Steno: Automatic Optimization of Declarative Queries

Derek G. Murray
University of Cambridge Computer Laboratory
Derek.Murray@cl.cam.ac.uk

Michael Isard   Yuan Yu
Microsoft Research Silicon Valley
{misard, yuanbyu}@microsoft.com

Abstract

Declarative queries enable programmers to write data manipulation code without being aware of the underlying data structure implementation. By increasing the level of abstraction over imperative code, they improve program readability and, crucially, create opportunities for automatic parallelization and optimization. For example, the Language Integrated Query (LINQ) extensions to C# allow the same declarative query to process in-memory collections, and datasets that are distributed across a compute cluster. However, our experiments show that the *serial* performance of declarative code is several times slower than the equivalent hand-optimized code, because it is implemented using run-time abstractions—such as iterators—that incur overhead due to virtual function calls and superfluous instructions.

To address this problem, we have developed Steno, which uses a combination of novel and well-known techniques to generate code for declarative queries that is almost as efficient as hand-optimized code. Steno translates a declarative LINQ query into type-specialized, inlined and loop-based imperative code. It eliminates chains of iterators from query execution, and optimizes nested queries. We have implemented Steno for uniprocessor, multiprocessor and distributed computing platforms, and show that, for a real-world distributed job, it can almost double the speed of end-to-end execution.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Optimization

General Terms Design, Performance

Keywords query optimization, abstract machines

1. Introduction

The declarative style of programming has long been proposed as a superior alternative to imperative programming. Recently, declarative programming has found an important application in data-center programming: systems such as MapReduce [10], DryadLINQ [32] and FlumeJava [7] allow users to compose a declarative specification of an application's logic, and execute it across hundreds or thousands of machines. However, these systems are implemented in common imperative languages (C++, C# and Java), and simulating the declarative style results in code that is less efficient than the equivalent imperative code. In this paper, we focus on one of these

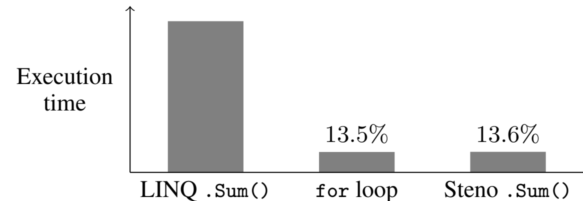

Figure 1. Relative execution time for computing the sum of squares of 10^7 `doubles` using LINQ, an imperative loop, and a Steno-optimized query. Steno achieves a $7.4\times$ speedup over LINQ.

systems—DryadLINQ—and describe a technique for code generation that significantly improves the end-to-end execution time of distributed jobs.

DryadLINQ is a system that takes a query written in .NET Language Integrated Query (LINQ) syntax, and generates a distributed query plan for executing the query across compute nodes in a data center [32]. The main advantage of DryadLINQ is that queries can include user-defined types and functions, which allows the developer to use the full .NET type system and class libary. DryadLINQ divides the query into *vertices* in a Dryad [19] task dependency graph: each vertex executes a portion of the query on a partition of the overall data. The generated code itself uses LINQ queries to achieve multiprocessor parallelism within a single machine. However, our experiments have shown that LINQ queries are far slower than the equivalent imperative code (Figure 1), for four reasons:

1. LINQ queries are lazily evaluated, and use *iterators* to communicate elements between stages of the query [2]. An iterator imposes the overhead of two virtual function calls per element per query operator.
2. LINQ queries may be nested, which involves each element flowing through multiple iterators. The iterator overhead is therefore multiplied by the number of nesting levels.
3. The lazy iterator implementation includes state machine logic to simulate coroutine behavior [22], which adds further per-element overhead.
4. An operator's behavior—such as a predicate or transformation function—is specified as a function object, which incurs a further virtual call per element per operator.

To address these overheads, we have implemented Steno: an optimizer for LINQ queries that generates the equivalent loop-based imperative code. Steno performs two optimizations: *iterator fusion* (Section 4), and *nested loop generation* (Section 5). Similar optimizers have been developed for functional languages [9, 31] and relational database query languages [15, 23]. However, Steno makes several contributions beyond existing work:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'11 June 4–8, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06…$10.00

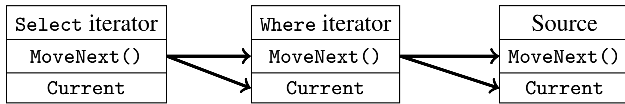

Figure 2. Call graph in a chain of query operators. Virtual function calls are represented by bold arrows.

- Steno integrates with an existing object-oriented language (C#), and does not require changes to the compiler or standard libraries (Section 3).
- Steno uses a novel automaton-based code generator to transform a sequence of query operators into loop-based code (Sections 4 and 5).
- Steno may be combined with DryadLINQ to generate code that runs in parallel across a data center, and across multiple processors on each machine (Section 6).
- We evaluate the performance of Steno-optimized queries running on a single machine and on a distributed compute cluster (Section 7).

Steno generates loop-based code, which is simple for a compiler to optimize and would be simple for a moderately-experienced programmer to write. The principal advantage of writing queries in a declarative style is that it is also possible to apply higher-level optimizations. In particular, DryadLINQ [32, 33], FlumeJava [7] and Pig [26] optimize many operations, including aggregation, joins and sorting, by applying high-level transformations on the query operator graph. The guiding principle in this work is that code should be written at the highest possible level, so that it can benefit from multiple levels of optimization. Steno provides further incentive to write at a high level, because the resulting code is almost as efficient as low-level imperative code. Our evaluation shows that, even in a distributed setting, our optimizations can significantly improve end-to-end execution times (Section 7).

Though this paper focuses on query optimization for LINQ and DryadLINQ, the techniques can be applied to any system that uses iterators to process streaming data or simulate lazy evaluation. However, to motivate our specific implementation, we begin with a brief description of how LINQ is currently implemented.

2. From queries to iterators

In this paper, we focus on the LINQ extensions that were added to .NET 3.5. LINQ extends C# with SQL-like *query comprehension* syntax, first-class *lambda expressions*, and a set of generic *query operators* that can be applied to any object implementing the enumerable interface (`IEnumerable<T>`) [2].

The following is an example of a simple LINQ query

```
IEnumerable<int> xs = ...;

var evenSquares = from x in xs
                  where x % 2 == 0
                  select x * x;
```

which the C# compiler desugars into

```
IEnumerable<int> evenSquares = xs.Where(x => x % 2 == 0)
                                 .Select(x => x * x);
```

The `where` and `select` clauses in the query comprehension are transformed into calls to the `Where()` and `Select()` methods, which—in this case—operate on and return an `IEnumerable<int>`. LINQ defines many other query operators, as we will describe in Sections 4 and 5.

Enumerable objects expose a single method, `GetEnumerator()`, which returns an object of type `IEnumerator<T>`, with the following (simplified) interface:

```
interface IEnumerator<T> {

    // Return the element at the current position.
    T Current { get; }

    // Advance to next element, returning false if no
    // more elements remain.
    bool MoveNext();

    // [Reset() and Dispose() methods not shown.]
}
```

In this interface, `Current` is a read-only *property*, which is syntactic sugar that allows accessor methods to be written in the same form as instance field accesses.

An enumerable object can be traversed using the `foreach(var x in xs)` statement. Again, this statement is syntactic sugar for operations on an `IEnumerable<T>` object (`xs`), and it translates to:

```
IEnumerator<T> enum = xs.GetEnumerator();
while (enum.MoveNext()) {
  T x = enum.Current;
  // foreach loop body...
}
enum.Dispose();
```

Note that since `MoveNext()` and `Current` are defined in an interface, they are virtual functions, so each iteration of the `while` loop involves at least two virtual calls.

LINQ uses iterators to implement lazy query evaluation. An iterator (`IEnumerator<T>`) object is typically implemented as a state machine that advances through the collection upon calls to `MoveNext()`, which update the `Current` property. Composable LINQ operators (such as `Select()`, `Where()` and `GroupBy()`) are implemented as iterators that consume elements from an upstream iterator, and *yield* (possibly-transformed) elements to downstream operators (Figure 2). Aggregate operators—which return a scalar (such as `Sum()`, `Min()` and `Average()`)—are eagerly evaluated and contain a `foreach` loop that consumes the upstream iterator.

Due to the use of iterators, each LINQ operator makes at least two virtual calls for each element that it processes[1]. A further virtual call per element is incurred when evaluating the predicate or transformation function. As Krikellas *et al.* explain, virtual calls impose a severe overhead on query execution because they are difficult to inline automatically [23]. As a result, each call causes an indirect branch, which inhibits instruction pipelining. Furthermore, iterators contain state machine logic in the `MoveNext()` function, which adds more instructions to the per-element overhead.

Writing the equivalent "hand-optimized" C# code for this query would be straightforward for most programmers:

```
foreach (int x in xs) {
    if (x % 2 == 0) {
        yield return x * x;
    }
}
```

The iterators for the `Where` and `Select` operators are *fused*, by storing the current element in the loop variable, `x`, and the predicate and transformation expressions are inlined. Steno automatically performs these optimizations on chains of query operators.

[1] An alternative approach, as used in Java, combines the `MoveNext()` and `Current` operations into a single virtual call. However, this precludes the use of non-reference types or null elements in collections, because the combined method returns `null` to indicate that there are no more elements.

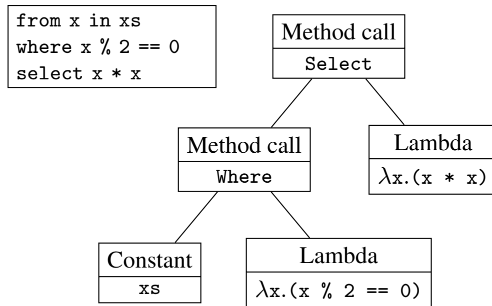

Figure 3. Translation from LINQ query syntax into AST form.

3. Outline of approach

At a high level, Steno converts queries into efficient, loop-based imperative code. Our main contribution is a novel technique for generating the imperative code. In this section, however, we first put this technique in context, by describing the main steps that comprise Steno optimization:

1. The query is transformed into an intermediate representation (QUIL) (§3.1).
2. The query AST is transformed into an equivalent C# AST (§3.2; see also §4 and §5).
3. The C# AST is compiled, loaded and invoked (§3.3).

Note that our implementation of Steno is a .NET library: it does not modify the compiler or the .NET core library. To apply Steno to a query, the `WithSteno()` extension method is applied to the source collection, as follows:

```
var evenSquares = from x in xs.WithSteno()
                  where x % 2 == 0
                  select x * x;
```

Since the C# compiler already performs source-to-source transformations on LINQ queries [2], Steno's optimizations could also be applied at compile-time. We speculate about how this approach could be taken further in Section 9.

3.1 Query extraction

The optimization process begins with a query expression, which can be written in query comprehension syntax or as a sequence of LINQ method invocations. To gain a usable representation of the query, we use the LINQ *query provider* facility, which reconstructs the query AST at run-time. A similar technique is used in DryadLINQ, which uses the information to build a distributed query graph [32]; and in LINQ-to-SQL, which turns a LINQ query into the equivalent SQL database query [25].

The query AST represents each LINQ operator as a method-call expression, and the operator arguments as lambda expressions (Figure 3). To simplify the code generation process, Steno translates this AST into a chain of operators, by post-order traversing the tree, and yielding a canonical[2] *operator* for each method-call expression. Steno also traverses the AST of the lambda expressions to identify nested queries, as we discuss further in Section 5.

3.2 Optimized code generation

The code generation process takes a (possibly nested) chain of operators and transforms it into optimized imperative code. To achieve this, Steno uses an automaton-based approach. Steno traverses the chain of operators (and any nested chains), and emits one or more symbols of *Query Intermediate Language* (QUIL) per operator. The code generator automaton recognizes any valid string of QUIL, and generates the appropriate C# code for each symbol. We introduce QUIL in Subsection 4.1.

The code generator automaton performs two principal optimizations. First, it eliminates the iterators from chains of operators (§4). It also identifies nested queries, and transforms these into nested loops (§5). As it parses a sequence of QUIL symbols, the code generator builds a new C# class with a single method that implements the optimized query. It builds the class using the .NET CodeDOM library, which provides an object model for the C# (and other .NET languages') AST. The automaton builds up the AST from loops, conditional statements, declarations and assignments, according to the QUIL symbols that it receives.

3.3 Final steps

Once the query class has been generated, it must be compiled and loaded in order to execute the query. This step invokes the C# compiler to build a dynamic link library (DLL) from the CodeDOM AST. Steno dynamically loads the resulting DLL, and instantiates a compiled query object using the reflection API.

Before the query is invoked, Steno must resolve any object references that were captured in the query. As a pre-processing step, Steno replaces all captured object references in the query with placeholder instance variables. Once the generated code has been loaded, Steno uses the reflection API to set the relevant fields of the compiled query object appropriately. Since using the dynamic loading and reflection APIs incurs a relatively high constant cost (§7.1), the query object may be cached between invocations. However, by integrating Steno with DryadLINQ, this cost can be eliminated by performing Steno code generation in the initial DryadLINQ code generation process (§6), and generating static code that sets the captured variables.

4. Iterator fusion

Iterator fusion replaces a chain of iterators with a sequence of imperative statements in a loop body. To achieve this, Steno generates type-specialized iteration code for the source collection, and inline element processing code for each operator in the query.

Table 1 classifies the LINQ operators according to their type, and maps each to a symbol in our intermediate language, QUIL. In Subsection 4.1, we describe QUIL, and specify a finite state machine that can parse QUIL sentences. Then, in Subsection 4.2, we show how Steno uses the state machine to generate optimized code without iterators. Finally, in Subsection 4.3, we show how Steno can use additional information from the operator graph to generate more-efficient specialized code.

4.1 Query Intermediate Language

The Query Intermediate Language (QUIL) serves three main purposes:

- It simplifies query optimization by reducing the large number of LINQ operators to six fundamental QUIL operators.
- It simplifies the code generator by allowing it to be structured as an automaton that recognizes the language.
- It enables extensibility, by specifying the interface that each operator must provide.

As Table 1 shows, we have defined six symbols, which comprise the QUIL alphabet and correspond to LINQ operators. In this subsection, we define the behavior of each operator, and show how operators may be combined to represent a LINQ query.

[2] For example, there are three overloaded versions of the `Aggregate` method, and eight overloaded versions of the `GroupBy` method.

Operator class	QUIL symbol	LINQ operators	Haskell equivalent	Input type	Output type
Source	Src	`Range`, `Repeat`	List constructor	—	`IEnumerable<T>`
Transform	Trans	`Select`	`map`	`IEnumerable<T>`	`IEnumerable<U>`
Predicate	Pred	`Where`, `Take`, `Skip`, etc.	`filter`	`IEnumerable<T>`	`IEnumerable<T>`
Sink	Sink	`GroupBy`, `OrderBy`, etc.	`foldl`	`IEnumerable<T>`	`IEnumerable<U>`
Aggregate	Agg	`Aggregate`, `Min`, `Sum`, etc.	`foldl`	`IEnumerable<T>`	`U`
Nested	—	`SelectMany`, `Join`	`concatMap`	`IEnumerable<T>`	`IEnumerable<U>`
Return	Ret	—	—	`IEnumerable<T>` or `T`	—

Table 1. The LINQ operator classes map on to QUIL symbols (§4.1). `T` and `U` are generic type variables. Nested operators map to multiple QUIL symbols, as explained in Section 5.

A QUIL expression begins with a Src symbol, and ends with a Ret symbol. The Src symbol represents an enumerable source collection, and may be annotated with the collection's run-time type, which enables Steno to produce efficient iteration code for the collection. LINQ collection generators, such as `Range(start, count)` are also represented by Src. The Ret symbol denotes the end of a query, and may appear after any other QUIL symbol. Therefore, a query may return either a collection or a scalar value.

The other QUIL operators are analogous to LINQ operators:

- The Trans operator applies an element-wise transformation to each element in the input collection, yielding a new collection. Trans is parameterized with a function, f : `T` $\rightarrow$ `U`, that transforms a single input element into an output element.
- The Pred operator applies a predicate to each element in the input collection, and yields an output collection containing only elements that match the predicate. Pred is parameterized with a function, f : `T` $\rightarrow \mathbb{B}$, that performs the predicate test on a single input element.
- The Sink operator transforms the input collection into an intermediate collection that may be enumerated subsequently. Typically, Sink builds up the intermediate collection in memory. Sink is parameterized with two functions. The first function, f : () $\rightarrow$ `IEnumerable<U>`, constructs an empty intermediate collection; and the second function, g : `IEnumerable<U>` $\times$ `T` $\rightarrow$ `IEnumerable<U>`, updates a collection with a new input element. Sink may also provide type-specialized iteration code for the intermediate collection.
- The Agg operator reduces elements into a single, scalar value. Agg is parameterized with two functions. The first function, f : () $\rightarrow$ `U`, returns the identity scalar value; and the second function, g : `U` $\times$ `T` $\rightarrow$ `U`, creates a new scalar value of type `U` from the current scalar value and a single input element.

For each operator, the function parameters may be specified as an appropriately-typed lambda expression, delegate function or functor object. Note that we can assume that the C# compiler has already type-checked the query expression, so Steno does not perform additional type-checking.

We now define QUIL by considering what constitutes a valid query. Every query begins with Src and ends with Ret. The Trans, Pred and Sink symbols transform one enumerable collection into another collection. Therefore, it is possible to chain together an unbounded number of these operators in an arbitrary order. As we will discuss in Section 5, a nested query may substitute for a Trans or Pred symbol. Finally, since Agg returns a scalar value, its result can only be consumed by Ret, and Agg may only appear as the penultimate symbol. These rules may be stated concisely using the following grammar:

$$\langle query \rangle \quad ::= \quad \mathsf{Src}\ (\ \mathsf{Trans} \mid \mathsf{Pred} \mid \mathsf{Sink} \mid \langle query \rangle)^{*}\ \mathsf{Agg}^{?}\ \mathsf{Ret}$$

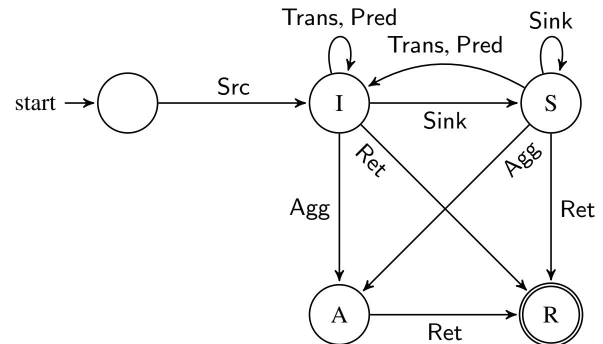

Figure 4. State machine used to perform iterator fusion in Steno.

To simplify the explanation, we will defer the discussion of nested queries to Section 5.

If we ignore the possibility of nested queries, QUIL is a regular language and can therefore be recognized by a finite state machine (FSM). Steno uses the FSM shown in Figure 4 to perform iterator fusion on QUIL expressions.

The QUIL FSM has five states. The initial Src symbol, and the element-wise Trans and Pred symbols cause a transition to the ITERATING state. The Agg symbol causes a transition to the AGGREGATING state, and the Sink symbol causes a transition to the SINKING state. Note that the SINKING and ITERATING states are separate, because the code generated on a transition from those states will differ (§4.2). Finally, the Ret symbol causes a transition to the terminal RETURNING state. In the following subsection, we show how these transitions can generate optimized code without iterators.

4.2 Code generation

Code generation is driven by transitions of the FSM shown in Figure 4. In this subsection, we describe the overall structure of the generated code, and explain the code that is emitted by each transition.

All QUIL queries begin with a Src operator, which corresponds to an enumerable collection, and therefore intuitively causes a new loop to be generated. Figure 5 shows the structure of the generated loop code: it contains a linked list of C# statements, with three internal pointers to positions where statements may be inserted. Insertion point α is the *loop prelude* (i.e. the list of statements immediately preceding the loop); μ is the *loop body*; and ω is the *loop postlude*. The loop prelude contains aggregation and sink variable declarations, the loop body contains element-wise operations, and the loop postlude may contain return statements.

The generated loop code depends on the run-time type of the source collection. For example, if the source is an array or array-backed collection, it is more efficient to use indexed element access than an iterator to access an element. The generated code is a loop

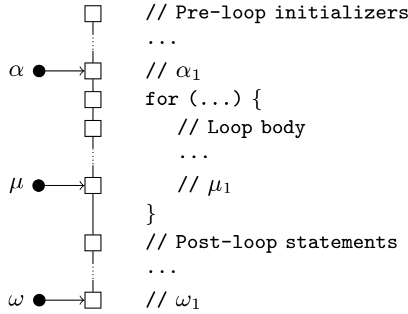

Figure 5. The generated code is maintained as a linked list of statements. Steno maintains three insertion points: the loop prelude (α), the loop body (μ), and the loop postlude (ω).

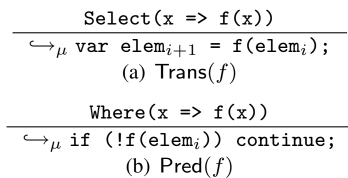

Figure 6. Generated code fragments for element-wise operators. **N.B.** The $\hookrightarrow_\mu$ symbol denotes that code is inserted at the μ pointer.

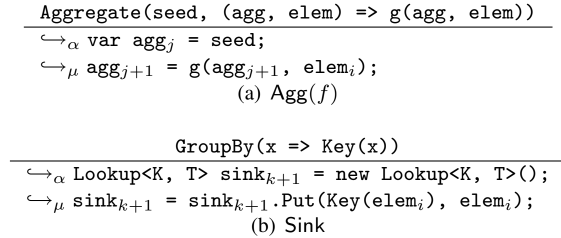

Figure 7. Generated code fragments for aggregating and sinking operators. In (b), `Lookup<K, T>` is a utility class that maintains a key-value multi-map, implements the `IEnumerable<IGrouping<K, T>>` interface, and provides a `Put` method that returns the updated collection.

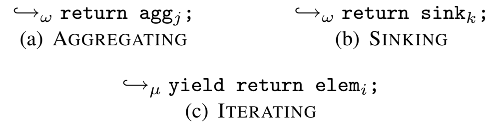

Figure 8. Generated code fragments for the Ret operator, which depend on the current state.

through the indices of the array, which also enables the compiler to hoist the array bounds check. Steno also provides Src implementations for the `Range` and `Repeat` operators, which generate new collections based on their parameters.

The code generator uses local variables to store intermediate values, such as the result of a transformation. There are three kinds of variable in the generated code:

$\texttt{elem}_i$	A (possibly-transformed) element in a collection
$\texttt{agg}_j$	The current scalar value of the jth Agg operator
$\texttt{sink}_k$	The current sink collection of the kth Sink operator

The code generator maintains the current names for each kind of variable as an integer index, which some transitions may increment. In the remainder of the paper, $\texttt{varname}_i$ is the current variable name, and $\texttt{varname}_{i+1}$ is the next variable name.

When the FSM is in the ITERATING state, the Trans and Pred element-wise operators insert code in the current loop body at the μ pointer. Figure 6 shows the code that is inserted in each case. Both operators operate on the current element ($\texttt{elem}_i$), and Trans creates a new element variable ($\texttt{elem}_{i+1}$). In addition, Steno inlines the transformation or predicate function, which eliminates the overhead of a virtual call on a function object per element.

The Agg and Sink operators reduce the current element into a scalar aggregate value or a sink collection, respectively. Therefore the code generator must insert code both for declaring and updating the reduction variable. Figure 7 shows the code that is inserted for the `Aggregate` and `GroupBy` operators. In both cases, the declaration is inserted at the α pointer before the loop, and the update statement is inserted at the μ pointer.

Following a Sink operator, the FSM is in the SINKING state, which behaves like the ITERATING state with one main exception: the following operator is applied to the sink collection. For example, a common pattern is to follow a `GroupBy` Sink operator with a `Where` Pred operator to filter the groups (cf. the `GROUP BY ...HAVING` pattern in SQL). To handle this, the code generator must insert a new loop that iterates through the sink collection. The loop is inserted at the current ω insertion point, and the insertion pointers are reset relative to the new loop.

Finally, the Ret symbol causes a transition to the terminal RETURNING state, which generates code that returns one or more values to the caller (Figure 8). If the FSM is in the SINKING or AGGREGATING states, this inserts a `return` statement (with the $\texttt{sink}_k$ or $\texttt{agg}_j$ variable respectively) at the ω insertion point. If it is in the ITERATING state, the transition inserts a `yield return` statement at the μ pointer: this turns the optimized query into an iterator, which enables the query to be lazily evaluated[3].

4.3 Operator specialization

By analysing the intermediate query representation, Steno can make high-level optimizations that are not possible when dealing with abstract iterators. In this subsection, we consider one concrete example: the `GroupBy-Aggregate` optimization.

The `GroupBy` operator (Figure 7(b)) is a Sink operator that builds a mapping from keys to bags of values, and is analogous to the `GROUP BY` clause in a SQL statement. LINQ provides several implementations of `GroupBy`, including some that allow the caller to specify a *result selector*, which applies a function to each key and the collection of values associated with that key.

The result selector is often used to aggregate—or *reduce*—the set of values associated with a key into a single scalar. Indeed, the `reduce()` function in MapReduce [10] has the same signature as the `GroupBy` result selector, and performs the equivalent operation [33]. If the result selector is an Agg operator (such as `Aggregate`, `Sum` or `Min`), we can save memory by storing per-key partial aggregates instead of the group of values.

To implement this optimization, Steno identifies `GroupBy` operators with an aggregating result selector when building the operator chain (§3.1), and inserts a specialized `GroupByAggregate` Sink operator in place of a conventional `GroupBy`. The `GroupByAggregate` operator performs the aggregation as each element is processed, and updates an intermediate hashtable sink object.

[3] It is often more efficient to store the elements in an array and return this to the caller, but it is unsafe to do this for some queries where the collection may be very large (or unbounded). Therefore, the caller can use the explicit `ToArray` Sink operator to enable this optimization.

The memory requirements can be decreased further (to $O(1)$ keys and reduction variables) if it is known that the collection is ordered by the same key as the grouping. DryadLINQ implements this optimization in order to aggregate key sets that are too large to fit in memory [33].

5. Nested loop generation

We now consider the case where a QUIL expression may contain a nested query. As defined in Subsection 4.1, a nested query may substitute for the transformation and predicate functions of Trans and Pred operators, respectively. In this section, we explain how Steno generates efficient nested loops for nested queries.

At first, it might seem that nested queries can be trivially supported by running a second instance of the optimizer on the nested query, and inserting a call to the optimized subquery in the parent. In fact, this is an appropriate solution if the nested query has a scalar result (i.e. if it contains an Agg operator). However, if the nested query returns an enumerable collection, this solution is inefficient, because the outer query obtains an opaque collection, and can only access the result elements through the iterator interface, which incurs two virtual calls per element (§2). The `SelectMany` operator illustrates this point:

```
int[] xs = ..., ys = ..., zs = ...;

int result = xs.SelectMany(x =>
                 ys.SelectMany(y =>
                        zs.Select(z => F(x, y, z)))
               .Sum();
```

This query computes the Cartesian product of three arrays—`xs`, `ys` and `zs`—applies `F` to each resulting element, and sums the results. The equivalent loop-based code is simple to write:

```
int total = 0;

for (int i = 0; i < xs.Length; ++i) {
    for (int j = 0; j < ys.Length; ++j) {
        for (int k = 0; k < zs.Length; ++k) {
            total += F(xs[i], ys[j], zs[k]);
        }
    }
}
```

However, a naïve implementation using nested FSM-based optimizers would not generate this code, because the `Sum` operator is part of the outermost query, yet it must inject code into the loop body of the innermost query. Without this ability, the `Sum` and nested `SelectMany` operators must consume from iterators, which limits the potential performance improvement.

The `SelectMany` operator *flattens* a collection of collections (one per original element) into a single collection. It is a fundamental operator in MapReduce, in which the `map()` function transforms a single element into zero or more key-value pairs [10]. DryadLINQ implements this functionality using `SelectMany` [32], and similar operators exist in FlumeJava [7] and Pig Latin [26], both of which execute on a MapReduce cluster. Furthermore, since it can implement the Cartesian product, `SelectMany` can also be used to implement joins across multiple collections:

```
int[] xs = ..., ys = ...;

var zs = xs.SelectMany(x => ys.Where(y => x == y)....);
```

The above example shows an equi-join on two arrays of integers. However, in practice, this is an inefficient way to implement joins on two large data sets, and partitioning or sorting is used to reduce the necessary amount of processing, I/O and/or memory [13, 32].

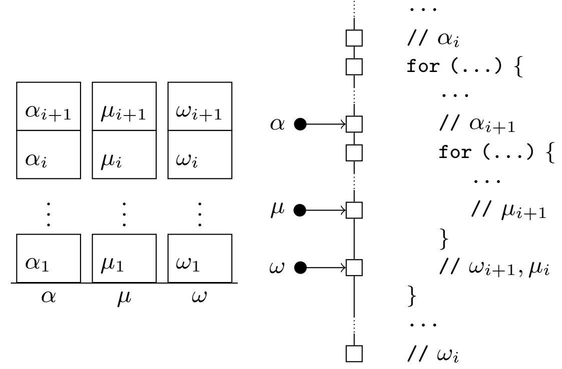

Figure 9. In the nested case, insertion pointers are arranged in a stack, with the innermost nesting level at the top. The current (α, μ, ω) pointers are read from the top of the stack.

Having motivated the need for nested query support, we now discuss how Steno uses QUIL to support nested queries (§5.1), and their effect on code generation (§5.2).

5.1 Adding a stack

Since a QUIL query can recursively contain another query, the language is context-free. Therefore, the FSM in Figure 4 is not powerful enough to recognize all valid QUIL queries. To recognize the full context-free language, we must add a *stack* to the code generator, making it a (deterministic) pushdown automaton. In order to reuse the iterator fusion optimization, the pushdown automaton is exactly equivalent to the FSM in the non-nested case. Therefore, in this subsection, we reintroduce the nested query transitions and how these manipulate the stack.

Recall that the code generator has α, μ and ω insertion pointers, which correspond to the current loop's prelude, body and postlude, respectively (Figure 5). Since all queries—nested or otherwise—involve iterating through a collection, a nested query will create a new loop, with a different prelude, body and postlude. After exiting a nested query, it may be necessary to recover the outer query's insertion points, so that code for subsequent operators may be inserted. Therefore, we define the stack, S, as containing $(\alpha_i, \mu_j, \omega_k)$ triples, where $i, j, k \geq 1$ are the nesting levels of each insertion point. The code generator uses the topmost triple as the current values for α, μ and ω.

S is initially empty. When a Src operator is encountered at nesting level $i = length(S)$, a new loop is inserted at position μ_i, and insertion points $(\alpha_{i+1}, \mu_{i+1}, \omega_{i+1})$ are pushed onto the stack. Figure 9 shows the state of the stack and the generated AST after entering nesting level $i + 1$.

5.2 Code generation

Following a nested Src operator, the automaton is in the ITERATING state. Subsequent Trans, Pred, Sink and Agg operators behave as in the non-nested case, though code is inserted at the nested insertion points. In addition, the nested query may refer to the current element in the outer query, as in this example:

```
xs.SelectMany(x => ys.Select(G(x, y)));
```

Therefore, before generating any code for the nested query, all occurrences of `x` in the nested query are rewritten with the current $\texttt{elem}_i$ variable name in the outer query.

The behavior on encountering a nested Ret operator depends on the current state of the automation. The SINKING and AGGREGATING cases are simpler. If the code generator is in either of these

```
↪ω var elem_{i+1} = agg_j;        ↪ω var elem_{i+1} = sink_k;
```

(a) AGGREGATING (b) SINKING

Figure 10. Generated code fragments for the nested Ret operator, in the AGGREGATING and SINKING states.

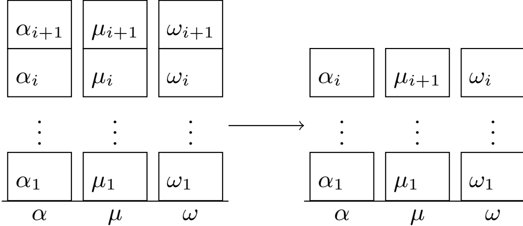

Figure 11. Code generator stack contents before and after encountering a nested Ret operator in the ITERATING state. After the transition, the α and ω pointers point to positions in the outer query, and the μ pointer is unchanged.

states, it assigns the current aggregation or sink variable to a new element variable in the nested loop postlude (Figure 10), then pops the current insertion pointer triplet from the stack.

If the code generator encounters a nested Ret operator while in the ITERATING state, the stack management is slightly more complicated. Recall that, in the non-nested case (§4.2), encountering Ret while ITERATING causes a `yield return` statement to be emitted. In this case, we want the code for subsequent operators to be inserted in the current nested loop body. Therefore we pop two insertion pointer triples from the stack, and push back the triple $(\alpha_{\text{outer}}, \mu_{\text{nested}}, \omega_{\text{outer}})$ (Figure 11). This ensures that the element-wise code for each operator (transformation, predicate testing and reduction variable updating) will be inserted in the nested loop, while any declarations or return statements will be placed in the outer scope.

This formulation means that nested `Select` and `SelectMany` have the same representation in QUIL. In a `Select` operator, the nested query will typically end with a Agg operator and return a scalar value; by definition, a `SelectMany` operator will yield many values. However, instead of creating an iterator using a `yield return` statement, the Ret in a nested query moves the insertion pointers so that subsequent operators consume the results of the nested query directly.

6. Optimizing parallel queries

A principal advantage of expressing a computation as a LINQ query is that it may be executed in parallel across multiple processors [29] or multiple machines in a data center [32]. However, the optimizations described above generate sequential code. In this section, we explain how Steno can be combined with DryadLINQ in order to improve the performance of distributed queries.

To execute a query on a large data set, a common strategy is to divide the data set into *partitions*, and execute the query in parallel on each partition [12]. If the query operators are *homomorphic* (i.e. apply to each element independently), the query may be applied to each partition in parallel, to yield a new set of partitions. Trans, Pred and nested queries are homomorphic. However, if a query operator performs an aggregation (Agg) or builds a sink collection (Sink), this requires coordination between the partitions.

To optimize a query that can execute in parallel, Steno traverses the QUIL representation of the query and identifies the homomorphic operators. Contiguous subsequences of homomorphic operators are combined into subqueries, and the subqueries are optimized

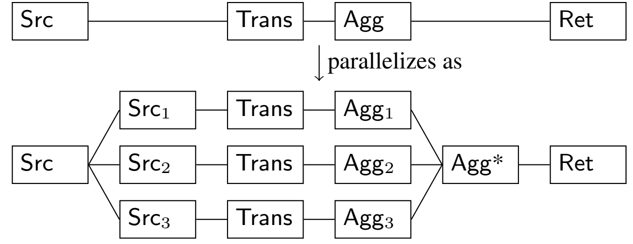

Figure 12. Parallel optimization of a `Select-Aggregate` query. Steno applies optimization to the Src_i-Trans-Agg_i subquery, which executes in parallel on the partitions of the data.

separately. Note that, if an *associative* Sink or Agg operator follows a subquery, a *partial* Sink_i or Agg_i operator can be appended to the ith subquery, which reduces the amount of coordination between partitions. For example, if the Agg operator represents `Sum`, the corresponding Agg_i operator computes a partial sum for each partition. Figure 12 shows how a simple query is parallelized. In that example, a special Agg* operator collects the partially-aggregated results from each partition.

We have integrated Steno with DryadLINQ, which executes queries in parallel across a compute cluster. DryadLINQ also identifies homomorphic query operators, and transforms a LINQ query into a directed acyclic graph of query operators, which the Dryad executes as a collection of parallel tasks [19]. In addition, DryadLINQ performs various static and dynamic optimizations on the query before generating the code for each task: for example, it transforms a `OrderBy` Sink operator into a distributed sort, which samples the data to estimate an appropriate partitioning, range-partitions the data based on that estimate, and sorts each resulting partition in parallel [32]. DryadLINQ also optimizes distributed aggregation where the aggregation function is commutative and associative [33]. After optimization, a DryadLINQ may contain a sequence of query operators: our modified version of DryadLINQ applies Steno-optimization to the subsequences of homomorphic operators in each task.

DryadLINQ can also execute subqueries in parallel within a single task. Previously, it used Parallel LINQ (PLINQ) to execute homomorphic subqueries using a thread-pool [29]. PLINQ provides the same operators as LINQ, but operates on a `ParallelEnumerable` collection, which uses a `Partitioner` object to assign elements to each thread. PLINQ uses iterators to compose query operators, and therefore suffers from similar virtual call overheads to sequential LINQ. To ameliorate this, we created a new PLINQ operator, called `HomomorphicApply`, which maps a function across partitions in parallel (as opposed to each element), and returns a new set of partitions. Our modified version of DryadLINQ invokes this operator with the compiled query method, which allows the optimized code to execute in parallel.

7. Evaluation

We now evaluate the performance improvement that Steno can achieve. We first evaluate a collection of single-machine microbenchmarks, which operate on in-memory data (§7.1). We then evaluate the impact of applying Steno to a real-world application that uses DryadLINQ on a distributed compute cluster (§7.2).

All experiments described in this paper were performed on our research cluster, from which we use up to 100 nodes. Each computer has two dual-core AMD Opteron 2218 HE processors running at 2.6 GHz, 16 GB of DDR2 RAM, and four 750 GB SATA hard drives in a RAID 0 (striped) configuration. The computers are connected using gigabit Ethernet, in a three-level tree topology.

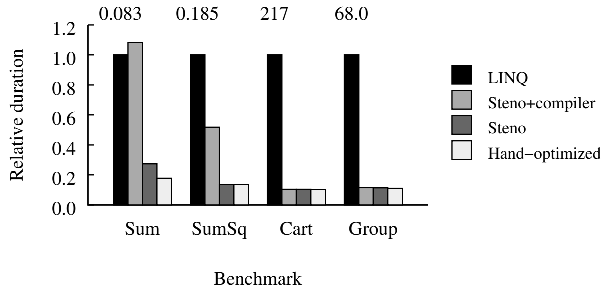

Figure 13. Relative performance of LINQ, Steno-optimized and hand-optimized queries for sequential microbenchmarks. Lower values are better. Each query is annotated with the absolute LINQ execution time, in seconds.

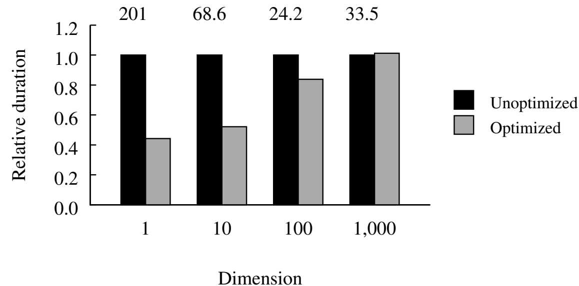

Figure 14. Relative performance of unoptimized and Steno-optimized k-means algorithm, running on DryadLINQ. Each query is annotated with the absolute unoptimized execution time for one iteration, in seconds.

All computers run the 64-bit version of Microsoft Windows Server 2003, and the 64-bit version of Microsoft .NET Framework 3.5[4].

7.1 Sequential microbenchmarks

We evaluated the performance of sequential Steno optimization using the following four queries on in-memory arrays:

Sum Calculate the sum of 10 million `double` values.

SumSq Calculate the sum of squares of 10 million `double` values.

Cart Calculate the Cartesian product of 10 million and 1000 `double`s, multiply together each pair, and sum.

Group Randomly generate 10 million `double` values according to a one-dimensional mixture-of-Gaussians distribution, and compute a binned histogram of the data.

Figure 13 shows four quantities for each benchmark: LINQ, Steno including compilation, Steno excluding compilation, and hand-optimized. In order to allow comparison of different benchmarks, the results are normalized to the LINQ execution time.

For all of the microbenchmarks, the Steno-optimized query is faster than the equivalent LINQ query. The speedup ranges from $3.32\times$ for Sum to $14.1\times$ for Group. As expected, the nested queries (Cart and Group) yield a bigger speedup due to the LINQ code using nested iterators. Compared to the hand-optimized, loop-based code, the worst overhead is 53%—for Sum, the simplest query—due to the .NET JIT compiler missing a possible temporary variable elimination in the Steno-generated code, which leads to two extraneous `movsd` instructions in the loop body. For the other queries, the overhead (compared to hand-optimized) is less than 3%.

However, Steno optimization carries a one-off cost per query, which is dominated by invoking the C# compiler and dynamically loading the optimized query class. The compiled query object can then be cached by the application. In our experiments, this cost was 69 milliseconds on average. Therefore, if an application contains mostly short, infrequently-executed queries, it is not worth invoking Steno at run-time. Summing 10 million `double`s with LINQ takes approximately 83 ms, whereas with Steno it takes 25 ms plus 69 ms for compilation. The break-even point is approximately 12 million `double`s. Note that, if the optimization were added to the C# compiler, this cost would be paid at compile-time. In addition, the optimized query object may be stored and reused in order to amortize the cost of compilation. In the current implementation, the user must explicitly instruct Steno to compile a given expression, but a query caching approach (based on Nectar [18]) could be added.

[4] The current version of DryadLINQ is not compatible with .NET 4.0. Steno is compatible with .NET 3.5 and 4.0. We confirmed that microbenchmark performance is the same using both versions.

7.2 Distributed k-means

We now evaluate the performance of Steno-optimized code for a representative distributed query: k-means clustering. A distributed cluster is a more challenging environment for Steno: since the data set is too large to fit in memory, it must be read from disk, and network communication is required to coordinate between partitions. By contrast, the microbenchmarks operate on in-memory arrays.

The k-means algorithm groups a set of data points into k clusters by estimating the centroid of each cluster, and iteratively updating the centroids by averaging the points in each cluster. The main computational step involves computing the (e.g. Euclidean) distance from each point to each centroid, and assigning each point to the cluster whose centroid is closest. We implemented the algorithm in DryadLINQ, and each iteration comprises two steps:

1. In parallel, for each data point (nested `Select`), compute the distance to each centroid (`Select`), and choose the cluster with the closest centroid (`Aggregate`). Then group these results by cluster ID (`GroupBy`) and compute partial sums of the points in each cluster (`Aggregate`).
2. Group the partial sums from each partition by cluster ID (`GroupBy`), add them together (`Aggregate`), and compute the new cluster centroids by taking the mean (`Select`).

This query exploits iterator fusion (§4), `GroupBy-Aggregate` specialization (§4.3) and nested loop generation (§5).

The benefits from Steno-optimization are greatest when (i) a large number of elements are processed, and (ii) the amount of work that each operator performs is small. The k-means algorithm offers us the opportunity to vary both quantities, but (since k-means has $O(n)$ complexity for n data points) it is more meaningful to vary to amount of work per element. We achieve this by varying the number of dimensions in each point, which is directly proportional to the number of floating-point operations in the Euclidean distance calculation. Figure 14 shows the effect of varying the dimension on the performance of unoptimized and Steno-optimized k-means. The overall size of the input data (number of points $\times$ dimension) is held constant at 10^9 `double`s (≈ 8 GB).

The most notable result is that Steno achieves substantial speed-up over LINQ for dimensions less than 1000, which demonstrates that iterator overheads can have a large effect on the performance of distributed query execution. Larger speedups are achieved for smaller dimensions: for example, $1.9\times$ for 10-dimensional points. However, as the dimension increases, Steno manages to speed up execution, and it achieves a 19% improvement for 100-dimensional data. Eventually, however, the execution converges with the unoptimized case, as the fraction of time spent in the distance computation becomes closer to 100%.

8. Related work

There is a large volume of related work on the optimization of declarative programs. Steno combines ideas from database systems, functional programming, XML processing and object-oriented programming to optimize the execution of declarative data-parallel programs. In this section, we survey the related work in each of these fields, and compare it to Steno.

8.1 Relational databases

The stated aim of Codd's relational algebra was to hide details of "how the data is organized in the machine (the internal representation)." [8] This set the precedent for database access using SQL, a declarative query language, which in turn influenced the design of LINQ [2]. Query evaluation can be implemented using iterators [3], which motivates use of `IEnumerator` objects to compose LINQ operators. Because relational databases are designed to hold large volumes of data, research into query efficiency has been carried out since the earliest implementations of the relational model.

IBM's System R implemented SEQUEL (a precursor of SQL) query evaluation by assembling fragments of System/370 machine code in a query-specific subroutine [6]. System R used a preprocessor to extract SEQUEL queries from COBOL or PL/1 source code, and replace them with calls to the generated machine code. However, the use of pre-defined code fragments led to overhead from procedure calls, and poor portability [15].

Freytag and Goodman refined this method by transforming a query into iterative C or Pascal code, which is then compiled by an existing compiler [15, 16]. Their system compiles queries written in a dialect of SQL—comprising operators for projection, filtering, left-join and aggregation—into iterative code, using rule-based transformations.

Most recently, Krikellas *et al.* revisited the problem of efficient query executed, and analysed the costs of the iterator model at the computer-architectural level [23]. They propose "holistic query evaluation" (HQE), which uses a code generator to turn SQL queries into C code, and then compiles it using an existing optimizing compiler. Like System R, HQE relies on a library of templates to provide the generated code. It offers a restricted form of nested loop generation for table joins, but does not support arbitrarily-nested query expressions such as those described in Section 5.

The techniques for optimizing query execution are similar to Steno, in that they propose generating lower-level code, which can be compiled or executed directly. However, all of these techniques target SQL, which is less general than LINQ (it lacks a `SelectMany` operator), and does not—at least, in the database systems described above—integrate with user-defined code in a general-purpose language. Furthermore, none of these techniques are shown to apply to nested queries. The automaton-based approach that we describe in this paper could be applied to SQL query optimization and may be useful in extending these systems with nested query support.

8.2 Functional programming

The LINQ execution framework shares several features with lazily-evaluated functional languages: in particular, the `Select`, `Where`, `Aggregate` and `SelectMany` operators correspond to Haskell's `map`, `filter`, `foldl` and `concatMap` functions [25]. Therefore, techniques for efficiently compiling a lazy functional program are related to (and can inform) our technique for efficiently compiling a LINQ expression.

Lazy evaluation can reduce the storage cost of some programs by only evaluating the values in a collection when they are needed by a consumer. However accessing the not-yet-evaluated portions of intermediate collections imposes a hidden cost that is analogous to the iterator overhead in LINQ. Wadler developed the "deforestation" algorithm to eliminate intermediate lists (and trees) from programs written in a restrictive, first-order, lazy functional language [31]. However, deforestation is considered impractical because it restricts programs to a "treeless" form that prohibits, amongst other things, intermediate data structures [17].

Practical versions of deforestation include "`build/foldr`" [17], "`destroy/unfoldr`" [28] and stream fusion [9]. These techniques use equational transformations in Haskell to transform functions that produce and consume lists into fused code that does not use intermediate lazy lists. Of the three techniques, stream fusion is the most powerful, and can optimize the nested `concatMap` function, which is analogous to `SelectMany` in LINQ. However, stream fusion is not well-suited to deeply-nested list computations because the Glasgow Haskell Compiler's optimizer cannot always generate efficient code from the fused intermediate form [9]. By contrast, Steno generates simple nested `for` loops that the C# compiler can easily optimize.

8.3 XML processing

The growing popularity of XML as a data interchange format has prompted research into efficient processing of XML documents and streams. XML query languages such as XQuery and XPath provide declarative syntax for computations over XML documents, and early query processing systems used an iterator-based approach for query evaluation [14].

Li and Agarwal developed a code generation technique called "Generalized Nested Loops" (GNLs), which can be used to optimize the evaluation of XQuery expressions on large data sets [24]. Although superficially similar to nested queries in QUIL (§5), GNLs can only represent nested loops that perform an associative aggregation function. Therefore, the GNL code generation technique could not be applied to nested queries that flatten a collection, such as `SelectMany` and `Join`.

Reichenbach *et al.* have studied the optimization of XML-processing computations that are embedded in the XJ imperative language, which is based on Java [27]. They developed a program analysis that identifies potential XPath queries that are latent in imperative code, and exploits opportunities for sharing results between queries. By contrast, Steno attempts to optimize the execution of explicit queries. However, since XJ supports LINQ-like query syntax, it may be possible to apply this analysis to C# programs, and identify opportunities for common subexpression evaluation between Steno-optimized queries.

8.4 Object-oriented programming

In an object-oriented programming language, virtual calls are expensive because the compiler does not have enough information to generate inline code at the call site: the receiver method is only known at run-time. To solve this problem, various devirtualization techniques have been proposed.

Calder and Grunwald proposed a simple transformation for C++ code that inlines the code for the most likely receiver of a virtual call, guarded by a run-time type check [5]. Since the guard code contains a simple branch instruction, hardware branch prediction can ameliorate the overhead of the type check. However, a LINQ operator may be used to consume from different iterator implementations at different points in the same program (or even in the same query), so the receiver for a particular call site is not usually predictable.

Dean *et al.* developed static class hierarchy analysis, which performs static analysis on the inheritance hierarchy and intraprocedural data-flow analysis to identify the precise type of an object (and hence potentially eliminate a virtual call) [11]. This technique is less useful for C#, which allows dynamic classloading, though a just-in-time approach that "revirtualizes" methods as appropriate has been developed [20]. Furthermore, since the `IEnumerator<T>`

interface has many implementing classes, it is rarely possible to make a precise static judgement about the run-time type of an iterator, without expensive interprocedural analysis.

Although devirtualization could improve the performance of LINQ queries, it is not implemented in the current version of the .NET Common Language Runtime (CLR). Furthermore, even if perfect devirtualization were achieved, the resulting inlined code would include the state machine logic from each iterator, which would be less efficient than the simple for loops generated by Steno. We are not aware of a compiler that could eliminate this semantically redundant logic and yield code that is as efficient as a Steno-optimized query.

8.5 Data-parallel computing

Steno was originally developed to optimize the performance of DryadLINQ programs, which run in parallel on a distributed compute cluster. The restrictions imposed by the declarative style of programming make it simple to parallelize a program written as a declarative query (§6), and several systems use this approach to exploit the computational resources of large compute clusters.

Dean and Ghemawat's MapReduce is an influential system for data-parallel programming [10]. In MapReduce, developers specify their computations by providing two functions: a *mapper*, which transforms input records into lists of key-value pairs (cf. `SelectMany`, §5), and a *reducer*, which aggregates all of the values corresponding to a single key (cf. `GroupBy-Aggregate`, §4.3). MapReduce uses an iterator in the reducer to provide access to the values corresponding to a single key; the open-source Hadoop implementation of MapReduce uses iterators extensively in the processing of records [1]. We expect that generating specialized code to eliminate iterators would yield a performance improvement similar to what Steno achieves.

Chambers *et al.* described FlumeJava, which (like DryadLINQ) uses lazy evaluation to build a distributed execution plan from a graph of operators [7]. In FlumeJava, however, the execution engine is Google's MapReduce implementation. We note the similarity of FlumeJava's `parallelDo()`, `groupByKey()`, `combineValues()` and `flatten()` functions to LINQ's `Select`, `GroupBy`, `Aggregate` and `SelectMany` operators, respectively. The optimizations that we have developed in Steno apply equally to FlumeJava programs. However, whereas LINQ uses `Expression` objects with an in-memory AST, FlumeJava uses classes that implement a functor interface to encapsulate operator behavior. This is less amenable to run-time optimization, and bytecode rewriting techniques would be necessary to eliminate virtual function calls from the optimized code.

Two contemporary projects have performed static analysis on (Hadoop) MapReduce programs in order to derive an optimized program. Manimal [4] and HadoopToSQL [21] analyse the Java bytecode of MapReduce programs in order to generate an equivalent relational algebra expression. The expression is then converted to SQL, and passed to a cluster of SQL databases. The intermediate representation bears some similarity to the LINQ expression tree that Steno uses. Therefore, the analyses that these systems perform could be combined with Steno-style code generation in order to generate optimized query execution code.

Since Steno integrates into DryadLINQ, it can take advantage of higher-level optimizations. For example, Yu *et al.* added a transformation to DryadLINQ that optimizes distributed aggregation (`GroupBy-Aggregate`) by partially performing associative and commutative aggregation functions before the communication step [33]. Gunda *et al.* developed a system called Nectar, which caches the results of previous DryadLINQ jobs, and uses the cached data to achieve common subexpression elimination across multiple DryadLINQ jobs [18]. Steno is applied after these transformations, which further improves the performance of DryadLINQ queries.

9. Conclusions

In this paper, we have presented Steno: an automatic optimizer for declarative queries that generates efficient, loop-based, imperative code. Our automaton-based approach provides a useful abstraction for structuring the code generator, and made it straightforward to implement support for our two key optimizations: iterator fusion and nested loop generation. Though these optimizations generate sequential code, we have integrated Steno with DryadLINQ, which extends the optimization to code running in parallel on a distributed compute cluster. Our evaluation showed that, even in the challenging distributed case, Steno can almost double the performance of real-world DryadLINQ jobs.

Our approach in developing Steno has been conservative. We did not modify the C# compiler or the .NET core libraries, and we faithfully reproduced the semantics of unoptimized LINQ. This leaves scope for further optimization. First, we can apply such optimizations as common subexpression elimination only if it is possible to prove that the subexpression has no side effects, and that materializing the subexpression does not exhaust memory. The difficulty of this problem suggests that there are opportunities for developer-guided optimization. Also, we have hitherto only considered C# as the target language, but it may be possible to gain more efficiency by directly generating bytecode, C or even native machine code. We note that many operators could benefit from vectorization, so SIMD execution using instruction-set extensions or GPGPUs would achieve greater efficiency [30].

The implementation of Steno as a library is a mixed blessing. The main advantage is that Steno works with a standard .NET toolchain, but as we have shown in Section 7, invoking the C# compiler at run-time introduces a one-off overhead of tens of milliseconds. As a result, the developer must be judicious in deciding when to optimize a query, by identifying frequently-executed or long-running queries. This problem could be addressed by modifying the C# compiler to perform Steno optimizations at compile-time. The compiler already desugars LINQ queries that are written in query comprehension syntax [2], and it would be conceptually straightforward to extend this compiler pass to use Steno.

The current implementation of Steno can only optimize the standard LINQ queries. Steno cannot optimize user-defined iterators that are created with the `yield return` statement, because this statement is syntactic sugar that is transformed into an iterator state machine at compile-time [22]. A typical user-defined iterator contains a loop over some internal data structure that `yield`s one or more elements. If Steno were able to access the original syntax tree for the iterator method, it could perform more-aggressive iterator fusion and nested loop generation, which would further improve the performance of iterator-based programs.

The overall lesson that we draw from this work is that it is advantageous to write in the declarative style wherever possible. The main benefit is simpler code, which is easier to develop, and which systems like PLINQ and DryadLINQ can automatically transform and optimize for parallel or distributed execution. Previously, developers were forced to choose between efficient serial execution and access to these transformations. Steno demonstrates that this is a false dichotomy, by making the performance of declarative queries competitive with hand-optimized code, while still exploiting higher-level transformations.

Acknowledgments

We wish to thank Frank McSherry, Mihai Budiu and the PLINQ team for helpful discussions during the development of Steno. We would also like to thank Steve Hand and the anonymous reviewers, whose comments on earlier drafts of this paper have been invaluable for improving the presentation of this work.

References

[1] Apache Hadoop. `http://hadoop.apache.org/`, accessed 17th November, 2010.

[2] G. M. Bierman, E. Meijer, and M. Torgersen. Lost In Translation: Formalizing Proposed Extensions to C$^\sharp$. In *Proceedings of OOPSLA*, 2007.

[3] P. Buneman, R. E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Trans. Database Syst.*, 7(2), 1982.

[4] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *Proceedings of WebDB*, 2010.

[5] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of POPL*, 1994.

[6] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of System R. *Commun. ACM*, 24 (10), 1981.

[7] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of PLDI*, 2010.

[8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6), 1970.

[9] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of ICFP*, 2007.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of OSDI*, 2004.

[11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP*, 1995.

[12] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6), 1992.

[13] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of SIGMOD*, 1984.

[14] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. In *Proceedings of VLDB*, 2003.

[15] J. C. Freytag and N. Goodman. On the translation of relational queries into iterative programs. *ACM Trans. Database Syst.*, 14(1), 1989.

[16] J. C. Freytag and N. Goodman. Translating aggregate queries into iterative programs. In *Proceedings of VLDB*, 1986.

[17] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of FPCA*, 1993.

[18] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Data Centers. In *Proceedings of OSDI*, 2010.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys*, 2007.

[20] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of OOPSLA*, 2000.

[21] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: a MapReduce query optimizer. In *Proceedings of EuroSys*, 2010.

[22] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: proof rules and implementation, 2005.

[23] K. Krikellas, S. D. Viglas, and M. Cintra. Generating Code for Holistic Query Evaluation. In *Proceedings of ICDE*, 2010.

[24] X. Li and G. Agrawal. Efficient evaluation of XQuery over streaming data. In *Proceedings of VLDB*, 2005.

[25] E. Meijer. Confessions of a used programming language salesman. *SIGPLAN Not.*, 42(10), 2007.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of SIGMOD*, 2008.

[27] C. Reichenbach, M. G. Burke, I. Peshansky, and M. Raghavachari. Analysis of imperative xml programs. *Information Systems*, 34(7), 2009.

[28] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of ICFP*, 2002.

[29] R. Tan, P. Nagpal, and S. Miller. Automated black box testing tool for a parallel programming library. In *Proceedings of ICST*, 2009.

[30] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *Proceedings of ASPLOS*, 2006.

[31] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of ESOP*, 1988.

[32] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of OSDI*, 2008.

[33] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of SOSP*, 2009.