counting operations. In cases of large parallel loops, reference counting overhead is amortized over the entire loop with batching.

IV. Evaluation of Compiler Optimizations

To characterize the impact of different optimization levels, we chose five benchmarks that capture common patterns of asynchronous task parallelism. **Sweep** is a parameter sweep with two nested loops and completely independent tasks with uneven task durations governed by a log-normal distribution, requiring dynamic assignment of tasks to resources. **ReduceTree** is a synthetic application comprising a massive reduction tree with the same structure as a recursive Fibonacci calculation. At full scale, the results of billions of asynchronously-executing tasks are reduced to a single result. **UTS** (Unbalanced Tree Search) is a benchmark that simulates a recursive search procedure with a highly irregular structure, requiring efficient load balancing [22]. The core of UTS in Swift/T is a six line recursive function that calls into the serial C code performing the UTS computation. The serial code executes until it has processed 1 million tree nodes or accumulated 128 unprocessed nodes. **Wavefront** is an application with the wavefront pattern in Figure 6 that executed a single function call to compute each grid cell, with runtime following a log-normal distribution with mean 5ms. **Annealing** is a science application comprising an iterative simulated annealing optimization algorithm implemented in $\sim$500 lines of Swift/T and a simulation implemented in $\sim$2,000 lines of C++. The objective function of the algorithm is a large ensemble of simulations, with up to 10,000-way parallelism, which is multiplied by the parallelism derived from multiple annealing runs for different parameters. Task runtimes are irregular and vary as a run progresses, requiring highly dynamic load balancing to redistribute tasks, especially to keep workers busy as straggler tasks from each objective function evaluation complete.

We implemented baseline versions of four benchmarks as C programs that directly use the **ADLB** [18] runtime library. These baselines aim to be equivalent to what a knowledgeable user familiar with ADLB would write. We strived to implement the ADLB baselines efficiently and scalably, but in a straightforward manner, that is, without any overly complex parallelization schemes. The Sweep ADLB baseline statically partitioned the outer loops between nodes, with up to four processes per node inserting tasks. The UTS ADLB baseline uses the same heuristics as were used in the Swift/T version and avoids all shared data operations, with each task spawning tasks directly. In the ReduceTree ADLB baseline, each task ($f(n)$) spawned two child tasks to compute $f(n-1)$ and $f(n-2)$, and a third data-dependent task to sum the results. The Wavefront ADLB baseline used a master process to manage data dependencies and launch work tasks.

To pare down Swift/T configurations to a manageable number, we grouped optimizations into four levels: **O0**: naïve compilation strategy with no optimization; **O1**: basic redundancy-reducing optimizations, namely, value numbering, constant folding, dead code elimination, loop fusion, frozen variable analysis, and refcount optimizations; **O2**: more aggressive optimizations, namely, asynchronous op inlining, task coalescing, arg localization, and hoisting; and **O3**: the remaining optimizations, namely, function inlining, pipeline fusion, loop unrolling, and instruction reordering. Automatic memory management was enabled in all cases.

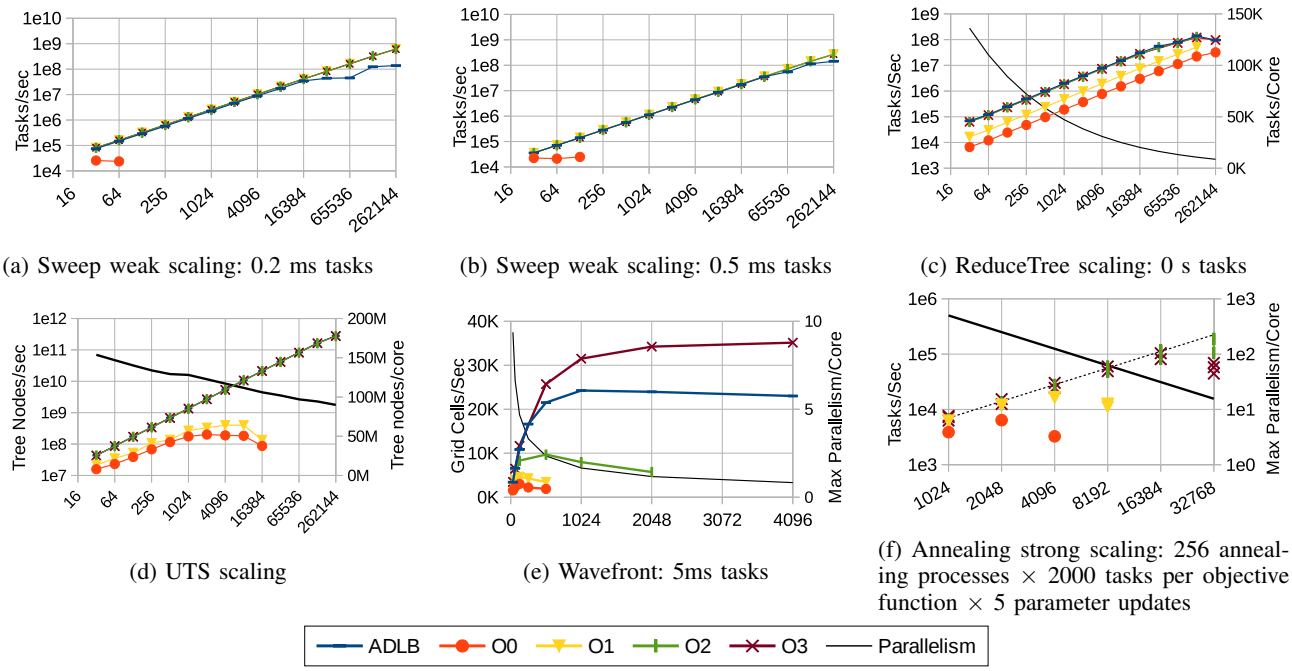
We expect that the O0 baseline configuration will perform poorly because of the nature of the Swift language, where language features such as implicit parallelism, transparent data movement, and monotonic data structures do not have general-purpose implementations that map efficiently to lower-level hardware and software interfaces. O1 is a stronger baseline configuration that includes basic compiler optimizations of the kind that would appear in most compilers. O1 also includes frozen variable analysis because it is critical in supporting the removal of shared data items. The O1 optimizations subsume most basic optimizations that could be implemented in the frontend or code generator, for example specialization of operations with literal constant arguments.

A. Method for Large-Scale Experiments

We evaluated the optimizations by running our benchmark applications at different combinations of scale and optimization levels. A prerelease version of Swift/T 0.6[1] was used for the experiments. We have made the benchmark source code publically available[2], with the exception of our collaborators' annealing code. We ran the benchmarks on the Cray XE6 nodes of the Blue Waters supercomputer [20], which have 2

[1] http://swift-lang.org/Swift-T

[2] https://github.com/swift-lang/exm-stc/tree/master/bench/suite



(a) Sweep weak scaling: 0.2 ms tasks

(b) Sweep weak scaling: 0.5 ms tasks

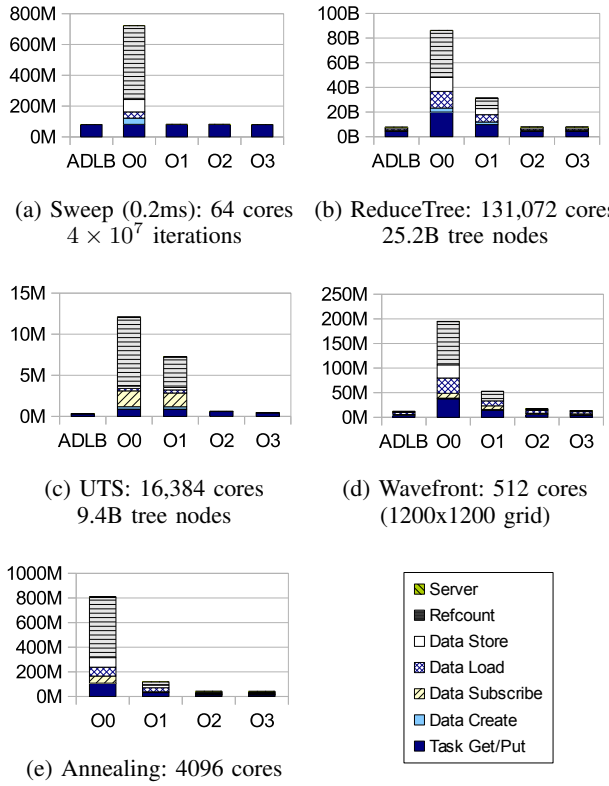
(c) ReduceTree scaling: 0 s tasks

(d) UTS scaling

(e) Wavefront: 5ms tasks

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

Fig. 10: Application speedup and scalability at different optimization levels. X axes show scale in cores. Primary Y axes show application throughput in application-dependent terms. Secondary Y axes show problem size or degree of parallelism where applicable.



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

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

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

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

(e) Annealing: 4096 cores

Fig. 11: Impact of optimizations on # of runtime operations issued to servers. Each additional level of optimizations reduces the number of operations required to execute the Swift program, which can lead to better throughput and scalability.

AMD Interlagos model 6276 CPUs with 32 cores total with clock speeds of >2.3 GHz and 64 GB of memory. We assigned one core per node to act as a server while the remainder were workers. Figure 10 shows application speedup, measured with the metric appropriate to each benchmark to quantify how rapidly and efficiently compiled Swift/T parallel coordination code can generate and distribute work. We increased the scale of each application at each optimization level until it failed to continue scaling.

To better understand the effect of optimizations, we instrumented the servers with performance counters that collect aggregate statistics with low overhead, including counts of each operation type invoked on servers and statistics about tasks and workstealing. Figure 11 shows operation counts summarized into several categories of runtime operations. *Data Creates* create new shared data items, *data loads* and *data stores* read and write them, and *data subscribes* implement notifications for data dependencies. *Task puts* and *task gets* add and remove tasks from the distributed task queue. *Refcount* operations are standalone refcount operations. *Server* operations include workstealing attempts and other communication between servers.

B. Discussion and Analysis of Large-Scale Experiments

These experimental results show that all applications benefit markedly from basic optimization at O1, but further optimizations often, but not always, provide further benefits of similar magnitude. By comparing Figure 10 with Figure 11, we see that reduction in operation counts leads directly to application speedup. This means that the the effectiveness of the compiler optimizations is not specific to our runtime system: some or all

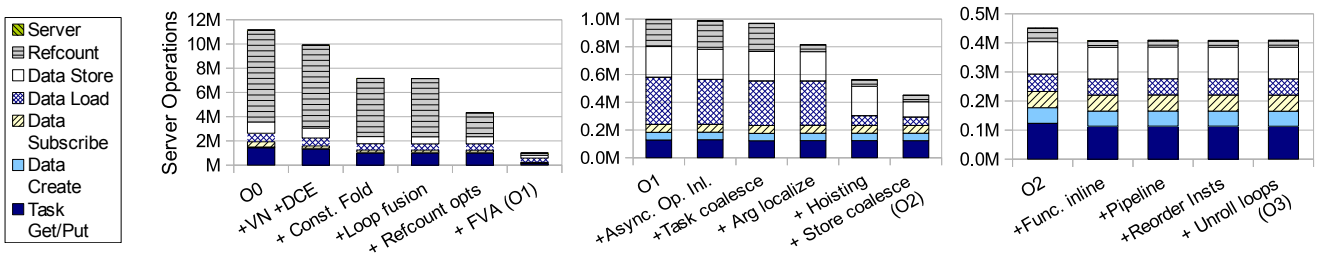


Fig. 12: Runtime operation counts for Annealing, measured in thousands of operations. Each bar includes previous optimizations, so the difference between adjacent bars shows the incremental impact of each optimization pass.

of these runtime operations will be bottlenecks to throughput and scaling in any task-parallel runtime system.

ReduceTree shows good scaling at all optimization levels with no scaling bottlenecks. Because of the short duration of the tasks, however, the superior efficiency of the code at O2 and O3 leads to an order of magnitude higher throughput compared with that of O0. UTS shows nearly perfect scaling, with performance of O2, O3, and ADLB nearly indistinguishable. O3 and ADLB were more economical with data operations than O2, but throughput was limited by computation of the UTS update hash function rather than work distribution. At lower optimization levels, input data to the UTS function quickly became a bottleneck and prevented further scaling. Our simple recursive UTS implementation reached a scale 4.7x larger than the previous largest reported UTS run, which was achieved by an X10 work-stealing algorithm [30].

The strong scaling results for Annealing show O0 and O1 failing to scale beyond a certain point as data operations became a bottleneck, while O2 and O3 continued scaling up until the point when work was relatively scarce. Figure 10f shows that O3 suffered a collapse in throughput when moving to 32,768 cores that was not suffered by O2, despite O2 using slightly more runtime operations. Work is underway to fix this problem, which we believe is caused by workers frequently transitioning from busy to idle in such a manner that the work stealing algorithm causes excessive congestion.

STC at O3 is competitive with the hand-coded ADLB baselines. In the case of UTS, O3 uses measurably more runtime operations, but this does not impact throughput to any great extent. In some cases it scales better because STC's dynamic, recursive partitioning of foreach loops is more friendly to the runtime's work-stealing algorithms than the static partitioning used by the ADLB baseline. In the Wavefront benchmark, O3 gradually overtook the ADLB baseline: the optimized code was less efficient but more scalable because management of data dependencies was automatically balanced between nodes.

C. Contribution of Individual Optimizations

To illustrate better how each optimization pass described can contribute to overall speedup, we analyzed the incremental contribution of each optimization level in smaller-scale runs of the Annealing benchmark. The results in Figure 12 show that frozen variable analysis and hoisting were effective at eliminating shared data and in hoisting shared array accesses

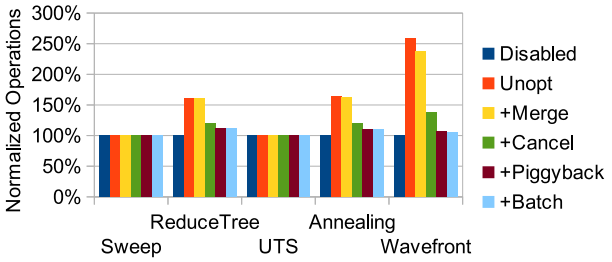


Fig. 13: Impact of unoptimized and optimized reference counting for memory management, normalized to the total count of runtime operations without memory management. Each bar includes previous optimizations.

out of loops. FVA is intimately connected to monotonic variables, while hoisting is a member of a widely used family of optimization techniques, a fact that clearly illustrates how both conventional and dataflow-specific optimizations are required for data-driven task parallelism. These optimizations also rely on basic optimizations, particularly VN and DCE to clean up and remove redundancy after major transformations of the IR.

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

D. Memory Management Overhead

We also performed experiments to understand the effectiveness of reference counting optimization, in particular to understand the overhead of automatic memory management and how much it could be mitigated. We ran scaled-down instances of the benchmarks under multiple configurations: **Off**, where read reference counts are not tracked and memory is never freed; **Unopt**, where all reference counting optimizations are disabled; and different levels where reference counting optimizations are incrementally enabled. All other optimizations were enabled; hence, for some benchmarks shared data was optimized out in all cases.

Figure 13 shows that the reference counting optimizations are effective: the overhead measured in operations from by automatic memory management after optimization is at most

12.2%. The additional operations only cause a proportional decrease in speed if the application is bottlenecked on runtime operations. In practice, this means that the overhead of automatic memory management leads to a 0–12% increase in the minimum task granularity that can be supported. This granularity increase is small enough that automatic memory management in Swift is very viable for large-scale computing. This is important because Swift semantics can require creation of many runtime data items for data dependencies, which cannot always be optimized out: automatic memory management is required to support the high-level programming model.

V. Related Work on Compiler Optimization

Other authors have studied optimization of parallel and distributed applications using a wide range of techniques. In this section we describe some closely related work.

Hardware data-flow-based languages and execution models received significant attention in the past, but there is a resurgence of interest in hardware-based [15], [23] and software-based [6], [7], [10], [25], [29] dataflow models because of their ability to expose parallelism, mask latency, and enable fault tolerance. Previous work has optimized data flow languages with arrays: SISAL [26] and Id [32]. Both have similarities to Swift/T; but emphasize generating low-level machine code. Id targets shared-memory dataflow machines, while STC targets a distributed software runtime. The SISAL runtime used fork-join parallelism, so compilation necessarily eliminated some potential parallelism. In STC, fully dynamic task parallelism required new techniques, such as task-graph-based transformations, and more complex reference counting algorithms.

Research on the APGAS family of programming languages [1], [4], [30] has resulted in optimization techniques for explicitly parallel programs executing in a partitioned global address space with async/finish synchronization, which contrasts with our work's focus on implicitly parallel programs without explicit partitioning and dataflow-driven synchronization. The main similarities are the first-class IR constructs that represent remote or asynchronous execution, and optimizations that understand and optimize these constructs.

Other authors have described parallel intermediate representations (IRs), which typically are sequential IRs with parallel extensions [39]. STC's IR semantics are, in contrast, fully based on a data-driven task parallel execution model with asynchronous execution and monotonic data structures built-in. This allows for aggressive optimization through exploitation of monotonicity and loose rules on statement reordering.

Related compiler techniques have been proposed in other contexts. Task creation and management overhead is a core challenge of task parallelism. Zhao et al. reduce this overhead by safely eliding or reducing strength of synchronization operations [40]. Arandi et al. show benefits from compiler-assisted resolution of task data dependencies with a shared-memory runtime [2]. Jagannathan's communication-passing transformation [13] is similar in spirit to STC's task pushdown, moving operations to execute at the place and time their inputs are produced. Previous work has addressed compile-time reference counting optimization [14], [24] but these techniques for sequential or explicitly parallel languages are substantially different from STC's.

Other work outside the compiler optimization literature has focused on optimization of distributed data-dependency driven *workflows*. This work has focused on scheduling the workflows with constraints of resource availability and data movement cost [27], [38], typically assuming that a static task graph is available. We focus on finer-grained parallelism in conjunction with a high-level, more general programming model, with the short duration of tasks making runtime overhead, in contrast, a dominant concern.

VI. Future Work

The intermediate representation and optimization techniques that we describe in this paper can provide the foundation for further research, both in compiler optimization, and in combined runtime/compiler approaches. For example, opportunities exist to implement further techniques from the extensive compiler optimization literature. More sophisticated control and data flow analyses could bring further incremental improvements to many applications, while more specialized techniques, such as for affine nested loops [5], would aid certain applications such as *wavefront*.

The compiler infrastructure presents opportunities for cross-layer optimization between the compiler and the distributed runtime. Past work [33] has identified opportunities for runtime systems to optimize data placement and movement for data-intensive applications given hints about future workload, which could be provided by compile-time analysis.

VII. Conclusion

We have described a suite of optimization techniques that can be applied to improving efficiency and scalability of distributed-memory task-parallel programs expressed in a high-level programming language. We applied these techniques to a particularly challenging case: high-level implicitly parallel scripts in the Swift/T programming language. However, these techniques are not specific to Swift/T: we expect that they could also be applied to other task-parallel programming models.

Our performance results support two major claims: that applying a wide spectrum of compiler optimization techniques can greatly improve performance and that compiler optimization can allow high-level implicitly parallel code to drive fine grained task-parallel execution at massive scales, rivaling the efficiency and scalability of hand-written parallel coordination code for common patterns of parallelism at scales from tens of cores to half a million cores for a range of task-parallel application patterns including iterative optimization, tree search, and parallel reductions.

ACKNOWLEDGMENTS

This research is supported by the U.S. DOE Office of Science under contract DE-AC02-06CH11357 and NSF award ACI 1148443. Computing resources were provided in part by NIH through the Computation Institute and the Biological Sciences Division of the University of Chicago and Argonne National Laboratory, under grant S10 RR029030-01, and by NSF award ACI 1238993 and the state of Illinois through the Blue Waters sustained-petascale computing project.

REFERENCES

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. PPoPP '07, pages 183–193, 2007.

[2] 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.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Conc. Comput.: Pract. Exper.*, 23(2):187–198, 2011.

[4] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimizations for distributed-memory X10 programs. In *IPDPS '11*, pages 1101–1113, May 2011.

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

[6] 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*.

[7] Z. Budimlić, 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*.

[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*.

[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. 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.

[13] S. Jagannathan. Continuation-based transformations for coordination languages. *Theoretical Computer Science*, 240(1):117 – 146, 2000.

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

[15] R. Knauerhase, R. Cledat, and J. Teller. For extreme parallelism, your OS is sooooo last-millennium. In *Proc. HotPar '12*, 2012.

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

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

[20] NCSA. Blue waters user portal, 2014. https://bluewaters.ncsa.illinois.edu/hardware-summary.

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

[22] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*, pages 235–250. Springer Berlin Heidelberg, 2007.

[23] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao. TIDeFlow: The time iterated dependency flow execution model. In *Proc. DFM '11*.

[24] Y. Park and B. Goldberg. Static analysis for optimizing reference counting. *Information Processing Letters*, 55(4):229 – 234, 1995.

[25] 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.

[26] V. Sarkar and D. Cann. POSC - a partitioning and optimizing SISAL compiler. *SIGARCH Comput. Archit. News*, 18(3b):148–164, June 1990.

[27] G. Singh, C. Kesselman, and E. Deelman. Optimizing grid-based workflow execution. *J. Grid Comp.*, 3(3), 2005.

[28] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proc. SC '09*.

[29] S. Taşirlar and V. Sarkar. Data-driven tasks and their implementation. In *Proc. ICPP '11*.

[30] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. PPoPP '14, pages 53–66, 2014.

[31] The MPI Forum. MPI: A message-passing interface standard, 2012.

[32] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report MIT-LCS-TR-370, Cambridge, MA, 1986.

[33] 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.

[34] 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.

[35] 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*.

[36] 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*.

[37] 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*.

[38] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. *Studies in Computational Intelligence*, 146:173–214, 2008.

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

[40] 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.

CATEGORIES AND SUBJECT DESCRIPTORS

D.3.2 [**Programming Languages**]: Language Classifications – Concurrent, distributed, and parallel languages; Dataflow languages.

GENERAL TERMS

Design; Languages; Performance

KEYWORDS

Parallel scripting; Swift; High-performance computing; Compiler optimization; Data-flow; Execution models; Programming models;