ABSTRACT

Title of Thesis:	OPTIMIZING THE EXECUTION OF BATCHES OF DATA ANALYSIS QUERIES
Degree candidate:	Suresh Aryangat
Degree and year:	Master of Science, 2004
Thesis directed by:	Professor Alan Sussman Department of Computer Science

Data analysis applications such as Kronos, a remote sensing application, and the Virtual Microscope, a telepathology application, require operating on and processing large datasets. In such situations, it is important that the storage, retrieval, and manipulation of these datasets is efficiently handled. Past research has focused on the creation of database systems that abstract the data analysis process into a framework facilitating the design of algorithms to optimize the execution of scientific queries and batches of queries. These optimizations occur at different levels in the query processing chain in the database system.

The present research deals with the optimizations performed by the database system when processing batches of queries. It includes an end-to-end process starting at parsing the declarative queries, converting them into imperative descriptions, merging the imperative descriptions into an execution plan, optimizing the plan for lowering

execution time by employing basic compiler optimization techniques, and, finally, optimizing the plan for lowering memory consumption. The last two steps essentially aim at reducing time and space for executing the batch. In particular, various algorithms to optimize the memory utilization of multiple data analysis queries are presented and the effect of each on query processing performance as well as their performance are investigated.

The query plan that is output from the time optimizations consists of a set of reorderable loops over ranges of a dataset or multiple datasets. These loops comprise the input set that is fed to the space optimization phase of the database system, which aims to reduce the average or maximum memory usage of the entire loop set by determining the order of execution of the loops. The methods used to optimize memory usage that are investigated in this research can be classified into two distinct categories: systematic and heuristic. The systematic methods, brute force and branch-and-bound, arrive at the optimal solution yielding the lowest memory usage metrics; however, their running times depend exponentially upon the size of the input set and thus cannot be employed for large numbers of loops. As a result, we have devised the "greedy" and "variable grouping" heuristics to address the need to optimize large numbers of loops. Each heuristic arrives at a near-optimal solution and has a running time that depends polynomially upon the size of the input set, with the variable grouping heuristic yielding particularly favorable results.

OPTIMIZING THE EXECUTION OF BATCHES OF DATA ANALYSIS QUERIES

by

Suresh Aryangat

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2004

DEDICATION

To Mom and Dad

ACKNOWLEDGEMENTS

I'd like to thank all those who have supported me and allowed me to complete my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Several applications exist today that process very large multi-dimensional datasets. An example of such an application is Kronos, which is used by earth scientists to process satellite images of the Earth. Another example is the Virtual Microscope, which provides realistic digital emulation of a high power light microscope. The common processing structure of these applications has been defined in previous work [4]. A database system has been developed to exploit the common processing structure of data analysis applications to perform various optimizations geared toward reducing turnaround time and improving throughput. The system accepts a declarative (SQL-style) form of a data analysis query, or batch of queries, and converts it to an imperative form, consisting of one or more loops over multidimensional ranges for some of the dataset attributes. Once in the imperative form, the system performs optimizations based on the algorithms commonly used by compilers to optimize intermediate or low level representations of program source code. The three algorithms implemented in the database system are loop fusion, common subexpression elimination, and dead code elimination.

After performing these optimizations, the original query is transformed into a sequence of loops iterating on subsets of the original dataset range. Each loop requires the use of one or more datasets. When a loop starts computing the result for a query, it must allocate memory for the query buffer holding intermediate aggregates and also for the query if the final results have not yet been partially processed by a previous loop. Additionally, once a query is returned to the client and completely computed it is deallocated from memory. The loops have the property that their execution order does not alter the result. Thus, the database system is able to arrange the execution of the loops in order to optimize memory usage, which is the subject of the current work.

Memory usage by the application is important because it affects the performance of the application, and may also affect performance of other applications sharing the same processor and memory space. The metrics that the database system attempts to optimize are the maximum and average memory usage of the loop set. The maximum memory usage of the loop set is the maximum amount of allocated memory at any one point in time during the execution of the entire loop set. The average memory usage of the loop set is the total amount of memory at every point in time during the execution of the loop set divided by the total amount of time the loop set requires to execute. Maximum memory usage is important because it determines whether page swapping will eventually be necessary during processing of the query. If the amount of memory required by the application exceeds the available system memory, then the amount of time for the loop set to complete will significantly increase due to page swapping. Average memory usage is important because a low average allows other applications to coexist on the same machine without these applications suffering from performance degradation. Lower memory usage is also desirable because it may improve the cache performance of the application.

In this paper, the author's contributions are described as follows:

- The implementation and adaptation of an extended SQL parser to serve as the front-end of the database that converts the declarative form of a query to its equivalent imperative form. The declarative form allows one to easily specify the query without worrying about the exact process by which the results are computed.

- The design and implementation of time optimization techniques in the current database multi-query planner borrowed from algorithms commonly used by compilers.
- The design and implementation of space optimization techniques in the current database multi-query planner. This includes the development of a novel heuristic, called "variable grouping", to perform memory usage optimizations that quickly arrive at a good loop ordering.

Chapter 2

Related Work

In this chapter, we discuss work done by other researchers that is related to the current work.

2.1 Register Allocation

The memory optimization problem is similar in many ways to the register allocation problem found in compiler optimization research. The register allocation problem consists of finding the best way to allocate registers in a code segment to minimize the amount of spill-over to memory. Current methods accomplish this by converting the problem to a graph where each node represents a variable and applying graph coloring heuristics with distinct colors representing registers, as described in [22]. This problem is of particular importance in embedded systems where the number of registers may be small and code execution time is critical [17].

2.2 Sparse Matrix Vector Multiplication

The current problem, particularly as formulated by the variable grouping heuristic that will be described in Section 4.6, is similar in many ways to the problem of optimizing sparse matrix vector multiplication. Sparse matrix vector multiplication involves the multiplication of a sparse matrix with a dense vector. In many situations, it is preferable that the non-zero elements of the sparse matrix appear near the diagonal of the matrix so that certain algorithms for computing the matrix vector multiplication can be applied which take advantage of the sparsity of the matrix. Much research has been done to formulate heuristics that attempt to minimize the “profile” or “bandwidth” of the sparse matrix ([9], [25], [11]). The profile of a matrix refers to the number of elements that appear between the first non-zero element in each row and the diagonal. The bandwidth of a matrix refers to the maximum number of elements that occur between the matrix diagonal and a non-zero element.

The problem of reordering the columns and rows of a matrix to minimize its profile and bandwidth is NP-hard [18]. Therefore, heuristics must be applied for large matrices. Many heuristics have been developed to optimize these quantities, notably the reverse Cuthill-Mckee ordering [10].

2.3 Relational Database Memory Optimizations

Much research has focused on the optimization of relational database memory usage, access, and structure. The performance of main-memory access is a growing bottleneck for database systems as the speed of CPUs outpace the speed of DRAM [20]. Vertically decomposed data structures can improve the cache performance of applications that require sequential access to data. Radix algorithms can typically improve the performance of random access operations. A new data organization model called PAX (Partition Attributes Across) is proposed by [1] that improves cache performance by grouping together attribute values in each page. This new model is demonstrated to have significant performance benefits over the N-ary storage model traditionally employed by relational database systems.

Chapter 3

Database Infrastructure

This chapter describes the infrastructure of the database system that has been developed to handle multiple data analysis queries on large datasets.

3.1 Database Architectural Overview

The database architecture we developed, a product of ongoing research, allows the efficient handling of multi-query workloads where user-defined operations are also part of the query plan [4, 6]. 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 methods are employed. This model relies on an "active semantic cache" that attaches semantic information to prior computed aggregates and permits the query optimizer to retrieve matching aggregates based on a query's meta-data description. 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 [4, 5, 6]. In essence, the previous 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. [16] 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.

3.1.1 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 meta-data information (i.e., the parameters and predicates specified for the query). Temporaries

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

Figure 3.1: General Data Reduction Loop.

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 [12], a range-aggregation query can be specified in the general loop format shown in Figure 3.1. A *Select* function identifies the sub-domain that intersects the query meta-data M_i for a query q_i. The sub-domain 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 sub-domain 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 3.1, we assume that there are n RHS collections of objects, denoted by $I_1, \ldots, I_n$, contributing to 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* [14], which must be commutative and associative operations.

3.2 Query Server

The compiler approach described in this work has been implemented as a front-end to the Query Server component of the database engine described in [2]. 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 [24] 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.

3.2.1 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 [6]. 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, called a processing chain, 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 3.2 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.

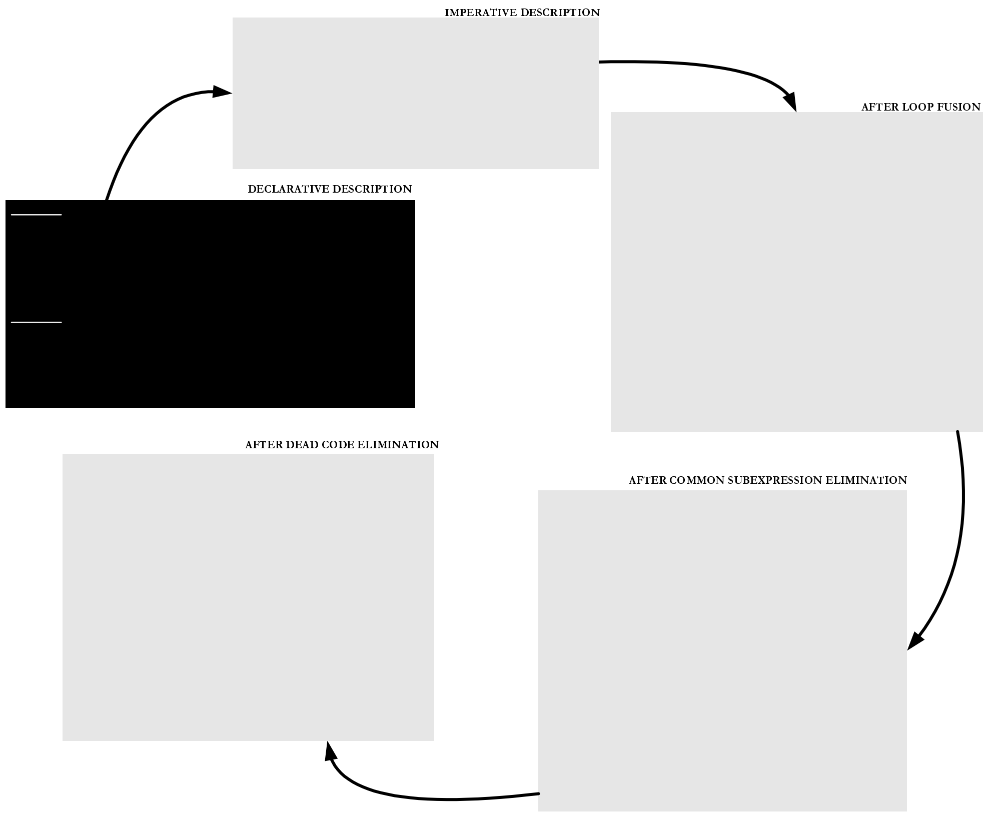

IMPERATIVE DESCRIPTION
AFTER LOOP FUSION
DECLARATIVE DESCRIPTION
AFTER DEAD CODE ELIMINATION
AFTER COMMON SUBEXPRESSION ELIMINATION

Figure 3.2: 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.

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 sub-sampled 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*). Query 2 selects data from the same collection as Query 1 with overlapping spatio-temporal boundaries. A different compositing method is used for this query.

Optimization for a query in a query batch occurs in a three-phase process in which the query is first integrated into the current plan, the redundancies are eliminated, and the loops comprising the imperative form of the query are reordered to reduce memory usage. 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.

3.3 Time Optimization Techniques

Time optimization involves the institution of methods to reduce the amount of time required to execute a query batch submitted to the database. Many projects have worked on database support for scientific datasets [8, 27]. Optimizing query processing for scientific applications using compiler optimization techniques to improve the speed at which queries are processed has attracted the attention of several researchers, including those in our own group. Ferreira et. al. [12, 13] 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 [13].

The database system we implemented uses 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. Sending 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.

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 [22].

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 [3] 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 $T_i \leftarrow copy(T_j)$, 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 [3] 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, hardware resources available for that purpose.

Chapter 4

Space Optimization Techniques

Space optimization involves the methods employed to make memory utilization as efficient as possible. This chapter details the notation and defines the terms used in this work. It also defines the evaluation metrics maximum and average memory usage. The brute force and branch-and-bound methods for obtaining optimal loop set orders are discussed, as well as the greedy and variable grouping heuristics for obtaining near-optimal loop orders.

4.1 Loop Sets

The multi-query planner of the database system performs optimizations on the submitted query batch and produces a sequence of loops that will be executed to compute the query results as seen in Chapter 3. The loops require the use of one or more query buffers to hold intermediate aggregates and the final results for the queries in the batch. The query planner uses both *temporary* and *output* query buffers. The output query buffer is used to store the final results of the query to be sent back to the client. The temporary query buffer is used to store intermediate results which are subsequently used to compute part of the contents of either the output query buffer or another temporary query buffer. For the purposes of this paper, each loop employs a set of variables and each variable requires a certain amount of memory. There is a one-to-one mapping between the set of variables and the set of query buffers. The set of variables required by loop i, here referred to as a “loop required variable set”, is denoted L_i where i ranges between 1 and the total number of loops in the loop sequence, denoted by N. Variables are referred to by the notation v_j, where j ranges between 1 and the total number of variables used by all loops, denoted by K. For example, if loop 1 uses variables v_1 and v_2, then the following relationship holds: L_1 = $\{v_1, v_2\}$.

The set of loop required variable sets L contains N loop required variable sets and is defined as follows:

$$L = \{L_1, L_2, ..., L_N\} \tag{4.1}$$

A permutation O defining the order of loop execution is used to order L and is defined as follows:

$$O = (o_1, o_2, ..., o_N) \tag{4.2}$$

Each element of O is a positive integer corresponding to a loop subscript in L such that $o_i = o_j$ if and only if $i = j$ and $1 \leq o_i \leq N$. The total ordering of loop set L with permutation O, denoted (L, O), is defined as follows:

$$(L, O) = (L_{o_1}, L_{o_2}, ..., L_{o_N}) \tag{4.3}$$

O defines an order in which loops are executed, and (L, O) defines an order in which the variables of each loop are allocated and deallocated.

The default sequence $O = (1, 2, ..., N)$ represents the order of the loops as they are submitted to the optimization algorithms. The output of the optimization algorithms is a new permutation $O' = (o'_1, o'_2, ..., o'_N)$ such that the resulting loop ordering has the minimum (or near-minimum) maximum or average memory usage.

4.2 Calculating the Metrics

The maximum and average memory usage metrics can be computed for a given loop ordering. The order of execution of the loops affects the allocation and deallocation of query buffers. Buffer space for a query is allocated for a given loop if it has not already been allocated by a previous loop. Thus, the first loop that is executed must allocate memory for all of the queries that it partially or completely computes. A variable corresponding to a query buffer that a loop partially or completely computes is referred to here as a “required variable”. The set of variables that are live during the execution of a particular loop are the “allocated variables” of the loop. If a particular variable is allocated but not required, then it is termed an “unused variable”. Buffer space for a query is deallocated after a loop completes and the query computation has finished and the results can be returned to the client. Thus, the last loop must deallocate all memory used by all of the query buffers that it uses once it completes its execution.

```
for each point in $bb_1$:(14.964, 19.954, 199206)(20.000,55.000,199206) {
    $v_2[bb_1]$    =    $\mathcal{R}etrieval(I)$
}
for each point in $bb_2$:(0.000, 15.972, 199206)(14.928, 65.000, 199206) {
    $v_3[bb_2]$    =    $\mathcal{C}omposite(v_2, MaxNDVI)$
}
for each point in $bb_3$:(14.964, 55.038, 199206)(20.000, 65.000, 199206) {
    $v_1[bb_3]$    =    $\mathcal{R}etrieval(I)$
}
for each point in $bb_4$:(14.964, 15.972, 199206)(20.000, 19.929, 199206) {
    $v_2[bb_4]$    =    $\mathcal{C}orrection(v_2, WaterVapor)$
}
```

Figure 4.1: Example Query in Imperative Form.

Loop	*Required Variables*	*Allocated Variables*	*Allocated Memory*	*Running Time*
1	$L_1 = \{v_1, v_2\}$	$\{v_1, v_2\}$	25	10
2	$L_2 = \{v_2, v_3\}$	$\{v_1, v_2, v_3\}$	45	30
3	$L_3 = \{v_1\}$	$\{v_1, v_2\}$	25	20
4	$L_4 = \{v_2\}$	$\{v_2\}$	15	15

Table 4.1: Example Loop Set. v_1 = 10, v_2 = 15, and v_3 = 20.

Consider the example given in Table 4.1, corresponding to the query shown in Figure 4.1. The first column of the table identifies the loop and the contents of the loop's required variable set is shown in the second column. The allocated variables are shown in the third column and the amount of memory allocated for these variables is given in the fourth column. The last column, *Running Time*, refers to the amount of time that the loop requires to execute and is a function of the loop iteration domain and the execution time of the individual loop body statements. In the example, there are a total of three variables used by all of the loops: v_1, v_2, and v_3. Suppose the table defines the loop execution order, with the first row corresponding to the first loop to be executed and the last row corresponding to the last loop to be executed. The first loop allocates variables v_1 and v_2. Once the first loop completes, it does not deallocate any variables because both variables are used by subsequent loops. The second loop allocates variable v_3 and leaves variable v_1 allocated even though it does not require the variable because v_1 is used in the next loop. The second loop deallocates variable v_3 once it completes execution because no subsequent loops compute results to be stored into this variable. The third loop does not need to allocate any variables because the variable it requires, v_1, has already been allocated by a previous loop. After the third loop completes, it deallocates variable v_1 because it will no longer be used, i.e. the query results stored in v_1 are completely computed and can be returned to the client. Finally, the last loop does not allocate any new variables and deallocates v_2 after it completes. The third column in the table gives the variables that are allocated while each loop executes as described. The general equation defining the allocated variable set of loop o_i given the set of loop required variable sets L ordered by the permutation $O = (o_1, o_2, ..., o_N)$, denoted $A(L_{o_i}, L, O)$, is as follows:

$$A(L_{o_i}, L, O) = \begin{cases} L_{o_i} \cup \left(\left(\bigcup_{j=1}^{i-1} L_{o_j} \right) \cap \left(\bigcup_{k=i+1}^{N} L_{o_k} \right) \right) & : \quad 2 \leq i \leq N-1 \\ L_{o_i} & : \quad i = 1, N \end{cases} \tag{4.4}$$

In order to determine the value of the maximum memory usage metric for this example loop ordering, the memory usage for each loop must be computed. Suppose variables v_1, v_2, and v_3 have sizes 10, 15, and 20, respectively. The units are omitted because they are not relevant to describing the method used to compute the metric. The amount of memory used by loop 1 is equal to v_1 + v_2 = 10 + 15 = 25. The Allocated Memory column in the table gives the amount of memory allocated for each loop. The maximum value in this column occurs at the second row for loop 2 and is 45, which is the maximum memory usage of this example loop ordering. The general equation defining the maximum memory usage of set L and permutation O denoted by $M_{max}(L, O)$ is as follows:

$$M_{max}(L, O) = \textbf{max}(|A(L_{o_1}, L, O)|, |A(L_{o_2}, L, O)|, ..., |A(L_{o_N}, L, O)|) \tag{4.5}$$

where the notation $|A(L_{o_i}, L, O)|$ is used to denote the summation of the sizes of all of the variables in set $A(L_{o_i}, L, O)$.

In order to determine the value of the average memory usage metric for this example loop sequence, both the memory usage and running times of each loop must be considered. The fourth column in Table 4.1 gives the running time of the loop. The

Loop	*Required Variables*	*Allocated Variables*	*Allocated Memory*	*Running Time*
3	$L_3 = \{v_1\}$	$\{v_1\}$	10	20
1	$L_1 = \{v_1, v_2\}$	$\{v_1, v_2\}$	25	10
2	$L_2 = \{v_2, v_3\}$	$\{v_2, v_3\}$	35	30
4	$L_4 = \{v_2\}$	$\{v_2\}$	15	15

Table 4.2: Example Loop Set After Reordering.

units of time are omitted because they are not relevant to describing the method used to obtain the average memory usage metric. The general equation defining the average memory usage of set L and permutation O, denoted $M_{avg}(L, O)$, is as follows:

$$M_{avg}(L, O) = \frac{\left(\sum_{i=1}^{N} |A(L_{o_i}, L, O)| R(o_i)\right)}{\left(\sum_{i=1}^{N} R(o_i)\right)} \tag{4.6}$$

where the notation $R(o_i)$ is used to denote the running time of loop o_i. The average memory usage metric for the example ordered input loop set has a value of 31.

To demonstrate the effect of loop order on the average and maximum memory usage metrics, consider Table 4.2 in which loop 3 has been moved to the beginning of the loop order from its original position shown in Table 4.1. The memory utilization for loop 2 decreased from 45 to 35. Similarly, the memory utilization for loop 3 decreased from 25 to 10. The maximum memory usage occurs at loop 2 and now has a lower value of 35. Indeed, the average memory usage decreased to 23 from its original value of 31. There is no allocated memory due to unused variables in this loop ordering because the set of required variables is equivalent to the set of allocated variables for every loop. As a result, one can conclude that the ordering is optimal with respect to both maximum and average memory usage metrics. However, it is important to note that the loop order yielding the minimum maximum memory usage may not always be the same as that yielding the minimum average memory usage.

4.3 Brute Force Method

In order to determine the order of the loops such that either maximum or average memory usage metrics are minimized, every possible loop set order has to be considered and the metrics computed for each. This is the essence of the "brute force" method. The order computed by this method always yields the optimal maximum or average memory usage. Unfortunately, the brute force method requires exponential time in the number of loops in the loop set and, therefore, can only be employed for very small numbers of loops. This is due to the fact that the method relies on computing all possible combinations of loop orderings. For a loop set of size N, the number of possible loop orderings is $N!$. This can be seen if one considers the possible loop orderings as forming a tree. The root node of this tree has N children, and each node at depth d has $N - d$ children, with each node corresponding to a particular loop. At depth d there will be $\frac{N!}{(N-d)!}$ nodes and the total number of nodes in the brute force tree for N loops, $T(N)$, is the sum:

$$T(N) = \sum_{d=0}^{N} \left(\frac{N!}{(N-d)!} \right) \tag{4.7}$$

4.4 Branch-and-Bound Method

A somewhat more effective technique is to use a "branch-and-bound" strategy [19] on the tree constructed during the brute force search. The branch-and-bound strategy is generally used to prune the brute force tree by maintaining a cost value or bound at each node and prioritizing the search along lower cost branches. The branch-and-bound technique was applied by employing a lower bound estimation of either the maximum or average memory usage of a prefix of a loop order, i.e. the sequence of nodes from the root to a non-leaf node. The technique allows us to prioritize the exploration of branches with the minimum lower bound and maximum depth, avoiding branches with confirmed poor orders, until a complete loop order is found. The branch-and-bound algorithm can significantly reduce the amount of time required to compute the optimal loop set order; however, the running time of this algorithm is still exponential in the number of loops in the worst case. This is the case because the lower bound may not be effective in pruning the brute force tree for certain loop sets.

The algorithm implementation uses tree and list data structures. The tree represents the loop ordering prefixes explored so far. The root of the tree represents the initial state of the execution of the loop set in which no variables are allocated. It has N children corresponding to the N possible choices for the first loop in the loop order, but the root itself does not represent a loop. The children of the root each have $N - 1$ children because the number of possible choices for the next loop to be executed has been reduced by 1. In general, a node in the tree at depth d has $N - d$ children because d loops have been chosen from the set of N loops and there are $N - d$ remaining loops.

The list data structure is maintained in order to keep track of the search order of the branches in the tree. The leaves of the tree form the elements of the list data structure. The list nodes are sorted in order of increasing lower bound for either maximum or average memory usage. The following formulas are used to compute the lower bound $B_{avg}(n, T, L)$ for average memory usage of a tree node, n, given tree T and loop list L:

$$Prefix(n, T) =$$

$$(LoopNum(Parent(n, T, d)),$$

$$LoopNum(Parent(n, T, d-1)),$$
$$..., LoopNum(Parent(n, T, 1)), LoopNum(n)) \tag{4.8}$$

$$LoopOrder(n, T, L) = Concatenate(Prefix(n, T), L - Prefix(n, T)) \tag{4.9}$$
$$AllocMem(n, L, T) = |A\left(Loop\left(n\right), L, LoopOrder\left(n, T, L\right)\right)| \tag{4.10}$$

$$\begin{aligned} &B_{avg}(n, T, L) = \\ &\quad B_{avg}\left(Parent\left(n\right), T, 1\right) + \\ &\quad \left(AllocMem(n, L, T) - |Loop\left(n\right)|\right) R\left(n\right) \end{aligned} \tag{4.11}$$

The $LoopNum(n)$ function, used in Equation 4.8, returns the loop number corresponding to the tree node passed as a parameter. The $Parent(n, T, d)$ function, used in Equations 4.8 and 4.11, returns dth ancestor of tree node n in tree T. The dth ancestor of a tree node n is the parent node of the $(d-1)$th ancestor of n. If $d = 1$ then the immediate parent node of n is returned. The $Concatenate(P, Q)$ function, used in Equation 4.9, concatenates the elements of list P to the end of list Q. Also in Equation 4.9, the subtraction operator with list operands, $P - Q$, results in a list containing all of the elements in list P that are not also contained in list Q. Equation 4.11 holds for all non-root tree nodes n. The root node, r, has its lower bound initialized to:

$$B_{avg}(r, T, L) = |L_1|R(1) + |L_2|R(2) + \cdots + |L_N|R(N) \tag{4.12}$$

where the elements of list L are represented as L_i for $1 \leq i \leq N$. The intuition behind this lower bound is that the minimum average memory usage must be the weighted average of the required variables of each loop, and increases based on the discovery of unused variables as nodes are added to the tree.

The following formula is used to compute the lower bound of maximum memory usage, $B_{max}(n, T, L)$, for node n in tree T and loop list L:

$$\begin{aligned} & B_{max}(n, T, L) = \\ & \quad max(\quad B_{max}(Parent(n), T, 1), \\ & \qquad\qquad |A(Loop(n), L, LoopOrder(n, T, L))|) \end{aligned} \tag{4.13}$$

Equation 4.13 holds for all non-root tree nodes n. The root node, r, has its lower bound initialized as follows:

$$B_{max}(r, T, L) = max(|L_1|, |L_2|, ..., |L_N|) \tag{4.14}$$

Once the lower bound has been computed for each list node, the list is ordered by increasing lower bound and secondarily by decreasing depth. The first list node corresponds to the tree node with the minimum lower bound and maximum depth. The implementation proceeds by replacing the first list node with list nodes corresponding to the child tree nodes of the tree node corresponding to the first list node. After this modification is made, the list is again sorted in order of increasing lower bound and decreasing depth. Branches in the tree are pruned by virtue of the fact that the first node of the list always has the lowest bound and is always chosen for further processing. Furthermore, if there are several nodes with the same lowest bound the node at the front of the list will have the maximum depth among these nodes. The process of replacing the first list node repeats until the first list node contains a tree node with depth N because at this point a complete optimal loop order has been found.

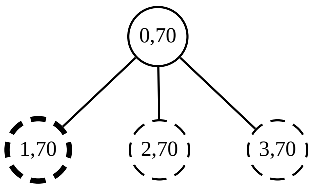
As an example of the operation of this algorithm when optimizing for average memory usage, consider the input loop set given in Table 4.3. For simplicity, the running time of each loop is 1. The variables v_1, v_2, and v_3 have sizes 10, 15, and 20, respectively. The state of the tree at different points in the processing of the algorithm

Loop	*Required Variables*	*Running Time*
1	$L_1 = \{v_1, v_2\}$	1
2	$L_2 = \{v_2, v_3\}$	1
3	$L_3 = \{v_1\}$	1

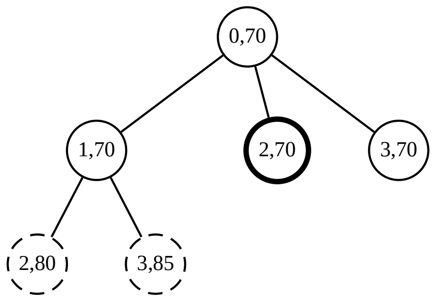
Table 4.3: Example Loop Set for Branch-and-Bound. v_1 = 10, v_2 = 15, v_3 = 20.

is depicted in Figures 4.2(a), 4.2(b), 4.2(c), and 4.2(d), with each tree node containing its associated loop number, i, and lower bound, b, in the pair (i, b). The algorithm begins by initializing the lower bound of the root node. Then the immediate children of the root node are added and their lower bounds computed, as seen in Figure 4.2(a). The most recently added nodes are bordered by a dashed line. The next node to be explored is marked by a thicker border. Since all of the leaf nodes have the same lower bound, the node to be explored next is simply the node with the lowest loop number. In Figure 4.2(b), the children of this node have been added to the tree and their lower bounds computed. The algorithm terminates with the tree in Figure 4.2(d), since the node at the front of the list has depth 3 and the number of loops N=3. The number of nodes in the tree explored using branch-and-bound before an optimal solution was found in this example is 9. In comparison, the brute force method would have searched the entire tree of 16 nodes.

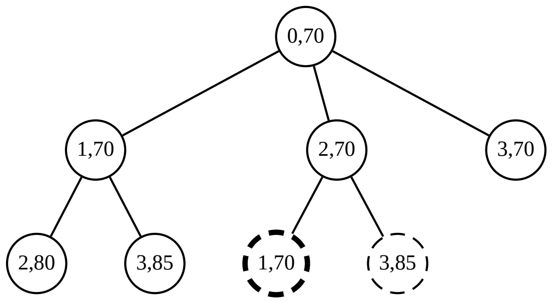
For certain input loop sets, it may be possible to divide the original set into two or more subsets that are independent of each other in terms of the variables that the loops share among the subsets. These subsets represent what are known as connected components [26]. Therefore, in our implementation the input loop set is preprocessed with a connected components analysis and each of the individual trees that result from this analysis is submitted separately to the branch-and-bound algorithm.



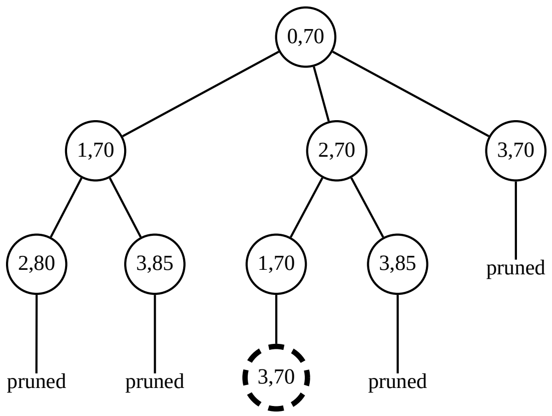
(a) First Iteration



(b) Second Iteration



(c) Third Iteration



(d) Fourth Iteration

Figure 4.2: Branch-and-bound Example

4.5 Greedy Heuristic

As we have previously stated, loops may sometimes have live variables (query buffers) allocated that are not needed by the loop. This condition arises because there may exist at least one loop before and after that loop in the loop set order computing the variable. The greedy heuristic attempts to identify the loops using the maximum amount of memory and reduce this maximum by rearranging the loops to eliminate the allocation of any unused variables. The rearrangement occurs using the following method:

1. The unused variable set of the loop with maximum memory usage, i_{max}, is determined.

2. For each unused variable v_u, the loops preceding i_{max} in the current loop order that use v_u are moved immediately after i_{max}. If this rearrangement results in a lower maximum or average memory usage, the rearrangement is kept and the new loop with maximum memory usage is determined; otherwise, the rearrangement is reversed.

3. If the previous step did not result in a new ordering with a lower maximum or average memory usage, the loops succeeding i_{max} in the current loop order that use v_u are moved immediately before i_{max}. If this rearrangement results in a lower maximum or average memory usage, the rearrangement is kept and the new loop with maximum memory usage is determined; otherwise, the rearrangement is reversed.

This process is iterated for each unused variable v_u until no rearrangement results in an improved memory usage. The process is then recursively applied to the loops preceding and succeeding i_{max}.

Loop	*Required Variables*	*Allocated Variables*	*Allocated Memory*
1	$L_1 = \{v_1, v_2\}$	$\{v_1, v_2\}$	25
2	$L_2 = \{v_2, v_3\}$	$\{v_1, v_2, v_3\}$	45
3	$L_3 = \{v_1\}$	$\{v_1, v_2\}$	25
4	$L_4 = \{v_2\}$	$\{v_2\}$	15

Table 4.4: Example Loop Set for Greedy Heuristic.

Loop	*Required Variables*	*Allocated Variables*	*Allocated Memory*
1	$L_1 = \{v_1, v_2\}$	$\{v_1, v_2\}$	25
3	$L_3 = \{v_1\}$	$\{v_1, v_2\}$	25
2	$L_2 = \{v_2, v_3\}$	$\{v_2, v_3\}$	35
4	$L_4 = \{v_2\}$	$\{v_2\}$	15

Table 4.5: Example Loop Set for Greedy Heuristic Reordered.

As an example of applying this algorithm when optimizing for maximum memory usage, consider the loop order given in Table 4.4. The loop with maximum memory usage is 2. The unused variable in this loop is v_1. Therefore, the loops after loop 2 that use variable v_1 are moved immediately before loop 2, as in Table 4.5. This reduces the maximum memory usage at loop 2 to 35 because variable v_1 is now deallocated before loop 2 begins execution. Since there are no longer any unused variables for loop 2 there is no way to reduce the memory usage for this loop further and, therefore, in this case the optimal loop order to minimize maximum memory usage has incidentally been found.

4.6 Variable Grouping Heuristic

The variable grouping heuristic considers each variable (query buffer) in turn and groups the loops using this variable together in the loop execution order. The intuition is that once the loops using a common variable are grouped together, there are no intervening loops that will require the memory space for a variable to be held unnecessarily. This section describes several aspects and variations of this heuristic.

4.6.1 Representing Loops Using Bit Vectors

Each loop in the input loop order is mapped to a bit vector and the mapping is maintained by map M. The bit vectors contain one bit per variable and each bit is a 1 or 0 depending on whether the loop uses or does not use the variable corresponding to that bit position. Bit vectors are denoted by b_i in this section, with the following general form:

$$b_i = (d_1, d_2, ..., d_K) \tag{4.15}$$

where d_j for $1 \leq j \leq K$ represents a binary digit in the bit vector b_i. d_j represents the usage of variable v_j and K represents the total number of variables used by all of the loops in the loop set. If two or more loops happen to use the exact same variables or, in other words, have the same "variable usage pattern" then they will be mapped to the same bit vector. For example, in Table 4.6 loops 1, 2, and 4 have the same variable usage pattern and, therefore, correspond to the same bit vector $(1, 1, 0)$. Although there are four loops in Table 4.6, there are only two distinct bit vectors representing two distinct variable usage patterns and the resulting mapping is shown in Table 4.7. After all of the loops are mapped to a bit vector, the resulting map M contains Q elements where Q can range anywhere between 1 and N depending upon how many

Loop	*Required Variables*	*Bit Vector*
1	$L_1 = \{v_1, v_2\}$	$(1,1,0)$
2	$L_2 = \{v_1, v_2\}$	$(1,1,0)$
3	$L_3 = \{v_3\}$	$(0,0,1)$
4	$L_4 = \{v_1, v_2\}$	$(1,1,0)$

Table 4.6: Example Loop Set for Variable Grouping Heuristic with Identical Loops.

Bit Vector	*Loops*
$(1,1,0)$	{1, 2, 4}
$(0,0,1)$	{3}

Table 4.7: Bit Vector-Loop Mapping M for Example Loop Set.

loops share the same variable usage pattern. In addition, the algorithm maintains a bit vector set B of size Q containing only the bit vector keys contained by map M:

$$B = \{b_1, b_2, ..., b_Q\} \tag{4.16}$$

This bit vector set is used in subsequent stages of the algorithm.

Once the input loop set is converted to a bit vector set B, the algorithm must first determine the order to process the bit positions, described in Section 4.6.2, and then order the bit vectors in B to produce a loop execution order that minimizes maximum or average memory usage, which is the objective of the $Aggregate$ function described in Section 4.6.3. The notation used here to define an ordering of set B is similar to that described in Section 4.2 to denote the ordering of set L. We use a permutation $W = (w_1, w_2, ..., w_Q)$ containing Q positive integers such that $w_i = w_j$ if and only if $i = j$ and $1 \leq w_i \leq Q$. The bit vector set B and the permutation W together form a total ordering of set B, denoted (B, W), defined as follows:

$$(B, W) = (b_{w_1}, b_{w_2}, ..., b_{w_Q}) \tag{4.17}$$

4.6.2 Variable-Bit Position Ordering Methods

After the bit vector set B corresponding to the input loop order has been created, the variable grouping heuristic decides the order in which to process the bit positions, and hence the variables, of the bit vectors representing the loops. This order is important in determining whether the algorithm produces a good result because an order that is imposed on the loop set by the $Aggregate$ method (see Section 4.6.3) for a given variable may affect any order that may be imposed by the processing of subsequent variables. The variable ordering methods used by the current work and the abbreviations for the corresponding algorithm are listed in Table 5.1.

The first variable ordering method, the "unused memory potential" method, is based on the observation that the cost that this algorithm attempts to minimize is the amount of unused memory allocated during the execution of any loop. Given a bit vector set B and a permutation $W = (w_1, w_2, ..., w_Q)$ of this bit vector set, one may compute the set of allocated variables for the loops mapped to any bit vector b_{w_i} in set B where $1 \leq i \leq Q$ using the following equation:

$$A(b_{w_i}, B, R) = V(b_{w_i}) \bigcup \left(\left(\bigcup_{j=1}^{i-1} V(b_{w_j}) \right) \bigcap \left(\bigcup_{k=i+1}^{N} V(b_{w_k}) \right) \right) \tag{4.18}$$

Function V in Equation 4.18 accepts as its only parameter a bit vector and returns the set of variables that are used by the loops mapped to this bit vector in map M described in Section 4.6.1. Equation 4.18 can be used to determine the set of unused variables $U(b_{w_i}, B, R)$ for the loops mapped to bit vector b_{w_i} contained in bit vector set B ordered by sequence W defined as follows:

$$U(b_{w_i}, B, W) = A(b_{w_i}, B, W) - V(b_{w_i}) \tag{4.19}$$

The unused memory cost function $C(b_{w_i}, B, W)$ gives the amount of memory that is allocated to unused variables for the set of loops that correspond to bit vector b_{w_i} and is defined as follows:

$$C(b_{w_i}, B, W) = |U(b_{w_i}, B, W)| R(b_{w_i}) \tag{4.20}$$

Equation 4.20 uses the function R to return the total running time of all loops that are mapped to bit vector b_{w_i} via map M. The total amount of unused memory $C_{total}(B, W)$ is the summation of Equation 4.20 over all of the bit vectors in B:

$$C_{total}(B, W) = \sum_{i=1}^{Q} (C(b_{w_i}, B, W)) \tag{4.21}$$

which is the quantity that the variable grouping heuristic seeks to minimize.

The notion of unused variables is displayed graphically for three variables in Figure 4.3 which depicts a Venn diagram consisting of three sets V_1, V_2, and V_3 such that set V_i contains the loops that use variable v_i for $1 \leq i \leq 3$. The three circles representing the sets divide the space into seven distinct subsets which have been labeled S_j for $1 \leq j \leq 7$ (omitting the eighth subset which lies outside all of the circles). Each subset S_j corresponds to a bit vector representing a distinct variable usage pattern. The dashed line indicates a possible "path" through these subsets which gives the order of loop execution. Entry of the line into a subset represents the fact that the loops that are contained by the subset have begun execution. Exit of the line from a subset represents the fact that the loops that are contained by the subset have all completed execution. It is possible for a subset to be empty representing the case where no loops have the variable usage pattern of the associated bit vector. In such a case, unused variables that may be allocated will not contribute to the unused memory cost function for this subset. The subsets visited by the path are listed in Table 4.8. The second column of the table “Equivalent” lists the set operations performed on V_i for $1 \leq i \leq 3$ to yield the corresponding subset. The associated bit vector is listed under the column “Bit Vector”. The last column “Unused Variables” lists the result of applying Equation 4.19 with bit vector b_{w_i} given by the “Bit Vector” column, bit vector set B given by the set containing all of the bit vectors in this column, and permutation $W = (1, 3, 2, 6, 7, 5, 4)$. The variable v_1 is allocated unnecessarily during the execution of loops contained by subsets S_2 and S_6. The part of the path in Figure 4.3 with this unused variable allocated is shown with a darker line.

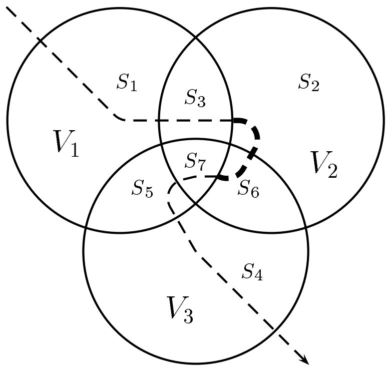


Figure 4.3: Variable Grouping Heuristic: Set Representation of Bit Vectors Showing Unused Memory on Loop Execution Path

Subset	*Equivalent*	*Bit Vector*	*Unused Variables*
S_1	$V_1 - V_2 - V_3$	$b_1 = (1, 0, 0)$	$\emptyset$
S_3	$(V_1 \bigcap V_2) - V_3$	$b_3 = (1, 1, 0)$	$\emptyset$
S_2	$V_2 - V_1 - V_3$	$b_2 = (0, 1, 0)$	$\{v_1\}$
S_6	$(V_2 \bigcap V_3) - V_1$	$b_6 = (0, 1, 1)$	$\{v_1\}$
S_7	$V_1 \bigcap V_2 \bigcap V_3$	$b_7 = (1, 1, 1)$	$\emptyset$
S_5	$(V_1 \bigcap V_3) - V_2$	$b_5 = (1, 0, 1)$	$\emptyset$
S_4	$V_3 - V_1 - V_2$	$b_4 = (0, 0, 1)$	$\emptyset$

Table 4.8: Subset Equivalent of Bit Vectors and Possible Loop Execution Order

$vghu$

Consequently, the permutation W chosen in the variable grouping heuristic to order the bit vector set B can be based on the amount of unused memory that a variable contributes given some loop order. The maximum amount of unused memory that a variable can contribute is termed its "unused memory potential", denoted $U(v_k, L)$ for variable v_k and set of loop required variable sets L, defined in the following equation:

$$U(v_k, L) = v_k \left(\sum_{i \in \{i : v_k \notin L_i\}} R(i) \right) \tag{4.22}$$

Equation 4.22 states that the unused memory potential of a given variable v_k is the product of the variable's size and the sum of the running times of all loops that do not require this variable to be allocated, where the notation $R(i)$ is used as defined in Section 4.2 to mean the running time of loop i. This quantity can be used to prioritize the variables used by all of the loops and assign an order in which to process the bit positions of the bit vector representations of the loops.

$vghs$

The second method is to prioritize variables solely based on their size. The reasoning behind this method is that the unused memory cost function (Equation 4.21) depends directly on the size of the variables used by the loops and, therefore, grouping loops together according to the size of the shared variable may reduce the amount of unused memory in the resultant loop execution order. This method ignores the running time of each loop as a factor in determining the variable order; therefore, it may be most productive when applied to an input loop set that has an even running time distribution among the loops.

$vghr$

The third method is to randomly choose an order in which to process the bit positions. This method could arrive at an order that may not be the best; however, if several random orders are tried and the $Aggregate$ function (see Section 4.6.3) is executed using each of the resulting bit position orders, it may be possible to arrive at a very good loop execution order by sheer chance. The implementation of this method in the *vghr* algorithm involves randomly generating 100 different bit position orders.

$vghd$

The fourth method is called “deterministic reordering” and reorders loops based on decreasing values for the unused memory cost function computed for each bit vector, using Equation 4.20 after an arbitrary bit position order has been tried and the $Aggregate$ function has been executed using this order. After loops have been thus reordered, the $Aggregate$ function is executed yielding a new loop execution order and the unused memory cost function for each bit vector is recomputed, resulting in a new bit position order. This process repeats until the average or maximum memory does not decrease for several iterations.

4.6.3 Aggregate Function

Once the variable bit position order has been chosen and each variable has been assigned a distinct bit position in the bit vectors that represent the loops, the $Aggregate$ function is invoked. The purpose of this function is to iteratively group bit vectors together according to the values of the bits in each bit position while, at the same time, preserving the groupings of previous iterations. The reason that previous groupings are preserved is that the function assumes the variable bit position order prioritizes the variables that potentially contribute the most to the unused memory cost function. Thus, the bit vector order imposed by processing a given bit position should not be disturbed by future orderings that may arise from processing subsequent bit positions in an attempt to avoid the introduction of unused memory by such a disruption.

The $Aggregate$ function uses a data structure, here called an $RSet$, that is recursively defined as a set containing the following elements:

- A bit vector
- An unordered set of $RSet$s
- An ordered set of $RSet$s

The input $RSet$ is initially equivalent to the bit vector set B described in Section 4.6.1. The $Aggregate$ function attempts to find an order for the bit vector set according to the bit values in the bit vectors contained by the $RSet$. The function imposes a partial or, in some cases, a total order on the input $RSet$ parameter by performing one of the following actions:

- Creating unordered subsets out of elements of an unordered set.
- Converting unordered sets to ordered sets.

The actions performed are driven by the value of the bit located at a particular bit position in each of the bit vectors contained by the $RSet$.

The input $RSet$ is denoted G and is represented as a set of Q bit vectors from set B described in Section 4.6.1:

$$G = \{b_1, b_2, ..., b_Q\} \tag{4.23}$$

Bit Vector Name	*Bit Vector Contents*
b_1	(0, 1, 0)
b_2	(1, 1, 0)
b_3	(0, 1, 1)
b_4	(1, 0, 1)

Table 4.9: Example Bit Vector Set Input for Aggregate Function

For example, consider the set of bit vectors in Table 4.9. There are four bit vectors with each vector containing a total of three bits. These bit vectors represent different variable usage patterns in the input loop set and each vector maps to one or more loops; however, the actual loops are unimportant in demonstrating the operation of the $Aggregate$ function. There are three variables in use among all of the loops in the input loop set, hence three bit positions appear per bit vector. The $Aggregate$ function will be invoked three times, once for each of the three bit positions.

The following $RSet$ will be passed to the $Aggregate$ function on the first invocation:

$$G_1 = \{010, 110, 011, 101\} \tag{4.24}$$

where the unordered set is signified with braces. The $Aggregate$ function accepts the G_1 parameter of type $RSet$ and processes the first bit position in each bit vector contained in bit vector set B. Once this processing is complete, the $Aggregate$ function will return the following $RSet$ as output:

$$G_2 = \{\mathbf{0}10, \mathbf{0}11, \{\mathbf{1}10, \mathbf{1}01\}\} \tag{4.25}$$

This demonstrates the partitioning effect that the $Aggregate$ function will have on unordered sets that contain 1s and 0s. The partitioning process results in a grouping of the 1 bits such that the output $RSet$ effectively groups loops together that use the variable corresponding to the bit position that was just processed, in this case the first bit position.

The function will also convert an unordered $RSet$ into an ordered one under the following conditions:

- A partition between 1 bits and 0 bits is not possible.
- A 0 bit is contained in a subset that also contains a 1 bit.

This conversion is performed to prevent the 0 bit from being relocated in between the 1 bits when a subsequent bit position is processed or when the final loop execution order is output resulting in additional unused memory from the allocation of an unused variable. As an example of this effect consider the invocation of $Aggregate$ on the G_2 $RSet$, which is the output of the previous invocation given in Equation 4.25, resulting in the following output:

$$G_3 = (1\mathbf{0}1, 1\mathbf{1}0, \{0\mathbf{1}0, 0\mathbf{1}1\}) \tag{4.26}$$

where the parentheses are used to denote the ordered set. Now the loops that use the variable mapped to the second bit position in each bit vector, v_2, will all be grouped together in the final loop execution order. The last invocation of $Aggregate$ results in the following output:

$$G_4 = (10\mathbf{1}, 11\mathbf{0}, 01\mathbf{1}, 01\mathbf{0}) \tag{4.27}$$

Equation 4.27 corresponds to a total ordering of bit vector set $B = \{b_1, b_2, b_3, b_4\}$ by permutation $W = (4, 2, 3, 1)$:

$$(B, W) = (b_4, b_2, b_3, b_1) \tag{4.28}$$

The final bit vector order and the contents of each bit vector is listed in Table 4.10.

Bit Vector Name	*Bit Vector Contents*
b_4	(1, 0, 1)
b_2	(1, 1, 0)
b_3	(0, 1, 1)
b_1	(0, 1, 0)

Table 4.10: Example Bit Vector Set Output for Aggregate Function

Consider the larger example input loop set given in Table 4.11 which was taken from a synthetic loop set used to obtain the results in Section 5.2. This example contains eight loops using a total of seven different variables, with the first variable v_1 unused. The loop set is converted to the bit vector set shown in Figure 4.4 such that each loop i is mapped to a bit vector b_i. Figure 4.5 shows the output bit vector set after the $Aggregate$ function completes processing the input bit vector set in Figure 4.4. A box is drawn around the groupings of 1 bits with the top-most 1 bit in each box representing the point at which the corresponding variable is allocated and the bottom-most 1 bit representing the point at which this variable is deallocated. Thus, the area of the boxes divided by the sum of loop running times approximates the average amount of memory allocated. These figures demonstrate that the $Aggregate$ function reduces the average memory usage of the loop set since the area of the boxes in the input bit vector set shown in Figure 4.4 is greater than the area of the boxes in the output bit vector set shown in Figure 4.5.

$$\begin{array}{c|cccccccc}
 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \\
\hline
b_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} \\
b_2 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & \boxed{1} \\
b_3 & 0 & 0 & 0 & 0 & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{0} \\
b_4 & 0 & 0 & 0 & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} \\
b_5 & 0 & 0 & 0 & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{1} \\
b_6 & 0 & 0 & \boxed{1} & 0 & \boxed{0} & \boxed{0} & 0 & 0 \\
b_7 & 0 & 0 & \boxed{1} & 0 & \boxed{1} & \boxed{1} & 0 & 0 \\
b_8 & 0 & \boxed{1} & \boxed{1} & 0 & \boxed{1} & \boxed{1} & 0 & 0
\end{array}$$

Figure 4.4: Variable Grouping Heuristic: Aggregate Function Input. Note that the boxes contain zeros.

$$\begin{array}{c|cccccccc}
 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \\
\hline
b_6 & 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 & 0 \\
b_7 & 0 & 0 & \boxed{1} & 0 & \boxed{1} & \boxed{1} & 0 & 0 \\
b_8 & 0 & \boxed{1} & \boxed{1} & 0 & \boxed{1} & \boxed{1} & 0 & 0 \\
b_3 & 0 & 0 & 0 & 0 & \boxed{1} & \boxed{1} & 0 & 0 \\
b_4 & 0 & 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 \\
b_5 & 0 & 0 & 0 & \boxed{1} & 0 & 0 & \boxed{1} & \boxed{1} \\
b_2 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & \boxed{1} \\
b_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1}
\end{array}$$

Figure 4.5: Variable Grouping Heuristic: Aggregate Function Output. Note that the boxes do not contain zeros.

Loop Set Name	*Required Variable Set*
1	$L_1 = \{v_8\}$
2	$L_2 = \{v_7, v_8\}$
3	$L_3 = \{v_5, v_6\}$
4	$L_4 = \{v_4\}$
5	$L_5 = \{v_4, v_7, v_8\}$
6	$L_6 = \{v_3\}$
7	$L_7 = \{v_3, v_5, v_6\}$
8	$L_8 = \{v_2, v_3, v_5, v_6\}$

Table 4.11: Variable Grouping Heuristic: Example Loop Set Input

Loop Set Name	*Required Variable Set*
6	$L_1 = \{v_3\}$
7	$L_2 = \{v_3, v_5, v_6\}$
8	$L_3 = \{v_2, v_3, v_5, v_6\}$
3	$L_4 = \{v_5, v_6\}$
4	$L_5 = \{v_4\}$
5	$L_6 = \{v_4, v_7, v_8\}$
2	$L_7 = \{v_7, v_8\}$
1	$L_8 = \{v_8\}$

Table 4.12: Variable Grouping Heuristic: Example Loop Set Output

Chapter 5

Performance Studies

This chapter presents the performance evaluation of the various memory optimizing algorithms for actual queries from an application that processes remote sensing data and for synthetic loops which allows us to explore a larger portion of the optimization space in a more controlled fashion. Table 5.1 lists the algorithms discussed in this paper and the abbreviations used in the performance result figures. The metrics associated with the optimal order obtained using the branch-and-bound method will be compared with the metrics obtained for the heuristics to assess their efficiency, or how close to the optimal ordering with respect to our performance metrics the heuristics can typically reach. In addition, the running times of the algorithms will be studied.

5.1 Case Study Application: Kronos

The performance of the loop ordering algorithms is studied in a real world setting using the Kronos remote sensing application. Remote sensing has become a very powerful tool for geographical, meteorological, and environmental studies [15]. Usually systems processing remotely sensed data provide on-demand access to raw data and user-specified data product generation. Kronos [15] is an example of such a class of

Method	*Abbreviation*
Unoptimized	none
Time optimized only	time
Original order (synthetic)	orig
Branch-and-bound method	bnb
Greedy heuristic	grd
Variable grouping heuristic, **u**nused memory potential	vgh**u**
Variable grouping heuristic, variable **s**ize	vgh**s**
Variable grouping heuristic, **r**andom	vgh**r**
Variable grouping heuristic, **d**eterministic reordering	vgh**d**

Table 5.1: Algorithm Abbreviations

applications. It targets datasets composed of remotely sensed AVHRR GAC level 1B (Advanced Very High Resolution Radiometer – Global Area Coverage) orbit data [23]. 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 a technical report [3].

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.

5.1.1 Solving the Multi-Query Optimization Problem

For our study, Kronos queries are defined as a 3-tuple: $[$ spatio-temporal bounding box and spatio-temporal resolution, correction method, compositing method $]$. The spatio-temporal bounding box 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 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 defines the aggregation level and function to be employed to coalesce multiple input grid points into a single output grid point.

5.1.2 Experimental Evaluation With Kronos

The evaluation of the techniques presented in this paper was carried out on the Kronos application (see Section 5.1.4). 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 5.1.3.

We designed several experiments to illustrate the impact of the time and space 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 2.9. We used a single processor of this machine to execute queries. Leverage from running in a multiprocessor 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.

5.1.3 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 [21]. 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 5.2 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

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 5.2: Transition probabilities.

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 possibly 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 5.2. More details about the workload model can be found in [6].

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).

5.1.4 Experimental Study

We studied the impact of the proposed optimizations varying the following quantities:

- The number of queries in a batch (from a 2-query batch up to a 32-query batch).
- The memory optimizations (none, bnb, grd, vghu, vghs, vghr, vghd) that were turned on. In all cases, the common subexpression elimination, dead code elimination, and loop fusion optimizations were enabled.
- 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 Spatial 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 metrics on average memory usage, maximum memory usage, and batch execution time, in addition to the running times of the space optimization methods.

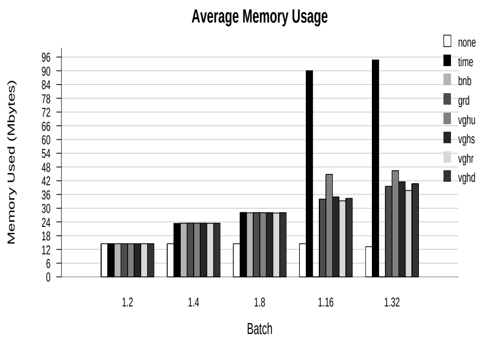
Average and Maximum Memory Usage

A comparison of average and maximum memory usage among the various memory optimization algorithms, as well as the unoptimized value, is presented in Figures 5.1(a)-5.2(d). The branch-and-bound algorithm was used to determine the optimal loop ordering in batches where the number of loops resulting from the time optimizations did not exceed 8. This was done because the amount of time required to compute the optimal order of larger numbers of loops was too long. Table 5.3 lists the number of queries and the corresponding number of loops for each workload profile as a result of applying the time optimizations. For example, the branch-and-bound algorithm was not applied for greater than 8 queries for workload 1 because the number of loops exceeded 8 for the higher number of queries. Thus, in Figure 5.1(a), the bar for *bnb* is missing for workload 1 with 16 and 32 queries.

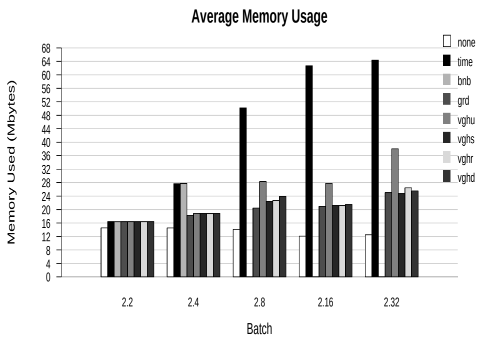
As shown in Figures 5.1(c) and 5.2(c), there are large savings in both average and maximum memory usage for workload 3 using 32 queries, in which average and maximum memory usage were reduced by in excess of 80% after application of the variable grouping heuristic. In Figures 5.1(d) and 5.2(d), large savings are also observable for workload 4 using 32 queries, albeit to a lesser extent, in which average and maximum memory usage were reduced by about 80% and 70%, respectively. These workloads involve less inter-dependent loop sets and ordering of the loops brings about more efficient memory usage. In no case does the memory usage increase as a result of applying the memory optimizations. The other algorithms performed well in reducing the amount of average and maximum memory used; however, there did not appear to be a clearly superior algorithm. The reduction in maximum memory when optimizing for average memory is about the same as when optimizing for maximum memory. In addition, the reduction in average memory when optimizing for maximum memory is about the same as when optimizing for average memory. This result indicates that only one application of the algorithms to optimize either maximum or average memory may be necessary to reduce both metrics.

Batch Execution Time

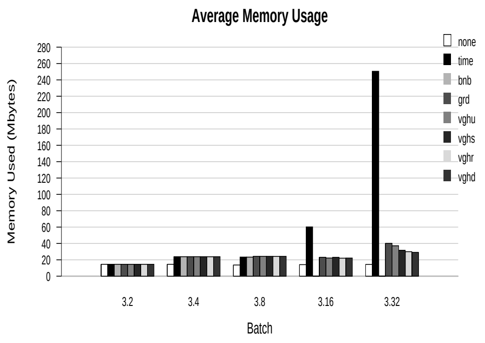
The amount of time required to execute the optimizations and the query are given in Figures 5.3 and 5.4. The amount of time required to execute time optimized query batches (*time*) that have not undergone memory optimization is significantly decreased from the unoptimized batch (*none*), due to the fact that after time optimization all of the queries are computed simultaneously. The execution time is not significantly affected by any of the heuristics from the execution time of the time optimized loops.



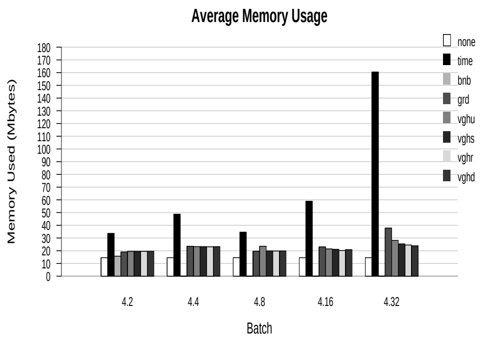
(a) Workload Profile 1



(b) Workload Profile 2

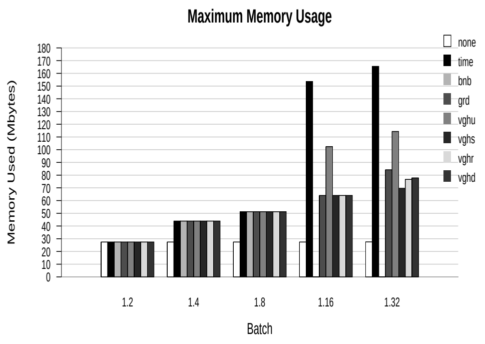


(c) Workload Profile 3

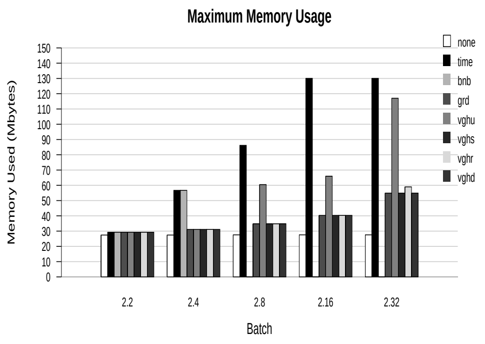


(d) Workload Profile 4

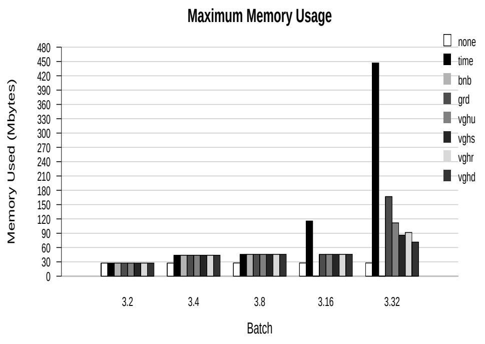
Figure 5.1: Average memory usage for Kronos queries



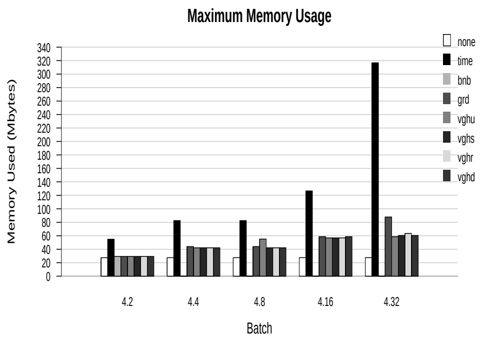
(a) Workload Profile 1



(b) Workload Profile 2

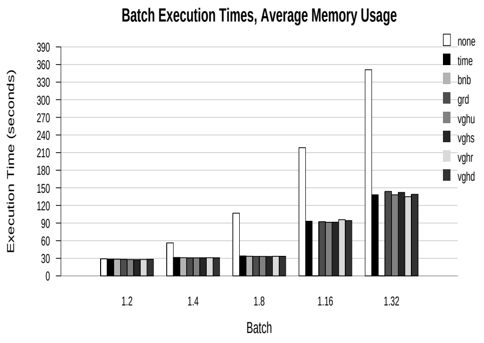


(c) Workload Profile 3

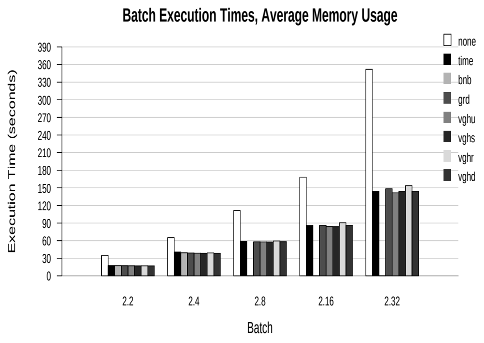


(d) Workload Profile 4

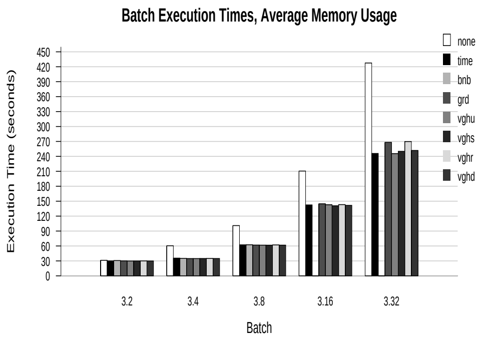
Figure 5.2: Maximum memory usage for Kronos queries



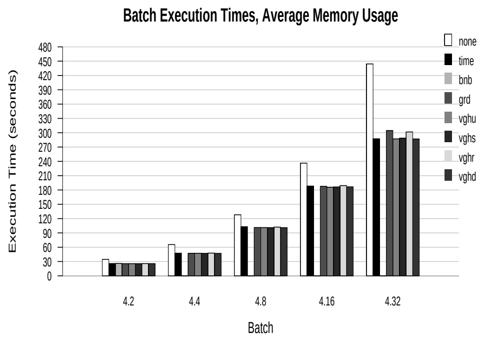
(a) Workload Profile 1



(b) Workload Profile 2

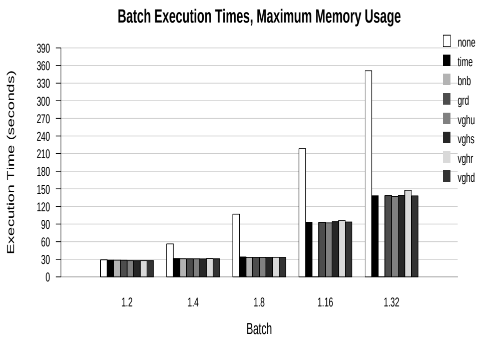


(c) Workload Profile 3

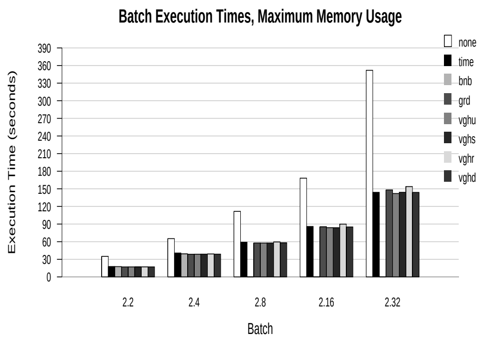


(d) Workload Profile 4

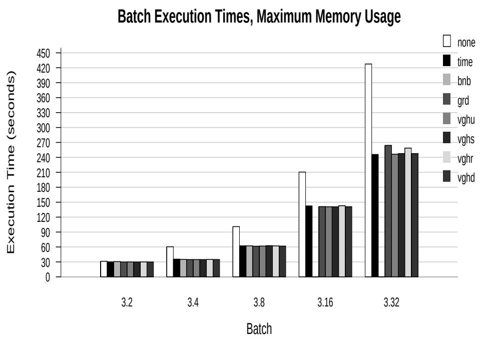
Figure 5.3: Batch execution times when optimizing average memory usage for Kronos queries



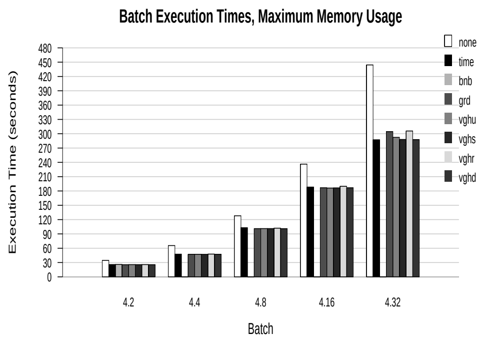
(a) Workload Profile 1



(b) Workload Profile 2



(c) Workload Profile 3



(d) Workload Profile 4

Figure 5.4: Batch execution times when optimizing maximum memory usage for Kronos queries

Workload 1	
Q	*N*
2	2
4	2
8	2
16	23
24	40
32	70

Workload 2	
Q	*N*
2	1
4	4
8	16
16	71
24	78
32	90

Workload 3	
Q	*N*
2	2
4	2
8	6
16	22
24	99
32	131

Workload 4	
Q	*N*
2	3
4	9
8	14
16	31
24	63
32	110

Table 5.3: Kronos Query Batches: Number of Loops (N) Per Number of Queries (Q)

5.2 Experimental Results With Synthetic Loops

We studied the impact of the number of loops on the performance of each loop ordering algorithm, as well as the effect of loop inter-dependency, using generated synthetic input loop sets. Performance results with synthetic loops were obtained for all of the algorithms mentioned. The synthetic loops were randomly generated with the parameters shown in Table 5.4. It should be noted that these loops do not correspond to actual queries, but represent the output of the time optimization performed on a query batch. The number of loops was varied to provide results on the dependency of the algorithms on this value, but values larger than 8 were not considered due to the amount of time required to determine the optimal result via the brute force method or the branch-and-bound method.

The “Variable Pool Size” determines the number of variables in the “variable pool”. The variable pool is a set of variables from which are drawn the variables that comprise the generated loops. The Variable Pool Size determines the degree of interdependency among the loops because a smaller number of variables in the pool increases the likelihood that loops share variables. This is true because the probability that two loops share a variable decreases as the total number of variables in the pool increases. To prove this, suppose the Variable Pool Size is given by μ and we are concerned with finding the probability that two loops 1 and 2 are dependent, i.e. share a variable. Furthermore, suppose loop 1 uses ω_1 variables and loop 2 uses ω_2 variables. The following equation gives the probability $P(\mu, \omega_1, \omega_2)$ that loops 1 and 2 share a variable:

$$P(\mu, \omega_1, \omega_2) = \sum_{i=1}^{\omega_{min}} \left(\frac{\binom{\omega_1}{i}\binom{\mu-\omega_1}{\omega_2-i}}{\binom{\mu}{\omega_2}} \right) \tag{5.1}$$

or, equivalently,

$$P(\mu, \omega_1, \omega_2) = \sum_{i=1}^{\omega_{min}} \left(\frac{\omega_1!\omega_2!\,(\mu-\omega_1)!\,(\mu-\omega_2)!}{(\omega_1-i)!i!\,(\omega_2-i)!\,(\mu-\omega_1-\omega_2+i)!\mu!} \right) \tag{5.2}$$

where $\omega_{min} = min(\omega_1, \omega_2)$. In Equation 5.2 it is also assumed that $\omega_1 + \omega_2 \leq \mu$. This condition is required because the equation is undefined when $\omega_1 + \omega_2 > \mu$; however, in this case it is certain, with probability 1, that loops 1 and 2 share a variable. The limit $\lim_{\mu \to \infty} P(\mu, \omega_1, \omega_2)$ is 0. This can be seen intuitively by considering the case where $\omega_1 = \omega_2 = 1$, in which case Equation 5.2 simplifies to $\frac{1}{\mu}$, which approaches 0 as μ approaches ∞.

We have chose two values, 8 and 20, for the Variable Pool Size. For a Variable Pool Size of 8 variables, the probability that two loops share at least one common variable, assuming that on average each loop has 3 variables, is 82%. This probability can be derived from Equation 5.1 by using $\mu = 8$ and $\omega_1 = \omega_2 = 3$. Synthetic loops generated with a Variable Pool Size of 8 are characterized as "highly dependent" loops. For a Variable Pool Size of 20 variables, the probability that two loops share at least one common variable, again assuming that on average each loop has 3 variables, is 40%. This probability can be derived from Equation 5.1 by using $\mu = 20$ and $\omega_1 = \omega_2 = 3$. Synthetic loops generated with a Variable Pool Size of 20 are characterized as "highly independent" loops.

The value of the other parameters were chosen arbitrarily because their exact values were considered unimportant in assessing the relative performance of the various algorithms. It should be noted, however, that the absolute magnitude of the performance gains are affected by the choice of variable sizes and loop running times.

The relative penalties of heuristics with respect to the optimal order as a function of loop set size is presented for both highly dependent and highly independent inputs in Figures 5.5 and 5.6. The results are presented in terms of percentage increase above the optimal value determined via brute force. All of the heuristics achieve results that are near-optimal, with the variable grouping heuristic using 100 randomized variable

Parameter	*Value*
Number of Loops	5, 6, 7, 8
Variable Pool Size	8, 20
Minimum Variables Per Loop	1
Maximum Variables Per Loop	5
Minimum Variable Size	10
Maximum Variable Size	100
Minimum Loop Running Time	10
Maximum Loop Running Time	100

Table 5.4: Synthetic Loop Parameters

orders (*vghr*) yielding the best results. *vghr* achieved within 0.06% of the optimal average memory usage for highly independent loop sets, as shown in Figure 5.5(b), and 0.1% of optimal for highly dependent loop sets, as shown in Figure 5.5(a). Furthermore, *vghr* achieved within 0.2% of the optimal maximum memory usage for highly independent loops, as shown in Figure 5.6(b), and achieved the optimal maximum memory usage for highly dependent loops. Among the variable bit position ordering methods, it appears the worst-performer was ordering according to the unused memory potential, *vghu*, which still achieved within 10% of the optimal average and maximum memory usage. Overall, there is a significant improvement in both average and maximum memory usage metrics over the original input loop set order.

The running times of the algorithms are presented in Figures 5.7 and 5.8. It is clear that the brute force method (*bru*) is correlated exponentially to the number of loops, since there is a linear increase in the logarithm of the running time with the number of loops. The same is true of the branch-and-bound method (*bnb*) albeit with a smaller

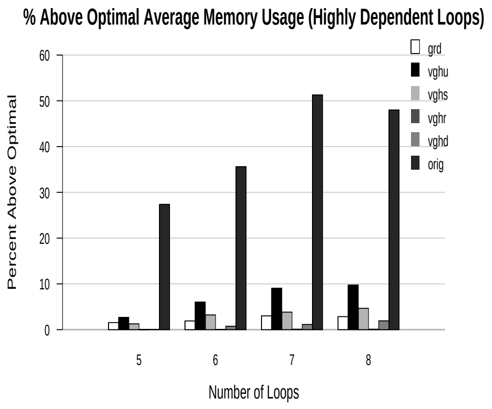
Parameter	*Value*
Number of Loops	16, 32, 64, 128, 256
Variable Pool Size	20, 40
Minimum Variables Per Loop	6
Maximum Variables Per Loop	10
Minimum Variable Size	10
Maximum Variable Size	100
Minimum Loop Running Time	10
Maximum Loop Running Time	100

Table 5.5: Synthetic Loop Parameters: Large Input Loop Sets

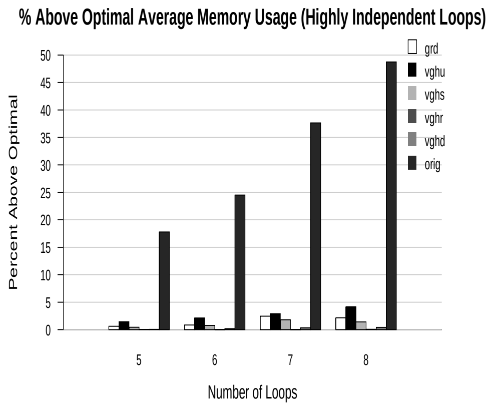
rate of increase. The running time of the greedy heuristic (*grd*) also increases with the number of loops. The running time of the variable grouping heuristics (*vghu*, *vghs*, *vghr*, *vghd*) do not appear to be correlated strongly with number of loops, at least for these small numbers of loops. The heuristic is polynomial in the number of loops and variables which makes the dependence on number of loops difficult to discern from Figures 5.7 and 5.8 given the small numbers of loops and the logarithm running time scale. The running time results as a whole suggest that the branch-and-bound method should be applied for small numbers of loops (e.g., less than 8). For larger numbers of loops, the variable grouping heuristic using several randomized variable orders should be applied.

The performance of the heuristics for larger synthetic loop sets was also studied. The larger sets were generated using the parameters given in Table 5.5. The highly dependent loop sets correspond to a Variable Pool Size of 20, which leads to a 99% probability of loop interdependency by application of Equation 5.2. The highly independent loop sets correspond to a Variable Pool Size of 40, which leads to an 86% probability of loop interdependency. The performance results are shown in Figures 5.9 and 5.10. The results are presented in terms of percentage decrease from the original average and maximum memory usage. The optimal values for these metrics could not be obtained because the amount of time required to determine them via brute force was too long. For highly dependent loops, the heuristics are able to decrease the average memory usage from the value for the original loop set order by up to 25% for 16 loops, but are less effective for higher numbers of loops. As was observed for lower numbers of loops, the best performer was the variable grouping heuristic using 100 randomized variable orders (*vghr*), which lowered the average memory usage by at least 20% for all numbers of loops tested. The greedy heuristic (*grd*) and variable grouping heuristic using deterministic reordering (*vghd*) also performed well. The worst performance was observed for variable grouping heuristic with the unused variable potential method (*vghu*) which still reduced average memory usage by 11%-15%. For highly independent loops, the heuristics performed significantly better than for highly dependent loops. The *vghr*, *vghd*, and *grd* heuristics all performed similarly, reducing average memory usage by up to 34%. Again, it was observed that *vghu* yielded the worst performance but still reduced average memory usage by 12%-20%. The performance results for reduction of maximum memory usage show that the heuristics are not effective for high numbers of highly dependent loops. It is observed that the reduction in maximum memory usage is closely correlated to the number of loops in the input loop set for both highly dependent and independent loops, and decreases markedly as the number of loops increases. For highly dependent loops, the heuristics are able to decrease maximum memory usage by 5%-15% for 16 loops. This percentage quickly goes to 0 for higher numbers of loops. For highly independent loops, the heuristics are able to decrease maximum memory usage by 14%-28%, but this percentage goes to near 0% for 256 loops. This phenomenon may be due to the fact that for higher numbers of loops it is more likely that for a given loop there exist several other loops that utilize the same variable in the same input loop set, which increases the interdependency of the loops. As a result, it is less likely that a reduction in the maximum memory usage metric can be found by the algorithms through loop reordering because any reordering based on a given variable is likely to be affected by the memory usage contributed by other variables that are shared by the loops.

The running times when optimizing the larger synthetic input loop sets for average and maximum memory usage are given in Figures 5.11 and 5.12. The running times for each heuristic appear to be correlated linearly with the number of loops in the input loop set. Optimizing highly independent loop sets required more time than the highly dependent loop sets because the variable pool size for highly independent loop sets was twice that of the highly dependent loop sets. This result indicates that the heuristics are also dependent on the total number of variables used by all of the loops in the input loop set, in addition to the number of loops in this set. There does not appear to be a significant difference in running time when optimizing for average memory usage versus maximum memory usage. The heuristics with the fastest running times are the variable grouping heuristics using the unused memory potential and variable size methods (*vghu*, *vghs*). These heuristics required between 0.02 and 0.2 seconds to complete optimization. The greedy heuristics required the most time to run and had a running time of up to 92 seconds when optimizing for average memory for highly independent loops.

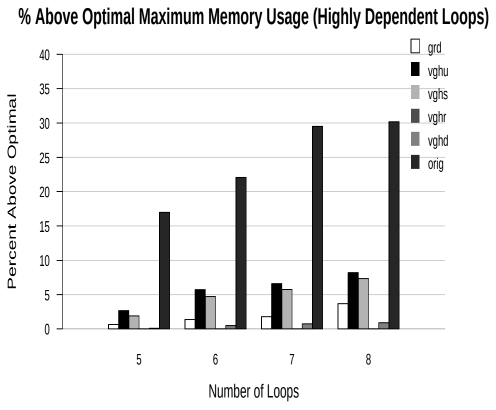


(a) Highly Dependent

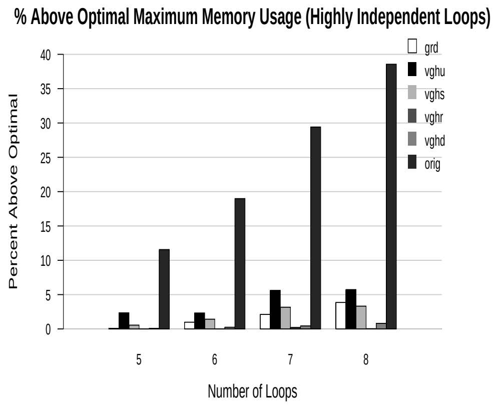


(b) Highly Independent

Figure 5.5: Percent above optimal average memory usage of heuristics applied to a) highly dependent loops and b) highly independent loops.

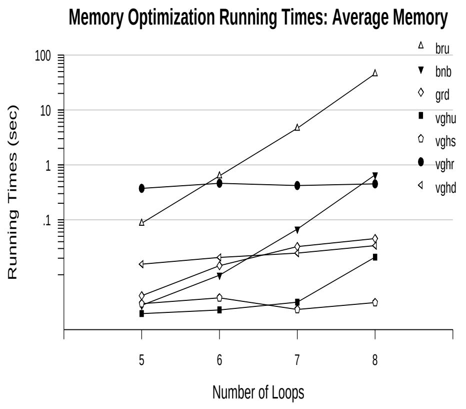


(a) Highly Dependent

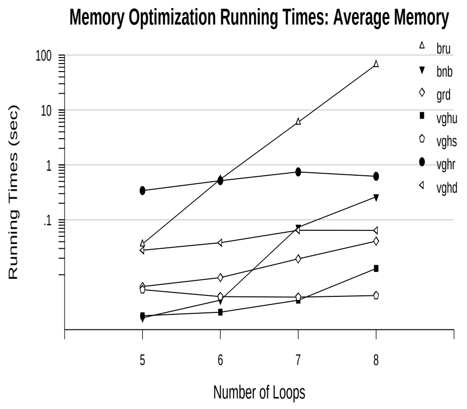


(b) Highly Independent

Figure 5.6: Percent above optimal maximum memory usage of heuristics applied to a) highly dependent loops and b) highly independent loops.

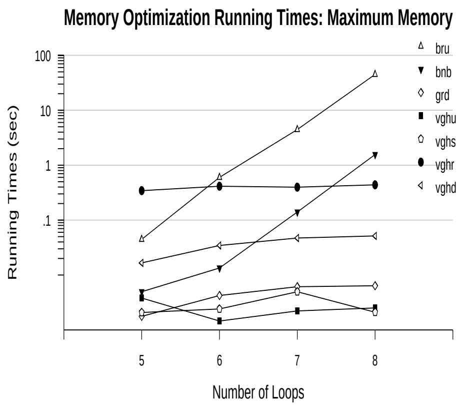


(a) Highly Dependent

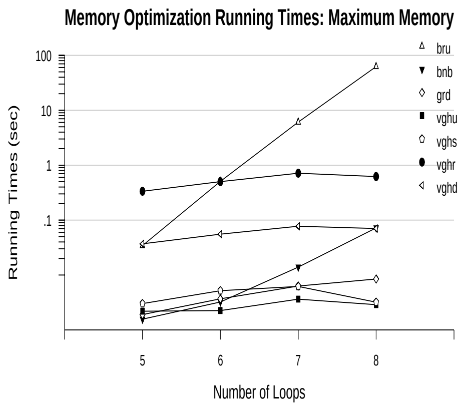


(b) Highly Independent

Figure 5.7: Running times of algorithms in seconds when optimizing for average memory usage for a) highly dependent loops and b) highly independent loops.

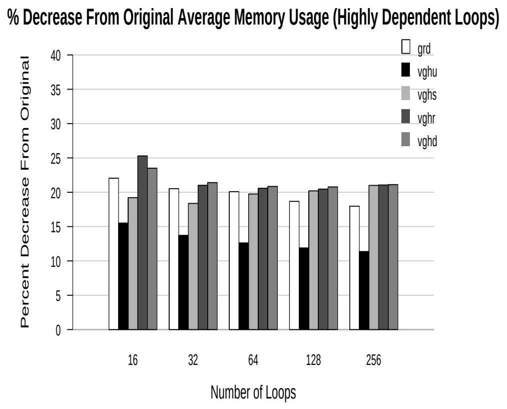


(a) Highly Dependent

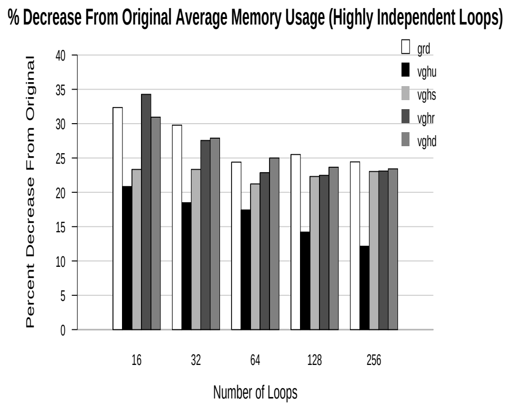


(b) Highly Independent

Figure 5.8: Running times of algorithms in seconds when optimizing for maximum memory usage for a) highly dependent loops and b) highly independent loops.

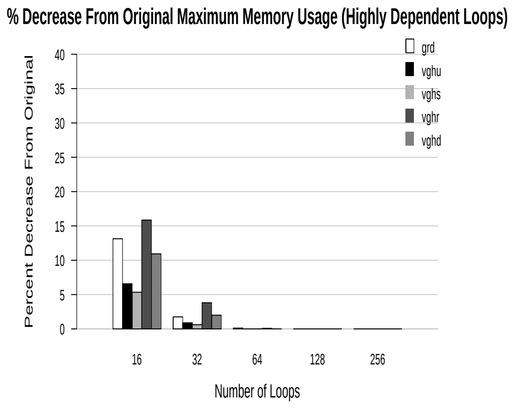


(a) Highly Dependent

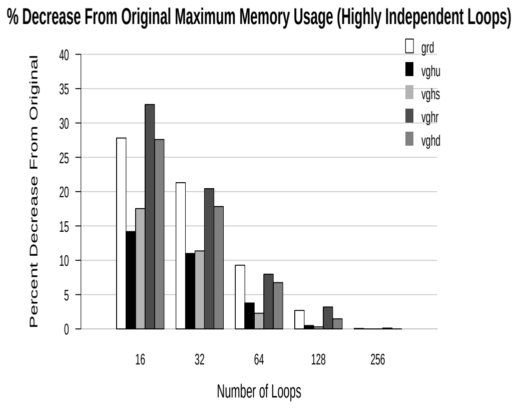


(b) Highly Independent

Figure 5.9: Percent decrease from original average memory usage of heuristics applied to a) highly dependent loops and b) highly independent loops.

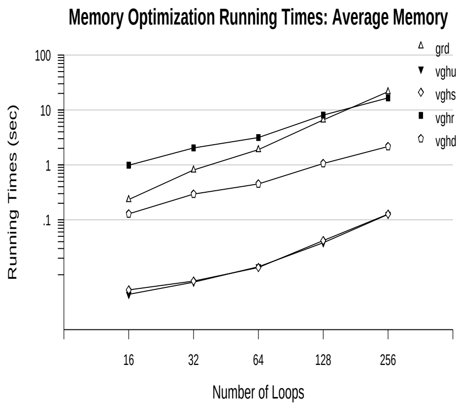


(a) Highly Dependent

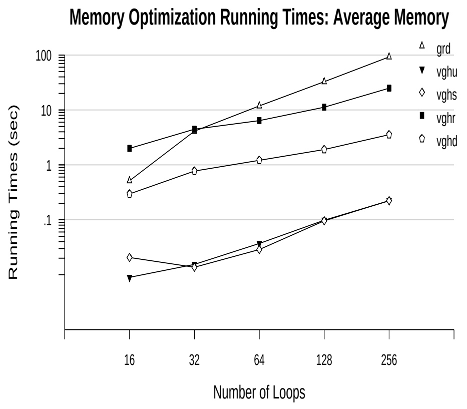


(b) Highly Independent

Figure 5.10: Percent decrease from original maximum memory usage of heuristics applied to a) highly dependent loops and b) highly independent loops.

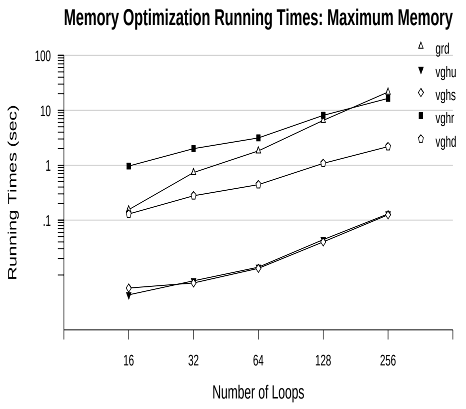


(a) Highly Dependent

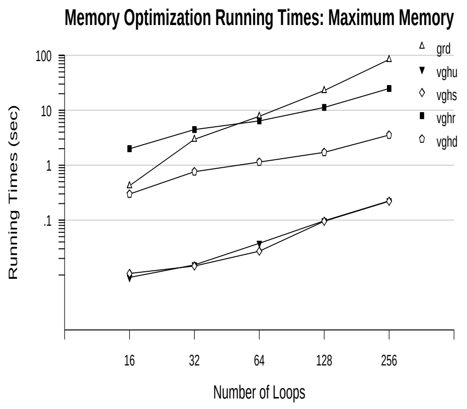


(b) Highly Independent

Figure 5.11: Execution times of algorithms in seconds when optimizing for average memory usage for a) highly dependent loops and b) highly independent loops.



(a) Highly Dependent



(b) Highly Independent

Figure 5.12: Execution times of algorithms in seconds when optimizing for maximum memory usage for a) highly dependent loops and b) highly independent loops.

Chapter 6

Conclusions and Future Work

In this chapter, we conclude by summarizing our contributions and by presenting some directions for future work.

6.1 Contributions

The current work presented various methods that may be used to optimize the execution time and memory footprint of multiple data analysis queries for a database system. Heuristics were presented that provide a good loop order and have a much shorter running time than the optimal brute force methods.

This work makes the following contributions:

- A method for the conversion of the declarative form of a query to its equivalent imperative form performed by our database system. The declarative form allows one to easily specify the query without worrying about the exact process by which the results are formed.

- A method for reducing the time to compute the results for query batches. These implemented optimizations are based on the algorithms commonly employed by compilers to reduce the execution time of compiled code.

- Methods for optimizing the memory footprint for an executing batch of queries. These optimizations, with the exception of the greedy heuristic, have been shown to improve the total query execution time without significantly impacting the execution time of the query. We have developed a novel heuristic, called "variable grouping", to perform memory usage optimizations that quickly arrive at a near-optimal loop ordering for executing a batch of range-aggregation queries. This heuristic has a low running time that does not grow significantly with the size of the input loop set and provides near-optimal results.

6.2 Future Work

Our work opens many opportunities for productive and innovative future research. It may be possible to improve the variable grouping heuristic further by devising a method to determine the variable bit position order in a way that utilizes the output of previous optimization attempts. In this way, the optimal loop ordering may be determined by iterating some number of variable bit position re-orderings and optimization attempts. Additionally, a combination of systematic and heuristic procedures can be used to optimize loop ordering, for example the partition of large query batches and the application of branch-and-bound method on these smaller batches. On a different note, an interesting course of study would be to prove the complexity of the loop ordering methods presented, or the difficulty of the loop ordering problem in general, i.e. proving its NP-completeness.

Furthermore, optimizations can be made to take advantage of memory hierarchies. For example, iteration over datasets can be done in such a way that improves cache performance by increasing spatial and temporal locality of data accesses. Certain integrated methods that optimize space and time simultaneously may be implemented that may be more desirable in certain situations where the optimal execution time or memory usage is not required. Composite methods can also be instituted in situations where the optimization of both space and time is not completely feasible. Loops may also be made to run in parallel to make use of parallel processor machines resulting in much faster absolute execution times.

BIBLIOGRAPHY

[1] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[2] Henrique Andrade. *Multiple Query Optimization Support for Data Analysis Applications*. PhD thesis, Department of Computer Science, University of Maryland, December 2002.

[3] Henrique Andrade, Suresh Aryangat, Tahsin Kurc, Joel Saltz, and Alan 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.

[4] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Efficient execution of multiple workloads in data analysis applications. In *Proceedings of the 2001 ACM/IEEE SC Conference*, Denver, CO, November 2001.

[5] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Active Proxy-G: Optimizing the query execution process in the Grid. In *Proceedings of the 2002 ACM/IEEE SC Conference*, Baltimore, MD, November 2002.

[6] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel 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.

[7] Chialin Chang. *Parallel Aggregation on Multi-Dimensional Scientific Datasets*. PhD thesis, Department of Computer Science, University of Maryland, April 2001.

[8] Josephine M. Cheng, Nelson Mendonça Mattos, Donald D. Chamberlin, and Linda G. DeMichiel. Extending relational database technology for new applications. *IBM Systems Journal*, 33(2):264–279, 1994.

[9] Gianna M. Del Corso and Francesco Romani. Heuristic spectral techniques for the reduction of bandwidth and work-bound of sparse matrices. Technical Report TR-01-02, Università di Pisa, January 2001.

[10] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM Press, 1969.

[11] Alessandra Esposito and Federico Malucelli. Bandwidth and Profile Reduction of Sparse Matrices: An Experimental Comparison of New Heuristics. In *Proceedings of Algorithms and Experiments*, Trento, Italy, Feb 1998.

[12] Renato Ferreira. *Compiler Techniques for Data Parallel Applications Using Very Large Multi-Dimensional Datasets*. PhD thesis, Department of Computer Science, University of Maryland, September 2001.

[13] Renato Ferreira, Gagan Agrawal, Ruoming Jin, and Joel 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.

[14] 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.

[15] Satya Kalluri, Zengyan Zhang, Joseph JáJá, David Bader, Nazmi El Saleous, Eric Vermote, and John 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.

[16] Myong H. Kang, Henry G. Dietz, and Bharat K. Bhargava. Multiple-query optimization at algorithm-level. *Data and Knowledge Engineering*, 14(1):57–75, 1994.

[17] David J. Kolson, Alexandru Nicolau, and Nikil Dutt. Optimal Register Assignment to Loops for Embedded Code Generation. *ACM Transactions on Design Automation of Electronic Systems*, 1(2):251–279, 1996.

[18] David Kuo and Gerard J. Chang. The profile minimization problem in trees. *SIAM Journal on Computing*, 23(1):71–81, 1994.

[19] Anany Levitin. *The Design and Analysis of Algorithms*. Addison Wesley, 2003.

[20] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[21] Daniel A. Menascé, Virgílio A. F. Almeida, and Larry W. Dowdy. *Capacity Planning and Performance Modeling.* Prentice Hall PTR, 2000.

[22] Steve S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[23] 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.*

[24] PostgreSQL 7.3.2 Developer's Guide. *http://www.postgresql.org*.

[25] John K. Reid and Jennifer A. Scott. Implementing Hager's Exchange Methods for Matrix Profile Reduction. *ACM Transactions on Mathematical Software*, 28(4):377–391, 2002.

[26] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1997.

[27] Michael Stonebraker. The SEQUOIA 2000 project. *Data Engineering*, 16(1):24–28, 1993.