Efficient Execution of Multi-Query Data Analysis Batches Using Compiler Optimization Strategies*

Henrique Andrade[1,2], Suresh Aryangat[1], Tahsin Kurc[2], Joel Saltz[2], and Alan Sussman[1]

[1] Dept. of Computer Science
University of Maryland
College Park, MD 20742
{hcma,suresha,als}@cs.umd.edu
[2] Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210
{kurc.1,saltz.3}@osu.edu

Abstract. This work investigates the leverage that can be obtained from compiler optimization techniques for efficient execution of multi-query workloads in data analysis applications. Our approach is to address multi-query optimization at the algorithmic level, by transforming a declarative specification of scientific data analysis queries into a high-level imperative program that can be made more efficient by applying compiler optimization techniques. These techniques – including loop fusion, common subexpression elimination and dead code elimination – are employed to allow data and computation reuse across queries. We describe a preliminary experimental analysis on a real remote sensing application that analyzes very large quantities of satellite data. The results show our techniques achieve sizable reductions in the amount of computation and I/O necessary for executing query batches and in average execution times for the individual queries in a given batch.

1 Introduction

Multi-query optimization has been investigated by several researchers, mostly in the realm of relational databases [6, 8, 14, 18, 19, 21]. We have devised a database architecture that allows efficiently handling multi-query workloads where user-defined operations are also part of the query plan [2, 4]. The architecture builds on a data and computation reuse model that can be employed to systematically expose reuse sites in the query plan when application-specific aggregation

* This research was supported by the National Science Foundation under Grants #EIA-0121161, #EIA-0121177, #ACI-9619020 (UC Subcontract #10152408), #ACI-0130437, #ACI-0203846, and #ACI-9982087, and Lawrence Livermore National Laboratory under Grant #B517095 and #B500288, NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

methods are employed. This model relies on an *active semantic cache*, in which semantic information is attached to prior computed aggregates that are cached by the system. This permits the query optimizer to retrieve the matching aggregates based on the metadata description of a new query. The cache is active in that it allows application-specific transformations to be performed on the cached aggregates so that they can be reused to speed up the evaluation of the query at hand. The reuse model and active semantic caching have been shown to effectively decrease the average turnaround time for a query, as well as to increase the database system throughput [2–4]. Our earlier approach leverages data and computation reuse for queries submitted to the system over an extended period of time. For a batch of queries, on the other hand, a global query plan that accommodates all the queries can be more profitable than scheduling queries based on individual query plans, especially if information at the algorithmic level for each of the query plans is exposed. A similar observation was the motivation for a study done by Kang et al. [14] for relational operators.

The need to handle query batches arises in many situations. In a data server concurrently accessed by many clients, there can be multiple queries awaiting execution. A typical example is the daily execution of a set of queries for detecting the probability of wildfire occurring in Southern California. In this context, a system could issue multiple queries in batch mode to analyze the current (or close to current) set of remotely sensed data at regular intervals and trigger a response by a fire brigade. In such a scenario, a pre-optimized batch of queries can result in better resource allocation and scheduling decisions by employing a single comprehensive query plan.

Many projects have worked on database support for scientific datasets [9, 20]. Optimizing query processing for scientific applications using compiler optimization techniques has attracted the attention of several researchers, including those in our own group. Ferreira et. al. [10, 11] have done extensive studies on using compiler and runtime analysis to speed up processing for scientific queries. They have investigated compiler optimization issues related to single queries with spatio-temporal predicates, which are similar to the ones we target [11].

In this work, we investigate the application of compiler optimization strategies to execute a *batch* of queries for scientific data analysis applications as opposed to a single query. Our approach is a multi-step process consisting of the following tasks: 1) Converting a declarative data analysis query into an imperative description; 2) Handing off the set of imperative descriptions for the queries in the batch to the query planner; 3) Employing traditional compiler optimization strategies in the planner, such as common subexpression elimination, dead code elimination, and loop fusion, to generate a single, global, efficient query plan.

2 Query Optimization Using Compiler Techniques

In this section, we describe the class of data analysis queries targeted in this work, and present an overview of the optimization phases for a batch of queries.

$\mathcal{R} \leftarrow Select(I, O, M_i)$
foreach(r $\in$ $\mathcal{R}$) {
  $O[\mathcal{S}_L(r)] = \mathcal{F}(O[\mathcal{S}_L(r)], I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)])$
}

Fig. 1. General Data Reduction Loop.

2.1 Data Analysis Queries

Queries in many data analysis applications can be defined as range-aggregation queries (RAGs) [7]. The datasets for range-aggregation queries can be classified as *input*, *output*, or *temporary*. **Input** (I) datasets correspond to the data to be processed. **Output** (O) datasets are the final results from applying one or more operations to the input datasets. **Temporary** (T) datasets (temporaries) are created during query processing to store intermediate results. A user-defined data structure is usually employed to describe and store a temporary dataset. Temporary and output datasets are tagged with the operations employed to compute them and also with the query metadata information (i.e. the parameters specified for the query). Temporaries are also referred to as *aggregates*, and we use the two terms interchangeably.

A RAG query typically has both spatial and temporal predicates, namely a multi-dimensional bounding box in the underlying multi-dimensional attribute space of the dataset. Only data elements whose associated coordinates fall within the multidimensional box must be retrieved and processed. The selected data elements are mapped to the corresponding output dataset elements. The mapping operation is an application-specific function that often involves finding a collection of data items using a specific spatial relationship (such as intersection), possibly after applying a geometric transformation. An input element can map to multiple output elements. Similarly, multiple input elements can map to the same output element. An application-specific aggregation operation (e.g., sum over selected elements) is applied to the input data elements that map to the same output element.

Borrowing from a formalism proposed by Ferreira [10], a range-aggregation query can be specified in the general loop format shown in Figure 1. A *Select* function identifies the subdomain that intersects the query metadata M_i for a query q_i. The subdomain can be defined in the input attribute space or in the output space. For the sake of discussion, we can view the input and output datasets as being composed of collections of objects. An object can be a single data element or a data chunk containing multiple data elements. The objects whose elements are updated in the loop are referred to as *left hand side*, or LHS, objects. The objects whose elements are only read in the loop are considered *right hand side*, or RHS, objects.

During query processing, the subdomain denoted by $\mathcal{R}$ in the *foreach* loop is traversed. Each point r in $\mathcal{R}$ and the corresponding *subscript functions* $\mathcal{S}_L(r), \mathcal{S}_{R1}(r), \ldots, \mathcal{S}_{Rn}(r)$ are used to access the input and output data elements

for the loop. In the figure, we assume that there are n RHS collections of objects, denoted by $I_1, \ldots, I_n$, contributing the values of a LHS object. It is not required that all n RHS collections be different, since different subscript functions can be used to access the same collection.

In iteration r of the loop, the value of an output element $O[\mathcal{S}_L(r)]$ is updated using the application-specific function $\mathcal{F}$. The function $\mathcal{F}$ uses one or more of the values $I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)]$, and may also use other scalar values that are inputs to the function, to compute an aggregate result value. The aggregation operations typically implement *generalized reductions* [12], which must be commutative and associative operations.

2.2 Case Study Application – Kronos

Before we present our approach and system support for multi-query optimization for query batches, we briefly describe the Kronos application used as a case study in this paper.

Remote sensing has become a very powerful tool for geographical, meteorological, and environmental studies [13]. Usually systems processing remotely sensed data provide on-demand access to raw data and user-specified data product generation. Kronos [13] is an example of such a class of applications. It targets datasets composed of remotely sensed AVHRR GAC level 1B (Advanced Very High Resolution Radiometer – Global Area Coverage) orbit data [16]. The raw data is continuously collected by multiple satellites and the volume of data for a single day is about 1 GB. The processing structure of Kronos can be divided into several basic primitives that form a processing chain on the sensor data. The primitives are: **Retrieval**, **Atmospheric Correction**, **Composite Generator**, **Subsampler**, and **Cartographic Projection**. More details about these primitives can be found in the technical report version of this paper [1].

All the primitives (with the exception of Retrieval) may employ different algorithms (i.e., multiple atmospheric correction methods) that are specified as a parameter to the actual primitive (e.g., Correction(T0,Rayleigh/Ozone), where Rayleigh/Ozone is an existing algorithm and T0 is the aggregate used as input). In fact, Kronos implements 3 algorithms for atmospheric correction, 3 different composite generator algorithms, and more than 60 different cartographic projections.

2.3 Solving the Multi-Query Optimization Problem

The objective of multi-query optimization is to take a batch of queries, expressed by a set of declarative query definitions (e.g., using the SQL extensions of PostgreSQL [17]), and generate a set of optimized data parallel reduction loops that represent the global query plan for the queries in the batch. Queries in a declarative language express what the desired result of a query should be, without proscribing exactly how the desired result is to be computed. Previous researchers have already postulated and verified the strengths of using declarative languages from the perspective of end-users, essentially because the process

of accessing the data and generating the data product does not need to be specified. Optimization of declarative queries is then a multi-phase process, in the which query definitions are first converted into imperative loops that conform to the canonical data reduction loop of Figure 1, and then those loops are optimized using various compiler techniques.

Consider Kronos queries as examples. For our study, queries are defined as a 3-tuple: [spatio-temporal bounding box and spatio-temporal resolution, correction method, compositing method]. The spatio-temporal bounding box (in the SQL WHERE clause) specifies the spatial and temporal coordinates for the data of interest. The spatio-temporal resolution (or output discretization level) describes the amount of data to be aggregated per output point (i.e., each output pixel is composed from x input points, so that an output pixel corresponds to an area of, for example, 8 Km^2). The correction method (in the SQL FROM clause) specifies the atmospheric correction algorithm to be applied to the raw data to approximate the values for each input point to the *ideal* corrected values. Finally, the compositing method (also in the SQL FROM clause) defines the aggregation level and function to be employed to *coalesce* multiple input grid points into a single output grid point. Two sample Kronos queries specified in PostgreSQL are illustrated in Figure 2. Query 1 selects the raw AVHRR data from a data collection named AVHRR_DC, for the spatio-temporal boundaries stated in the WHERE clause (within the boundaries for latitude, longitude, and day). The data is subsampled in such a way that each output pixel represents 4 KM^2 of data (with the discretization levels defined by *deltalat*, *deltalon* and *deltaday*). Pixels are also corrected for atmospheric distortions using the *WaterVapor* method and composited to find the maximum value of the Normalized Difference Vegetation Index (*MaxNDVI*).

Figure 2 presents an overview of the optimization process. The goal is to detect commonalities between Query 1 and 2, in terms of the common spatio-temporal domains and the primitives they require. In order to achieve this goal, the first step in the optimization process is to parse and convert these queries into imperative loops conforming with the loop in Figure 1. That loop presents the high-level description of the same queries, with the spatio-temporal boundaries translated into input data points (via index lookup operations). Therefore, loops can iterate on points, blocks, or chunks depending on how the raw data is stored, declustered, and indexed.

We should note that we have omitted the calls to the subscript mapping functions in order to simplify the presentation. These functions enable both finding an input data element in the input dataset and determining where it is placed in the output dataset (or temporary dataset). In some cases, mapping from an absolute set of multidimensional coordinates (given in the WHERE clause of the query) into a relative set of coordinates (the locations of the data elements) may take a considerable amount of time. Thus, minimizing the number of calls to the mapping operations can also improve performance.

As seen in Figure 2, once the loops have been generated, the following steps are carried out to transform them into a global query plan. First, the imperative

Fig. 2. An overview of the entire optimization process for two queries. *MaxNDVI* and *MinCh1* are different compositing methods and *Water Vapor* designates an atmospheric correction algorithm. All temporaries have local scope with respect to the loop. The discretization values are not shown as part of the loop iteration domains for a more clear presentation.

descriptions are concatenated into a single ***workload program***. Second, the domains for each of the *foreach* loops are inspected for multidimensional overlaps. Loops with domains that overlap are fused by moving the individual loop bodies into one or more combined loops. Loops corresponding to the non-overlapping domain regions are also created. An intermediate *program* is generated with two parts: combined loops for the overlapping areas and individual loops for the non-overlapping areas. Third, for each combined loop, common subexpression elimination and dead code elimination techniques are employed. That is, redundant RHS function calls are eliminated, redundant subscript function calls are deleted, and multiple retrievals of the same input data elements are eliminated.

3 System Support

In this section, we describe the runtime system that supports the multi-query optimization phases presented in Section 2. The runtime system is built on a database engine we have specifically developed for efficiently executing multi-

query loads from scientific data analysis applications in parallel and distributed environments [2, 3]. The compiler approach described in this work has been implemented as a front-end to the Query Server component of the database engine.

The Query Server is responsible for receiving *declarative* queries from the clients, generating an *imperative* query plan, and dispatching them for execution. It invokes the Query Planner every time a new query is received for processing, and continually computes the best query plan for the queries in the waiting queue which essentially form a query batch.

Given the limitations of SQL-2, we have employed PostgreSQL [17] as the declarative language of choice for our system. PostgreSQL has language constructs for creating new data types (CREATE TYPE) and new data processing routines, called user-defined functions (CREATE FUNCTION). The only relevant part of PostgreSQL to our system is its parser, since the other data processing services all are handled within our existing database engine.

3.1 The Multi-Query Planner

The multi-query planner is the system module that receives an imperative query description from the Query Server and iteratively generates an optimized query plan for the queries received, until the system is ready to process the next query batch. The loop body of a query may consist of multiple *function primitives* registered in the database catalog. In this work, a function primitive is an application-specific, user-defined, minimal, and indivisible part of the data processing [4]. A primitive consists of a function call that can take multiple parameters, with the restriction that one of them is the input data to be processed and the return value is the processed output value. An important assumption is that the function has no side effects. The function primitives in a query loop form a chain of operations transforming the input data elements into the output data elements. A primitive at level l of a processing chain in the loop body has the dual role of consuming the *temporary dataset* generated by the primitive immediately before (at level $l-1$) and generating the temporary dataset for the primitive immediately after (at level $l+1$).

Figure 2 shows two sample Kronos queries that contain multiple function primitives. In the figure, the spatio-temporal bounding box is described by a pair of 3-dimensional coordinates in the input dataset domain. *Retrieval*, *Correction*, and *Composite* are the user-defined primitives. *I* designates the portion of the input domain (i.e., the raw data) being processed in the current iteration of the *foreach* loop and *T0* and *T1* designate the results of the computation performed by the *Retrieval* and *Correction* primitive calls. *O1* and *O2* designate the output for Query 1 and Query 2, respectively.

Optimization for a query in a query batch occurs in a two-phase process in which the query is first integrated into the current plan, and then redundancies are eliminated. The integration of a query into the current plan is a recursive process, defined by the spatio-temporal boundaries of the query, which describe the loop iteration domain. The details of this process are explained in the next sections and in more detail in the technical report version of this paper [1].

Loop Fusion The first stage of the optimization mainly employs the bounding boxes for the new query, as well as the bounding boxes for the set of already optimized loops in the query plan. The optimization essentially consists of *loop fusion* operations – merging and fusing the bodies of loops representing queries that iterate at least partially over the same domain $\mathcal{R}$. The intuition behind this optimization goes beyond the traditional reasons for performing loop fusion, namely reducing the cost of the loops by combining overheads and exposing more instructions for parallel execution. The main goal of this phase is to expose opportunities for subsequent common subexpression elimination and dead code elimination.

Two distinct tasks are performed when a new loop (*newl*) is integrated into the current query batch plan. First, the query domain for the new loop is compared against the iteration domains for all the loops already in the query plan. The loop with the largest amount of multidimensional overlap is selected to incorporate the statements from the body of the new loop. The second task is to modify the current plan appropriately, based on three possible scenarios: 1) The new query represented by *newl* does not overlap with any of the existing loops, so *newl* is added to the plan as is; 2) The iteration domain for the new loop *newl* is exactly equal to that of a loop already in the query plan (loop *bestl*). In this case, the body of *bestl* is *merged* with that of *newl*; 3) The iteration domain for *newl* is either subsumed by that of *bestl*, or subsumes that of *bestl*, or there is a partial overlap between the two iteration domains. This case requires computing several new loops to replace the original *bestl*. The first new loop iterates only on the common, overlapping domain of *newl* and *bestl*. The body of *newl* is merged with that of *bestl* and the resulting loop is added to the query plan (i.e., *bestl* is replaced by *updatedl*). Second, loops covering the rest of the domain originally covered by *bestl* are added to the current plan. Finally, the additional loops representing the rest of the domain for *newl* are computed, and the new loops become *candidates* to be added to the updated query plan. They are considered candidates because those loops may also overlap with other loops already in the plan. Each of the new loops is recursively inserted into the optimized plan using the same algorithm. This last step guarantees that there will be no iteration space overlap across the loops in the final query batch plan.

Redundancy Elimination After the loops for all the queries in the batch are added to the query plan, redundancies in the loop bodies can be removed, employing straightforward optimizations – common subexpression elimination and dead code elimination. In our case, common subexpression elimination consists of identifying computations and data retrieval operations that are performed multiple times in the loop body, eliminating all but the first occurrence [15].

Each statement in a loop body creates a new available expression (i.e., represented by the right hand side of the assignment), which can be accessed through a reference to the temporary aggregate on the left hand side of the assignment. The common subexpression algorithm [1] performs detection of new available expressions and substitutes a call to a primitive by a *copy* from the temporary

aggregate containing the redundant expression. The equivalence of the results generated by two statements is determined by inspecting the *call site* for the primitive function invocations. Equivalence is determined by establishing that in addition to using the same (or equivalent) input data, the parameters for the primitives are also the same or equivalent. Because the primitive invocation is replaced by a copy operation, primitive functions are required to not have any side effects.

The removal of redundant expressions often causes the creation of useless code – assignments that generate *dead variables* that are no longer needed to compute the output results of a loop. We extend the definition of *dead* variable to also accommodate situations in which a statement has the form $T_i \leftarrow \mathbf{copy}(T_j)$, where T_i and T_j are both temporaries. In this case, all uses of T_i can be replaced by T_j. We employ the standard dead code elimination algorithm, which requires marking all instructions that compute essential values. Our algorithm computes the def-use chain (connections between a *definition* of a variable and all its *uses*) for all the temporaries in the loop body. The dead code elimination algorithm [1] makes two passes over the statements that are part of a loop in the query plan. The first pass detects the statements that define a temporary and the ones that use it, A second pass over the statements looks for statements that define a temporary value, checking for whether they are utilized, and removes the unneeded statements.

Both the common subexpression elimination and the dead code elimination algorithms must be invoked multiple times, until the query plan remains stable, meaning that all redundancies and unneeded statements are eliminated. Although similar to standard compiler optimization algorithms, all of the algorithms were implemented in the Query Planner to handle an intermediate code representation we devised to represent the query plan. We emphasize that we are not compiling C or C++ code, but rather the query plan representation. Indeed, the runtime system implements a virtual machine that can take either the unoptimized query plan or the final optimized plan and execute it, leveraging any, possibly parallel, infrastructure available for that purpose.

4 Experimental Evaluation

The evaluation of the techniques presented in this paper was carried out on the Kronos application (see Section 2.2). It was necessary to re-implement the Kronos primitives to conform to the interfaces of our database system. However, employing a real application ensures a more realistic scenario for obtaining experimental results. On the other hand, we had to employ synthetic workloads to perform a parameter sweep of the optimization space. We utilized a statistical workload model based on how real users interact with the Kronos system, which we describe in Section 4.1.

We designed several experiments to illustrate the impact of the compiler optimizations on the overall batch processing performance, using AVHRR datasets and a mix of synthetic workloads. All the experiments were run on a 24-processor

SunFire 6800 machine with 24 GB of main memory running Solaris 2.8. We used a single processor of this machine to execute queries, as our main goal in this paper is to evaluate the impact of the various compiler optimization techniques on the performance of query batches. Leverage from running in a multi-processor environment will be investigated in future work, to obtain further decreases in query batch execution time. A dataset containing one month (January 1992) of AVHRR data was used, totaling about 30 GB.

4.1 A Query Workload Model

In order to create the queries that are part of a query batch, we employed a variation of the Customer Behavior Model Graph (CBMG) technique. CBMG is utilized, for example, by researchers analyzing performance aspects of e-business applications and website capacity planning. A CBMG can be characterized by a set of n states, a set of transitions between states, and by an $n \times n$ matrix, $P = [p_{i,j}]$, of transition probabilities between the n states.

In our model, the first query in a batch specifies a geographical region, a set of temporal coordinates (a continuous period of days), a resolution level (both vertical and horizontal), a correction algorithm (from 3 possibilities), and a compositing operator (also from 3 different algorithms). The subsequent queries in the batch are generated based on the following operations: another *new point of interest*, *spatial movement*, *temporal movement*, *resolution increase* or *decrease*, applying a different *correction algorithm*, or applying a different *compositing operator*. In our experiments, we used the probabilities shown in Table 1 to generate multiple queries for a batch with different workload profiles. For each workload profile, we created batches of 2, 4, 8, 16, 24, and 32 queries. A 2-query batch requires processing around 50 MB of input data and a 32-query batch requires around 800 MB, given that there is no redundancy in the queries forming the batch and also that no optimization is performed. There are 16 available points of interest; for example, Southern California, the Chesapeake Bay, the Amazon Forest, etc. This way, depending on the workload profile, subsequent queries after the first one in the batch may either remain around that point (moving around its neighborhood and generating new data products with other types of atmospheric correction and compositing algorithms) or move on to a different point. These transitions are controlled according to the transition probabilities in Table 1. More details about the workload model can be found in [4].

For the results shown in this paper each query returns a data product for a 256×256 pixel window. We have also produced results for larger queries – 512×512 data products. The results from those queries are consistent with the ones we show here. In fact, in absolute terms the performance improvements are even larger. However, for the larger data products we had to restrict the experiments to smaller batches of up to 16 queries, because the memory footprint exceeded 2 GB (the amount of addressable memory using 32-bit addresses available when utilizing gcc 2.95.3 in Solaris).

Transition	*Workload 1*	*Workload 2*	*Workload 3*	*Workload 4*
New Point-of-Interest	5%	5%	65%	65%
Spatial Movement	10%	50%	5%	35%
New Resolution	15%	15%	5%	0%
Temporal Movement	5%	5%	5%	0%
New Correction	25%	5%	5%	0%
New Compositing	25%	5%	5%	0%
New Compositing Level	15%	15%	10%	0%

Table 1. Transition probabilities.

4.2 Experimental Study

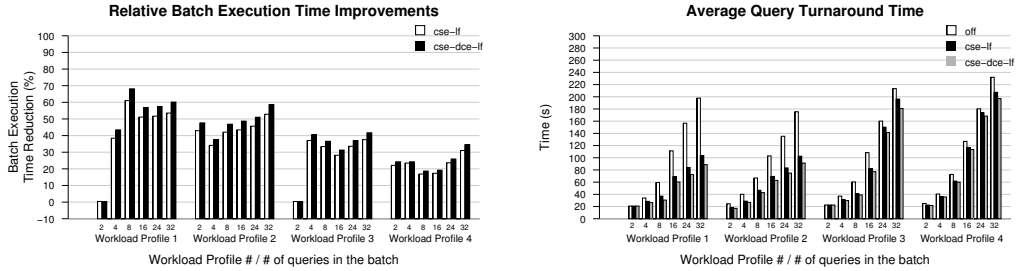
We studied the impact of the proposed optimizations varying the following quantities: **1)** The number of queries in a batch (from a 2-query batch up to a 32-query batch). **2)** The optimizations that are turned on (none, only common subexpression elimination and loop fusion – CSE-LF; or common subexpression elimination, dead code elimination, and loop fusion – CSE-DCE-LF). **3)** The workload profile for a batch. Workload 1 represents a profile with high probability of reuse across the queries. In this workload profile, there is high overlap in regions of interest across queries. This is achieved by a low probability for the New Point-of-Interest and Movement values, as seen in the table. Moreover, the probabilities of choosing new correction, compositing, and resolution values are low. Workload 4, on the other hand, describes a profile with the lowest probability of data and computation reuse. The other profiles – 2 and 3 – are in between the two extremes in terms of the likelihood of data and computation reuse.

Our study collected five different performance metrics: batch execution time, number of statements executed (loop body statements), average query turnaround time[3], average query response time[4], and plan generation time (i.e., the amount of time from when the parser calls the query planner until the time the plan is fully computed).

Batch Execution Time The amount of time required for processing a batch of queries is the most important metric, since that is the main goal of the optimizations we employ. Figure 3 (a) shows the reduction in execution time for different batches and workload profiles, comparing against executing the batch *without any optimizations*. The results show that reductions in the range of 20% to 70% in execution time are achieved. Greater reductions are observed for larger batches using workload profile 1, which shows high locality of interest. In this

[3] Query turnaround time is the time from when a query is submitted until when it is completed.

[4] Query response time is the time between when a query is submitted and the time the first results are returned.



(a) (b)

Fig. 3. (a) The reduction in batch execution time. (b) Average query turnaround time.

profile, there is a very low chance of selecting a new point of interest or performing spatial movement (which implies high spatial and temporal locality as seen in Table 1). Therefore, once some data is retrieved and computed over, most queries will reuse at least the input data, even if they require different atmospheric correction and compositing algorithms. Additionally, there are only 16 points of interest as we previously stated, which means that across the 32 queries at least some of the queries will be near the same point of interest, which again implies high locality. On the other hand, when a batch has only 2 queries, the chance of having spatio-temporal locality is small, so the optimizations have little effect. The 2-query batches for workload profiles 1 and 3 show this behavior (note that the y-axis in the chart starts at -10% improvement). In some experiments we observe that the percent reduction in execution time decreases when the number of queries in a batch is increased (e.g., going from a 4-query batch to an 8-query batch for Workload 3). We attribute this to the fact that queries in different batches are generated randomly and independently. Hence, although a workload is designed to have a certain level of locality, it is possible that different batches in the same workload may have different amounts of locality, due to the distribution of queries. The important observation is that the proposed approach takes advantage of locality when it is present.

Query Turnaround Time A query batch may be formed while the system is busy processing other queries, and interactive clients continue to send new queries that are stored in a waiting queue. In this scenario, it is also important for a database system to decrease the average execution time per query so that interactive clients experience less delay between submitting a query and seeing its results. Although the optimizations are targeted at improving batch execution time, Figure 3 (b) shows that they also improve average query turnaround time. In these experiments, queries are added to the batch as long as the system is busy. The query batch is executed as soon as the system becomes available for processing it. As seen from the figure, for the workload profiles with higher

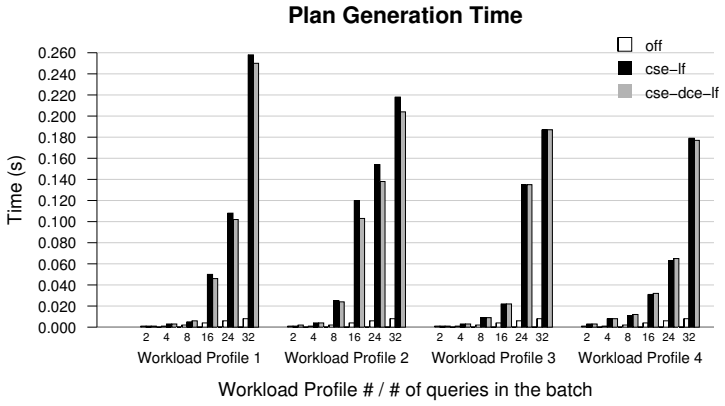


Fig. 4. Time to generate a batch execution plan.

locality (1 and 2), execution time decreases by up to 55%. Conversely, for batches with low locality there is little decrease in execution time, as expected.

Plan Generation Time The application of compiler optimization strategies introduces costs for computing the optimized query plan for the query batch. Figure 4 illustrates how much time is needed to obtain the execution plan for a query batch. There are two key observations here. The first observation is that the planning time depends on the number of exploitable optimization opportunities that exist in the batch (i.e., locality across queries). Hence, if there is no locality in the query batch, the time to generate the optimized plan (which should be the same as the unoptimized plan) is roughly equivalent to the time to compute the non-optimized plan. The second observation is that the time to compute a plan for batches that have heavily correlated queries increases exponentially (due to the fact that each spatio-temporal overlap detected produces multiple new loops that must be recursively inserted into the optimized plan). However, even though much more time is spent in computing the plan, executing the query batch is several orders of magnitude more expensive than computing the plan. As seen from Figures 3 and 4, query batch planning takes milliseconds, while query batch execution time can be hundreds of seconds, depending on the complexity and size of the data products being computed. Finally, a somewhat surprising observation is that adding dead code elimination to the set of optimizations slightly decreases the time needed to compute the plan. The reason is that the loop merging operation and subsequent common subexpression elimination operations become simpler if useless statements are removed from the loop body. This additional improvement is doubly beneficial because dead code elimination also decreases batch execution time, as seen in Figures 3 (a) and (b).

5 Conclusions

In this paper we have described a framework for optimizing the execution of scientific data analysis query batches that employs well-understood compiler optimization strategies. The queries are described using a declarative representation – PostgreSQL – which in itself represents an improvement in how easily queries can be formulated by end users. This representation is transformed into an imperative representation using loops that iterate over a multidimensional spatio-temporal bounding box. The imperative representation lends itself to various compiler optimizations techniques, such as loop fusion, common subexpression elimination, and dead code elimination. Our experimental results using a real application show that the optimization process is relatively inexpensive and that when there is some locality across the queries in a batch, the benefits of the optimizations greatly outweigh the costs.

Two important issues we plan to address in the near future are batch scheduling for parallel execution and resource management. Use of loop fusion techniques not only reduces loop overheads, but also exposes more operations for parallel execution and local optimization. In fact, because of the nature of our target queries (i.e., queries involving primitives with no side effects and generalized reduction operations), each statement of the loop body can be carried out in parallel. This means that scheduling the loop iterations in a multithreaded environment or across a cluster of workstations can improve performance, assuming that synchronization and communication issues are appropriately handled. With respect to resource utilization, there are complex issues to be addressed, in particular with regard to memory utilization. When two or more queries are fused into the same loop, all the output buffers for the queries need to be allocated (at least partially) to hold the results produced by the loop iteration. Moreover, those buffers may need to be maintained in memory for a long time, since all the iterations required to complete a query may be spread across a large collection of loops that may be executed over a long time period (i.e., the first and last loop for a query may be widely separated in the batch plan). Another extension we plan to investigate in a future prototype is to integrate the active caching system and the batch optimizer. In that case, the batch optimizer can also leverage the cache contents when performing common subexpression eliminations.

References

1. H. Andrade, S. Aryangat, T. Kurc, J. Saltz, and A. Sussman. Efficient execution of multi-query data analysis batches using compiler optimization strategies. Technical Report CS-TR-4507 and UMIACS-TR-2003-76, University of Maryland, July 2003.
2. H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Efficient execution of multiple workloads in data analysis applications. In *Proceedings of the 2001 ACM/IEEE Supercomputing Conference*, Denver, CO, November 2001.
3. H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Active Proxy-G: Optimizing the query execution process in the Grid. In *Proceedings of the 2002 ACM/IEEE Supercomputing Conference*, Baltimore, MD, November 2002.

4. H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Exploiting functional decomposition for efficient parallel processing of multiple data analysis queries. Technical Report CS-TR-4404 and UMIACS-TR-2002-84, University of Maryland, October 2002. A shorter version appears in the Proceedings of IPDPS 2003.
5. Caltech. Sensing and responding – Mani Chandy's biologically inspired approach to crisis management. *ENGenious – Caltech Division of Engineering and Applied Sciences*, Winter 2003.
6. U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *Proceedings of the 12th VLDB Conference*, pages 384–391, 1986.
7. C. Chang. *Parallel Aggregation on Multi-Dimensional Scientific Datasets*. PhD thesis, Department of Computer Science, University of Maryland, April 2001.
8. F.-C. F. Chen and M. H. Dunham. Common subexpression processing in multiple-query processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):493–499, 1998.
9. J. M. Cheng, N. M. Mattos, D. D. Chamberlin, and L. G. DeMichiel. Extending relational database technology for new applications. *IBM Systems Journal*, 33(2):264–279, 1994.
10. R. Ferreira. *Compiler Techniques for Data Parallel Applications Using Very Large Multi-Dimensional Datasets*. PhD thesis, Department of Computer Science, University of Maryland, September 2001.
11. R. Ferreira, G. Agrawal, R. Jin, and J. Saltz. Compiling data intensive applications with spatial coordinates. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing*, pages 339–354, Yorktown Heights, NY, August 2000.
12. High Performance Fortran Forum. High Performance Fortran – language specification – version 2.0. Technical report, Rice University, January 1997. Available at http://www.netlib.org/hpf.
13. S. Kalluri, Z. Zhang, J. JáJá, D. Bader, N. E. Saleous, E. Vermote, and J. R. G. Townshend. A hierarchical data archiving and processing system to generate custom tailored products from AVHRR data. In *1999 IEEE International Geoscience and Remote Sensing Symposium*, pages 2374–2376, 1999.
14. M. H. Kang, H. G. Dietz, and B. K. Bhargava. Multiple-query optimization at algorithm-level. *Data and Knowledge Engineering*, 14(1):57–75, 1994.
15. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.
16. National Oceanic and Atmospheric Administration. *NOAA Polar Orbiter User's Guide – November 1998 Revision*. compiled and edited by Katherine B. Kidwell. Available at http://www2.ncdc.noaa.gov/docs/podug/cover.htm.
17. PostgreSQL 7.3.2 Developer's Guide. *http://www.postgresql.org*.
18. P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM-SIGMOD Conference*, pages 249–260, 2000.
19. T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.
20. M. Stonebraker. The SEQUOIA 2000 project. *Data Engineering*, 16(1):24–28, 1993.
21. K. L. Tan and H. Lu. Workload scheduling for multiple query processing. *Information Processing Letters*, 55(5):251–257, 1995.
22. J. D. Ullman. *Database and Knowledge-Base Systems*. Computer Science Press, 1988.