Transparent and Efficient I/O
for Statistical Computing

by

Yi Zhang

Department of Computer Science
Duke University

Date: __________________________

Approved:

Jun Yang, Supervisor

Pankaj K. Agarwal

Shivnath Babu

Meichun Hsu

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2012
ABSTRACT

Transparent and Efficient I/O for Statistical Computing

by

Yi Zhang

Department of Computer Science
Duke University

Date: __________________________
Approved:

Jun Yang, Supervisor

Pankaj K. Agarwal

Shivnath Babu

Meichun Hsu

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University 2012
Abstract

Statistical analysis of massive array data is becoming indispensable in answering important scientific and business questions. Most analysis tasks consist of multiple steps, each making one or multiple passes over the arrays to be analyzed and generating intermediate results. In the “big data” setting, storage and I/O efficiency is a key to efficient analytics. Because of the distinct characteristics of disk-resident arrays and the operations performed on them, we need a computing environment that is easy to use, scalable to big data, and different from traditional, CPU- and memory-centric solutions.

R is a popular computing environment for statistical/numerical data analysis. Like many such environments, R performs poorly for large datasets. This dissertation presents RIOT (R with I/O Transparency), a framework to make R programs I/O-efficient in a way transparent to users. RIOT-DB, an implementation of RIOT using a relational database system as its backend, significantly outperforms R in many big-data scenarios. RIOT users are insulated from the data management backend and I/O optimization specifics. Because of this transparency, RIOT is easy to adopt by the majority of the R users.

While RIOT-DB demonstrates the feasibility of transparent I/O efficiency and the potential of database-style inter-operator optimizations, it also reveals significant deficiencies of database systems in handling statistical computation. To improve the efficiency of array storage, RIOT uses a novel storage structure called Linearized-
Array B-tree, or LAB-tree. LAB-tree supports flexible array layouts and automatically adapts to varying sparsity across parts of an array and over time. It also implements splitting strategies and update batching policies with good theoretical guarantees and/or practical performance.

While LAB-tree removes many I/O inefficiencies that arise in accessing individual arrays, programs consisting of multiple operators need further optimization. To this end, RIOT incorporates an I/O optimization framework, RIOTShare, which is able to jointly optimize I/O sharing and array layouts for a broad range of analysis tasks expressible in nested-loop forms. RIOTShare explores the middle ground between high-level, database-style operator-based query optimization and low-level, compiler-style loop-based code optimization.

In sum, combining a transparent language binding mechanism, an efficient and flexible storage engine, and an accurate I/O sharing and array layout optimizer, RIOT provides a systematic solution for data-intensive array-based statistical computing.
# Contents

Abstract iv  
List of Tables v  
List of Figures vi  
Acknowledgements viii  

## 1 Introduction  
1.1 Challenges in Array-Based Analytics 2  
1.2 Contributions 4  

## 2 Related Work  
2.1 Statistical Computing Languages and Environments 9  
2.2 Low-Level Languages and Compiler Optimizations 10  
2.3 I/O-Efficient Algorithms and Libraries 11  
2.4 Database Systems 12  
2.5 Array Storage Solutions 13  

## 3 Transparent I/O Efficiency for R  
3.1 Introduction 15  
3.2 Related Work 18  
3.3 Opportunities for Improving R 21  
3.4 RIOT-DB: Database as a Solution 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1</td>
<td>Towards Inter-Operation Optimization</td>
<td>27</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Experiments</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>RIOT: The Next Generation</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>I/O-Efficient Array Storage</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Related Work</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Overview of LAB-Tree</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Efficiency Through Better Space Utilization</td>
<td>48</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Splitting Strategy Revisited</td>
<td>49</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Dynamic Leaf Storage Format</td>
<td>55</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Experiments</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>Update Batching</td>
<td>70</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Flushing Policies and Analysis</td>
<td>70</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Implementation</td>
<td>76</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Experiments</td>
<td>79</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusion</td>
<td>82</td>
</tr>
<tr>
<td>5</td>
<td>I/O Sharing Optimization</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Related Work</td>
<td>89</td>
</tr>
<tr>
<td>5.3</td>
<td>Overview</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>A Polyhedral I/O Optimization Framework</td>
<td>93</td>
</tr>
<tr>
<td>5.4.1</td>
<td>The Polyhedral Model</td>
<td>94</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Problem Definition</td>
<td>97</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Dependences and I/O Sharing Opportunities</td>
<td>99</td>
</tr>
<tr>
<td>5.5</td>
<td>The Optimizer</td>
<td>102</td>
</tr>
</tbody>
</table>
List of Tables

4.1 LAB-tree, B-tree, DAF: total running time of \texttt{dgemm} on UFSparse and dense matrices. ........................................... 69

5.1 Constraints implied by a sharing opportunity. ......................... 110

5.2 Matrix sizes in the addition-multiplication experiment. ............ 121

5.3 Sharing opportunities in the multiplication-multiplication experiment. 124

5.4 Matrix sizes in the multiplication-multiplication experiment. ........ 125

5.5 Matrix sizes in the linear regression experiment. ..................... 127

6.1 Matrix blocking factors of selected plans for $C = AB; \ E = AD$. .... 165

6.2 Matrix sizes for $C = AB; \ E = DA$. .................................. 167

6.3 Matrix blocking factors of selected plans for linear regression. ....... 169
### List of Figures

3.1 Performance of R vs. RIOT-DB variants. ............................................. 33
3.2 I/O costs of a chain of three matrix multiplications. ........................... 38
4.1 Splitting strategies, with sparse leaf storage format. .......................... 57
4.2 Splitting strategies, with dynamic leaf storage format. ......................... 59
4.3 Splitting strategies: scalability test. .................................................. 60
4.4 Impact of buffer pool size. ................................................................. 62
4.5 LAB-tree, B-tree, DAF: 20000 × 20000 dense matrix. .......................... 63
4.6 LAB-tree, B-tree, DAF: 40000 × 40000 sparse matrix.......................... 63
4.7 LAB-tree, B-tree, DAF: scalability test. ............................................. 65
4.8 LAB-tree, B-tree, DAF: more insertion patterns. .................................. 66
4.9 LAB-tree, B-tree, DAF: UF Sparse matrices. ........................................ 67
4.10 I/O counts of flushing policies. ......................................................... 80
4.11 Flushing policies on local drive. ....................................................... 80
4.12 Flushing policies on NFS. ................................................................. 80
4.13 I/O counts of flushing policies, with larger matrix. ............................ 83
4.14 Flushing policies on local drive, with larger matrix. ............................ 83
5.1 Transformed code for a two-operator program. ................................... 87
5.2 Architecture of RIOTShare. ............................................................... 93
5.3 Intricacy of multiplicity reduction for many-many. ............................... 106
5.4 Matrix addition and multiplication. All plans. ..................................... 121
Acknowledgements

I am sincerely grateful to my advisor, Jun Yang, for his mentoring and support during my Ph.D. study. With his patient guidance, I have learned methods of critical thinking and independent research, which is the most valuable reward of my Ph.D. journey. He has also shown me how to precisely and concisely present technical contents, both verbally and in writing. I truly feel lucky to have been his student.

I would also like to thank my committee members, Pankaj K. Agarwal, Shivnath Babu, and Meichun Hsu, for their continuous feedback and guidance. Meichun has spent a generous amount of time discussing my research and given helpful suggestions. During my internship at the Microsoft Jim Gray Systems Lab, David DeWitt and Alan Halverson have mentored me on distributed query processing; I want to thank them and other colleagues for the wonderful experience.

Some parts of this dissertation are based on results of collaboration with other faculty and students at Duke University. Kamesh Munagala provided invaluable insights into the theoretical guarantees of LAB-tree’s update batching policies. Herodotos Herodotou helped with the initial design and implementation of RIOT-DB. Weiping Zhang participated in the implementation of RIOT’s array storage engine.

Last but not least, I give my special thanks to my family for their support. My parents have always been supportive of my Ph.D. pursuit. I want to express my deepest gratitude to my wife, Weina, for her love and support along the way. Because of you, Weina, the past five years have been a beautiful journey.
Recent technological advances have enabled collection of massive amounts of data in science, business, and society. Online social interactions are generating terabytes to petabytes of data each day. In various fields of science, improvement of equipment and technology has also enabled fast collection of large amounts of data. The state of the art in Flow Cytometry, for instance, allows 72,000,000 samples to be collected each hour.\textsuperscript{1} These large, high-resolution datasets have brought us closer than ever before to solving important problems such as improving business efficiencies and decoding human genomes.

For many statistical analysis tasks, arrays, one of the fundamental data types, are the most natural representation of data. As we apply increasingly sophisticated analysis to bigger and bigger datasets, efficient handling of large arrays is rapidly gaining importance. There are several critical challenges in supporting efficient analytics involving large arrays.

\textsuperscript{1} http://biology.berkeley.edu/crl/moflo.html
1.1 Challenges in Array-Based Analytics

**Language-Data Interface** When it comes to data-intensive computing, there has always been a barrier between programming languages and the underlying data management facilities (e.g., databases) that makes it cumbersome and often inefficient to program against big data. Although both the language layer and the data management layer have their own set of advanced optimization techniques, they are often connected through a very simple and rudimentary interface that excludes the possibility of joint optimization across the boundary. The result is that writing efficient code often requires a great amount of expertise and effort. It is critical to explore possible mechanisms that seamlessly integrate programming languages like R [R D08] and data management backends, to make automatic optimization possible. Note that our goal here is not to create new languages, but rather to seek such opportunities within the confinement of existing languages. We believe such an approach will place the least amount of burden on language users.

A more concrete question to ask is what it takes to convey enough information from user written programs to the data management and processing engine such as a database for effective optimization, and more importantly, what kind of information needs to be conveyed. We have already seen some answers for some particular applications, e.g., LINQ [MBB06] and Pig Latin [ORS+08]; however, the problem of optimizing array-based statistical and numerical programs has not been considered yet.

**High-Level Inter-Operation Optimizations** The I/O efficiency of data-intensive analytical programs can be tackled at different levels. On a low level, there has been abundant research on I/O-efficient out-of-core algorithms for individual operations like matrix multiplication, LU factorization, etc [VV96, TG96]. However, optimization opportunities on a higher level have not been studied. To be more specific, we consider
optimization across individual operators, expressions, and even blocks of code. There are potentially more opportunities when user code is optimized in a larger context. The context we consider is larger in terms of space, as optimization is applied across operators; it is also larger in terms of time, as information across repeated iterations of code execution (e.g., a loop) can also be utilized.

**Efficient Array Storage** Scalable data-intensive analytics require efficient handling of disk-resident arrays. Our target applications make prevalent use of high-level, whole-array operators such as matrix multiply, inverse, and factorization, but low-level, element-wise reads and writes are also possible. We must support different array access patterns, handle updates, and make the storage format automatically adapt to array sparsity that can vary across parts of an array and over time. Although there has been a myriad of approaches to storing arrays on disk, many fail to meet all requirements above. It is a challenge to design an array storage engine that meets the requirements and also has good theoretical properties and practical performance.

**I/O Sharing Optimization** Data analysis has become more sophisticated—it may use linear algebra instead of relational operations as building blocks, and each step may exhibit a complex, multi-pass access pattern over its input and output. Since the same array may be accessed multiple times, by a single or multiple operations, it is desirable and often necessary to share the accesses in order to achieve overall I/O efficiency. Optimizing I/O in this setting is challenging. First, passively identifying sharing opportunities is not enough; to maximize I/O efficiency, it is often necessary to actively create sharing opportunities via non-trivial program transformations. Second, sharing I/O implies buffering reused data in memory for later reuse, resulting in a direct tradeoff between I/O savings and memory usage. Third, sharing I/O may change the original execution order of the program and thus should be carefully checked against legality. Fourth, a program may exhibit many I/O sharing opportunities and some
may conflict others. Finally, even given the same program template, the appropriate strategy for sharing I/O may depend heavily on the parameters of the template (e.g., array sizes). Can the solution be parameterized?

**Block-Based Array Layout Optimization** Program I/O efficiency heavily depends on array layouts, as they directly affect spatial locality of program execution. Among others, block-based layout is the most popular; it is used by standard linear algebra algorithms. Even if we consider only the block-based layout, there is freedom in choosing the blocking factors for arrays. A single array may be involved in multiple operations, whose costs can be modeled by different functions in their operands’ blocking factors. When two operations both accessing array $A$ require different blocking factors of $A$ in order to achieve their respective optimal I/O cost, is it better off to satisfy both operations’ needs and pay the cost of a layout conversion, or use the same layout for both operations and live with suboptimal costs of individual operations? Again, the layout optimization problem should be approached by a cost-based fashion, with various factors such as array sizes and number of passes on the array made by the operations taken into account. Furthermore, layout optimization is not independent of I/O sharing optimization, because each operation’s cost function may change due to the I/O sharing enabled and affect the global solution. The two factors need to be jointly optimized.

### 1.2 Contributions

In this dissertation my goal is to design a framework for I/O-efficient statistical computing, in particular array-centric data-intensive programs involving block-based array accesses describable by nested loops. The proposed solution, RIOT, transparently achieves I/O efficiency by blurring the boundary between the host language, R, and the data management and I/O optimization components underneath. In addition,
with an array-oriented storage engine and an I/O sharing and block-based array layout optimizer, RIOT is able to achieve significant improvements in I/O performance. The proposed solution also generalizes to other computing environments and target applications. Specifically, this dissertation makes the following contributions.

**Interfacing Language and Data Management** RIOT makes it completely transparent to users how efficient I/O is supported. Transparency means no SQL, or any new language to learn. Transparency also means that existing code should run without (or with minimal) modification, and automatically gain I/O-efficiency. In addition to taking advantage of existing high-level constructs in the host language, RIOT interfaces the host language environment and the out-of-core data processing backend in a clever way to enable high-level, inter-operator optimizations. Chapter 3 describes the mechanism behind this using R as the host language and a relational database as the backend. Chapter 3 also reveals the deficiencies of a relational database backend and the need for an efficient array-oriented storage and processing engine.

It is important to note that the techniques for achieving transparency and enabling high-level optimization can be applied to other languages or environments as well, such as MATLAB, as long as some form of operator overloading is supported. Also, the deficiencies of a relational database backend does not mean we have to entirely abandon the database platform. What we propose next to address these inefficiencies can still be integrated to database systems as an extension, for example through user-defined types and functions. In this case, SQL is used as a driver while our custom storage engine and I/O optimizer are implemented outside the existing database query optimization.

**Intelligent Array Storage** RIOT uses LAB-tree as a solution for storing arrays on disk to support scalable analysis. It uses linearization to provide flexible array layouts, and automatically adapts to varying sparsity across parts of an array and
over time. In designing LAB-tree, we reexamine the B-tree splitting strategy for handling insertions and the flushing policy for batching updates, and show that common practices may be suboptimal. Through theoretical and empirical studies, we show LAB-tree’s advantages compared to existing solutions. Chapter 4 discusses the motivation, design, and evaluation of LAB-tree as the core of RIOT’s storage engine.

What differentiates our approach and existing storage schemes for general structured data or scientific array data is that we exploit characteristics unique to array data. Specifically, we realize that array indices are discrete and unique. To the best of our knowledge, our work is the first to take advantage of this fact and use it in designing an efficient array storage solution with proven theoretical properties. Moreover, to the best of our knowledge, we provide for the first time theoretical understanding and practical performance comparison of existing and new flushing policies for batching updates.

I/O Sharing Optimization Given an array-based, loop-centric program, RIOT captures the I/O patterns of the program in a way that is high-level enough to allow automatic exaction and reasoning, yet not too high-level to impede optimization flexibility (as black-box operators do). Based on this representation, RIOT’s I/O optimizer, RIOTShare, considers a rich space of plans and is able to accurately determine their legality, I/O costs, and memory requirements. Chapter 5 describes the details of RIOTShare’s inter-operation I/O sharing optimization capabilities.

Our major contribution here is that we propose a precise, flexible representation for structured data accesses (describable by affine functions), based on which I/O sharing among multiple operators can be accurately modeled and optimized. Our automatic optimization approach is in contrast to previous manual analysis and optimization of individual operators [RT01, Tol99a, BDHS11]. It has been shown that implementing a program or a composite operator using individually optimized basic operators may
be suboptimal; instead, operators may have to be jointly considered to achieve optimal performance [BDHS11]. Since it is impossible to manually analyze and optimize all possible combinations of operators, an automatic, cost-based optimization approach like ours is necessary.

In traditional database query optimization, the optimizer can only select a physical implementation from a fixed, limited pool of alternatives for each logical operator and thus only enables limited I/O sharing such as pipelining. Operators are treated as black boxes with certain known properties, e.g., a nested-loop join operator scans the outer table once and the inner table multiple times, where the number of times is proportional to the size of the outer table. Although this approach has enjoyed tremendous success in relational data processing, it is not suited for the array- and loop-centric applications that we consider, because the operators in our case have a far wider variety of implementation alternatives with complex data access patterns governed by many parameters. Therefore, we choose to “open up” operators to allow explicit modeling of their data access patterns and I/O costs. A direct consequence is that we are able to interleave the execution of multiple operators, which offers more I/O sharing opportunities than databases’ iterator-style execution.

**Block-Based Array Layout Optimization** RIOTShare is also extended with the capability of jointly optimizing I/O sharing and blocking. By doing so, RIOTShare is able to consider a even richer space of plans, where arrays’ blocking factors are allowed to change at operation boundaries. RIOTShare incorporates blocking factors into the cost model, and utilizes complementary geometric programming to decide the optimal execution plan. Chapter 6 discusses the details of this blocking optimization extension.

Deciding the optimal array layout is an intricate problem, as it reflects the contention on memory usage and data reuse among multiple operators. Because of
its complexity, especially when jointly optimized with I/O sharing, an automatic optimizer is called for. Our proposed solution is the first to address this problem via cost-based global optimization (see Section 6.2 for related previous work).

Together, these features make RIOT an appealing platform for transparent, I/O-efficient statistical computing. End users can run their R existing programs on big data with no or minimal modification. Throughout this dissertation, we show that in terms of I/O and total execution time, our solution beats plain R by orders of magnitude, and leads a state-of-the-art array database by a factor of more than 30.
Across application domains, much of advanced data analysis is done with programs custom-developed by statisticians, scientists, and engineers. Some rely heavily on numerical and statistical computing environments such as R and MATLAB, while others turn to specialized array management and processing systems. However, these existing solutions fail to address all the challenges discussed in Section 1.1 and are inadequate in supporting big array analytics.

This chapter reviews past work that is broadly related to this dissertation. Detailed discussion of related work that is specific to individual components appear in their respective chapters.

2.1 Statistical Computing Languages and Environments

R [R D08] is an open-source implementation of the S programming language, the de facto standard among statisticians for developing data analysis tools. R has a huge user and developer base; CRAN, the Comprehensive R Archive Network, lists more than 3,500 packages (which extend R’s functionalities) available as of February 2012. Like MATLAB [MAT], R provides a high level of abstraction to simplify programming
of numerical and statistical computation. Most users learn to program at the level of vectors and matrices instead of using explicit loops to iterate through arrays.

Such computing environments as R, however, are seriously challenged by the ever-growing size of data, because they typically assume that all data fits in main memory. If the physical memory cannot hold all data, the operating system’s virtual memory mechanism starts to swap data to and from disk, often causing thrashing. This problem stems from the difficulty for the operating system in predicting the program’s data access pattern and optimizing I/O accordingly.

2.2 Low-Level Languages and Compiler Optimizations

When thrashing happens, many R users’ first reaction is to rewrite the program to manage I/O explicitly, often in a lower-level language like C or FORTRAN. Rewriting code and hand-optimizing I/O require a huge amount of effort and expertise. Also, what is hand-optimized for one computer may perform poorly on another.

A more promising approach is to rely on compilers to automatically optimize user programs for improved data locality. The compiler community has developed a plethora of techniques for such automatic optimization [WL91, CMT94, BF03, BHRS08]. Although I/O optimization can be cast as a particular form of data locality problem, the data locality problem compilers solve is a different one. Their traditional focus is minimizing the traffic between CPU cache and memory, while ours is minimizing disk I/O. Although similar at a first glance, the two problems are fundamentally different. Traffic between cache and memory is hardware-managed and has peculiarities such as cache associativity; therefore, optimization tends to be best-effort, and does not produce a program that controls data sharing precisely. Traffic between memory and disk, on the other hand, is completely under our control, making precise control and analysis possible and preferable for our approach.
Polyhedral Optimizations The polyhedral model is a powerful abstraction of programs containing loop nests with affine loop bounds and array subscripts. It dates back to the seminal work of Karp, Miller and Winograd on uniform recurrence equations [KMW67]. Because of its power of algebraic abstraction and transformation expressiveness, it has gain traction in the compiler field on some important optimization problems [Fea91, Fea92a, Fea92b, GVB+06, GR07, BHRS08]. Although mainly used for cache-level optimizations previously, we show in this dissertation that it can be a viable foundation to build an I/O optimizer because it admits higher-level program analysis.

2.3 I/O-Efficient Algorithms and Libraries

There have been many approaches towards making data-intensive programs I/O-efficient without placing too much burden on users. One approach is to use an I/O-efficient library of routines which implement expert-crafted algorithms. Domain experts have designed algorithms with theoretical bounds for various linear algebra kernels. For example, as early as 1981, Hong and Kung proved a lower bound on the amount of I/O needed to perform a conventional matrix multiplication using a fixed small memory [JWK81]. Since then, abundant research has been devoted to improving the communication costs (between disk and memory or between distributed machines) of important linear algebra kernels (for a survey, see [Tol99a]). Recently Demmel et al. proved a lower bound for a general class of dense and sparse linear algebra algorithms and showed that existing algorithms and some new ones they proposed indeed achieve the lower bound [BDHS11]. One critical observation they made is that simply composing calls to optimal implementations of basic operators may result in suboptimal overall performance for composite operators such as LU factorization. Sometimes interleaving instructions from different operators is required to achieve the lower bound. However, there is no existing general algorithm for doing
this. Overall, designing I/O-efficient algorithms is a manual process which requires a
great deal of expertise and ingenuity.

There are a number of I/O-efficient numerical computing libraries (e.g., SO-
LAR [TG96], DRA [NF96] and TPIE [VV96]). While developing such libraries is
an important first step, simply having I/O-efficient implementations of individual
operations is not enough. Many sources of I/O-inefficiency in a program remain at
a higher, inter-operation level: e.g., how intermediate results are passed between
operations, how much I/O could be shared if certain operations are reordered and
interleaved, etc.

2.4 Database Systems

**Relational Model and Query Language** Database systems have also been used
for managing large datasets. Designed to be I/O-efficient, they feature a high-level
language (SQL) that enables advanced optimization. Most numerical and statistical
computing environments provide ways to connect to databases, but SQL is awkward for
capturing nontrivial computation and advanced array layouts, and general-purpose
database systems are highly inefficient for dense arrays [Bau94, MS02, SBČ+07,
CDD+09, Bro10]. While there has been work on making database systems more
efficient for array-based storage and computation [BDF+99, MS99, SBČ+07], much
of that work is highly database-centric. Effective users must become SQL experts.
Unless all their computational needs can be completely satisfied by a database system,
they are faced with the difficult challenge of deciding what processing should be done
by the database versus the host programming language. Inexperienced users should
not be forced to draw such explicit boundaries.

**Work Sharing** QPipe [HSA05] proposes an on-demand simultaneous pipelining
paradigm for maximizing data and work sharing across concurrent queries. It detects
overlapping scans at run time and exploit the sharing opportunities using *circular scans*. Cooperative scans [ZHNB07] is based on a similar idea, but coordinates I/O sharing using a more effective policy. Multi-query optimization [Sel88, RSSB00] tries to match common subqueries so that query processing can be partially shared. The recent DataPath system [ADJ+10] relaxes the condition of sharing by employing a data-centric, push-based approach.

The database-like, operator-based approach does not allow full-fledged inter-operator optimization. Although this approach has enjoyed tremendous success in relational data processing, it is not suited for the array- and loop-centric analytical applications that we consider, as the operators in our case have a wide variety of implementation alternatives with complex data access patterns governed by many parameters. When put together for co-optimization, the operators cannot be treated as black boxes but need to be “opened up” so that the optimizer can reason about I/O and achieve savings.

2.5 Array Storage Solutions

**R/MATLAB** In terms of array storage, R and MATLAB offer separate dense and sparse storage formats, but these formats do not adapt to varying sparsity across parts of an array and over time, and users must choose one format in advance. For sparse arrays, the *Compressed Sparse Column* format used by MATLAB does not support updates or random accesses for disk-resident arrays.

**Databases** Most array databases (e.g., *RasDaMan* [Bau94], *ArrayDB* [MS02], and *SciDB* [SBW11]) divide arrays into rectangular chunks, and often rely on spatial indexing to retrieve chunks in high-dimensional arrays. While this works for ad hoc region-based retrieval, whole-matrix operations with more predictable but specific access patterns require different storage optimization techniques.
**B-Trees and UB-Trees** Multidimensional arrays must be *linearized* in some way to be stored on 1-d linear storage media such as disks. UB-tree [Bay97], for example, linearizes arrays using Z-order [Mor66]. After linearization, B-tree [Com79] can be leveraged to store the array elements, with the linearized index as the key. Various B-tree tricks [BU77, Lom01] can then be applied to improve the storage efficiency. Prefix B-tree compression, for example, is a general form of compression, though its generality also carries some overhead. There is also work on alternative splitting strategies, such as avoid splitting by scanning adjacent nodes for free space [Küs83]. These are techniques that apply to general B-tree storage. In this dissertation we investigate new space-saving methods that take advantage of unique characteristics of array data.

Work on update batching on B-trees dates back to Lohman et al. [LM77]. Instead of a complete reorganization, Lang et al. [LDJ86] propose accumulating insertions in a batch, sorting them by key, and applying them to B-tree by traversing from left to right and backtracking along root- to-leaf paths when necessary. Our contribution to the update batching problem lies in analyzing and questioning the standard practice of flushing all buffered updates.
This chapter describes how RIOT achieves transparent I/O efficiency for the R language. RIOT-DB, a proof-of-concept implementation of RIOT using a relational database backend, is presented. RIOT-DB inherits RIOT’s language-data interface design, and thus demonstrates many high-level opportunities of I/O optimization. By exploiting these opportunities, RIOT-DB achieves orders of magnitude speedup compared to plain R. This convincingly demonstrates the advantage of RIOT’s language-data interface. The use of a relational database backend in RIOT-DB, however, reveals significant inefficiencies in databases’ support for statistical computing. This motivates our subsequent work to design and implement a backend for RIOT that is specially optimized for array-centric statistical computing.

3.1 Introduction

Scientists and engineers rely heavily on numerical computing environments such as R [R D08] and MATLAB [MAT]. Like MATLAB, R provides a high level of abstraction to simplify programming of numerical and statistical computation. Most R users
learn to program at the level of vectors and matrices instead of using explicit loops to iterate through arrays.

The ever-growing sizes of datasets, however, pose serious challenges to these numerical computing environments. R, for example, assumes that all data fits in main memory. When the physical memory can no longer hold all data, the operating system’s virtual memory mechanism starts to swap data to and from disk, often causing the program to thrash and run unbearably slow. This problem stems from the difficulty for the operating system in predicting the program’s data access pattern and optimizing I/O accordingly.

There are many ways to achieve better I/O efficiency; however, as discussed in Chapters 1 and 2, they all have their limitations. Rewriting code in a low-level language and hand-optimizing I/O require a huge amount of effort and expertise, especially if the solution needs to be portable. Using I/O-efficient numerical computing libraries makes individual operations run faster, but misses optimizations at a higher, inter-operation level. We will show in this chapter the importance of optimizing I/O at this level. Database systems are efficient in I/O by design, but has a rigid, simplistic interface with the host language. For detailed discussions of these approaches, refer to Chapter 2.

To have a practical impact on the majority of the users of R (or any other language for numerical computing), we believe that a better approach is to make it completely transparent to the users how we support efficient I/O. Transparency means no SQL, or any new language to learn. Transparency means that existing code should run without modification, and automatically gain I/O-efficiency.

Achieving transparency is challenging. First, besides making data layout and elementary operations I/O-efficient, what are the higher-level optimization opportunities and how critical are they? Second, what does it take to integrate I/O-efficient data layout, algorithms, and optimizations seamlessly into an existing language environ-
ment? R and MATLAB are interpreted; is it possible to implement the higher-level optimizations without switching to compilation instead? To what extent do we need to modify the existing implementation of these language environments?

This chapter presents the language-level mechanisms of RIOT (R with I/O Transparency), a system that makes R I/O-efficient in a transparent fashion, without requiring users to learn a new language or rewrite their code. Although we are currently focusing on R, we expect many of our techniques to work for other numerical computing environments such as MATLAB. We will first report our experience of implementing RIOT-DB, a prototype system that uses a generic relational database system as a backend (while hiding it completely from users). Use of a relational database system allows us to explore both limitations and opportunities offered by the full range of database system features. Some features, such as the view facility, while seemingly unrelated to I/O-efficiency, turned out to be indispensable to RIOT-DB. We show that it is indeed possible to achieve I/O-efficiency transparently, not only because R already has high-level language constructs, but also because we are able to interface the host language environment and the database system in a clever way to enable high-level, inter-operator optimizations, such as avoiding materialization of intermediate results, deferring and reordering operations, etc. We demonstrate that, despite the overhead and inadequacy of generic database systems in handling array data and numerical computation, RIOT-DB’s high-level optimizations enable it to significantly outperform R in many common scenarios. Finally, based on our experience with RIOT-DB, we highlight the required features of a more efficient array processing engine, which are subsequently designed and implemented in RIOT and also presented in following chapters.
3.2 Related Work

Many relational database systems have introduced support for arrays (standardized in SQL99): e.g., Oracle’s VARRAY and nested table, and PostgreSQL’s ARRAY. Data cubes [GCB+97] can also be regarded as high-dimensional arrays, and many systems offer good storage and query support for them; however, these systems focus on OLAP-style queries instead of numerical computation over arrays. Furthermore, all these solutions fall under the database-centric approach discussed in Section 3.1, and therefore it is difficult to gain traction in scientific and statistical user communities.

There have been a number of database systems specialized in array processing; because of limited space, we describe only a few representatives here. RasDaMan [BDF+99] provides extensive support for multidimensional arrays with its own query language, RasQL, which extends SQL92. Queries are translated into an array algebra and optimized using a large collection of transformation rules. The storage manager utilizes various array tiling strategies to support different access patterns. AML [MS99] is another declarative language for manipulating arrays, along with a suite of query processing and optimization techniques. The system allows MATLAB users to issue AML queries and bring their results into MATLAB for further processing. Again, in contrast to our fully transparent approach, such systems can be considered database-centric (even though AML is not SQL-based), because users must explicitly draw a boundary between database processing and processing by the host programming language. Also, RasDaMan and AML target different application domains from ours and therefore do not treat operations such as matrix multiplication as first-class citizens; therefore, high-level optimizations involving these operations are difficult.

ASAP [SBÇ+07] is an array processing system that supports primitive operations oriented towards scientific computing. It also features ChunkyStore, a storage manager
highly optimized for storing multi-dimensional arrays. The work in \[SB\mathbf{C}^+ 07\] focuses primarily on demonstrating the I/O-efficiency brought by ChunkyStore to individual operations; on the other hand, in this dissertation we emphasize more on the high-level, inter-operation optimizations. Also, we have the additional goal of making an existing language environment I/O-efficient in a transparent manner to users.

Lots of work from the scientific and high-performance computing communities has gone into developing I/O-efficient libraries (e.g., [TG96, NF96, VV96]). Toledo gives an excellent survey [Tol99a] on out-of-core linear algebra algorithms. This line of work provides a solid foundation for us to build on. However, as we have mentioned in Section 3.1 and will show in the rest of this chapter, higher-level optimizations are just as critical in ensuring I/O-efficiency; a library-only approach is insufficient by itself.

Finally, we note that there are many interesting connections between our techniques (often databases-inspired) and those from the programming languages community. Many of our optimizations have analogies in compiler research: e.g., deferred evaluation, forward substitution, loop fusion, and array contraction. However, there are also notable differences. On the highest level, work on programming languages tends to focus on improving performance of memory-resident programs and/or their parallelization; I/O issues are often not considered.

Traditional compiler optimization techniques for array languages (e.g., Fortran 90) first translate array statements into scalar operations (expressed as loops), and then perform data dependence analysis and code transformations such as loop reversal and loop fusion. This approach, however, as Lewis et al. [LLS98] pointed out elegantly, “solves the problem at a greater conceptual distance from the source of the problem and at a greater cost.” Instead, RIOT optimizes at the higher level of array operations, an approach also used by [GW78, HLJ95, LLS98, JB03, RMI06].

Guibas and Wyatt [GW78] studied delayed evaluation in APL code compilation,
i.e., deferring the computation of intermediate results in an APL expression until the moment they are needed. During evaluation, intermediate results are “streamed” in time, instead of being materialized in temporary arrays. Hwang et al. [HLJ95] generalized the idea and applied it to Fortran 90. They also support statement merges, whereby certain adjacent statements can be merged into one and processed as a single loop without materializing temporary arrays. Lewis et al. [LLS98] perform dependence analysis among statements and identify clusters of statements that are “contractible” into a single loop. Joisha and Banerjee [JB03] studied how to minimize array storage in MATLAB, using program analysis to identify opportunities to reuse storage allocated for one array on another. Rosenkrantz et al. [RMI06] performs inter-statement optimization to avoid materializing temporary arrays that can be obtained by “shuffling” other stored arrays. Although RIOT effectively performs similar optimizations as those cited above, our techniques are different due to the interpreted nature of R. Also, RIOT is far more aggressive in deferring evaluation (e.g., converting assignments to deferrable function evaluations) and performing optimization (e.g., considering different data layouts and algorithms, and reordering matrix multiplications) than these works.

Menon and Pingali [MP99] presented a framework for detecting high-level matrix operations written in loop-oriented codes. The codes are represented in an intermediate form, which is then optimized using heuristic rule-based transformations; RIOT considers a broader set of optimizations. Iu and Zwaenepoel [IZ06] proposed a Java bytecode rewriting tool that automatically detects and converts compiled Java code that directly manipulates database tables as Java collections (instead of using JDBC) into more efficient SQL queries. Techniques from this line of work complement RIOT in the sense that we can apply them to programs written with lower-level loops (as opposed to high-level array operations) and then make them amenable to RIOT’s optimizations.
With the recent interest in blurring the boundary between programming and query languages (e.g., declarative networking [LCG+06], Pig Latin [ORS+08], LINQ), we expect that more and more connections between programming languages and databases communities will become relevant to this research.

3.3 Opportunities for Improving R

To illustrate sources of I/O-inefficiency in R and opportunities for improvement, consider the following example.

Example 1. We are given a large number of points in a 2-d space, whose coordinates are stored by vectors \( x[1:n] \) and \( y[1:n] \). Given two other points \((xs,ys)\) and \((xe,ye)\), we want to compute the lengths of paths between them via each of the points given earlier. We then draw 100 such lengths at random. The following R code accomplishes this task. Note that most operations in R are vectorized (e.g., \(^2\) squares every element of a vector and returns the results in a new vector).

\begin{enumerate}
\item \( d \leftarrow \sqrt{(x-xs)^2+(y-ys)^2} + \sqrt{(x-xe)^2+(y-ye)^2} \)
\item \( s \leftarrow \text{sample}(\text{length}(x),100) \) # draw 100 samples from \( 1:n \)
\item \( z \leftarrow d[s] \) # extract elements of \( d \) whose indices are in \( s \)
\end{enumerate}

Avoiding Intermediate Results It is common for an expression to involve multiple operations, such as Line (1) of Example 1. R would generate an intermediate result for each of these operations: first \( x-xs \), then \((x-xs)^2\), and so on. If memory can hold all data objects including intermediate results, there is no problem. However, when intermediate results accumulate and leave insufficient memory, thrashing can occur. Consider Line (1) again. R would generate a total of twelve intermediate results, all vectors of length \( n \). Even with a smart garbage collector that immediately reclaims memory as soon as an intermediate result is no longer needed, there can be
multiple intermediate results alive at the same time. When evaluating \((y-ye)^2\), for example, three intermediate results are alive: \(\sqrt{(x-xs)^2+(y-ys)^2}\), \((x-xe)^2\), and \((y-ye)\). Together with \(x\) and \(y\), we have five \(n\)-vectors that can easily cause thrashing if \(n\) is large.

If we were to hand-code Line (1), we could in fact compute \(d\) without materializing any of the twelve intermediate results, by using an explicit loop over \(1:n\) and computing one element of \(d\) at a time. This strategy would require a negligible amount of memory beyond the two inputs and the output. The question, of course, is how we can accomplish this optimization automatically.

**Deferred and Selective Evaluation** A closer examination of the R code in Example 1 reveals that not all elements of \(d\) need to be computed; in fact, only 100 of them are eventually used on Line (3). Nonetheless, R will happily compute the entire \(d\), wasting both computation and I/Os. If we could somehow defer the evaluation of \(d\) until we know which 100 elements are needed, we would selectively compute them by accessing the corresponding elements in \(x\) and \(y\). With this optimization, we would reduce the cost of accessing \(x\) and \(y\) (especially if they had been swapped out previously), and even avoid materializing the named object \(d\).

One might argue that a programmer should know enough to avoid useless computation, but such code is not uncommon for those without formal training in programming. Our hope is that a programmer can focus on what they want to compute instead of how they want to compute it, and leave the rest to RIOT.

**Example 2.** Given three matrices \(A\) (with dimensions \(n_1 \times n_2\)), \(B\) (\(n_2 \times n_3\)), and \(C\) (\(n_3 \times n_4\)), we want to compute \(A \%*\% B \%*\% C\), their product expressed in the syntax of R. R would first multiply \(A\) and \(B\), and then multiply the result and \(C\).

Internally, R implements matrix multiplication as follows, where \(T\) denotes the result of \(A \%*\% B\). This algorithm performs a total of \(n_1n_2n_3\) multiplications. R by
default uses a column layout for matrices; i.e., elements are stored in the column-major order.

```r
for (j in 1:n3)
  for (i in 1:n1) {
    T[i,j] <- 0
    for (k in 1:n2)
  }
```

**Optimizing Data Layout and Algorithms** When the size of data exceeds the memory capacity, data has to be swapped in and out of memory in blocks. For efficiency, how we lay out data should correspond to how we access it, so that each disk block we read will bring in a maximum amount of useful data. The data access pattern of matrix multiplication in Example 2 is significantly more complex than the sequential access pattern for most vector operations in Example 1. Therefore, a closer look is warranted.

Suppose the size of a disk block is $B$, and the size of available memory is $M$, where $M \ll \min\{n_1 n_2, n_2 n_3, n_3 n_4\}$. In the algorithm of Example 2, to compute each column of $T$, we must access one column of $B$, and the entire $A$ in row-major order. If both $A$ and $B$ use column layout (R default), each access to $A$ would likely result in a page fault, bringing the total I/O cost to a huge $\Theta(n_1 n_2 n_3)$. Had we been smarter to choose row layout for $A$, the total I/O cost would have been reduced to $\Theta(n_1 n_2 n_3 / B)$.

Higher I/O-efficiency can be gained by further tweaking the access pattern of matrix multiplication. Borrowing the idea from block nested-loop join, we could read as many rows of $A$ as possible into memory while leaving enough memory to update the corresponding rows of $T$ and a block to scan $B$ in column-major order. The total I/O cost would be reduced to $\Theta(\frac{n_1 n_2 n_3 (n_2 + n_3)}{BM})$. As we will show later, however, there
are even better strategies if we move beyond the built-in row and column layouts supported by R.

**Reordering Computation** R computes a chain of matrix multiplications in the order specified by the program. For Example 2, this strategy requires \( n_1 n_2 n_3 + n_1 n_3 n_4 \) multiplications and a commensurate number of I/Os. However, noting that matrix multiplications are associative, we could instead compute \( A \times (B \times C) \), which would require \( n_2 n_3 n_4 + n_1 n_2 n_4 \) multiplications. Depending on the values of \( n_1, \ldots, n_4 \), reordering the multiplication may significantly reduce both computation and I/O. The challenge is to let RIOT make such optimization decisions, much in the same way as a database query optimizer.

3.4 RIOT-DB: Database as a Solution

Having discussed some sources of I/O-inefficiency in R, we now describe our experience of implementing RIOT-DB, a prototype system that addresses these sources of inefficiency using a relational database system as a backend. While doing so, RIOT-DB maintains complete transparency; i.e., existing R programs can benefit from RIOT-DB without any modification. We are aware of the overhead and inadequacy of relational database systems for this task, as shown by previous work; ASAP, for example, revealed gross inefficiency of such systems at the storage level [SBC+07]. We still chose this option because it enabled rapid prototyping and offered an opportunity to investigate not only the limitations but also the potential of leveraging other relational database features (e.g., views, query optimization) in a new context.

**Interfacing with R** Instead of rebuilding R from scratch to make it I/O-efficient, we decided on a minimally invasive approach. We would build RIOT-DB as an R package using R’s extensibility features, and avoid modifying the core R code whenever possible. RIOT-DB can be dynamically plugged into an R environment,
and immediately adds I/O-efficiency to R programs. The decision to be modular and minimally invasive is important, since we plan to apply our techniques to other environments (e.g., MATLAB).

RIOT-DB defines three new data types, `dbvector`, `dbmatrix`, and `dbarray` (with an arbitrary number of dimensions), which correspond to R’s built-in `vector`, `matrix`, and `array`. The new types implement the same interfaces as their built-in counterparts. Users do not need to know whether an object they are dealing with has a RIOT-DB type or a built-in type. R’s `generics` mechanism [Cha98] enables this transparency. Briefly put, a generic function in R is associated with a collection of concrete methods which share the same formal arguments, but differ in the classes of the arguments. When a call to a generic function is evaluated, a method is selected according to the classes of the actual arguments. This is analogous to method overloading in some object-oriented programming languages like C++.

We illustrate how to define a new class and to use the generics mechanism by an example. Suppose we want to add an `+` operator for the new `dbvector` class. Below are the steps we follow.

1. Define a new class for `dbvector` by
   
   ```r
   setClass("dbvector",representation(size="numeric",⋯)),
   
   where `representation(⋯)` defines the names and types of `dbvector`'s members.
   
2. Define a method for adding two `dbvector`s and register it with the `+` generic method:

   ```r
   setMethod("+", signature(e1="dbvector", e2="dbvector"),
   
   function(e1,e2) {
     .Call("add_dbvectors", e1, e2)
The above code specifies that when the two operands of the + call are both of \texttt{dbvector} type, the provided function should be invoked. The provided function further calls a C function \texttt{add\_dbvectors}. Note that \texttt{.Call} can call C functions at runtime from dynamically loaded libraries (.dll on Windows platform or .so on Unix-like platforms).

3. Implement the addition logic in a C function:

\begin{verbatim}
SEXP add_dbvectors(SEXP e1, SEXP e2) {
    /* implementation */
}
\end{verbatim}

4. When two \texttt{dbvector} objects are added in user R code, e.g., \texttt{a+b}, all functions registered with the + generic will be checked for argument type match. The result is that our function in Step 2 is selected and eventually our custom C code is executed.

Fortunately, the object-oriented programming facility as illustrated above is not peculiar to R. Many other statistical computing environments, such as MATLAB, also provide mechanisms for registering new classes and overloading operators. Thus, we expect our general design to be portable to other environments.

\textbf{A Strawman Solution} A straightforward way for RIOT-DB to leverage a relational database system is to map every RIOT-DB object to a database table. A \texttt{dbvector} object would be mapped to a table with schema \((I, V)\), where the primary key \(I\) stores an index, and \(V\) stores the corresponding vector element. An \(n\)-dimensional \texttt{dbarray} object would be mapped to schema \((I_1, \ldots, I_n, V)\), where the array indexes
(I_1, \ldots, I_n) serve as the primary key. The function \texttt{add_dbvectors}, which adds two \texttt{dbvector} objects corresponding to tables \texttt{E1} and \texttt{E2}, would compute the result using the following SQL query:

\texttt{SELECT E1.I, E1.V+E2.V AS V FROM E1, E2 WHERE E1.I=E2.I}

The result of the above query would be stored in another database table, which is then associated with the \texttt{dbvector} object representing the result of addition. Execution of this query would carry a very small memory footprint.

As shown in [SBÇ⁺07], however, storing array indexes in tables incurs significant storage and processing overhead, which grows linearly with the number of dimensions. Database query processing also carries overhead, and usually cannot match R’s raw performance.

Some of these problems will go away if we move to a more specialized database system that uses, for example, a smarter storage manager like ChunkyStore [SBÇ⁺07]. However, deeper issues still remain. In particular, this strawman approach leverages the power of a database system only at an intra-operation level, and fails to address any I/O-inefficiency that exists at the inter-operational level. One could argue that R is partly to blame because RIOT-DB can only take control of individual operations involving RIOT-DB types. Next, we show how to enable higher-level optimizations within the confines of this interface between R and RIOT-DB.

### 3.4.1 Towards Inter-Operation Optimization

Interestingly, views provide a natural mechanism for RIOT-DB to tap more into database systems’ advanced features through its limited interface with R. Recall that when we define a view using a query (over tables or other views), the system simply records this query without evaluating it. When the system executes a query involving a view, the query is expanded by replacing references to the view by its definition
query.

We map each RIOT-DB object to a database table or view. The result of operating on RIOT-DB objects becomes a view, whose definition encapsulates the computation involved in generating this result. However, no computation actually takes place (yet). For example, RIOT-DB’s `add_dbvectors` function simply defines the following view to capture the result of adding two `dbvector` objects, and associates the view with the result object:

```
CREATE VIEW E3(I,V) AS
```

For a complex R expression such as Line (1) of Example 1, RIOT-DB would define one view for each intermediate result object. The view definition for `d`, when expanded by the database system, would look like the following:

```
CREATE VIEW D(I,V) AS
SELECT TMP1.I, TMP1.V+TMP2.V
FROM (SELECT I, SQRT(V) AS V
      FROM (SELECT TMP3.I AS I, TMP3.V+TMP4.V AS V...)) TMP1,
      (SELECT I, SQRT(V) AS V
WHERE TMP1.I=TMP2.I
```

In effect, the view mechanism allows RIOT-DB to build up, one operation at a time, bigger and bigger view definitions that correspond to more and more complex R expressions. With a view representing a complex, multi-operation expression, we are now ready to unleash other features of database systems.
Avoiding Intermediate Results To compute the result of a complex R expression, RIOT-DB evaluates the definition query of the corresponding view. Most database systems optimize and compile the query into a tree-shaped plan, and use an iterator-based model to execute it. Query execution proceeds in a recursive fashion, where each plan operator obtains its input tuples one at a time, as needed, from its child operators. Leveraging this execution model, RIOT-DB effectively pipelines processing among plan operators, and eliminates the need to materialize intermediate results (although in some cases the database system can still decide to materialize for performance). Compared with the strawman approach, RIOT-DB avoids storing intermediate results in temporary tables on disk. Compared with plain R, RIOT-DB avoids creating large intermediate results in memory; in a memory-constrained setting, I/O savings resulted from fewer virtual memory swaps can be substantial.

For example, to compute \( d \) in Example 1, RIOT-DB only needs a single pass over the tables associated with \( x \) and \( y \), and incurs no additional I/Os for intermediate results.

Deferred and Selective Evaluation RIOT-DB does not restrict the use of views to unnamed intermediate results produced within a single R expression. Named objects can be created with views as well,\(^1\) effectively deferring their evaluation. If a named object is subsequently referenced, RIOT-DB simply uses the view associated with that object. In Example 1, object \( z \) on Line (3) would correspond to the following view (note how dereferencing a vector with a vector of indices translates cleanly to a join between them):

```
CREATE VIEW Z(I,V) AS
```

\(^1\) There is one technicality here. Assignments introduce dependencies on views created by RIOT-DB, but R does not notify RIOT-DB of assignment operations. To be able to safely drop views, RIOT-DB must track such dependencies. Therefore, we had to introduce this additional hook for R assignments (which was the only modification we made to the core R code).
SELECT S.I, D.V FROM D, S WHERE D.I=S.V

At this point, RIOT-DB has defined view D, but not yet computed its content. Suppose we now want the result in z. RIOT-DB will effectively compute the following query, where $xs$, $xe$, $ys$, and $ye$ should be replaced with their actual values:

SELECT S.I, SQRT(POW(X.V-$xs$,2)+POW(Y.V-$ys$,2))
+ SQRT(POW(X.V-$xe$,2)+POW(Y.V-$ye$,2))
FROM X, Y, S WHERE X.I=Y.I AND X.I=S.V

Since S is very small, a reasonable database query optimizer would pick an index nested loop plan, which probes X and Y with each S.V value and computes the SELECT clause. Hence, RIOT-DB is able to compute just those d elements that are actually used, thereby saving both computation and I/O.

**Optimizing Algorithms and Data Layout** Although generic database systems handle vectors reasonably well, optimizations beyond vectors are still worrisome. For example, RIOT-DB defines matrix multiplication (cf. Example 2) as follows:

SELECT A.I, B.J, SUM(A.V*B.V) AS V
FROM A, B WHERE A.J=B.I GROUP BY A.I, B.J

The optimized query plan\(^2\) does a hash join on $A.J=B.I$, and then sorts the result by $(A.I,B.J)$ to perform group-by and aggregation. Unfortunately, as we will show in Section 3.5, this plan is far from the optimum. We believe the problem lies in that SQL is too low-level for representing many linear algebra operations; optimizing at this level is much less effective than if we know the high-level semantics of these operations. We will revisit this point when presenting our design for the next generation of RIOT in Section 3.5.

\(^2\) Although we implemented RIOT-DB with MySQL, we obtained this plan on a commercial database system with a well-regarded optimizer.
In terms of matrix data layout, there is little in a generic database system for RIOT-DB to leverage. RIOT-DB can specify either row or column layout by changing the order of index attributes in the primary key. With automated database design techniques, it might be possible to automate the choice between these two layouts. However, it is awkward to specify more advanced layouts such as tiling (further discussed in Section 3.5) in a generic database system, let alone finding the best tiling strategy.

3.4.2 Experiments

To evaluate the performance of RIOT-DB and the savings obtained by its various optimizations, we compare four approaches:

- Plain R;
- RIOT-DB/Strawman, as described earlier in this section (with no views);
- RIOT-DB/MatNamed, which uses views, but materializes all named objects;
- RIOT-DB (the full version).

Our RIOT-DB uses MySQL with MyISAM storage engine as the backend. All experiments are conducted on a Solaris 10 machine with an AMD Opteron 275 processor. To limit the burden placed on the testing machine, we did not test with very large vectors whose sizes are greater than the actual amount of physical memory. Instead, we simulated a limited-memory environment by using a simple program to lock down a large portion of the memory. The program uses the `shmat(2)` system call on Solaris, with the `SHM_SHARE_MMU` flag. This operation has the consequence of locking down the allotted memory pages in physical memory such that they will never be paged out.

31
We run the code in Example 1, capping the available amount of physical memory at 84MB, just enough to hold the R runtime plus two vectors with $2^{22}$ elements each. To be fair, we set the same memory cap for RIOT-DB variants (which include the MySQL server). To force computation of $z$, we add a final line: `print(z)`.

Two metrics are used to compare the performance of the four approaches: execution time and disk I/O. To measure the amount of I/O, we utilize the DTrace facility on the Solaris platform. From *Solaris Dynamic Tracing Guide*, “DTrace is a comprehensive dynamic tracing facility that is built into Solaris that can be used by administrators and developers on live production systems to examine the behavior of both user programs and of the operating system itself.” We use DTrace to monitor different statistics for plain R and RIOT-DB:

- For plain R, I/O is caused by the swapping of data into and out of the physical memory. We thus monitor virtual memory paging statistics.

- For RIOT-DB, virtual memory paging activity is negligible assuming there is enough memory to run R and MySQL; most I/O is caused by the MySQL database server reading and writing its data and index files. Therefore we monitor disk I/O statistics pertaining to MySQL files.

Results are shown in Figure 3.1. When vectors are sized at $2^{21}$ and $2^{22}$, RIOT-DB/Strawman underperforms plain R, even though R already suffers significantly from thrashing. The reason lies in the overhead of MySQL for storage (additional columns for array indexes) and numerical computation, and in that MySQL is used only at an intra-operation level. Intermediate results are particularly damaging to RIOT-DB/Strawman, because it writes them all into database tables. Nonetheless, MySQL-managed I/Os are mostly bulky and sequential, and therefore do not impact the execution time as much as the virtual memory I/Os incurred by R; also, performance
Figure 3.1: Performance of R vs. RIOT-DB variants.
of RIOT-DB/Strawman degrades linearly with the data size, much more gracefully than plain R.

Once we enable inter-operation optimizations, performance improves dramatically. RIOT-DB/MatNamed, by avoiding materialization of nameless objects and pipelining query execution, already nets significant gains over R. RIOT-DB is barely visible in the figures because it is so much faster than others. By further deferring evaluation across statements and using the database optimizer to avoid unnecessary evaluation, it is able to outperform plain R by orders of magnitude. These results demonstrate the importance and potential of inter-operation optimizations.

3.5 RIOT: The Next Generation

RIOT-DB has shown the feasibility of bringing I/O-efficiency to R in a transparent manner, revealed the overhead and inadequacy of generic database systems for numerical computation, and demonstrated the potential of higher-order, database-style optimizations. Based on these lessons, we now outline our design for the next generation of RIOT, whose components are presented in Chapters 4, 5 and 6.

Data Storage and Layout Options To address well-understood inefficiencies of the simple relational representation for arrays, RIOT incorporates a storage engine specially optimized for array data, which supports flexible array layouts and automatically adapts to varying sparsity across parts of an array and over time. The key indexing structure RIOT uses, LAB-tree, implements space-saving strategies and algorithms alternative to long-standing common practices. In Chapter 4, we discuss the detailed design of RIOT’s storage engine and show its theoretical guarantees and practical performance.

Program and Operator Representation RIOT-DB made extensive use of SQL views to piece together individual operations into larger expression for optimization
and execution. However, we do not need SQL views per se for this purpose. With an appropriate high-level representation, RIOT can build up an expression DAG, operation by operation, in the same fashion as RIOT-DB.

RIOT has operators built in for standard linear algebra kernels, such as matrix multiplication. Moreover, users can implement custom operators that can be incorporated into RIOT for optimization together with other built-in or custom operators. All operators are described using the polyhedral model (discussed in 5.4), and can be composed to form a representation of a complete input program suitable for I/O optimization. Chapter 5 discusses the details of this representation and the optimizations it enables.

**Optimization** Given an already-analyzed input program, RIOT first carries out operator-level rewrites using heuristics based on algebraic laws, such as the associativity of matrix multiplications. It further works on the polyhedral representation of the program and applies I/O sharing optimization (Chapter 5) and blocking optimization (Chapter 6). Below we show how to optimize the matrix multiplication chain in Example 2. First, consider multiplying two matrices \( A(n_1 \times n_2) \) and \( B(n_2 \times n_3) \) with memory \( M \) and block size \( B \). Previous work has established a lower bound [JWK81, Tol99a] of \( \Omega(p n_1 n_2 n_3/(B \sqrt{M})) \) I/Os for algorithms requiring \( \Theta(n_1 n_2 n_3) \) multiplications,\(^3\) and proposed an algorithm with matching complexity [Tol99a], which we describe below. We present an alternative proof for this lower bound in Appendix A.1 that gives more insight into how to achieve it, which leads naturally to the algorithm.

This algorithm divides available memory into three equal parts, each storing a \( p \times p \) (where \( p = \sqrt{M/3} \)) square submatrix: one from \( A \), one from \( B \), and the other for the result \( T \). For simplicity, assume that \( n_1, n_2, \) and \( n_3 \) are multiples of \( p \). The algorithm, captured by the pseudocode below, looks similar to the one in Example 2,\(^3\) There exist algorithms that use fewer multiplications, such as Strassen’s, although they are harder to implement and numerically less stable.
except that it operates on the level of submatrices:

```r
for (i in 1:(n1/p))
  for (j in 1:(n3/p)) {
    Tsub <- matrix(0, p, p)
    for (k in 1:(n2/p)) {
      read Asub <- A[(i*p-p+1):(i*p),(k*p-p+1):(k*p)] from disk
      read Bsub <- B[(k*p-p+1):(k*p),(j*p-p+1):(j*p)] from disk
      Tsub <- Tsub + Asub %*% Bsub
    }
    write Tsub as T[(i*p-p+1):(i*p),(j*p-p+1):(j*p)] to disk
  }
```

A storage layout strategy that works well with this algorithm is to use square tiles of area $B$, such that each $p \times p$ submatrix requires $O(p^2/B)$ I/Os. The total number of I/Os is $\Theta(n_1n_2n_3/(B\sqrt{M}))$, matching the lower bound. For large matrices, this algorithm beats the one in Section 3.4 inspired by block nested-loop join, which uses row and column layouts.

Stepping up a level, we now discuss how to optimize a chain of matrix multiplications. With dynamic programming [HS80], we can find a multiplication order that minimizes the total number of multiplications. Let $N$ denote this number. Using the matrix multiplication algorithm above and square tiling for all matrices, RIOT can compute the chain in $\Theta(N/(B\sqrt{M}))$ I/Os. Asymptotical optimality is shown in Appendix A.2.

To illustrate the effectiveness of RIOT optimizations related to matrix multiplications, we compare four strategies for computing $A \times B \times C$: RIOT-DB uses a plan consisting of two hash-join-sort-aggregate subplans (Section 3.4), one to first multiply $A$ and $B$, and the other to multiply with $C$; BNLJ-Inspired assumes
that the matrices use row, column, and column layouts respectively, and performs
the matrix multiplications in order, using the algorithm in Section 3.4 inspired by
block nested-loop join; **Square/In-Order** assumes square tiling for all matrices,
and performs the matrix multiplications in order, using the algorithm described in
this section; **Square/Opt-Order** also employs square tiling and the same matrix
multiplication algorithm as Square/In-Order, but first uses dynamic programming to
find the best multiplication order.

Suppose $A$, $B$, and $C$ have dimensions $n \times \frac{n}{s}$, $\frac{n}{s} \times n$, and $n \times n$, respectively,
where $s > 1$ is a skewness factor, which causes Square/Opt-Order to choose the
multiplication order $A(BC)$. The block size $B = 1024$. Figure 3.2(a) compares the
calculated I/O costs of the four strategies$^4$ for $n = 100000$ and 120000, and for
memory sizes of 2GB and 4GB. We see a progression of improvements as more
optimizations are introduced, and this trend is consistent for all parameter settings
tested. Figure 3.2(b) shows the results when we vary $s$, the skewness factor. Memory
is set at 2GB and $n = 100000$. RIOT-DB is no longer shown because it performs far
worse than others. As $s$ increases, the performance gap between Square/Opt-Order
and others widens, demonstrating the importance of optimizing the multiplication
order.

$^4$ To focus our comparison on evaluation strategies, Figure 3.2 excludes the overhead of storing
array indexes in RIOT-DB; this adjustment has no effect on the relative ordering of performance.
Figure 3.2: I/O costs of a chain of three matrix multiplications.
This chapter describes RIOTStore, the array storage engine of RIOT, in particular its core data structure (LAB-tree) and associated I/O-efficient algorithms and techniques.

4.1 Introduction

As mentioned in Chapter 1, arrays are one of the fundamental data types. Vectors and matrices, in particular, are the most natural representation of data for many statistical analysis and machine learning tasks. As we apply increasingly sophisticated analysis to bigger and bigger datasets, efficient handling of large arrays is rapidly gaining importance.

Scalability requires efficient handling of disk-resident arrays. Our target applications make prevalent use of high-level, whole-array operators such as matrix multiply, inverse, and factorization, but low-level, element-wise reads and writes are also possible. We have identified the following requirements for an array storage engine:

1. We must support different array access patterns (including those that appear
random). Our storage engine should allow a user or optimizer to select from a variety of storage layouts, because many whole-array operators have access patterns that prefer specific storage layouts: e.g., I/O-efficient matrix multiply prefers row, column, or blocked layouts, while FFT prefers the bit-reversal order. Moreover, a single array may be used in operators with different access patterns; instead of converting the storage layout for every use, sometimes it is cheaper to allow access patterns that do not match the storage layout, even though it makes accesses more random. Finally, some operators inherently contain some degree of randomness in their access patterns that cannot be removed by storage layouts, e.g., LU factorization with partial pivoting.

2. We must handle updates. One common update pattern is populating an array one element at a time in some order, which may or may not be the same as the storage layout order. Handling updates goes beyond bulk loading: some operators, such as LU factorization, iteratively update an array and read previously updated values, which means that we cannot simply log all updates without efficiently supporting interleaving (and sometimes random) reads to updated values.

3. We want the storage format to automatically adapt to array sparsity. For a sparse array, we want to avoid wasting space for elements that are zero (or some other default value), which can be done by storing array indices and values only for non-zero elements. On the other hand, for a dense array, we want to avoid the overhead of storing array indices by densely packing the values and inferring their indices from storage positions. In practice, there is no obvious delineation between “sparse” and “dense”; sparsity often varies across parts of an array and over time, and is difficult to predict in advance. For example, consider an application program that updates an initially empty
(all-zero) matrix one element at a time in random order according to some ongoing computation. The matrix may turn out dense, sparse, or partly dense (e.g., mostly upper-triangular); regardless of its final content, our storage engine should store the matrix in a way that provides good performance throughout the update sequence, without user intervention.

There has been a myriad of approaches to storing arrays on disk, but many fail to meet all requirements above. Targeting in-memory computation, popular platforms for statistical computing such as R and MATLAB offer separate dense and sparse storage formats, but these formats do not adapt to varying sparsity across parts of an array and over time, and users must choose one format in advance. Compressed sparse column, used by MATLAB and representative of popular sparse formats, does not support updates or random accesses for disk-resident arrays.\(^1\) Alternatively, a database system can store an array as a table with columns representing array index and value, but the overhead is high for dense arrays. It is generally believed that special support for arrays is needed in database systems, either through user-defined extensions or by completely new designs [Bau94, MS02, CDD09, Bro10]. Section 4.2 surveys additional related work.

A promising approach is to leverage B-tree [Com79]. To handle multi-dimensional arrays, we use linearization, which maps a multi-dimensional coordinate to a 1-d array index according to a linearization function that offers control over data layout. To adapt to varying sparsity, we apply the idea of compression, allowing each B-tree leaf to switch dynamically between sparse and dense formats according to the array density within the leaf. Simply outfitting B-tree with these features, however, falls short of offering optimal performance for arrays, as illustrated below.

**Example 3.** Consider sequentially inserting elements of array into an empty B-tree,

\(^1\) For memory-resident arrays, this format is easier to search but still inefficient to update.
which is a very common update pattern. Suppose the array has size 12 and a B-tree
leaf can hold at most 4 records. When a leaf overflows, the standard strategy is
split-in-middle, which divides the leaf into two with equal number of records (or as
closely as possible). The leaf level of the B-tree after the insertion sequence looks
as follows (only record keys are shown). About half of the space is empty, which is
particularly wasteful as no future insertions can possibly fill it. The suboptimal space
utilization also hurts access performance; e.g., array scans become twice as costly.

While one can handle sequential insertions as a special case, other patterns that
lead to waste are difficult to detect. Are there alternative splitting strategies that
are provably resilient against such waste, without knowing the insertion sequence in
advance?

Example 4. A popular trick to speed B-tree updates is to batch them by keeping
individual record updates in a memory buffer. When the buffer fills up, we flush
the buffered updates by applying them in key order. This approach reduces I/Os by
applying multiple updates to the same B-tree leaf with a single leaf access. A large
buffer also helps make the leaf accesses more sequential. However, for the following
update sequence, the conventional policy of flushing all buffered updates when the
buffer is full is not optimal. Here, $K$ denotes the number of updates that the buffer
can hold, and each $P_i$ represents an update of some record on leaf $P_i$.

$$
\underbrace{P_1, \ldots, P_{\frac{K}{2}}}^{K/2}, \underbrace{P_2, \ldots, P_\frac{K}{2}}^{K}, \underbrace{P_3, \ldots, P_{\frac{3K}{2}}}^{K}, \underbrace{P_4, \ldots, P_{K}}^{K}, \ldots
$$

The flush-all policy incurs two leaf writes (of $P_i$ and $P_{i+1}$) for every $K$ updates.
However, the optimal policy would flush all $P_i$ updates after the $(K/2)$-th update;
subsequently, only one leaf write would be incurred for every $K$ updates.
For this simple insertion sequence, flush-all is only a factor of 2 worse than the optimal. As we will see later, however, there exist sequences for which flush-all is a factor of $\Omega(\sqrt{K})$ worse. Are there flushing policies that offer better competitive ratio in theory or perform better in practice?

In this chapter, we present LAB-tree (Linearized Array B-tree), the backbone of the RIOT array storage engine, which meets all requirements identified earlier. LAB-tree offers flexible layouts via linearization; it inherits from B-tree efficient support for accesses and updates; and it adapts to varying sparsity by switching between dense and sparse storage formats automatically on a per-leaf basis. LAB-tree reexamines the leaf splitting strategies and batched update flushing policies, for which common practices have been rarely questioned. We present theoretical and empirical results that contribute to the fundamental understanding of these problems.

These results challenge the common practices. For leaf splitting, exploiting the fact that the domain of array indices is bounded and discrete, we devise a strategy that naturally produces trees with “no-dead-space,” often twice as efficient as those produced by split-in-middle. This advantage does incur a fundamental trade-off—in the worst-case, split-in-middle has competitive ratio 2, while this strategy has 3, which is the best possible for any “no-dead-space” strategy. Nonetheless, on common workloads, this strategy consistently and significantly outperforms split-in-middle.

For update batching, we give a flushing policy with competitive ratio $O(\log^3 K)$ in the worst case, beating flush-all’s $\Omega(\sqrt{K})$. For common workloads, however, flush-all actually performs better in practice. On the other hand, starting from a simple policy with a poor competitive ratio of $\Omega(K)$, we devise a randomized variant that incurs fewer number of I/Os than flush-all for some workloads (and comparable numbers for others). Our approach can be seen as bringing to the update batching problem the same level of rigor as in the study of caching (though results do not carry over
because of fundamental differences in their problem definitions).

Finally, we note that our techniques are easy to implement as they do not require intrusive modifications to the conventional B-tree. Also, many of our results generalize to other settings: the idea of “no-dead-space” splitting makes sense for other discrete, ordered key domains; theoretical analysis of update batching generalizes to other block-oriented or distributed data structures.

4.2 Related Work

Database systems have been extended with support for arrays, and more specifically, linear algebra. Besides storing arrays as tables whose rows correspond to individual array elements, UDTs and UDFs are popular implementation options (e.g., [OGG06, CDD+09]). In general, these approaches can be seen as dividing an array into chunks and storing each chunk in a database row as a unit of access. SQL can express many linear algebra operations by calling UDFs that operate on chunks or pairs of chunks. Database indexing is used for accessing chunks. While this dissertation does not store arrays in databases, many ideas, such as linearization, dynamic storage format, and update batching, are readily applicable by regarding a table of chunks as a block-oriented storage structure.

There has also been work building database systems specializing in arrays (e.g., RasDaMan [Bau94], ArrayDB [MS02], and SciDB [Bro10]). These approaches divide arrays into rectangular chunks, and often rely on spatial indexing to retrieve chunks in high-dimensional arrays. Our approach of linearization supports more layouts (e.g., bit-reversed) and avoids the difficulty of high-dimensional indexing. One reason for this different approach is that we focus less on ad hoc region-based retrieval, but more on whole-matrix operations with more predictable but specific access patterns. Nonetheless, it would be interesting to see how our ideas can be applied in their settings (e.g., linearization, alternative index reorganization and update buffering
methods) and vice versa (e.g., allowing replication of boundary elements between neighboring chunks as in SciDB).

Linearization is frequently used for multi-dimensional indexing. UB-tree [Bay97] is the most related to our work in this regard. While UB-tree linearizes arrays using Z-order, LAB-tree provides more linearization options to match different application needs (with a similar goal as RodentStore [CMWM09], but at a different level). More importantly, we reexamine index reorganization and update buffering practices, which UB-tree does not address.

There is no shortage of B-tree tricks [BU77, Lom01] aimed at improving its efficiency. Prefix B-tree compression, for example, is a more general form of compression than our dynamic leaf format, though its generality also carries some overhead. There is also work on alternative splitting strategies, such as avoid splitting by scanning adjacent nodes for free space [Küs83]. Most of these techniques are orthogonal to ours and may further improve LAB-tree in some cases. We are not aware of any previous work on alternative splitting strategies for bounded, discrete key domains and how they interact with compression. Work on update batching dates back to Lohman et al. [LM77]. Like us, instead of a complete reorganization, Lang et al. [LDJ86] propose accumulating insertions in a batch, sorting them by key, and applying them to B-tree by traversing from left to right and backtracking along root-to-leaf paths when necessary. Our contribution to the update batching problem lies in analyzing and questioning the standard practice of flushing all buffered updates.

4.3 Overview of LAB-Tree

Based on B-tree, LAB-tree introduces modifications and extensions designed for arrays: linearization (this section), new leaf splitting strategies (Section 4.4.1), dynamic leaf storage format (Section 4.4.2), and alternative flushing policies for update batching (Section 4.5).
Each LAB-tree has a linearization function that specifies the storage layout of the array. For an array of dimension $d$ and size $N_1 \times \cdots \times N_d$, a linearization function $f : [0, N_1) \times \cdots \times [0, N_d) \rightarrow [0, N_1 \times \cdots \times N_d)$, where all intervals are over $\mathbb{N}_0$, is a bijection that maps each $d$-d array index to a 1-d array index. When $d = 1$, $f$ is a permutation. Conceptually, LAB-tree indexes the values of array elements by their linearized array indices; i.e., the element of array $A$ with index $i = \langle i_0, \ldots, i_d \rangle$ is indexed as the key-value pair $(f(i), A[i])$. LAB-tree supports arbitrary user-defined linearization functions; for convenience and efficiency, however, the frequently used ones have support built into LAB-tree, with examples given below.

**Example 5.** Besides the commonly seen row-major, column-major and block-based linearizations, a plethora of space-filling curves can also serve as linearization functions, providing various spatial locality properties. For example, the Z-order (Morton order) linearizes a 2D coordinate $(x, y)$ by interleaving the bits of the binary forms of $x$ and $y$. Let $\langle x_{n-1}x_{n-2}\ldots x_0 \rangle$ denote the binary representation of $x$, i.e., $x = \sum_{i=0}^{n-1} 2^i x_i$, where each $x_i \in \{0, 1\}$. Given the 2D coordinate $(x, y)$, its linearized coordinate is $f_Z(x, y) = \langle x_{n-1}y_{n-1}\ldots x_0y_0 \rangle$. As another example of linearization backed by bit operations, consider the bit-reversal order, which reverses the bits in the binary form of a coordinate. Row-wise bit-reversal linearization, a key component in 2D FFT, has function $f_{BR}(x, y) = xn + \langle y_0y_1\ldots y_{n-1} \rangle$, where $n$ is the number of columns. Column-wise bit-reversal can be similarly defined. Other extensively studied space-filling curves include Hilbert Curve and Peano Curve, whose mathematical definitions are more involved; interested readers can refer to standard texts [Sag94].

Each LAB-tree also has a default value (often 0) for array elements. Conceptually, LAB-tree only indexes elements whose values differ from the default. A new, “empty” array is filled with the default value. Setting a default-valued element to non-default value amounts to an insertion; the inverse operation amounts to a deletion. For
convenience and without loss of generality, we will assume the default value to be 0 for the remainder of the chapter.

With LAB-tree, we support three types of array accesses:

- Accessing an element by its array index \( \vec{i} \), which amounts to accessing the LAB-tree with key \( f(\vec{i}) \).

- Accessing elements of an array via an iterator with linearization function \( g \), which specifies the access order and may differ from the linearization function \( f \) used for controlling the storage order. The \( i \)-th element in the access order has LAB-tree key \( f(g^{-1}(i)) \). We implement various optimizations to speed up key calculation, including incremental computation of \( f \circ g^{-1} \) and detecting the special (but common) case of \( f = g \). Further details can be found below. We also support an option to iterate over only non-zero elements.

- Reading/writing elements in a specified hyper-rectangle in the array index space. This type of access is common in I/O-efficient matrix algorithms (such as multiply) that process matrices a chunk at a time, whose size depends on the amount of available memory. Supporting such accesses as batch operations allows us to avoid the overhead of iterator calls and provide more efficient implementation for built-in linearization functions.

**Speeding Up Linearization** Accessing elements of an array with storage linearization function \( f \) via an iterator with linearization function \( g \) is flexible but sometimes may incur considerable computational overhead. To speed up key calculation, we implement various optimizations. First note that the \( i \)-th element in the access order has LAB-tree key \( f(g^{-1}(i)) \), which would be \( f().linearize(g().unlinearize(i)) \) if translated into our API (\( f \) and \( g \) are instances of some subclass of Linearization class).
• For the special case $f = g$, the result is simply $i$. Our API requires all linearization subclasses to implement an `equals()` method to allow such detection.

• We also support incremental computation. Having already computed $f(g^{-1}(i - 1))$, we rely on another API function that can be optionally supplied by the user for linearization objects to speed up the calculation of $f(g^{-1}(i))$. Specifically, each linearization subclass can implement a `move (base coord, key diff)` method. Semantically, it computes `unlinearize (linearize (base coord) + key diff)`, but it can potentially use the previously computed result, `base_coord`, and adjust it with `key_diff` as determined by the linearization function. For example, for row-major linearization, `move` can be implemented by simply adding `key_diff` to the column index of `base_coord`, with “carrying” considered. This is computationally cheaper than a complete `unlinearize` call, where a mod and a divide operation are needed. In the iterator case, we can simply pass in $g^{-1}(i - 1)$ (cached) as `base_coord` and 1 as `key_diff`. Depending on the concrete linearization, incremental computation may or may not be faster than direct computation; in the latter case, the linearization implementor can always omit `move` or declare that it should not be used.

4.4 Efficiency Through Better Space Utilization

This section tackles B-trees’ efficiency problem from two angles: splitting strategy (Section 4.4.1) and leaf storage format (Section 4.4.2). Both aim at improving space utilization, which, as pointed out in [RS81] and validated by our empirical study (Section 4.4.3), is largely in line with the goal of improving time efficiency as well. We show that by exploiting the special characteristics of arrays, LAB-trees can achieve much better performance than conventional B-trees.
4.4.1 Splitting Strategy Revisited

As motivated in Section 4.1, the standard B-tree splitting strategy can lead to lots of wasted space within leaves that will never get used. In the following, we formalize the desirable properties of a splitting strategy, propose several alternatives, and discuss their properties.

We begin with some terminology. Let $\kappa$ denote the leaf capacity, or the maximum number of records that can be stored in a leaf of the index. Each leaf has a (key) range, which contains all keys of records stored in this leaf. The set of all leaf ranges forms a disjoint partitioning of the key domain. Since our index stores a 1-d array, a leaf range is an interval $[l, u)$, where $l$ and $u$ are the lower bound (inclusive) and upper bound (exclusive) of the 0-based array indices stored in the leaf. We define the density of a leaf $\ell$, denoted $\rho(\ell)$, as the number of records in $\ell$ divided by its capacity. Density can be similarly defined for a set of leaves or the entire index.

When a record needs to be inserted into a leaf with range $[l, u)$ and already $\kappa$ records (thereby causing it to overflow), a splitting strategy chooses a splitting point $x$, such that the original leaf is split into two leaves with ranges $[l, x)$ and $[x, u)$. A splitting strategy operates in an online fashion; i.e., it processes the current insertion without knowledge of future insertions. To ensure low runtime overhead, we consider only local splitting strategies, i.e., ones that do not read or modify leaves other than the one being inserted into. Also, we focus on leaf splitting strategies; splitting at upper levels of the index has little impact on the overall space and efficiency, and we simply follow the standard B-tree strategy.

The standard B-tree leaf splitting strategy is as follows:

- **Split-in-Middle.** Given an overflowing leaf with $\kappa + 1$ records with keys $i_0, i_1, \ldots, i_\kappa$, this strategy chooses the splitting point to be $x = i_j$, where $j = \lceil (\kappa + 1)/2 \rceil$. 


There are two desirable properties that a good splitting strategy should have: *bounded space consumption* and *no dead space*. The space consumption of a splitting strategy can be measured by its competitive ratio with respect to an optimal offline algorithm. Formally, a splitting strategy $\Sigma$ is $\alpha$-competitive if, for any insertion sequence $S$, the number of leaves produced by $\Sigma$ at the end of $S$ is less than $\alpha$ times that produced by an optimal offline algorithm, within an additive constant. Knowing the entire $S$, the optimal offline algorithm basically stores all non-zero array elements compactly, so an array with range $[0, N)$ and $n \leq N$ non-zero elements can be stored in $\lceil n/\kappa \rceil$ leaves.\(^2\)

Split-in-middle is clearly 2-competitive, because it always generates leaves that are half full. It turns out that this competitive ratio is the best we can hope for: we show that no deterministic local splitting strategy can have a competitive ratio of less than 2 (Theorem 3 in Appendix B).

A second desirable property of splitting strategies is *no-dead-space*. By “dead space” we mean empty slots in leaves that can never be filled by future insertions. For example, every leaf except the last one in Example 1 has two slots of dead space. Note that the notion of dead space is special to unique indexes with discrete key domains such as our setting. General B-tree leaves do not have dead space; it is always possible to insert a record with a duplicate key, or a record between two adjacent existing keys (up to some limit—precision of floating-point keys or maximum length of string keys). Formally, we define the no-dead-space property as follows. Without loss of generality, assume that the array size is a multiple of $\kappa$.\(^3\)

**Definition 1** (No-Dead-Space). A splitting strategy $\Sigma$ is no-dead-space if for any index state $\Sigma$ may result in, there exists a future insertion sequence that causes all

\(^2\) We assume standard B-tree leaf format for now; optimizations for dense array regions are discussed later in Section 4.4.2.

\(^3\) Otherwise, for an array with range $[0, N)$, the last leaf can have $N - \kappa \lfloor N/\kappa \rfloor$ slots of dead space.
leaves to be full under $\Sigma$.

As we have seen Section 4.1, split-in-middle does not have this property. But how important is no-dead-space, given that split-in-middle already has the best possible competitive ratio? Consider any array (or a region within an array) with density $\varrho$. A strategy that is no-dead-space would be guaranteed to have a competitive ratio of no more than $1/\varrho$ for storing the array (or the dense region). In contrast, regardless of density, split-in-middle may well take twice the minimum space required, as illustrated in Example 3. Thus, split-in-middle is less attractive than a no-dead-space strategy when $\varrho > 1/2$, which is a rather common case in our setting. For example, all dense matrices fall into this case, unless they are at the early stage of being populated in non-sequential order. Hence, no-dead-space is an important property that focuses less on the worst case and more on the common case of dense matrices or dense regions in matrices.

We propose a novel strategy that is naturally no-dead-space:

- **Split-Aligned.** Given an overflowing leaf $\ell$ with range $[l, u)$, this strategy chooses the splitting point $x$ to be a multiple of $\kappa$ that minimizes the difference between the number of records in $[l, x)$ and that in $[x, u)$. If multiple values of $x$ satisfy the condition, the one that minimizes $|x - \frac{l+u}{2}|$ is chosen.

In other words, split-aligned favors a split that is most balanced, like split-in-middle, but under the condition that the splitting point aligns with $0, \kappa, 2\kappa, \ldots$, i.e., endpoints of the leaf ranges had we laid out all array elements (zero or non-zero) compactly. For example, with $\kappa = 5$, split-aligned will choose the following split:

<table>
<thead>
<tr>
<th>10</th>
<th>11</th>
<th>14</th>
<th>21</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>27</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

51
It is easy to see that, starting with a single leaf with range \([0, N]\), split-aligned is no-dead-space.

An obvious question is how split-aligned does on competitive ratio. Unfortunately, there is a fundamental trade-off between no-dead-space and bounded space consumption—we show that any no-dead-space splitting strategy must have a competitive ratio of at least 3 (Theorem 5 in Appendix B), which is worse than split-in-middle in the worst case. We also show that split-aligned indeed has a competitive ratio of 3; i.e., it is the best no-dead-space strategy possible (Theorem 6 in Appendix B). This bound is non-trivial, considering that split-aligned may generate near-empty leaves.

Besides split-in-middle and split-aligned, we also consider:

- **Split-off-Dense.** Given a leaf to split with range \([l, u]\), this strategy first considers two candidate splitting points \(l + \kappa\) and \(u - \kappa\), which would result in a leaf with range \([l, l + \kappa]\) or one with range \([u - \kappa, u]\), respectively. Note these leaves will never be split further. If either leaf has density greater than 0.5, we choose the splitting point that would result in the leaf with the higher density. Otherwise, we fall back to split-in-middle. Intuitively, this strategy can be seen as a tweak to split-in-middle that first tries to split off a dense leaf that will not split again in the future. It is not hard to see that split-off-dense is no worse than split-in-middle in terms of competitive ratio, but split-off-dense may sometimes do better, e.g., the sequential insertion sequence in Example 3.

- **Split-Defer-Next.** This strategy tries to choose a splitting point that delays the split of either result leaf as much as possible. Suppose we split a leaf \(\ell\) with range \([l, u]\) and keys \(i_0, \ldots, i_\kappa\) into leaves \(\ell_1\) and \(\ell_2\) with splitting point \(x\). Assuming that each future insertion hits each missing key with equal probability, we can calculate \(\tau(x)\), the expected number of future insertions into \([l, u]\) that will cause the first split of either \(\ell_1\) or \(\ell_2\), using a formula involving \(l, u,\) and
\(i_0, \ldots, i_\kappa\) as shown below. Split-defer-next choose the splitting point to be 
\[
\arg \max_x \tau(x).
\]
Let \(k_1 = 1 + \arg \max_j i_j < x\) and \(k_2 = \kappa + 1 - k_1\) be the numbers of records initially in \(\ell_1\) and \(\ell_2\), respectively. Let \(b_1 = \kappa - k_1\) and \(b_2 = \kappa - k_2\) be the numbers of free storage slots initially in \(\ell_1\) and \(\ell_2\), respectively. Let \(d_1 = x - l - k_1\) and \(d_2 = u - x - k_2\) be the numbers of keys initially missing from the ranges of \(\ell_1\) and \(\ell_2\), respectively. If \(d_1 \leq b_1\) and \(d_2 \leq b_2\), \(\tau(x) = \infty\); otherwise,
\[
\tau(x) = \sum_{k \in [b_1, b_1 + b_2]} (k + 1) \cdot \frac{\binom{d_1}{b_1} \binom{d_2}{k-b_1}}{\binom{d_1 + d_2}{k}} \cdot \frac{d_1 - b_1}{d_1 + d_2 - k} + \sum_{k \in [b_2, b_1 + b_2]} (k + 1) \cdot \frac{\binom{d_2}{b_2} \binom{d_1}{k-b_2}}{\binom{d_1 + d_2}{k}} \cdot \frac{d_2 - b_2}{d_1 + d_2 - k}.
\]
This expectation is calculated over all possible sequences of \(d_1 + d_2\) insertions into \([l, u]\). We give the intuition behind the first summation; the second summation is analogous. Each summand in the first summation corresponds to the case that \(\ell_1\) splits after \(k + 1\) insertions into \([l, u]\). For this case to happen, the first \(k\) insertions must have completely filled up \(\ell_1\)‘s space, and the last insertion still goes into \(\ell_1\)’s range. The second term in the summand, \(\frac{\binom{d_1}{b_1} \binom{d_2}{k-b_1}}{\binom{d_1 + d_2}{k}} \cdot \frac{d_1 - b_1}{d_1 + d_2 - k}\), calculates the fraction of all insertion sequences whose first \(k\) insertions fill up \(\ell_1\) completely. The third term, \(\frac{\binom{d_1}{b_1} \binom{d_2}{k-b_1}}{\binom{d_1 + d_2}{k}} \cdot \frac{d_2 - b_2}{d_1 + d_2 - k}\), further calculates the fraction of these insertion sequences whose \((k + 1)\)-th insertion goes into \(\ell_1\)’s range.

The above formula can be further simplified as
\[
\tau(x) = \sum_{k \in [0, b_1 + b_2]} (k+1) \binom{k}{b_1} \binom{d_1 + d_2 - k}{d_1 + d_2 - b_1 - 1} + \binom{k}{b_2} \binom{d_1 + d_2 - k}{d_1 + d_2 - b_2 - 1} \binom{d_1 + d_2}{k}.
\]
Unfortunately, the formula for \(\tau(x)\) is quite involved, and we have no closed-form solution for this maximization problem; therefore, we resort to trying every \(x \in \{i_1, \ldots, i_\kappa\}\) in a brute-force fashion.
• **Split-Balanced-Ratio.** This strategy shares the same goal as split-defer-next, but uses a simpler optimization objective that is computationally easier. Given a leaf $\ell$, consider the ratio $\chi(\ell)$ between the number of free storage slots in $\ell$ and the number of keys missing from (and hence can be later inserted into) $\ell$’s range. Intuitively, a bigger $\chi(\ell)$ means $\ell$ is less likely to split in the future. Split-balanced-ratio picks the splitting point that maximizes the minimum of the two resulting leaves’ ratios. Specifically, given an overflowing leaf with range $[l, u]$ and keys $i_0, i_1, \ldots, i_\kappa$, this strategy sets $x = i_k$, where

$$k = \arg \max_j \left( \min \left( \frac{\kappa - j}{(i_j - l) - j}, \frac{\kappa - (\kappa + 1 - j)}{(u - i_j) - (\kappa + 1 - j)} \right) \right).$$

Section 4.4.3 compares these strategies with split-in-middle and split-aligned using various metrics, and evaluates their performance in practice with common workloads for matrices.

**Deletions** We have only discussed insertions so far. For all splitting strategies discussed in Section 4.4.1, it is possible to devise a strategy for merging LAB-tree leaves analogous to that for B-tree. On the other hand, except for split-in-middle, there is no analogy of stealing from an adjacent leaf, because it might undo the careful choice of leaf range endpoints.

For our LAB-tree implementation, we in fact adopt a simpler approach taken by many practical B-tree implementations. Namely, we do not merge index nodes when underflow occurs; a node is deleted only when it is completely empty. The competitive ratio for space consumption will be broken for workloads involving deletions, but it is acceptable because deletions are rare in our workloads and hence less important to overall performance.
4.4.2 Dynamic Leaf Storage Format

As discussed in Section 4.1, plain B-trees are not efficient for dense arrays. We want LAB-tree to be efficient for dense arrays as well as arrays whose sparsity varies over time and across different regions inside them. To this end, LAB-tree supports two leaf storage formats, *sparse* and *dense*. Different leaves can have different storage formats, and each leaf can switch between the two formats dynamically. A *sparse*-format leaf stores each non-zero array element in its range as a key-value pair; zeros are not stored. Let $\kappa_s$ denote the *sparse leaf capacity*, i.e., the maximum number of records that can be stored by a sparse-format leaf. A *dense*-format leaf, on the other hand, stores all values (zero or non-zero) of array elements from a continuous subrange of its key range. The key that starts the subrange is also stored, but the other keys in the subrange are not, because they can be simply inferred from the starting key and the entry positions. Let $\kappa_d$ denote the *dense leaf capacity*, i.e., the maximum length of the subrange, or the maximum number of records that can be stored by a dense-format leaf. Clearly, $\kappa_d > \kappa_s$. For example, if the keys are 64-bit integers and values are 64-bit doubles, then $\kappa_d \approx 2\kappa_s$. This two-format approach can be regarded as a simple compression method, which we feel provides a good trade-off between storage space and access time. More sophisticated compression methods are certainly possible, but they will likely add non-trivial decompression overhead to data accesses.

LAB-tree automatically switches between the two formats when a leaf is written. We call the *effective range* of a leaf $\ell$ to be the tightest interval containing all keys stored in $\ell$. The effective range of $\ell$ is always contained in the range of $\ell$. If an insertion overflows a sparse-format leaf $\ell$, and the length of $\ell$’s effective range (containing all $\kappa_s + 1$ keys) is no greater than $\kappa_d$, then we switch $\ell$ to the dense format without splitting $\ell$. Conversely, if an insertion into a dense-format leaf $\ell$ expands the length of its effective range to greater than $\kappa_d$ but the total number of records is still below
\( \kappa_s \), then we switch \( \ell \) to the sparse format without splitting \( \ell \).

The splitting strategies in Section 4.4.1 need to be modified to work with the dynamic leaf format. For split-aligned, we require the splitting point to be a multiple of \( \kappa_d \). Other necessary modifications are not difficult to devise, but care is needed to cover all possible cases. Because of limited space, we will illustrate just one intricacy with an example. With \( \kappa_s = 4 \) and \( \kappa_d = 8 \), consider the following overflowing dense leaf upon the insertion of key 97:

```
0 5 6 7 8 9 10 11 12 256
```

Without modification, split-aligned would choose 8 (a multiple of \( \kappa_d \)) as the splitting point. However, the result right leaf cannot store all of 8, 9, 10, 11, 12, and 97, with either dense or sparse format. Hence, it is necessary to further modify split-aligned to rule out infeasible splitting points. In this case, 8 will be ruled out, and 96 will be chosen instead.

### 4.4.3 Experiments

**Setup** We ran all our experiments on a Dell Optiplex 960 running Fedora 14 (kernel version 2.6.35.11), with Intel Core 2 Duo E8500 3.16GHz CPU, 8GB of memory, and a 160GB SATA hard drive. We used the `systemtap` tool to measure I/O and time costs, with `systemtap`’s built-in system call probes as well as some user-space probes in our code. We verified that instrumentation overhead was negligible. To make it easier to understand results, we used the `ext2` file system (unless otherwise noted), as it does not have journaling that would unnecessarily complicate result interpretation; we also implemented our own buffer pool manager with LRU and 8KB-sized pages, and turned off file system caching using the `O_DIRECT` flag.
Figure 4.1: Splitting strategies, with all leaves using the sparse format. In the first three graphs (for seq, str, and int), horizontal axes show the percentage of elements inserted so far; each plot contains one data point every 1000 insertions, and shows one tick every $10^8$ insertions. In the last figure, the vertical axis shows the break-down of running time into I/O and CPU, with CPU on top.4

**Splitting Strategies on Common Insertion Patterns** We first compare the performance of various splitting strategies, for now assuming sparse formats across all leaves. We consider the following patterns for populating an initially empty matrix with row-major layout: **seq**uential inserts elements in row-major order; **str**orted inserts elements in column-major order; **int**erleaved inserts elements in row- and column-major orders in an interleaving fashion (as in LU factorization); and **ran**dom inserts elements in random order.

Figure 4.1 summarizes the results for a $20000 \times 20000$ matrix and a 320MB buffer pool. Results on other scales are similar. For this experiment, **ran** is too expensive
to run to completion; it takes an hour just to process 4% of the insertions. As its performance is clearly unacceptable regardless of the choice of splitting strategy, we do not discuss ran further here. We will, however, revisit ran in Section 4.5.3 because update batching helps improve its performance.

From the first three graphs in Figure 4.1, we see that standard split-in-middle uses about twice as much space as others throughout the course of each workload. From the last graph, we see that split-in-middle’s simpler splitting logic is not enough to make up for its loss in I/O efficiency. On the other hand, split-aligned maintains a noticeable lead ahead split-in-middle in running time, and is the best strategy overall in both space and time efficiency.

As for other strategies, split-off-dense has curiously high running time for str despite its low number of I/Os (whose plots are not shown here but are consistent with the first three graphs); a closer examination of the traces reveals that split-off-dense’s tendency to generate far more unbalanced leaves than others leads to very scattered I/Os. Split-balanced-ratio has no better space utilization than split-aligned but carries higher CPU overhead. We omit split-defer-next here and subsequently, because it has prohibitive CPU overhead but offers no significant space savings.

Next, we repeat the experiments with dynamic leaf storage format, to study how this feature further affects performance. Figure 4.2 summarizes the results. All strategies benefit from this feature, but split-aligned benefits more, thanks to its ability to produce leaves that are better aligned (and hence better “prepared”) for the dense format. For the more interesting patterns of str and int, its advantage over split-in-middle widens to a factor of more than 3.5 in terms of space, and more than 1.7 in terms of time; its advantage over other strategies are also more pronounced.

Note that our CPU time accounting includes time spent outside system calls on behalf of I/Os. In particular, time spent on I/Os served from our buffer pool without hitting the disk is counted towards the CPU time instead of the I/O time. In this figure, the CPU time’s significant proportion is in part explained by the effectiveness of our buffer pool for these workloads.
than in Figure 4.1. Moreover, the relative performance differences stay the same over the course of the workloads (plots are omitted here, but exhibit the same linear trends as the first three graphs in Figure 4.1). In conclusion, split-aligned is a clear winner.

Finally, note that these experiments only report the running time of populating the matrix. Split-aligned, with its highest space efficiency, becomes even more appealing if we consider the cost of accessing the matrix subsequently. For other strategies, one could bulk load (and compact) the array at end of the insertion sequence to make subsequent scans more efficient, but doing so would further add to the running time and, for a dense matrix, result in a final tree no better than split-aligned.

**Scalability Test** The experiments above are all performed on a $20000 \times 20000$ matrix (with 400 million elements). We also vary the matrix size and plot the normalized total running time (obtained by dividing the total running time by that of split-in-middle) in Figure 4.3. The results show a consistent relative gap between split-in-middle and split-aligned, with or without the dynamic leaf storage format. In terms of absolute running time (not plotted here), both strategies scale linearly with the matrix size. It is clear that split-aligned’s space efficiency advantage extends to different data scales.
Figure 4.3: Splitting strategies: scalability test with sparse (top row) and dynamic (bottom row) leaf formats. X-axes show the scale of matrix ($\times 10^6$ elements), while y-axes show the normalized total running time (in-middle as baseline).
Buffer Pool Settings We next replicate the experiments in Figure 4.2 with different buffer pool sizes: a smaller 200MB and a bigger 440MB. The I/O and CPU time breakdown for the four splitting strategies with dynamic leaf page format is shown in Figure 4.4. Split-aligned and split-off-dense are generally able to better exploit a larger buffer pool to reduce their I/O time, although a larger-than-enough buffer pool does not bring further benefit, and in some case may even cause extra CPU overhead (namely split-aligned with 440MB pool under seq). Split-in-middle and split-balanced-ratio are relatively insensitive to the size of buffer pool. In this sense, their performance is more predictable. However, even if the memory resource is scarce, split-aligned still has considerable advantage over them.

LAB-Tree, B-Tree, and Directly Addressable File We now step up a level and compare the performance of LAB-tree (with split-aligned and dynamic leaf storage format), standard B-tree (with split-in-middle and sparse leaf format), and directly addressable file (DAF). DAF stores all array values compactly in a file, enabling direct lookups and eliminating the need to store array indices or to use extra indirections for indexing. File system optimizations allow us to allocate disk pages for DAF lazily: if a page has never been written (because it contains all zeros), it is never allocated.

First, we repeat the same experiments for a 20000 × 20000 matrix in Figure 4.2, and summarize the results in Figure 4.5. In terms of space utilization, LAB-tree is on par with DAF, the best possible in this case; B-tree is four times worse, because it lacks the dense format and its leaves are mostly half-full. As for running time, the break-down into CPU and I/O offers interesting insights. In terms of CPU time, DAF is the fastest, and B-tree is the slowest; the reasons are that DAF’s direct address calculation is simpler than tree lookups, and that searching with the sparse leaf format (which B-tree uses exclusively) is more expensive than the dense format. In terms of I/O time, B-tree suffers from a larger number of I/Os. Surprisingly, DAF has the
Figure 4.4: Impact of buffer pool size.
Figure 4.5: LAB-tree, B-tree, DAF: 20000 × 20000 dense matrix.

Figure 4.6: LAB-tree, B-tree, DAF: 40000 × 40000 sparse matrix.

worst I/O time for str and int, even though it incurs a similar number of I/Os (not plotted here) as LAB-tree. A closer look shows that DAF generates very scattered I/Os because column-major insertions hit faraway portions of the file. In this regard, LAB- and B-trees are better at placing and moving array elements during the course of these workloads. This observation offers the insight that it can be suboptimal to simply place each element where it should be at the end of the insertion sequence, as the intermediate states of the data structure also affect performance.

In the second set of experiments, we populate a sparse 40000 × 40000 matrix with 10% randomly distributed non-zero elements. Figure 4.6 summarizes the results. As expected, DAF really suffers while B-tree shines, as there are not even locally dense regions in this matrix. Despite being unable to exploit any density, LAB-tree
maintains comparable performance to B-tree, except that LAB-tree has slightly higher I/O time due to slightly more random I/Os.

From the above two sets of experiments, which straddle the opposite ends of the dense-sparse spectrum, we see that LAB-tree is able to automatically achieve optimal (or close to optimal) performance without manual tuning.

**Scalability Test** We also scaled the experiments above with different matrix sizes (Figure 4.7). While LAB-tree and B-tree scale linearly under all tests, DAF’s scalability is not linear. For dense matrices under non-sequential insertion pattern, DAF’s performance degrades quickly and becomes inferior to LAB-tree as the matrix size increases. For sparse matrices, DAF is always substantially slower than LAB-tree and B-tree. Also note that across all scales LAB-tree is able to maintain a factor of 2 performance advantage over B-tree for dense matrices, while having comparable performance for sparse matrices.

**More Interesting Insertion Patterns** We have only considered three common yet fundamental insertion patterns so far, namely seq, str and int. Note that these patterns are independent from the storage layout or access pattern; instead, an insertion pattern is generated by a combination of two linearizations—access and storage. For instance, str can happen if a row-major layout matrix is populated in column-major order, or vice versa. Now, we are ready to test two other patterns obtained by inserting into matrices with a block-based layout.

Given a matrix, we choose a block-based linearization as its layout. We set the block size to be $31 \times 31$ (the biggest size that can still fit in a disk page). Within every block, elements are laid out in row-major order, and so are the blocks themselves. On top of this fixed block storage layout, we consider two ways of populating a matrix: row-major order and row-wise bit-reversal order. We call the two resulting patterns row/block and bit-reversal/block, respectively. Note that the second access pattern is
Figure 4.7: LAB-tree, B-tree, DAF: scalability on dense and sparse matrices. X-axes show the scale of matrix ($\times 10^6$ elements, including zeros in case of sparse matrix), while y-axes show the normalized running time (B-tree as baseline).
an essential part of the 2D FFT algorithm. Combining the block storage layout with these two access patterns, the resulting patterns hitting the linear storage medium become more complicated and interesting.

We test the two patterns on a $20000 \times 20000$ dense matrix. Figure 4.8 plots the results. Again, in terms of space utilization, LAB-tree is the same as DAF, the best possible in both cases. B-tree is more than three times worse due to its lack of dense format. In terms of both I/O time and total time, B-tree is also the worst, not surprisingly. For row/block, LAB-tree’s I/O time is on par with DAF’s, but it has more CPU overhead; so the result is similar to seq in Figure 4.5. For bit-reversal/block, LAB-tree’s I/O time is only 60% of DAF’s, which is enough to compensate for its higher CPU time. Overall, the results from these two new insertion patterns agree with previous results in Figure 4.5 and do not change our conclusion.

![Figure 4.8](image)

**Figure 4.8:** LAB-tree, B-tree, DAF: more insertion patterns on blocked $20000 \times 20000$ dense matrix.

**BLAS on UF Sparse** Stepping up yet another level, we examine how LAB-tree compares with B-tree and DAF for linear algebra operations involving real-world matrices. For the operation, we test matrix multiply, an essential and often performance-critical building block of more sophisticated analysis. We use an I/O-efficient version of the block matrix multiply algorithm, which computes the result matrix one block
(submatrix) at a time by reading and multiplying pairs of blocks from the input matrices and accumulating the multiplication results in memory. For multiplying submatrices in memory, we use the BLAS routine \texttt{dgemm} if both submatrices have density greater than 0.5, or the CHOLMOD [CDHR08] routines \texttt{cholmod_ssmult} or \texttt{cholmod_sdmult} otherwise.

For input, we use matrices from UFSparse, the University of Florida Sparse Matrix Collection [DH]. To test each storage method, we prepare the input matrices with this method using a blocked layout that matches the pattern of blocks accessed by the I/O-efficient matrix multiply. We multiply each input matrix with itself, and save the result using the same storage method as the input. Here, we discuss results for two matrices, \texttt{human\_gene2} and \texttt{TSOPF\_RS\_b2383} (Figure 4.9). We report the total running time, which excludes input preparation but includes writing the result.

For \texttt{human\_gene2} (14340 $\times$ 14340 and density 8.79%), we use 1500 $\times$ 1500 blocks, and the total running time is 386sec for LAB-tree, 461sec for B-tree, and 1170sec for DAF. DAF suffers from a bloated input file. LAB- and B-trees both perform well, with LAB-tree leading by about 16%. Their input trees are comparable in size, because \texttt{human\_gene2} looks uniformly sparse. The result matrix turns out fairly dense, so the LAB-tree result is more compact.
For TSOPF_RS_b2383 (38120 × 38120 and density 1.11%), we use 4000 × 4000 blocks, and the total running time is 388sec for LAB-tree, 615sec for B-tree, and 1088sec for DAF. Unlike human_gene2, this matrix has a dense region despite its overall sparsity. LAB-tree is able to exploit this local density to widen its lead over B-tree to a factor of 1.6. Its lead over DAF narrows slightly, but is still more than a factor of 2.8.

Results on more matrices are presented in Table 4.1. The conclusion is consistent: for sparse matrices, LAB-tree performs much better than DAF, and as well as or better than B-tree (depending on the uniformity of sparsity); for the full matrix, LAB-tree has comparable performance to DAF, which is the best, while B-tree really suffers from its space inefficiency.
Table 4.1: LAB-tree, B-tree, DAF: total running time of \texttt{dgemm} on UFSparse and dense matrices.

<table>
<thead>
<tr>
<th>Name (ID)</th>
<th>Size</th>
<th>#Nonzeros</th>
<th>LAB-tree (s)</th>
<th>B-tree (s)</th>
<th>DAF (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt1 (1271)</td>
<td>$15449 \times 15449$</td>
<td>1930655</td>
<td>8.5</td>
<td>8.7</td>
<td>110.1</td>
</tr>
<tr>
<td>ramage02 (1274)</td>
<td>$16830 \times 16830$</td>
<td>2866352</td>
<td>22.0</td>
<td>22.1</td>
<td>167.1</td>
</tr>
<tr>
<td>ship_001 (1277)</td>
<td>$34920 \times 34920$</td>
<td>3896496</td>
<td>51.8</td>
<td>50.0</td>
<td>792.9</td>
</tr>
<tr>
<td>std1_Jac2 (1334)</td>
<td>$21982 \times 21982$</td>
<td>1248213</td>
<td>31.6</td>
<td>32.4</td>
<td>258.6</td>
</tr>
<tr>
<td>Ga3As3H12 (1352)</td>
<td>$61349 \times 61349$</td>
<td>5970947</td>
<td>131.6</td>
<td>132.4</td>
<td>4583.8</td>
</tr>
<tr>
<td>net75 (1392)</td>
<td>$23120 \times 23120$</td>
<td>1489200</td>
<td>13.3</td>
<td>16.9</td>
<td>364.3</td>
</tr>
<tr>
<td>human_gene2 (2281)</td>
<td>$14340 \times 14340$</td>
<td>18068388</td>
<td>386</td>
<td>461</td>
<td>1170</td>
</tr>
<tr>
<td>TSOPF_RS_b2383 (2219)</td>
<td>$38120 \times 38120$</td>
<td>16171169</td>
<td>388</td>
<td>615</td>
<td>1088</td>
</tr>
<tr>
<td>Dense</td>
<td>$10000 \times 10000$</td>
<td>$100000000$</td>
<td>3349</td>
<td>5200</td>
<td>3292</td>
</tr>
</tbody>
</table>
4.5 Update Batching

We now turn to the problem of batching index updates in a memory buffer to consolidate writes to disk. To support index access while updates are being buffered, we organize this buffer as an index over the buffered updates; a record lookup would be first checked against this in-memory index. Whenever the buffer is full, we need to flush updates, i.e., applying them in a batch to the underlying disk-resident indexes. As discussed in Section 4.1, we question the common practice of flushing all buffered updates whenever the buffer is full. Section 4.5.1 presents alternative policies and a theoretical analysis of their performance. Section 4.5.2 discusses implementation issues and Section 4.5.3 presents an empirical evaluation.

4.5.1 Flushing Policies and Analysis

To simplify theoretical analysis, we make some assumptions. First, we view each update to a record \( r \) as a request for the disk page (leaf) that contains \( r \) or will contain \( r \), and we assume that we know the identities of all requested pages before each flushing action (see Section 4.5.2 for implementation details). Second, we assume that each flush incurs a fixed cost per update plus a fixed cost per page; multiple updates requesting the same page incur the per-page cost only once for the flush, reflecting the benefit of batching. Because the sum of per-update costs in the end remain the same no matter how we flush, we focus on minimizing the sum of per-page costs over time. Note that this analytical model is an imperfect simplification of reality. For example, it ignores the cost of obtaining page identities (Section 4.5.2) and that of splitting (which depends on factors such as the splitting strategy). Nonetheless, it provides a reasonable estimate of the true cost, and makes our analysis more generalizable to other batch processing settings.

---

5 The buffer in this context should not be confused with the system buffer pool. This buffer batches updates while the buffer pool caches disk pages.
With these assumptions, we now formally define the problem.

**Definition 2.** There are a set of pages $P$ on disk, and a buffer of capacity $K$ in memory for buffering requests. Every request refers to a page and takes unit space in the buffer.\(^6\) A flushing policy selects subsets of requests to flush as needed to keep the buffer size capped at $K$ at all times. Flushing requests for the same page incurs unit cost. We are interested in an online flushing policy that minimizes the total cost over a request sequence.

For brevity, by “buffered” requests we mean all requests eligible for flushing, which include the incoming request. Without loss of generality, we assume a policy only flushes when the buffer is full (any policy can be modified to do so without affecting the cost). We can also assume that if a policy flushes any request for $P$, it flushes all buffered requests for $P$; in this case, we simply say it flushes $P$.

As it may have occurred to the reader, this problem looks similar to cache replacement [ST85]. Unfortunately, known results on caching do not carry over. Although caching has been generalized to cases where pages can have varying sizes and eviction cost can be a function of the page size, an underlying assumption remains that the cache space devoted to a page $P$ does not change as the number of requests to $P$ increases. On the contrary, with our problem, $n$ requests to the same page take $n$ units of buffer space. This difference turns out to be fundamental. While we can develop flushing policies analogous to well-studied cache replacement policies, we will see that their performance differs both analytically and experimentally; new policies specialized for flushing are needed.

We now present our flushing policies. Here we summarize our theoretical results; see Appendix B for formal statements and proofs. We measure the performance of a flushing policy by its competitive ratio against $OPT$, the optimal offline policy.

---

\(^6\) Updates to currently buffered records are simply applied to the buffer, and are not counted as new requests. Therefore, $n$ requests, even if they are for the same page, would take $n$ units of space.
which knows the entire request sequence in advance. OPT can be implemented by an exponential-time search; the algorithmic details are irrelevant here. As a side note, the optimal offline cache replacement policy, *furthest-in-future* [Bel66], is not optimal for flushing, as can be seen from Example 6 below.

**Example 6.** Consider the following request sequence (both subscripts and superscripts are used to differentiate pages):

\[
P_0, P_0, \ldots, P_0, P_1^1, P_2^1, \ldots, P_{K-1}^1, P_0, P_{K-1}^1, P_{K-1}^1, \ldots, P_1^1,
\]

Starting from an initially empty buffer of size \(K\), it is easy to verify that furthest-in-future produces the following sequence of buffer states, for all \(i \geq 1\). Here, \([P]^n\) denotes \(n\) requests for page \(P\).

- **Immediately before the first** \(P_1^1\), we have: \([P_0]^{(K-1)}\).

- **Upon the first** \(P_j^1\), where \(1 \leq j \leq K - 1\), we flush \(P_{j-1}^1\) (if any),\(^7\) and get: \([P_0]^{(K-1)}, P_j^1\).

- **Upon the** \(P_0\) **immediately following the first** \(P_{K-1}^1\), we flush all \(P_0\), and get: \(P_{K-1}^1\).

- **Upon the second** \(P_j^i\), where \(K - 1 \geq j \geq 1\), we have, without flushing: \(P_j^i, P_{j+1}^i, \ldots, P_{K-2}^i, [P_{K-1}^i]^{*2}\).

- **Upon the** \(j\)-th \(P_0\) **following the second** \(P_1^1\), where \(1 \leq j \leq K - 2\), we flush \(P_j^1\), and get: \([P_0]^{*j}, P_{j+1}^i, \ldots, P_{K-2}^i, [P_{K-1}^i]^{*2}\).

\(\small{\text{\[^7\] This behavior of flushing the incoming request differs from the traditional furthest-in-future [Bel66], as all cache replacement policies by definition cache the incoming request. Flushing policies do not have this constraint, and it makes sense to not consider the incoming request as a “future” one.}}\)
Upon the $(K - 1)$-th $P_0$ following the second $P_i^1$, we flush $P_{K-1}^i$, and get: $\{[P_0]^{*(K-1)}\}$.

For each $i \geq 1$, furthest-in-future repeats the above steps, incurring $2K - 2$ flushes. On the other hand, the strategy that produces the following sequence of buffer states incurs only $K + \lfloor \frac{K}{2} \rfloor \leq 1.5K$ flushes for each $i$:

- Immediately before the first $P_i^1$, we have: $\{[P_0]^{*(K-1)}\}$.

- Upon the first $P_i^j$, where $1 \leq j \leq \lfloor \frac{K}{2} \rfloor$, we flush the (one) previously buffered page, and get: $\{P_i^j\}$.

- Upon the first $P_i^j$, where $\lfloor \frac{K}{2} \rfloor < j \leq K-1$, we get, without flushing: $\{P_i^j, \ldots, P_i^1\}$.

- Upon the $P_0$ immediately following the first $P_{K-1}^i$, we flush the new $P_0$ request, and get: $\{P_i^1, \ldots, P_i^i\}$.

- Upon the second $P_j^i$, where $K - 1 \geq j > \lfloor \frac{K}{2} \rfloor$, we have, without flushing: $\{P_i^j, \ldots, P_j^1, [P_j]^{*2}, \ldots, [P_{K-1}]^{*2}\}$.

- Upon the second $P_j^i$, where $\lfloor \frac{K}{2} \rfloor < j \geq 1$, we flush the incoming request, and get: $\{\}$.

- Upon the second $P_j^i$, where $\lfloor \frac{K}{2} \rfloor > j \geq 1$, we flush the incoming request, and get: $\{\}$.

- Upon the $j$-th $P_0$ following the second $P_i^j$, where $1 \leq j \leq K - 1$, we get, without flushing: $\{[P_0]^{*j}\}$.

We show that any policy is $O(K)$-competitive (Lemma 7). (Had we been dealing with caching instead, this competitive ratio would have been the best that any deterministic policy can offer.) The most commonly used flushing policy actually does better:
• **Flush-All (ALL).** This policy simply flushes the entire buffer whenever the buffer is full. We show that ALL is $\Omega(\sqrt{K})$- and $O(\sqrt{K}\log K)$-competitive (Theorems 8 and 11).

We can generalize the lower bound above to what we call $c$-recent flushing policies (Definition 9 in appendix), which do not buffer a request for a page if there has been no request for that page during the past $cK$ requests. Clearly, ALL is 1-recent. We show that any $c$-recent policy is $\Omega(\sqrt{K}/c)$-competitive (Theorem 9).

The next few flushing policies have analogies in caching:

• **Least-Recently-Used (LRU).** This policy always flushes the page whose most recent request is the oldest (among all pages’ most recent requests). It is analogous to the classic cache replacement policy of the same name. We show that LRU is $\Omega(\sqrt{K})$-competitive (Corollary 10) by noting that LRU is 1-recent. (Note that for caching, LRU is optimally competitive, with a competitive ratio of $K$.)

• **Smallest-Page (SP).** This policy always flushes the “smallest” page, i.e., one with the smallest number of currently buffered requests. It is analogous to the LFU (least-frequently-used) cache replacement policy. While LFU is widely used for caching, SP does not make much sense for flushing. Intuitively, SP flushes small pages, but flushing larger ones is more profitable as more requests can be processed with one page write. While SP attempts to preserve large pages, pages have little chance to grow large because they may get flushed when still small.

We show that SP is $\Theta(K)$-competitive (Lemma 7 and Theorem 13). The example constructed in the proof of Theorem 13 makes the above intuition concrete. (Note that for caching, LFU’s competitive ratio is unbounded.)
• **Largest-Page (LP).** This policy alwaysflushes the “largest” page, i.e., one
with the largest number of currently buffered requests. It is analogous to the
MFU (most-frequently-used) cache replacement policy. LP avoids SP’s problem
of flushing small pages. On the other hand, LP may flush a page prematurely
just because it is currently the largest; however, that page may grow even larger
if it not immediately flushed.

We show that, just like SP, LP is $\Theta(K)$-competitive (Lemma 7 and Theorem 12).
The proof of Theorem 12 gives a concrete example of the premature flushing
problem.

Next, we present two new polices: the first is a randomized variant of LP, while
the second is a novel policy aimed at achieving a fundamentally better competitive
ratio than the policies above.

• **Largest-Page-Probabilistically (LPP).** This policy randomly flushes a page
with probability proportional to the number of requests currently buffered for
this page. It can be seen as a randomization of LP. Intuitively, LPP is designed
to avoid the problems of LP and SP: larger pages have a higher chance of
being flushed, but all pages have a chance to survive and grow larger. Another
attractive feature of LPP is its efficiency of implementation, as we shall see in
Section 4.5.2.

• **Largest-Group (LG).** This policy partitions buffered requests into groups:
Group $i$, where $0 \leq i \leq \lfloor \log K \rfloor$, contains a page $P$ if the number of buffered
requests for $P$ is in the range $[2^i, 2^{i+1})$. We define the size of a group to be the
total number of buffered requests for its constituent pages. When the buffer is
full, LG flushes the group with the largest size.

LG is a novel policy designed specifically for the update batching problem.
Intuitively, LG’s practice of flushing a group at a time offers better protection against an adversary than flushing a page at a time. With \([\log K] + 1\) groups, the largest group has at least \(\frac{K}{[\log K]+1}\) requests, so LG always flushes a sizable number of requests. Even if LG had chosen a wrong subset of requests to flush, this mistake cannot be repeated until the buffer is full again, which only happens after at least \(\frac{K}{[\log K]+1}\) more requests. In contrast, an adversary can more easily penalize policies that may flush a few requests.

We show that LG has a competitive ratio of \(O(\log^3 K)\) (Theorem 15), making it the theoretically best among our policies.

### 4.5.2 Implementation

**Obtaining Page Identities and Ranges** All policies above except ALL require obtaining the page identity and key range for a buffered request. Such information is readily available by executing a “partial” lookup for the requested key in the LAB-tree, without visiting the leaf page containing the key. Only one partial lookup is needed for requests to the same page, because once we obtain page \(P\)’s range, we can check whether a request refers to \(P\) by comparing the requested key with \(P\)’s range. Since only non-leaf levels are visited, a generic system buffer pool (not to be confused with the update buffer) is effective in reducing I/Os.

**LP, SP, and LRU** At the time of flush, these policies make one pass over the buffered requests in key order. In the process, we find the identity and range of each requested page \(P\), using one partial lookup (as opposed to one per request to \(P\), as explained above). Policy-specific details are given below.

For LP and SP, for each requested page \(P\) we identify, we count the number of buffered requests to \(P\) using \(P\)’s range, and move to the first request to a different page. We remember the page with the largest or smallest (for LP or SP, respectively)
number of buffered requests encountered so far. LP can terminate the process early once we know that the largest page has been found, e.g., when its number of requests is no less than the number of remaining requests to be examined. SP can terminate early as soon as it finds a page with a single request.

For LRU, we record the time when each buffered request entered the buffer. This information requires additional space and therefore reduces the number of requests that can be buffered. However, this information can be compressed at the expense of accuracy. During the pass over the buffered requests, for each requested page $P$ we identify, we scan the buffered requests in $P$’s range and determine the last time when $P$ is requested. We remember the page with the earliest such time among the pages we have encountered.

To further reduce page identification cost, we maintain a cache for all the above policies that remembers the identity and range for up to a configurable number of pages. At the next flush, we avoid the cost of identifying such pages. Of course, this page information cache consumes space that could otherwise be devoted to buffering requests, which we account for in our empirical evaluation in Section 4.5.3.

**LPP** At the first glance, LPP seems to require knowing the counts of buffered requests for all pages. A far more efficient implementation is possible, however. We simply need to pick one buffered request uniformly at random, find the identity and range of its page, and flush that page (i.e., all buffered requests within its range). Clearly, this implementation picks a page with probability proportional to the number of buffered requests to this page.

Specifically, we store all buffered requests in an array $\mathcal{A}$ in memory in no particular order. We maintain an ordered search tree $\mathcal{T}$ on top of $\mathcal{A}$, which allows us to find the locations of requests in $\mathcal{A}$ given a request key range. To pick a page to flush, we simply pick a random location in $\mathcal{A}$ (which is full at the time of the flush). With the
key of the buffered request at this location, we obtain the key range for the page to be flushed, using a partial LAB-tree lookup. We then use this key range to search \( T \) for all buffered requests that we need to flush. As we flush them, we chain the reclaimed locations in \( \mathcal{A} \) into a linked list, whose links can be stored in \( \mathcal{A} \) itself. This linked list is then used for adding new requests to \( \mathcal{A} \) efficiently.

**LG** At the time of flush, LG makes one pass over the buffered requests in key order. For each requested page \( P \), we find and record the identity and range of \( P \); using \( P \)'s range, we count the number of buffered requests to \( P \); using this count, we determine and record the group number of \( P \); finally, we add the count to a running sum that maintains the size of \( P \)'s group. After this process, we make a second pass to flush the group with the largest size; requests in this group are those with keys that fall within the ranges of its constituent pages. Like LP, SP, and LRU, LG also maintains a page information cache across flushes to reduce the cost of page identification. We populate this cache in the second pass with a subset of pages that are not flushed.

**Batching Updates to Multiple Arrays** An application often updates multiple arrays simultaneously, so we have the problem of allocating buffer space among multiple LAB-trees. It is possible to have a dedicated buffer per array, and intelligently allocate space among multiple arrays according to their shares of the workload. This approach works well with the knowledge of future access patterns, which is often difficult to obtain without user input. Thus, we give the user the option to specify the amount of buffer to allocate to each array (or a subset of arrays).

In the absence of user input, we take a simple default approach—we have one single buffer for all arrays being updated by the application, and we rely on the flushing policy to determine what to buffer and what to flush across arrays. After all, for many applications, boundaries among different arrays are murky in the first place, and access patterns vary across different regions in the same array. For example, LU
decomposition often stores the result of decomposition—an upper-triangular matrix and a lower-triangular matrix—together as one single matrix. Hence, the approach of using a unified buffer is both clean and natural in general settings.

4.5.3 Experiments

We evaluate the flushing policies using seq, str, str, and ran, the four insertion patterns from Section 4.4.3. Here we discuss results for a matrix of size 4000 \times 4000 and a 32MB buffer pool. Updates are buffered in a separate 3MB memory buffer, which holds about 200,000 requests for ALL but fewer for others because of their extra space overhead. LRU, LP, SP, and LG maintain a cache that remembers information about 8000 pages; this space is charged against the update buffer. The scale of these experiments is smaller than those in Section 4.4.3, but allows us to obtain a complete set of results including those for the most demanding ran workloads. Additional results for larger scales are at the end of the section. Because of space constraints, we also omit LRU and SP; they incur unacceptably high CPU cost like LP, which is explained later when we discuss Figure 4.11.

Figure 4.10 shows the total number of actual I/Os incurred by each policy (which excludes those serviced by the buffer pool without hitting storage). This metric is unaffected by the characteristics of the underlying storage substrate. LPP, the randomized version of LP, turns out a winner: across all patterns, LPP is either the best or comes close to the best. ALL is noticeably worse than LPP for str (13% more I/Os), and much worse for ran (73% more I/Os). LG, despite its attractive worst-case theoretical guarantee, fails to distinguish itself for these common insertion patterns. LP has reasonable I/O counts, but we will soon see its crippling disadvantage.

As storage substrates grow more diverse and sophisticated, the relationship between I/O count and running time has become increasingly dependent on the system specifics. Therefore, we compare running time for two different storage substrates: ext2 on a
Figure 4.10: I/O counts of flushing policies.

Figure 4.11: Flushing policies on local drive.

Figure 4.12: Flushing policies on NFS.
local hard drive, and NFS over network-attached storage (NAS).

Figure 4.11 summarizes the results for the local hard drive. Here ALL really shines. A closer inspection reveals that LPP suffers from random I/Os because of its inherent randomness; its I/O times are higher than ALL even when its I/O counts are much lower. The high CPU overhead destroys LP, because at the time of each flush, it needs to scan the entire buffer and identify all pages requested. Although the buffer pool is efficient in reducing I/O needed for page identification (as evidenced in Figure 4.10), the CPU overhead remains. Another interesting observation is the stark contrast between LP’s high CPU overhead and LG’s low CPU overhead, since LG’s flushing procedure seems more costly than LP’s. However, by flushing only large groups, LG flushes much less frequently than LP, so the amortized cost of its procedure becomes much lower.

On the other hand, Figure 4.12, which summarizes the results for NAS through NFS, tells a very different story. Here, the I/O component of the total time is more consistent with the I/O count in Figure 4.10. For this reason, and because of LPP’s low CPU overhead, LPP’s I/O count advantage over ALL carries over, and LPP becomes the overall winner.

We next present results for a matrix of size $20000 \times 20000$ and a 160MB buffer pool. The update batching buffer has a size of 800MB (equivalent to 50,000,000 requests for ALL). Because of its exorbitant CPU cost, LP took far more time to run than the other policies; so we omit its results here. We saw a long running time for LPP under run again due to random I/Os. Therefore we terminated the run experiments for all policies after 37.5% matrix elements were inserted. The I/O counts in Figure 4.13 are mostly consistent with Figure 4.10, except that now LPP does better than ALL under int and LG better than ALL under run. What remains

---

8 As an interesting side note, in some cases we see better performance on NFS than on a local drive, because of fast network as well as better hardware and performance enhancements in NAS.
unchanged is that LPP has a very clear advantage over the others. Figure 4.14 plots the running time on the local hard drive. In terms of I/O time, ALL is similar to LG, both much better than LPP again due to LPP’s excessive random I/Os. All policies have much higher CPU cost compared to Figure 4.11, because of the cost of maintaining order among requests in the now much larger buffer. Overall, ALL (and also LG) is the winner because of its sequential I/O pattern despite its larger number of I/Os. We do not have the complete results for the larger dataset on NFS, because of the amount of time they take to finish; it would be interesting to see if they agree with the results for the smaller dataset.

4.6 Conclusion

We have presented LAB-tree as a solution for storing arrays on disk to support scalable analysis. It uses linearization to provide flexible array layouts, and has a dynamic leaf format that adapts to varying sparsity across space and time. Experiments on common workloads and real data confirm its advantage over B-tree and directly addressable files. We have also called into question the standard B-tree strategy for splitting overflowing leaves and the common flush-all policy for update batching. Based on our theoretical analysis and empirical evaluation, we conclude with some recommendations. 1) We believe split-aligned should replace split-in-middle as the choice of splitting strategy for array data, because of its good theoretical properties and practical performance. 2) For update batching, when the difference between random and sequential writes is obscured (e.g., log-structured file system) or nonexistent (e.g., phase-change memory), we recommend flush-largest-page-probabilistically, with fewer I/Os and low CPU overhead. On conventional hard drives, however, the best bet remains flush-all, for which we also prove a reasonable competitive ratio.
Figure 4.13: I/O counts of flushing policies, with larger matrix.

Figure 4.14: Flushing policies on local drive, with larger matrix.
In data-intensive statistical computing, large arrays are often accessed many times by the same operator or multiple operators. This chapter describes the optimization of I/O sharing in RIOT.

5.1 Introduction

As mentioned in Chapter 1, it is often impossible or uneconomical to fit data in today’s analytical problems entirely in memory, even after partitioning and distribution in a parallel or cluster setting. Besides the big input data, analysis can write out big intermediate and/or final results. Thus, big array analytics today are I/O-intensive, and I/O optimization is critical in achieving high overall performance. Furthermore, data analysis has become more sophisticated—it may use linear algebra instead of relational operations as building blocks, and each step may exhibit a complex, multi-pass access pattern over its input and output. Optimizing I/O in this setting is challenging.

Example 7. Consider a program with two steps, a matrix addition and a matrix
multiplication: $C = A + B$, $E = CD$. Suppose the matrices are stored on disk in blocks. The blocks here are a logical storage and access unit, usually large in size and not to be confused with physical disk blocks. The following C-style program describes the operations involved. In the following (and the rest of this chapter), each array access (such as $C[i,j]$ below) represents a block access, not an element access.

```c
for (i=0; i<n1; ++i)
    for (k=0; k<n2; ++k)
        C[i,k] = A[i,k] + B[i,k]; // s1
for (i=0; i<n1; ++i)
    for (j=0; j<n3; ++j)
        for (k=0; k<n2; ++k)
            E[i,j] += C[i,k] * D[k,j]; // s2
```

There are two statements in the program: $s_1$ and $s_2$. If we regard each array access as an I/O, $A$ and $B$ are both read once, $C$ is written once and then read $n_3$ times, $D$ is read $n_1$ times, and $E$ is written $n_2$ times and read $n_2 \cdot n_1$ times.\(^1\) However, it is not hard to find some I/O-saving opportunities:

1. $E[i,j]$ in $s_2$ can be kept in memory until the innermost loop is done, and written once.

2. $C[i,k]$ in $s_2$ can be read once and kept in memory for the innermost loop, if we make $j$ the innermost loop.

3. If 2 is done, the two loop nests can be merged and $C$ in $s_1$ does not have to be written to disk at all.

4. $D[k,j]$ in $s_2$ can be read once and kept in memory for the innermost loop, if we make $i$ the innermost loop.

\(^1\) The listed code is simplified. Statement $s_2$ should actually be:

```c
if (k==0) E[i,j] = C[i,k] * D[k,j];
else E[i,j] += C[i,k] * D[k,j];
```
The above example illustrates an important optimization idea—*I/O sharing*. When data is accessed repeatedly within the same processing step or by multiple steps in a data-intensive application, sharing I/O—i.e., retaining data in memory to avoid subsequent I/O—can reduce the overall running time. Although the example is simple in nature, it already reflects some intricacies of the I/O sharing problem, as listed below.

**Memory requirement** Each I/O sharing opportunity in Example 7 results in different amount of I/O savings and requires different amount of memory in order to keep certain array blocks in memory. The opportunity that results in the most I/O savings may require more memory than available. It is necessary to analyze both factors.

**Legality** Some opportunities (2, 3 and 4) change the original execution order of statement instances, which may or may not preserve the semantics of the program.

**Incompatibility of I/O sharing opportunities** Some opportunities conflict and cannot be applied together. For example, Opportunity 1 requires \( k \) to be the innermost loop for \( s_2 \), Opportunity 2 and 3 require \( j \) to be the innermost loop, and Opportunity 4 requires \( i \) to be the innermost loop.

**Dependence on parameters** The optimal solution depends on not only the operators involved, but also the input parameters, namely sizes of arrays and their blocks. In Example 7, in the special case \( n_3 = 1 \), \( s_2 \) is surrounded by essentially only two loops. In this case, Opportunity 1 does not contradict Opportunities 2 and 3 any more; they can all be realized by the transformed program in Figure 5.1(a). While this special-case solution is easy for a human to produce, the general case of \( n_3 \geq 1 \) is not. If \( E[i, j] \) is pinned in memory for continuous self accumulation (Opportunity 1), then it is impossible to avoid writing \( C \) (Opportunities 2 and 3) unless \( n_3 = 1 \). However,
it is still possible to “merge” the two loop nests and save a single pass of reading \( C \), as shown in Figure 5.1(b). This solution subsumes the one in Figure 5.1(a). For a human to devise such a solution is nontrivial and error-prone; we would rather achieve this optimization automatically. As we will show in this chapter, our optimizer can indeed find a parameterized solution for the most general case automatically.

for (i=0; i<n1; ++i) {
    // init E[i,0] with 0 in memory
    for (k=0; k<n2; ++k) {
        // read A[i,k] and B[i,k]
        C[i,k] = A[i,k] + B[i,k];  // s1
        // pipeline C[i,k] from s1 to s2
        // read D[k,0]
        E[i,0] += C[i,k] * D[k,0];  // s2
    }
    // write E[i,0]
}
(a) Special case of \( n_3 = 1 \).

for (i=0; i<n1; ++i) {
    for (j=0; j<n3; ++j) {
        // init E[i,j] with 0 in memory
        for (k=0; k<n2; ++k) {
            if (j == 0) {
                // read A[i,k] and B[i,k]
                C[i,k] = A[i,k] + B[i,k];  // s1
                // write C[i,k]
            }
            // read D[k,j]
            // pipeline C[i,k] if j==0
            // read C[i,k] if j>0
            E[i,j] += C[i,k] * D[k,j];  // s2
        }
        // write E[i,j]
    }
}
(b) General case of \( n_3 \geq 1 \).

**Figure 5.1**: Transformed code for Example 7.

Existing database and compiler techniques fall short of solving the I/O sharing...
problem in our setting. First, a database-like, operator-based approach does not allow full-fledged inter-operator optimization. With this approach, users can write their programs in terms of logical operators such as matrix addition and multiplication. The system can provide for each logical operator a variety of physical implementations, each corresponding to a particular way of structuring the loops that implement the operator. This approach does allow for some I/O sharing opportunities such as pipelining between operators. Although this approach has enjoyed tremendous success in relational data processing, it is not suited for the array- and loop-centric applications that we consider, because the operators in our case have a far wider variety of implementation alternatives with complex data access patterns governed by many parameters. When putting our operators together for co-optimization, they cannot be treated as black boxes but need to be “opened up” so that the optimizer can tweak their inner workings further.\(^2\) Otherwise, even a program as simple as in Example 7 cannot be handled. For instance, a database-like approach may be able to find a pipelining opportunity for \(C\) if \(n_3 = 1\), but will not be able to exploit “partial” pipelining as in Figure 5.1(b) if \(n_3 > 1\).

Second, traditional compiler techniques cannot solve our I/O sharing problem because they lack explicit control over data reuse. The compiler community has developed a plethora of techniques for automatic optimization of data locality [WL91, CMT94, BF03, BHRS08]. Their traditional focus is minimizing the traffic between CPU cache and memory, while ours is minimizing disk I/O. Although similar at a first glance, the two problems are fundamentally different. Traffic between cache and memory is hardware-managed and has peculiarities such as cache associativity; therefore, optimization tends to be best-effort, and does not produce a program that

\(^2\) One might wonder if the need to “open up” operators can be avoided by making them more fine-grained. However, a complex operation often cannot be represented simply by a tree of fine-grained operators; instead, loop constructs would be required, which traditional database optimization does not handle. Indeed, our approach offers ways to reason with loops.
controls data sharing precisely. Traffic between memory and disk, on the other hand, is completely under our control, making precise control and analysis possible for our approach. Nonetheless, we have found the polyhedral model, which has been applied in a number of compiler optimizations [KMW67, Fea91, Fea92a, Fea92b, GR07], to be a viable foundation to build on because it admits higher-level program analysis.

**Contributions** In this chapter, we present RIOTShare, for optimizing I/O of loop-centric data-intensive programs. Building on the polyhedral model, we develop a new framework for capturing the I/O patterns of a program that is high-level enough to allow automatic extraction and reasoning of the I/O patterns, yet not too high-level to impede optimization flexibility (as black-box operators do). With this framework, we develop an optimizer that considers a rich space of plans (transformed programs), and is able to accurately determine their legality, I/O costs, and memory requirements. The optimizer employs Apriori-like search algorithm to enumerate different combinations of sharing opportunities and to look for legal, I/O-efficient plans under memory constraints. We demonstrate the effectiveness and accuracy of our optimizer through experiments.

5.2 Related Work

Database systems rely on the buffer pool mechanism for sharing common I/O. This approach is rather low-level, opportunistic, and extremely sensitive to timing and the replacement policy used. There has also been much work on proactive work sharing. QPipe [HSA05] proposes an on-demand simultaneous pipelining paradigm for maximizing data and work sharing across concurrent queries. It detects overlapping scans at run time and exploit the sharing opportunities using circular scans. Cooperative scans [ZHN07] is based on a similar idea, but coordinates I/O sharing using an active buffer manager and a policy called relevance, which is shown to
be more effective than circular scans. Both approaches fall under the category of execution-time optimization, which is different from the principled, systematic optimization developed in this dissertation. Multi-query optimization [Sel88, RSSB00] tries to match common subqueries so that query processing can be partially shared. The recent DataPath system [ADJ+10] relaxes the condition of sharing by employing a data-centric, push-based approach.

The aforementioned database-like, operator-based approaches have limited applicability in statistical and scientific data analysis workloads for two reasons. First, analytical operations typically have much more complex, parameter-governed data access patterns than most sequential-scan database operators. To support optimization of these complex I/O patterns across operators, operators need to be “opened up” so that the optimizer can reason about I/O sharing. Second, support for user-defined operators implementing customized analytical algorithms is a must. The system can no longer base optimization solely on some built-in knowledge of a static list of (physical) operators. This optimizable extensibility requirement again calls for a representation upon which both user-defined and built-in operators can be reasoned.

The compiler community has been working on automatic locality (data reuse) optimization of programs for decades. Most of the efforts have been devoted to locality at the cache level; examples include [WL91, CMT94, BF03]. Since the cache is hardware-managed and its behavior depends on the machine’s runtime state, the optimizer does not have explicit control over the data reuse at that level. Cache associativity further complicates the problem and only admits heuristic, instead of exact, solutions. Our problem is to optimize locality at the memory level and the key difference is that memory is explicitly managed by software, allowing us to develop a precise optimizer.

Tiling [BF03, BHRS08], also called blocking or chunking, is a common technique to increase data locality. We solve a different problem in this chapter: we assume
tiling is already done and try to share the I/O of tiles by restructuring the data access patterns. The coarse-grained tiling optimizer of PLuTo [BHRS08] aims to increase parallelism and locality simultaneously, by minimizing the maximum of all reuse distances in the input program. In contrast, we directly optimize the total amount of data reuse, because the memory-level (as opposed to cache-level) data reuse can be precisely characterized.

Some compiler optimization ideas have been successfully applied in database systems. For example, MonetDB/X100 [BZN05] adopts vector processing in place of the traditional tuple-at-a-time paradigm to achieve high CPU efficiency. HIQUE [KVC10] uses a set of highly efficient code templates to customize code generation during query evaluation. It abandons the CPU-unfriendly iterator model and takes advantage of existing compiler optimizations to achieve high in-memory execution efficiency. Our work also has close ties to the compiler field, but we attack a different problem at a higher level.

Our optimizer represents and reasons about I/O sharing opportunities using the polyhedral model. The polyhedral model dates back to the seminal work of Karp, Miller and Winograd on uniform recurrence equations [KMW67]. Because of its power of algebraic abstraction and transformation expressiveness, it has gain traction in the compiler field on some important optimization problems [GVB+06, BHRS08]. However, to the best of our knowledge, this is the first time that it is applied to the I/O sharing problem.

5.3 Overview

Figure 5.2 shows the architecture of RIOTShare. The input to the system is a representation of an input program whose I/O we want to optimize. The representation is based on the polyhedral model, further discussed in Section 5.4.1, and can capture loop nests with conditional statements. We require the unit of I/O to be logical
blocks, which is a standard practice to increase locality and reduce I/O overhead. Since we focus on optimizing I/O, we care only about read and write accesses to these blocks; the actual in-memory computation on them is unimportant. To obtain the representation for the input program in the polyhedral model, we can start with the program written using a library of high-level operators (such as matrix addition and multiplication), where the polyhedral representations of their implementations are already provided and can be assembled into a presentation for the entire program. Alternatively, we can obtain this representation by analyzing user-supplied pseudo-code (such as in Example 7) or source code for a user-defined operator or program, using standard code analysis tools like Clan.³ The details of this preprocessing step are beyond the scope of this dissertation.

The next step is to identify data dependences as well as individual I/O sharing opportunities. Basically, a sharing opportunity signifies a data reuse relationship between two statements in the program. Note that the two statements can be the same one, in which case a self sharing opportunity occurs. We capture both dependences and sharing opportunities precisely, down to the instance level, i.e., individual accesses to the same block (as opposed to statements operating on the same array). In Section 5.4.3, we show how to express dependences and sharing opportunities concisely in polyhedral forms, which avoid costly enumeration of individual accesses and enable optimization.

Given the dependences and sharing opportunities, our optimizer explores the space of plans (or schedules of data accesses) to find the I/O-optimal plan that is legal (i.e., respects all dependences) and meets the memory requirement. Intuitively, dependences and sharing opportunities translate to constraints on plans. The optimizer considers combinations of sharing opportunities using a strategy similar to the Apriori algorithm [AS+94], to prune infeasible combinations of sharing opportunities.

³ http://www.cse.ohio-state.edu/~pouchet/software/pocc
Legal plans are fed into a costing module, which evaluates their memory requirements and I/O costs. Finally, the best plan given the current available memory resource is chosen and further converted into an executable plan. Section 5.5 discusses how the optimizer searches for and costs plans.

Finally, Section 5.6 demonstrates through experiments the effectiveness of our I/O sharing optimization framework. Section 5.7 concludes the chapter.

5.4 A Polyhedral I/O Optimization Framework

We first introduce some well-established concepts in the polyhedral model (Section 5.4.1), which serves as the foundation of our optimization framework. Next, in we define the I/O sharing optimization problem (Section 5.4.2) and show how to characterize data dependences and I/O sharing opportunities in our framework (Section 5.4.3).
5.4.1 The Polyhedral Model

**Static-Control Programs** In this dissertation we focus on data-intensive programs that make “regular” accesses of out-of-core data. In particular, we assume the I/O patterns of the program can be described by a set of static-control loop nests and if conditionals, where the loop bounds, conditionals, and array access functions are affine combinations (linear combination plus a constant) of the enclosing loop variables and global parameters (e.g., array sizes). Note that this encompasses a large body of scientific and analytical programs, such as matrix addition, multiplication and factoriation, linear regression, table scans and nested loop joins in traditional databases, FILTER and FOREACH commands in Pig, etc. More general data flow programs, such as those with data-dependent control and non-affine conditionals, can also be cast into a static-control form by techniques like safe over-approximation [BPCB10].

**Iteration Domains** A program consists of a set $\mathcal{S}$ of statements. Each statement $s \in \mathcal{S}$ has an iteration domain, denoted $\mathbb{D}_s$, which describes the set of all executed instances of this statement. Each instance of $s$ is identified by the values of the loop variables surrounding $s$. In Example 7, $(i = 0, k = 0)$ is an instance of $s_1$, which is contained in its iteration domain $\mathbb{D}_{s_1} = \{(i, k) \in \mathbb{Z}^2 \mid 0 \leq i < n_1, 0 \leq k < n_2\}$. If $s$ is enclosed by a loop nest of depth $d_s$, then $\mathbb{D}_s$ is a parametric integer polyhedron [Fea88] which is a subset of $\mathbb{Z}^{d_s}$ and contains the statement instances as integer points. $\mathbb{D}_{s_1}$ is parameterized by $n_1$, $n_2$ and $n_3$. Geometrically, a polyhedron is a union of convex polyhera, each of which is the intersection of finitely many half-spaces and can be described by a system of linear inequalities. For example, $\mathbb{D}_{s_1}$ above can be written as a system of linear inequalities: $(i \geq 0) \land (-i + n_1 - 1 \geq 0) \land (k \geq 0) \land (-k + n_2 - 1 \geq 0)$,
or equivalently in matrix form as:

\[
\Delta_s \vec{x}_s = \Delta_s \begin{bmatrix} \vec{l}_s \\ \vec{p} \\ 1 \end{bmatrix} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 & -1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & -1
\end{pmatrix} \begin{pmatrix}
i \\
k \\
n_1 \\
n_2 \\
n_3 \\
1
\end{pmatrix} \geq \vec{0}.
\]

We call \( \vec{l}_s \) 's loop iteration vector, \( \vec{x}_s \) the extended iteration vector, and \( \vec{p} \) the parameter vector. Different statements in a program can have (partly or completely) different iteration domains. For the brevity of presentation, we may drop the parameter vector and refer to the extended iteration vector simply as the iteration vector.

**Array Accesses** A statement \( s \) can access multiple arrays. Each access is defined as a tuple \( a = \langle s, t, A, \Phi \rangle \), where \( s \) is the statement performing the access, \( t \in \{R, W\} \) the type of access (read or write), \( A \) the array accessed, and \( \Phi \) a matrix describing the affine access function which maps \( \vec{x}_s \) (the extended iteration vector of \( s \)) to a subscript in \( A \). Each point in \( A \)'s subscript space corresponds to a block of array elements. \( \Phi \) has as many rows as \( A \)'s dimensionality and as many columns as \( \vec{x}_s \)'s dimensionality. Note that \( \Phi \) is required to uniquely identify an access because \( s \) may access multiple parts of \( A \). For example, there are three accesses in a statement \( s \)

\[
\]

\[
\langle s, W, A, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \rangle, \langle s, R, A, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \rangle, \langle s, R, A, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} \rangle.
\]

\( A[i,j] \) and \( A[i,j+1] \) are regarded as different read accesses because they access different parts of \( A \); the write (assignment) of \( A[i,j] \) is regarded as a different access from the reads of \( A[i,j] \) because of different access types. However, note the two reads of \( A[i,j] \) are treated as one access and have the same tuple representation, because they can always be serviced with only one I/O. In this dissertation, we assume
each statement can have only one write access, which holds in most programming languages.

**Schedules** Each program has a schedule, which maps all statement instances in the program to an execution time. Formally, we define a *statement schedule* for statement $s$ to be an affine function (or matrix) $\Theta_s$ mapping $\mathbb{D}_s$, the iteration domain of $s$, to a *multidimensional time domain*, and a *program schedule* to be the set of all statement schedules in the program $\Theta = \{\Theta_s \mid s \in S\}$. The time domain is a totally ordered set of vectors, where the order is lexicographic (the vector components can be thought of as, for example, year, month, day, etc.): $(x_1, \ldots, x_m) < (y_1, \ldots, y_m) \equiv \exists r \in [1, m], (\forall i \in [1, r-1], x_i = y_i) \land (x_r < y_r)$. For the code in Example 7, one possible program schedule is (loop variables of $s_2$ are renamed to avoid confusion):

$$\Theta_{s_1} \vec{x}_{s_1} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ k \\ n_1 \\ n_2 \\ n_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 1 \end{pmatrix},$$

$$\Theta_{s_2} \vec{x}_{s_2} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i' \\ j' \\ k' \\ n_1 \\ n_2 \\ n_3 \end{pmatrix} = \begin{pmatrix} 1 \\ i' \\ j' \end{pmatrix}.$$  

Because of the 0 and 1 in the first component of the result time vectors, all instances of $s_1$ are scheduled before those of $s_2$. Also, $i$ appearing before $k$ in $\Theta_{s_1}$ corresponds to the fact that $i$ is an outer loop than $k$. Note that there are many equivalent schedules for the same program as far as the execution order is concerned. For example, changing $\Theta_{s_2} \vec{x}_{s_2}$ to $(2, i' + 1, j', k')$ does not change the relative execution
order of statement instances in the program. Our optimizer works no matter which one of the equivalent schedules is specified for the input program.

5.4.2 Problem Definition

With the above preliminaries, the problem we tackle is the following. Given a memory cap and a static-control input program, whose iteration domains, array references, and original schedule are specified under the polyhedral model, find a legal transformation (represented by a schedule) of the given program such that I/O sharing is maximized (i.e., total I/O cost minimized) and memory consumption does not exceed the cap. A legal program schedule is one under which all data dependences in the original program schedule are observed; we formalize the notion of dependences in Section 5.4.3. We discuss how to compute I/O cost and memory consumption for a program schedule in Section 5.5.4.

Why Explicitly Capping Memory? We impose an explicit memory cap instead of relaxing the restraint and relying on the virtual memory mechanism. As verified in Chapter 3, the virtual memory mechanism fails to utilize application-level memory usage information to optimally orchestrate contending consumers, and as a result, may lead to excessive paging for the types of programs we consider. Thus, we choose to impose a memory cap and control memory data reuse explicitly.

Schedule Search Space Recall that a (program) schedule is a set of affine functions, one for each statement, which maps the iteration instances to their scheduled execution time. It has been shown [Fea92b] that we can always find a schedule with dimension \( \bar{d} + 1 \) for any static-control program, where \( \bar{d} = \max_{s \in S} d_s \) and \( d_s \) is the depth of the loop nest enclosing \( s \) in the original program. Thus, we can safely restrict our search to \( (\bar{d} + 1) \)-dimensional schedules only. Statement \( s \)'s schedule is then a list of \( (\bar{d} + 1) \) 1-d affine functions: \( \Theta_s = (\theta_s^1, \ldots, \theta_s^{\bar{d}+1}) \), where each function corresponds to a row in
the matrix and maps the iteration vector to a scalar time component.

As shown in [Fea92b], it is possible to make the last schedule dimension a constant, i.e., $\theta^{d+1}_s = c_s$, for all $s \in S$, where $c_s$ denotes the textual position of $s$ in the transformed program under the schedule. For example, $\Theta_{s_1} \vec{x}_{s_1} = (k, 0, 1), \Theta_{s_2} \vec{x}_{s_2} = (i + n, j, 1), \Theta_{s_3} \vec{x}_{s_3} = (i + n, j, 2)$ is a schedule describing the program below:

```c
for (k=0; k<n; ++k)
  // s1
  for (i=0; i<n; ++i)
    for (j=0; j<n; ++j) {
      // s2 and s3
    }
```

All instances of $s_1$ are scheduled before those of $s_2$ and $s_3$ due to the first schedule dimension. The order of the two instances of $s_2$ and $s_3$ within the same loop iteration are determined by the last constant dimension, which specifies the textual order of the two statements.

**Why the Polyhedral Model?** We choose the polyhedral model as the “language” for solving the I/O sharing problem for three reasons. First, it is well known that the polyhedral model captures a large space of transformations, such as loop interchange, reverse, skew, fusion, etc., and their compositions [Fea92a, GVB+06, BHRS08]. It can also handle programs with more general code than static-control loops [BPCB10].

Second, analysis of data flow in the polyhedral model is at the level of statement *instances* as opposed to just statements. This level of detail makes it possible to build a precise cost-based optimizer that captures individual block accesses. For example, our optimizer is able to identify the “partial” sharing opportunity in Figure 5.1(b).

Third, the polyhedral model abstracts program transformations to make them amenable to automatic and systematic search. This is in direct contrast with traditional *syntactic* analysis, where programs are represented in abstract syntax trees and go through a series of pattern-matching and transformation steps, which does
not lend itself to structured search [GVB+06].

5.4.3 Dependences and I/O Sharing Opportunities

Building on the polyhedral model, we show how to represent data dependences and I/O sharing opportunities, which are essential in determining the legality, I/O cost, and memory requirement of a program transformation (schedule). We begin with the notions of co-accesses and their extent polyhedra.

**Definition 3** (Co-Access and Extent Polyhedron). Let $A$ denote the set of all array block accesses in the program. A co-access, denoted $a \rightarrow a'$, is a pair of accesses in $A \times A$ to the same array; i.e., $a.A = a'.A$. The type of co-access $a \rightarrow a'$ is $a.t \rightarrow a'.t$, which is one of $R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, and $W \rightarrow W$.

Suppose the original program schedule is $\Theta = \{\Theta_s | s \in \mathcal{S}\}$. The (extent) polyhedron of co-access $a \rightarrow a'$, where $a = \langle s, t, A, \Phi \rangle$ and $a' = \langle s', t', A, \Phi' \rangle$, is $P(a \rightarrow a') = \{(x, x') | x \in D_s, x' \in D_{s'}, \Phi x = \Phi' x', \Theta_s x < \Theta_{s'} x'\}$.

Thus, the extent polyhedron of a co-access is a polyhedron in the product space of the iteration domains of the two statements involved. Intuitively, it contains all pairs of statement instances that access the same array block ($\Phi x = \Phi' x'$), where the source instance executes before the target instance in the original schedule ($\Theta_s x < \Theta_{s'} x'$).

In the following, when no ambiguity exists (specifically, if statements $s$ and $s'$ each make only one access of a given type to the array $A$), we may omit $\Phi$ and $\Phi'$ and denote a co-access by $stA \rightarrow s't'A$.

Using the notion of co-accesses, we can now define data dependences and I/O sharing opportunities.

**Definition 4** (Dependence). A (data) dependence is a co-access $a \rightarrow a'$ with type $R \rightarrow W$, $W \rightarrow R$, or $W \rightarrow W$ (i.e., at least one access is a write) and $P(a \rightarrow a') \neq \emptyset$. Let $D$ denote the set of all dependences in the original program.
Intuitively, the polyhedron for a dependence specifies all data dependences among individual array block accesses. Given a dependence, for any pair of statement instances \((\vec{x}, \vec{x}')\) in its polyhedron, \(\vec{x}\) must execute before \(\vec{x}'\) under any legal schedule. Note that \(R \rightarrow R\) co-accesses are not dependences because exchanging the order of two reads by itself does not affect program semantics.

I/O sharing opportunities are also defined using co-accesses, but they are different from dependences in subtle yet important ways.

**Definition 5** (Sharing Opportunity). *An I/O sharing opportunity is a co-access \(a \rightarrow a'\) with type \(W \rightarrow R, W \rightarrow W\), or \(R \rightarrow R\) and \(P(a \rightarrow a') \neq \emptyset\). Let \(O\) denote the set of all sharing opportunities in the original program.*

Intuitively, the polyhedron for a sharing opportunity identifies all possibilities for sharing I/O among individual array block accesses. Given a sharing opportunity, for any pair of statement instances \((\vec{x}, \vec{x}')\) in its polyhedron, \(\vec{x}\) and \(\vec{x}'\) may share I/O in accessing the same array block (at \(\Phi \vec{x} = \Phi' \vec{x}'\)). Note that a sharing opportunity merely indicates the potential for sharing but does not guarantee it; whether the potential is realized depends on the final schedule chosen. Specifically, depending on the type of the opportunity, sharing may happen in the following ways (see Section 5.5.2 for details on how they are considered by the optimizer):

- **\(R \rightarrow R\)**: A read followed by a read. The I/O for the second read may be eliminated using the in-memory copy of the shared block used to serve the first read (assuming there is no write of the same block in between). Alternatively, a new schedule may be able to reorder the reads, such that the first read in the original schedule becomes the second in the new schedule and saves its I/O. In either case, to realize the sharing opportunity, the shared block has to be kept in memory until the reuse.
• $W \rightarrow R$: A write followed by a read. The I/O for the read may be eliminated using the in-memory copy of the shared block used to serve the write (assuming here is no other write of the same block in between). To realize this opportunity, the shared block has to be kept in memory until the read. Unlike the case of $R \rightarrow R$, the new schedule cannot reorder these accesses because it would violate the $W \rightarrow R$ dependence.

• $W \rightarrow W$: A write followed by a write. The first write may be eliminated since it will be overwritten by the second one (assuming there is no other write of the same block in between). The shared data need not be kept in memory. Like the case of $W \rightarrow R$ but unlike $R \rightarrow R$, the new schedule cannot reorder these accesses because it would violate the $W \rightarrow W$ dependence.

Note that a $R \rightarrow W$ co-access does not make a sharing opportunity because neither the read nor the write can be saved.\(^4\)

The resemblance between Definitions 4 and 5 is not a coincidence: two accesses to the same data may impose an execution order on the two statement instances and thus induce a dependence (if either access is a write), or may represent an opportunity for reducing I/O (if the co-access is not $R \rightarrow W$). However, their differences should also be clear. First, dependences capture the ordering constraints that must be preserved for any transformation, whereas sharing opportunities capture data reuse relationships that may potentially lead to I/O savings. Second, because of their distinct purposes, they stem from different subsets of co-access types: $R \rightarrow W$ can be a dependence but not a sharing opportunity, whereas $R \rightarrow R$ can be a sharing opportunity but not a dependence.

\(^4\) Here we assume that a write operation does not include first reading that data. If a statement performs a read-modify-write, the read and the write are modeled as two separate accesses. This assumption still holds in the presence of disk blocks, because the unit of I/O is a logical array block as opposed to an individual array element.
It is important to point out that the extent polyhedron of a co-access characterizes fine-grained, \textit{instance-level} relationships, not a coarse-grained, \textit{statement-level} relationship. Nonetheless, the polyhedron can be succinctly represented in an algebraic form (system of inequalities), instead of literal enumerations of integer points in the polyhedron. For example, in Example 7, $s_1 WC \rightarrow s_2 RC$ is both a dependence and a sharing opportunity, and $\mathbb{P}(s_1 WC \rightarrow s_2 RC) = \{(i, k, i', j', k') \mid i = i', k = k', 0 \leq i, i' < n_1, 0 \leq k, k' < n_2, 0 \leq j < n_3\}$. On the other hand, $s_2 RC \rightarrow s_1 WC$ is neither, because $\mathbb{P}(s_2 RC \rightarrow s_1 WC) = \emptyset$ (no instance of $s_2$ executes before any instance of $s_1$ in the original program).

It is also worth noting that the arrow in $stA \rightarrow s't'A$ does not necessarily imply $s$ should textually precede $s'$ in the original program. As a more dramatic example, consider the code below:

```c
for (i=0; i<n; ++i) {
    A[i] = B[i];  // s1
    C[i] = A[n-1-i]; // s2
}
```

Two dependences (and sharing opportunities) with opposite directions exist at the same time, with polyhedra $\mathbb{P}(s_1 WA \rightarrow s_2 RA) = \{(i, i') \mid i + i' = n - 1, 0 \leq i \leq (n-1)/2\}$ and $\mathbb{P}(s_2 RA \rightarrow s_1 WA) = \{(i', i) \mid i' + i = n - 1, 0 \leq i' \leq (n-2)/2\}$.

5.5 The Optimizer

This section presents the design and implementation of our I/O sharing optimizer. At a high level, the optimizer translates individual data dependences and sharing opportunities, after necessary preprocessing, to constraints on schedules, which represent possible program transformations. The optimizer then uses an Apriori-like algorithm to efficiently enumerate feasible combinations of sharing opportunities while satisfying all dependences. Each feasible combination leads to a legal plan, which is then evaluated in terms of its memory requirement and total I/O cost. Finally,
given the amount of memory available, the plan with the least I/O cost is chosen and converted into code for compilation and execution.

### 5.5.1 Preprocessing and Pruning

Before passing the sets of dependences and sharing opportunities on to the rest of the optimizer, we preprocess them by pruning out possibilities that either can be safely ignored by optimization, or need to be ignored to make optimization tractable. In this section, we describe two pruning techniques in these respective categories, and then discuss how extraction and preprocessing of dependences and sharing opportunities are implemented. Both pruning techniques stem from our concept of linear sharing model, which we first introduce below.

With any program schedule, every statement instance is executed at a specific time, which defines a linear ordering of all statement instances. I/O sharing effectively only happens between consecutive accesses to the same data in time order. To understand this statement, imagine that when a statement instance touches a shared piece of data, it becomes the owner of the data. A subsequent reuse of the data is always “charged” to the owner, and the new user becomes the new owner. We call this the linear sharing model. By considering sharing only between consecutive accesses, we avoid the problem of over-counting reuses. For example, consider three consecutive reads to the same data. There are only two pairs of consecutive accesses in time order, corresponding to two reuses. Including the non-consecutive accesses (the first and the third) would give us three, which is too much.

**No Write in Between** In light of the linear sharing model, the “no-write-in-between” rule states that, given a sharing opportunity \( a \rightarrow a' \), any pair of statement instances \((\vec{x}, \vec{x}') \in P(a \rightarrow a')\) can be removed from the polyhedron if there is a write to the same array block that executes between \( \vec{x} \) and \( \vec{x}' \) in the original program. This rule makes sense because, to preserve program semantics, no legal schedule can move the
write before \(\vec{x}\) or after \(\vec{x}'\); hence, in no legal schedule will \(\vec{x}\) and \(\vec{x}'\) ever be consecutive accesses.

The no-write-in-between rule also applies to dependences: given a dependence \(a \rightarrow a'\), any pair of statement instances \((\vec{x}, \vec{x}') \in P(a \rightarrow a')\) can be removed from the polyhedron if there is a write to the same array block by some statement instance \(\vec{y}\) that executes between \(\vec{x}\) and \(\vec{x}'\) in the original program. The rule is applicable in this setting because the ordering constraint between \(\vec{x}\) and \(\vec{x}'\) would be redundant, as it is implied by the constraints between \(\vec{x}\) and \(\vec{y}\), and between \(\vec{y}\) and \(\vec{x}'\).

**Multiplicity Reduction** We define the *multiplicity* of a sharing opportunity as follows. A sharing opportunity is *many-one* if each source instance is related to at most one target instance in the extent polyhedron, *one-many* if each target instance is related to at most one source instance, *one-one* if both, or *many-many* if neither. For a sharing opportunity that is not one-one, there exists an instance \(\vec{x}\) related to multiple other instances. However, by the linear sharing model, only one of these instances can possibly form a real sharing relationship with \(\vec{x}\). Ideally, the optimizer should explore all possibilities when realizing the sharing opportunity; however, doing so is impractical because it would blow up the search space. As a practical alternative, we perform a multiplicity reduction step to make all sharing opportunities one-one. Care is taken to minimize the impact on optimality. For details and potential intricacies, see the paragraphs that follow. Experiment results in Section 5.6 confirm that such reduction does not miss interesting solutions. Note that multiplicity reduction is not applied to dependences, because a legal schedule must preserve the execution order of all statement instances in a dependence’s polyhedron.

*Intricacies in Multiplicity Reduction* First note that a “many” side can only be a read access, because if a statement instance \(\vec{x}\) is related to multiple instances with a write, only the instance closest to \(\vec{x}\) in execution time forms a real sharing opportunity.
with \( \vec{x} \) due to the no write in between rule, in which case the “many” side is not really a “many” side.

The algorithm we use works as follows. If a sharing opportunity is one-many or many-one, we reduce the multiplicity of the “many” side to “one”. Specifically, we keep for any instance on the “one” side only the instance closest to it in execution time on the “many” side. Such reduction does not reduce the amount of I/O savings for this sharing opportunity, because each instance from the original “one” side can share I/O with at most one instance from the “many” side anyway, by the linear sharing model.

For many-many sharing opportunities, we first reduce them to many-one and then apply the reduction described above. The reduction from many-many to many-one cannot use the same idea as above, however; the potential problem is illustrated in Figure 5.3(a). If we keep for each source instance the target instance closest to it in execution time, many target instances may be ignored and thus reduce the amount of potential I/O savings. We solve this problem by ensuring the rank, or degree of freedom in the iteration variables, of both sides after the reduction do not decrease below the minimum of the original ranks of both sides. Continuing with the above example, suppose originally both sides are depth-1 loops: \texttt{for (i=0; i<2; ++i)}, both with rank 1 because \( i \) is a free variable. After the first reduction in Figure 5.3(a), the target side has constraint \( i=0 \), which means the rank decreases to 0 and renders the reduction invalid. Our algorithm carefully adds rank-preserving equality constraints and would produce a result as shown in Figure 5.3(b).

Note that the multiplicity reduction problem can be cast into a maximum bipartite matching problem and solved in \( O(|V||E|) \) time. However, in our case, \( |V| \) corresponds to the size of the iteration domain, which is symbolic and usually takes a big value, making regular algorithms inapplicable. Our algorithm, in contrast, works in \( O(d_i d_j) \) time, where \( d_i \) and \( d_j \), source and target loop nest depths, are very small constants.
Although our multiplicity reduction algorithm does not reduce the amount of I/O sharing for each sharing opportunity, it is still possible that the reduced sharing opportunity cannot be satisfied with others at the same time while the original one can. This is expected in order to reduce the exponential search space and make optimization tractable.

![Diagram of multiplicity reduction for many-many](image)

**Figure 5.3**: Intricacy of multiplicity reduction for many-many.

**Extracting and Preprocessing Dependences and Sharing Opportunities**

We use *isl* [Ver10], a library for manipulating integer points in polyhedra, for extracting program dependences. The algorithm used by *isl* was first introduced in [Fea91]. Because of the similarity of dependences and sharing opportunities, we adapt the algorithm to extract sharing opportunities. The library supports removing transitively-covered dependent statement instances, which we use to produce no-write-in-between dependences and sharing opportunities. We then apply our multiplicity reduction algorithm on the sharing opportunities only. Henceforth, all dependences and sharing opportunities are assumed to be no-write-in-between; in addition, all sharing opportunities are assumed to be one-one. With a slight abuse of notation, we still use $P(a \rightarrow a')$ to denote the polyhedron of a dependence or sharing opportunity after the aforementioned preprocessing.

### 5.5.2 Deriving Constraints

There are three types of constraints imposed on a schedule.
Dimensionality Constraints At the very least—not even considering data dependences and sharing opportunities—a legal schedule must map every statement instance in the original program to a unique execution time. This requirement can be satisfied by ensuring 1) instances belonging to the same statement are mapped to different times, and 2) any two instances belonging to different statements are mapped to different times. Below we explain how 1) can be translated into concrete constraints on the schedule. 2) is handled by the optimizer’s search algorithm as it involves schedules of multiple statements and thus needs global coordination.\(^5\)

A schedule \(\Theta_s\) of statement \(s\) is essentially a linear map, \(\Theta_s : D_s \rightarrow \mathbb{Z}^{\tilde{d}+1}\) (see Section 5.4.2 for the definition of \(\tilde{d}\) and \(d_s\) used below). Ensuring all instances in \(D_s\) map to different images means \(\Theta_s\) has to be injective. Thus, the null space of \(\Theta_s\) should have dimension 0, i.e., \(\text{dim null} \Theta_s = 0\). By the rank-nullity theorem in linear algebra, \(\text{dim } D_s = \text{dim null } \Theta_s + \text{rank } \Theta_s\), and the fact that \(\text{dim } D_s = d_s\), we have \(\text{rank } \Theta_s = d_s\). In other words, the matrix representation of \(\Theta_s\) should have exactly \(d_s\) linearly independent rows out of the first \(\tilde{d}\) rows (as the last dimension is a constant which does not contribute to the dimensionality).

The optimizer finds the matrix representation of \(\Theta_s\) in a row-by-row fashion. When choosing each row, we use Algorithm 1 below to enumerate whether the current row should be linearly independent of previously found rows.

Dependence Constraints By definition, a schedule \(\Theta = \{\Theta_s \mid s \in S\}\) is legal only if for any dependence \(a \rightarrow a', \forall (\vec{x}, \vec{x}') \in \mathbb{P}(a \rightarrow a'), \Theta_{a,s} \vec{x} < \Theta_{a',s} \vec{x}'\). This \(<\) condition should not be confused with the one in the definition of extent polyhedron (Definition 3): that condition is in terms of the original schedule of the program, whereas this condition applies to a new schedule. To translate the dependence

\(^5\) The optimizer either assigns different constants for the last schedule dimensions of different statements, or tries to separate instances of different statements at an earlier dimension. Due to the space limit, we omit the technical details.
Algorithm 1: EnumRow(i,j,k)

**Input**: Statement ID $i$, current row index $j$ (1-based), and $k$, number of independent rows before row $j$

**Output**: a list of Booleans indicating whether row $j$ can be linear independent of previous rows

1. if $\tilde{d} - j = d_{s_i} - k$ then
2. return $\{1\}$
3. else
4. return $\{0,1\}$

Constraint into a linear form, we first let

$$X_{s,s',s}^q = (\theta_{s}^0 \bar{x} = \theta_{s}^1 \bar{x}') \land \cdots \land (\theta_{s}^{q-1} \bar{x} = \theta_{s}^q \bar{x}') \land (\theta_{s}^q \bar{x} < \theta_{s}^q \bar{x}')$$

Then, by definition of $<$, we can write the dependence constraint as

$$\forall (\bar{x},\bar{x}') \in \mathbb{P}(a \rightarrow a'), \ X_{a,s,a',s}^1 \lor \cdots \lor X_{a,s,a',s}^{d+1}$$

since $<$ can be satisfied at any depth. Note that the $X$ terms are mutually exclusive; i.e., only one of them can be true. If the $q$-th term is true, we say the dependence is strongly satisfied at depth $q$.

The next question is how to handle the quadratic-form constraints, i.e., vector inner products such as $\theta_{s}^q \bar{x}$. As a concrete example, let us examine the dependence $s_2WE \rightarrow s_2WE$ in Example 7. Its polyhedron is $\mathbb{P} = \{(i,j,k,i',j',k') \mid i' - i = 0, j' - j = 0, k' - k - 1 = 0\}.$

Suppose we want to find the constraint on a schedule dimension $q$ such that $\theta_{s_2}^q \cdot (i,j,k) < \theta_{s_2}^q \cdot (i',j',k')$. If we let $\theta_{s_2}^q = (\alpha, \beta, \gamma)$, the target constraint can be rewritten as $\alpha i + \beta j + \gamma k < \alpha i' + \beta j' + \gamma k'$, or $-\alpha i - \beta j - \gamma k + \alpha i' + \beta j' + \gamma k' - 1 \geq 0$.

Note that this constraint is quadratic and thus does not directly fit in the polyhedral model. Fortunately, the following lemma provides a powerful mechanism to linearize such constraints.

---

6 For the case of presentation, we have omitted the parameter dimensions $n_1, n_2, n_3$ and the constant dimension; they are unimportant for the purpose of this example.
Lemma 1 (Affine Form of the Farkas Lemma [Sch98]). Let $\mathbb{P}$ be a nonempty polyhedron defined by $p$ affine inequalities:

$$\tilde{a}_k \tilde{x} + \tilde{b}_k \geq 0, \ k = 1, \ldots, p. \quad (5.1)$$

Then $\forall \tilde{x} \in \mathbb{P}, \tilde{\theta} \tilde{x} \geq 0$ iff there exist $\lambda_0, \ldots, \lambda_p \geq 0$ such that

$$\tilde{\theta} \tilde{x} = \lambda_0 + \sum_k \lambda_k (\tilde{a}_k \tilde{x} + \tilde{b}_k).$$

Note that constraints of forms other than $\geq 0$ as in (5.1) can be rewritten so that the affine form of the Farkas Lemma applies. For example, $\theta^g_x \tilde{x} < \theta^g_{x'} \tilde{x}'$ can be rewritten as $\theta^g_x \tilde{x} - \theta^g_{x'} \tilde{x}' - 1 \geq 0$; an equality can be split into two inequalities, $\geq 0$ and $\leq 0$.

Continuing the above example, by Lemma 1, $-\alpha i - \beta j - \gamma k + \alpha i' + \beta j' + \gamma k' - 1 \equiv \lambda_0 + \lambda_1 (i' - i) + \lambda_2 (i - i') + \lambda_3 (j' - j) + \lambda_4 (j - j') + \lambda_5 (k' - k - 1)$. Comparing the coefficients of the iteration variables on both sides gives

$$-1 = \lambda_0 - \lambda_5, \ \alpha = \lambda_1 - \lambda_2, \ \beta = \lambda_3 - \lambda_4, \ \gamma = \lambda_5, \ \lambda_0, \ldots, \lambda_5 \geq 0.$$

By eliminating $\lambda_0, \ldots, \lambda_5$, we obtain $\alpha \in \mathbb{Z}, \ \beta \in \mathbb{Z}$, and $\gamma \geq 1$, which indeed preserve the execution order of dependent iterations.

A legal schedule should strongly satisfy each dependence in the program at a certain depth. As we have seen through the above example, the affine form of the Farkas Lemma helps us translate the condition of “strongly satisfying a dependence at a depth $q$” into a polyhedral constraint on the schedule’s coefficients at dimension $q$. The union of such polyhedra for different depths characterizes the space of valid schedules for the statements involved in this particular dependence. The intersection of the results of these unions across all dependences characterizes the space of legal schedules for the entire program. Conceptually, the optimizer uses the dependence constraints to narrow the search space down to legal schedules only; practically, it employs a less expensive, depth-by-depth algorithm as discussed in Section 5.5.3.
Table 5.1: Constraints on statement schedules $\Theta_s$ and $\Theta_{s'}$ that realize a sharing opportunity $a \rightarrow a'$. Here, $P = P(a \rightarrow a')$, $s = a.s$, and $s' = a'.s$.

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Non-Self ($s \neq s'$)</th>
<th>Self ($s = s'$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W \rightarrow R, W \rightarrow W$</td>
<td>$\exists c &gt; 0, \forall (\vec{x}, \vec{x'}) \in P, \Theta_{s'} \vec{x'} - \Theta_s \vec{x} = (0, \ldots, 0, 0, c)$</td>
<td>$\forall (\vec{x}, \vec{x'}) \in P, \Theta_{s'} \vec{x'} - \Theta_s \vec{x} = (0, \ldots, 0, 0, 0)$</td>
</tr>
<tr>
<td>$R \rightarrow R$</td>
<td>$\exists c \neq 0, \forall (\vec{x}, \vec{x'}) \in P, \Theta_{s'} \vec{x'} - \Theta_s \vec{x} = (0, \ldots, 0, 0, c)$</td>
<td>$\exists c \in {-1, 1}, \forall (\vec{x}, \vec{x'}) \in P, \Theta_{s} \vec{x} - \Theta_{s'} \vec{x}'' = (0, \ldots, 0, c, 0)$</td>
</tr>
</tbody>
</table>

**Sharing Opportunity Constraints** Realizing a sharing opportunity in a schedule also imposes certain constraints on the schedule. To reduce the duration for which the shared data has to remain in memory, we require that related statement instances in a non-self sharing opportunity $a \rightarrow a'$ (where $a.s \neq a'.s$) be scheduled to times that differ only in the last constant time dimension. However, this requirement would not work for a self sharing opportunity $a \rightarrow a'$ (where $a.s = a'.s$), because related statement instances have the same constant for their last schedule dimension and thus enforcing it would schedule two instances to the same time. Thus, we instead require them to be scheduled to consecutive times, ignoring the last constant time dimension. In mathematical terms, realizing a sharing opportunity means satisfying the constraints listed in Table 5.1 according to its type. We special-case $R \rightarrow R$ sharing opportunities because a pair of related statement instances may have their execution order reversed by a new schedule without altering the original program semantics. Note that the polyhedra in Table 5.1 are after the preprocessing steps discussed in Section 5.5.1.

Each dimension (except the last one) of these constraints can easily be converted into linear constraints on the schedules by applying the affine form of the Farkas Lemma. The last constant dimension for all statements can be determined by a simple algorithm based on topological sort.
5.5.3 *Search Algorithm*

The goal of optimization is to find a schedule that minimizes I/O cost, or maximizes I/O savings, for a program given a certain amount of available memory. I/O savings come from the realization of sharing opportunities. It is important to note the following:

- Not all sharing opportunities can be realized simultaneously; some may be in direct conflict, as shown in Example 7.

- Maximizing the number of realized sharing opportunities does not necessarily minimize the total I/O cost, because the memory requirement may exceed the given cap, or because the amount of I/O saved by individual sharing opportunities varies depending on the sizes of array blocks and their iteration domains.

A naïve approach is to enumerate the power set of $\mathcal{O}$, the set of all sharing opportunities, and for each candidate subset check if its member sharing opportunities can all be realized by a schedule while satisfying all dimensionality and dependence constraints. We propose a better algorithm based on the following key observation:

**Lemma 2** (Apriori Property). *If a set of sharing opportunities cannot be realized simultaneously, nor can any of its supersets.*

This lemma immediately suggests an algorithm similar to Apriori [AS+94]. Algorithm 2 shows the details. The algorithm proceeds in the order of increasing size of sharing opportunity combinations. A set of $k$ sharing opportunities is considered a candidate only if all its subsets of size $k - 1$ are found to be feasible already (Line 5). A candidate is feasible only if it survives the *FindSchedule* test (details below), which attempts to find a schedule that realizes all the sharing opportunities in the candidate while satisfying all dimensionality and dependence constraints.
Algorithm 2: Apriori-like search.

**Input**: Set containing all sharing opportunities \( O \), set containing all dependences \( D \)

**Output**: Set of legal schedules, each satisfying a different combination of sharing opportunities

```plaintext
1 \( C_1 \leftarrow \{ o \mid o \in O, \text{ FindSchedule}(\{o\}, D) \neq \emptyset \} \)
2 \( T \leftarrow \{ \text{FindSchedule}(\{o\}, D) \mid o \in C_1 \} \)
3 \( k \leftarrow 2 \)
4 while \( C_{k-1} \neq \emptyset \) and \( k \leq |O| \) do
5 \( C_k \leftarrow \{ c \mid c \subseteq O, |c| = k, c \text{'s subsets of size } k-1 \text{ all in } C_{k-1} \} \)
6 foreach \( c \in C_k \) do
7 \( t \leftarrow \text{FindSchedule}(c, D) \)
8 if \( t = \emptyset \) then \( C_k \leftarrow C_k \setminus \{c\} \) else \( T \leftarrow T \cup \{t\} \)
9 \( k \leftarrow k + 1 \)
10 return \( T \)
```

FindSchedule (Algorithm 3), repeatedly called by the search algorithm, searches for a legal schedule for a candidate sharing opportunity set. Because each dependence constraint can be satisfied at any of the \( \tilde{d} + 1 \) depths and it is computationally too expensive to consider all possibilities, the algorithm tries to satisfy each dependence constraint in a greedy fashion, from depth 1 to depth \( \tilde{d} + 1 \). Satisfying a dependence constraint at an early depth may lead to “over-separation” of statements (through enforcing the lexicographic order of statement instances) and prevent I/O sharing. To address this issue, we give higher priority to sharing opportunity constraints (Lines 13) and lower priority to dependence constraints (Lines 15–19). A similar greedy algorithm is used to satisfy dimensionality constraints (Lines 14). Note that Algorithm 3 involves many basic polyhedral operations, e.g., intersection and applying the affine form of the Farkas Lemma. The isl library [Ver10] provides efficient implementation of these operations.

One subtlety is worth noting. For two feasible sharing opportunity sets \( Q \) and \( Q' \) where \( Q \subset Q' \), FindSchedule may produce the same schedule. Indeed, a schedule satisfying the sharing opportunity constraints for \( Q \) may happen to also satisfy those for \( Q' \setminus Q \), because the algorithm does not explicitly consider other sharing
opportunities when trying to satisfy $\mathcal{Q}$. However, as discussed earlier, “accidentally” realizing more sharing opportunities may not be desirable, as it may increase memory requirement. Thus, when generating code for a schedule (Section 5.5.5), we consider the set $\mathcal{Q}$ of sharing opportunities it is supposed to realize, and inject appropriate code to exploit only $\mathcal{Q}$. In any case, the search will eventually consider superset $\mathcal{Q}' \supset \mathcal{Q}$, so no good schedule will be missed.

---

$^7$ This brings up the point that a schedule alone does not completely dictate what and how I/O sharing is achieved; it only specifies the execution timing of statement instances, a necessary but not sufficient condition for sharing. Code generation must ensure that appropriate I/O and memory buffer management actions are taken to enable sharing.
Algorithm 3: FindSchedule($\mathcal{Q}, \mathcal{D}$).

**Input:** Sharing opportunity set $\mathcal{Q}$, dependence set $\mathcal{D}$

1. $n \leftarrow |\mathcal{S}|$, the number of statements
2. $\hat{d} \leftarrow \max_{s \in \mathcal{S}} d_s$
3. $\mathcal{Q}_{sw} \leftarrow$ self sharing opportunities of types $W \rightarrow R$, $W \rightarrow W$
4. $\mathcal{Q}_{sr} \leftarrow$ self sharing opportunities of type $R \rightarrow R$
5. $\mathcal{Q}_{nw} \leftarrow$ non-self sharing opportunities of types $W \rightarrow R$, $W \rightarrow W$
6. $\mathcal{Q}_{nr} \leftarrow$ non-self sharing opportunities of type $R \rightarrow R$
7. $k_1, \ldots, k_n \leftarrow 0$; $\Theta_1, \ldots, \Theta_n \leftarrow \emptyset$
8. Let $\theta^d$ denote $(\theta^d_1, \ldots, \theta^d_n)$, the $d$-th dimension of schedules

9. for $d \leftarrow 1$ to $\hat{d}$ do
   // Initialize the space of schedules for dimension $d$
   10. $X_d \leftarrow$ the polyhedron containing all integer points
   // Weakly satisfy remaining dependence constraints
   11. foreach $a \rightarrow a' \in \mathcal{D}$ do
       12. $X_d \leftarrow X_d \cap \{\theta^d \mid \forall (x, x') \in \mathcal{P}(a \rightarrow a'), \theta^d_{a', s} x' - \theta^d_{a, s} x \geq 0\}$
   // Sharing opportunity constraints
   13. $X_d \leftarrow \text{SatisfySharingOpportunities}(X_d, \mathcal{Q}_{sw}, \mathcal{Q}_{sr}, \mathcal{Q}_{nw}, \mathcal{Q}_{nr})$
   // Dimensionality constraints
   14. $X_d \leftarrow \text{SatisfyDimensionality}(X_d, n, d, k_1, \ldots, k_n, \Theta_1, \ldots, \Theta_n)$
   // Strongly satisfy remaining dependence constraints
   15. foreach $a \rightarrow a' \in \mathcal{D}$ do
       16. $T \leftarrow \{\theta^d \mid \forall (x, x') \in \mathcal{P}(a \rightarrow a'), \theta^d_{a', s} x' - \theta^d_{a, s} x > 0\}$
       if $X_d \cap T \neq \emptyset$ then
           18. $X_d \leftarrow X_d \cap T$
           19. $\mathcal{D} \leftarrow \mathcal{D} \setminus \{a \rightarrow a'\}$
       if $X_d = \emptyset$ then return $\emptyset$
       21. $\theta^d_1, \ldots, \theta^d_n \leftarrow$ sample a point from $X_d$
   22. foreach $i \leftarrow 1$ to $n$ do $\Theta_i \leftarrow \Theta_i \cup \{\theta^d_i\}$
   23. Find constants for the last dimensions of $\Theta_1, \ldots, \Theta_n$
   24. return $\Theta = \{\Theta_1, \ldots, \Theta_n\}$
Algorithm 4: SatisfySharingOpportunities($\mathcal{X}_d, Q_{sw}, Q_{sr}, Q_{nw}, Q_{nr}$).

1. foreach $a \rightarrow a' \in Q_{nw} \cup Q_{nr}$ do
2.   $X_d \leftarrow X_d \cap \{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{P}(a \rightarrow a'), \theta^d_{a',s} \vec{x}' - \theta^d_{a,s} \vec{x} = 0 \}$
3. if $d < \tilde{d}$ then
4.   foreach $a \rightarrow a' \in Q_{sw} \cup Q_{sr}$ do
5.     $X_d \leftarrow X_d \cap \{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{P}(a \rightarrow a'), \theta^d_{a',s} \vec{x}' - \theta^d_{a,s} \vec{x} = 0 \}$
6. else
7.     foreach $a \rightarrow a' \in Q_{sw} \cup Q_{sr}$ do
8.       $X_d \leftarrow X_d \cap \{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{P}(a \rightarrow a'), \theta^d_{a',s} \vec{x}' - \theta^d_{a,s} \vec{x} = 1 \}$
9.     foreach $a \rightarrow a' \in Q_{sw} \cup Q_{sr}$ do
10.    $X_d \leftarrow X_d \cap \left\{ \begin{array}{l}
\{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{P}(a \rightarrow a'), \theta^d_{a',s} \vec{x}' - \theta^d_{a,s} \vec{x} = -1 \} \cup \\
\{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{P}(a \rightarrow a'), \theta^d_{a',s} \vec{x}' - \theta^d_{a,s} \vec{x} = 1 \} \end{array} \right\}$
11. return $X_d$

Algorithm 5: SatisfyDimensionality($\mathcal{X}_d, n, d, k_1, \ldots, k_n, \Theta_1, \ldots, \Theta_n$)

1. for $i \leftarrow 1$ to $n$ do
2.   $f \leftarrow false$
3.   foreach $l \leftarrow EnumRow((i, d - 1, k_i))$ do
4.     if $l = 0$ then $T \leftarrow$ space spanned by $\Theta_i$
5.     else $T \leftarrow$ null space of $\Theta_i$
6.     if $X_d \cap T \neq \emptyset$ then
7.       $X_d \leftarrow X_d \cap T$
8.       $k_i \leftarrow k_i + l$
9.       $f \leftarrow true$
10. break
11. if $f = false$ then return $\emptyset$
12. return $X_d$
5.5.4 Cost Evaluation

The search algorithm returns a list of legal schedules, each satisfying a particular combination of sharing opportunities. Next, we evaluate each schedule in terms of memory requirement and I/O cost.

Memory Requirement We want to compute the maximum amount of memory required by a schedule $\Theta$ found for a sharing opportunity set $Q$. First consider the baseline, where no sharing opportunities are realized. For a statement to successfully execute at time $\bar{\tau}$, all array blocks it accesses must all be in memory at $\bar{\tau}$. Therefore, the baseline memory requirement at time $\bar{\tau}$, $M(\bar{\tau})$, can be computed by first finding the iteration instance $\bar{x} = \Theta^{-1}(\bar{\tau})$, and then summing up all the sizes of the array blocks $\bar{x}$ accesses. Each realized sharing opportunity except those of type $W \rightarrow W$ can require additional memory for keeping the shared array block until the reuse occurs. For each sharing opportunity $a \rightarrow a' \in Q$, for each $(\bar{x}, \bar{x}') \in P(a \rightarrow a')$, the shared array block $a.A[a.\Phi \bar{x}]$ has to be kept in memory between time $\Theta_{a,s}(\bar{x})$ and $\Theta_{a',s}(\bar{x}')$. Thus we can find for each time $\bar{\tau}$ all the additional array blocks that have to be in memory at that time, and add their sizes to $M(\bar{\tau})$. Finally, taking the maximum of $M(\bar{\tau})$ across all $\bar{\tau}$’s gives the memory requirement of the schedule.

I/O Cost We adopt a simple I/O cost model that predicts the total I/O time as a linear function of the total read and write volumes (in the number of bytes). More refined models (e.g., charging an overhead for each I/O request) can be easily incorporated, though as shown by the experiments in Section 5.6, our simple model already provides very accurate estimates (thanks to our framework’s ability to capture instance-level I/O sharing).

Without realizing any sharing opportunity, the baseline I/O cost for a statement can be computed by summing up the sizes of all array blocks it accesses over its
iteration domain. Realized I/O sharing opportunities involving the statement can save some of the baseline I/O operations and cut down the cost. For example, for a sharing opportunity \( a \rightarrow a' \) where \( a' = \langle s', R, A, \Phi' \rangle \), at any target iteration \( x' \in \{ x' \mid (x, x') \in P(a \rightarrow a') \} \) a read of array block \( A[\Phi' x'] \) is saved; for one where \( a = \langle s, W, A, \Phi \rangle \) and \( a' = \langle s', W, A, \Phi' \rangle \), at any source iteration \( x \in \{ x \mid (x, x') \in P(a \rightarrow a') \} \) a write of array block \( A[\Phi x] \) is saved. With a union operation across all realized sharing opportunities, we can find the I/O savings for every iteration instance. Summing up all the savings over all iteration instances gives the total I/O savings of the given schedule; subtracting it from the baseline I/O cost gives the actual I/O cost of the schedule.

**Remark** All computation above happens in a symbolic, algebraic fashion under the polyhedral model. The memory requirement and I/O cost of a schedule are represented as polynomials (piecewise quasipolynomials to be exact) in the global parameters \( \bar{p} \). The advantage of this approach is that schedule search and evaluation need to be done only once for a given program “template”; should the parameters (array and block sizes) change, we only need to plug the new values into the polynomials instead of performing optimization all over again.

5.5.5 **Code Generation**

The schedule chosen by the optimizer is subsequently transformed into C code with **for** and **if** control structures for compilation and execution. This process is the reverse of the program analysis that happens before optimization: program analysis extracts a polyhedral representation from code, while code generation converts the optimized polyhedral representation back to code. Recent advances in polyhedral compiler construction have produced efficient code generation tools such as CLooG [Bas04, VBC06], which is incorporated in production compilers such as GCC and also used by us.
As a concrete example, let us see what code is generated for the program in Example 7. Suppose the following three sharing opportunities are satisfied: $s_1 WC \rightarrow s_2 RC$, $s_2 WE \rightarrow s_2 RE$, $s_2 WE \rightarrow s_2 WE$. Note that whether this yields the optimal I/O cost depends on the values of the parameters; here we use the parameterized schedule to demonstrate code generation only. The optimizer produces the following schedule for this set of sharing opportunities: $\Theta_{s_1} \bar{x}_{s_1} = (0, -i, k, 0)$, $\Theta_{s_2} \bar{x}_{s_2} = (j, -i, k, 1)$.

The generated code for this schedule is listed below. It is easy to verify that this is equivalent to the hand-generated solution in Figure 5.1(b).

```c
for (i=-n1+1; i<=0; i++)
    for (k=0; k<=n2-1; k++) {
        C[-i,k] = A[-i,k] + B[-i,k]; // s1
        E[-i,0] += C[-i,k] * D[k, 0]; // s2
    }
for (j=1; j<=n3-1; j++)
    for (i=-n1+1; i<=0; i++)
        for (k=0; k<=n2-1; k++)
            E[-i,j] += C[-i,k] * D[k,j]; // s2
```

For brevity, the code above does not contain explicit I/O operations. In practice, RIOTShare injects additional code to ensure that all array block accesses are fulfilled either by blocks already buffered in memory or by I/O (and displacing appropriate buffered blocks when necessary); the details are omitted. In general, RIOTShare relieves the burden of manually managing I/O from library and application developers.

5.6 Experiments

**Setup** All our experiments were run on a desktop computer with an Intel Core i7-2600 four-core CPU, 8GB of memory, and a WD Caviar Black 7200RPM hard drive, running Ubuntu Linux 11.10. I/O and CPU time of plan execution was collected using the systemtap instrumentation tool. We verified that instrumentation overhead

---

8 $C$ is not written in the case ($n_3 = 1$) shown in Figure 5.1(b). Although not reflected in the code shown here, our optimizer and execution engine check the value of $n_3$ and decide if $C$ needs to be written to disk.
was negligible. To make it easier to understand results, we used the ext2 file system, as it does not have journaling that would necessarily complicate result interpretation. To make I/O measurements meaningful, we turned off file system caching using the \texttt{O\_DIRECT} flag when opening files. Under this setting, we benchmarked the I/O rates of the hard drive and found that sustained reads and writes were 96MB/s and 60MB/s, respectively. These numbers were used by the optimizer to convert the predicted I/O volume of plans to estimated I/O time. The in-core computation of tested programs was done by calling GotoBLAS2 [GOT], an optimized implementation of BLAS which is able to utilize all four cores on our machine.

**Storage Scheme** In the following experiments, matrices are stored in large, logical blocks. The blocks are laid out on disk in column-major order, and so are the elements within each block. Since every element in a matrix has a predetermined storage position, its index (row and column numbers) is not stored. This is a highly efficient storage scheme for dense matrices. We use our storage library, RIOTStore (Chapter 4), for managing the storage and performing I/O. RIOTStore implements the LAB-tree (Linearized Array B-tree) and the DAF (Directly Addressable File) storage formats, both of which provide the storage scheme we want and work virtually identically for dense matrices.

**A Note on Optimization Time** The optimization time for all experiments below are reasonably short: 0.6 second for the matrix addition and multiplication program in Section 5.6.1, 2.1 seconds for the two matrix multiplications in Section 5.6.2, and 156.7 seconds (more on this next) for the linear regression program in Section 5.6.3. Even though it optimizes at the instance level, our optimizer avoids enumerating all statement instances by working with polyhedra. Hence, the complexity of optimization depends on the complexity of the program (such as the number of statements and dimensionalities of iteration domains) instead of the size of the data it operates on or
the number of iterations each loop takes. This property can be seen directly from the algorithms in Section 5.5, and has been further confirmed by experiments on datasets of different scales.

The optimization time for the linear regression experiment is longer, because there are 7 operators (statements) and 16 sharing opportunities, and also because our optimizer is implemented in Python and single-threaded—we expect a multithreaded C implementation to be significantly more efficient. Nevertheless, our optimizer is able to cut the search space from 65536 plans down to 4100. As we shall see later, the optimization overhead is dwarfed by the I/O savings. Moreover, this overhead is not affected by the size of the dataset; it becomes more negligible when the dataset is larger. If needed, we can further improve optimization time for larger, more complex programs by localizing optimization to the most expensive code fragments, and by combining plan enumeration and costing so we can terminate the search early as soon as acceptable plans are found.

Datasets of Different Scales We have run the following experiments with datasets of different scales and found consistent results. Also, as expected, optimization time for the same program does not change with the scale of the dataset. Due to the space limit, below we present only results on the largest dataset tested.

5.6.1 Matrix Addition and Multiplication

We first test our optimizer with the program shown in Example 7. It consists of a matrix addition followed by a matrix multiplication. The actual matrix sizes used in this experiment are listed in the table below. The optimizer finds 8 legal execution plans (including the unmodified original schedule). We plot these plans in Figure 5.4.

Figure 5.4(a) shows each plan’s memory footprint and I/O time as estimated by the optimizer. The circles (◦) represent the 8 plans considered by the optimizer (the ♠ is explained below). We notice that a plan’s memory footprint can only take one
Table 5.2: Matrix sizes in the addition-multiplication experiment.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Block size</th>
<th># Blocks</th>
<th>Total size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B, C$</td>
<td>$6000 \times 4000$</td>
<td>$12 \times 12$</td>
<td>$25.6$GB</td>
</tr>
<tr>
<td>$D$</td>
<td>$4000 \times 5000$</td>
<td>$12 \times 1$</td>
<td>$1.8$GB</td>
</tr>
<tr>
<td>$E$</td>
<td>$6000 \times 5000$</td>
<td>$12 \times 1$</td>
<td>$2.7$GB</td>
</tr>
</tbody>
</table>

of three possible values, because there are limited combinations of which matrices’ blocks to keep in memory. Among plans with identical memory footprint, all have different I/O costs. The plan with the lowest I/O cost, Plan 7 in the lower right corner, takes 836 seconds, while the original plan, Plan 0 in the upper left corner, takes 2394 seconds. The code generated from Plan 7 is equivalent to the one shown in Figure 5.1(b) for the general case. Since $n_3 = 1$ (the number of blocks in the second dimension of $D$ and $E$) in this experiment, Plan 7 is also effectively equivalent to the special case solution in Figure 5.1(a). Plan 7 satisfies three sharing opportunities: $s_1WC \rightarrow s_2RC$, $s_2WE \rightarrow s_2RE$ and $s_2WE \rightarrow s_2WE$. Note that because $n_3 = 1$, sharing opportunity $s_2RC \rightarrow s_2RC$ does not exist.

One may argue the comparison between Plan 0 and 7 is not fair: Plan 0 underutilizes the memory and could reduce I/O by buffering more data. Extra memory can be used to support sophisticated I/O sharing schedules as Plan 7, or to simply
allow bigger array blocks. Which approach is better? To answer this question, we took Plan 0 and increased the number of rows in $A$, $B$, $C$, and $E$’s blocks from 6000 to 9000, and plotted it (♦) also in Figure 5.4(a). This modified plan consumes more memory than Plan 7, but still incurs far more I/O cost than it. This shows that blindly enlarging array blocks is not the best way of utilizing extra memory; cost-driven optimization like ours can give much better results.

Figure 5.4(b) compares the optimizer-predicted I/O cost with the actual I/O cost of executing the plan. This comparison shows our optimizer is impressively accurate in estimating the the I/O cost of plans; the average error is merely 1.7%. This high accuracy should be no surprise, because our optimizer is precise down to instance-level sharing and can calculate the exact number and amount of I/O. The only source of error is the simple I/O cost model we employ for predicting the I/O time from the amount of I/O; however, the error is small and does not affect optimization decisions.

Figure 5.4(b) also breaks the actual execution time of each plan down into CPU and I/O time. Because our optimizer only optimizes I/O, the CPU time is the same across all plans. With or without optimization, the program remains I/O-dominant. Therefore, maximizing I/O sharing brings the total execution time from the original 3180 seconds down to 1560 seconds, a 50.9% improvement.

Comparing to Matlab and SciDB  Matlab and SciDB [Bro10] represent state-of-the-art scientific computing and scientific data management systems, respectively. However, neither has RIOTShare’s I/O sharing optimization capabilities. We have tested them using the same input program (properly translated) and the same data (properly converted and loaded). As with RIOTShare, both systems are allowed to use all four CPU cores of the machine. Running the test program without blocking in Matlab immediately gives a not-enough-memory error, due to the large data size. With blocking, Matlab’s running time is 2.65 times that of our best plan. This
suggests that not only is Matlab unable to optimize I/O, but it also has considerable control and storage overhead. Manually implementing our best plan in Matlab makes a big difference—the performance becomes 6% better than ours. The minor advantage may come from Matlab’s higher in-memory math performance. This demonstrates that the ideas developed in this paper is readily transferable to existing systems and have great, platform-independent potential.

Although given a larger memory usage, SciDB takes 33.08 times more time than our best plan. This could be a result of not using a BLAS library or using an unoptimized one, and also not sharing I/O. While SciDB focuses on parallelization, its handling of execution on a single node seems to leave a big room for improvement on I/O efficiency.

5.6.2 Two Matrix Multiplications

![Figure 5.5: Two matrix multiplications. Selected plans. Config A.](image)

We next present results on a program with slightly more control structures and a larger search space. Matrix multiplication is a fundamental building block of many data analysis algorithms and routines and has long been the target of optimization efforts by researchers and HPC vendors. It is common to have multiple matrix multiplications in the same program. In the following, we consider two
matrix multiplications, $C = AB$, $E = AD$, to be executed together. There are 9 sharing opportunities. We have tested this program with two different matrix size configurations; relevant information is summarized in the tables below.

for (i=0; i<n1; ++i)
  for (j=0; j<n2; ++j)
    for (k=0; k<n3; ++k)
      C[i,j] += A[i,k] * B[k,j]; // s1

for (i=0; i<n1; ++i)
  for (j=0; j<n4; ++j)
    for (k=0; k<n3; ++k)
      E[i,j] += A[i,k] * D[k,j]; // s2

Under both configurations, the optimizer produced 40 plans. For the sake of presentation, we select four plans for demonstration below. Plan 0 enables no sharing opportunities, Plan 1 enables o5–o8, Plan 2 enables o1 plus o5–o8, and Plan 3 enables o1–o3. Intuitively, Plan 1 uses two separate loop nests to accumulate $C$ and $E$ blocks.
Table 5.4: Matrix sizes in the multiplication-multiplication experiment.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Matrix</th>
<th>Block size</th>
<th># Blocks</th>
<th>Total size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (Figure 5.5)</td>
<td>A</td>
<td>8000 × 7000</td>
<td>6 × 6</td>
<td>15.2GB</td>
</tr>
<tr>
<td></td>
<td>B, D</td>
<td>7000 × 3000</td>
<td>6 × 10</td>
<td>9.2GB</td>
</tr>
<tr>
<td></td>
<td>C, E</td>
<td>8000 × 3000</td>
<td>6 × 10</td>
<td>10.8GB</td>
</tr>
<tr>
<td>B (Figure 5.6)</td>
<td>A</td>
<td>2000 × 8000</td>
<td>18 × 6</td>
<td>12.8GB</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>8000 × 6000</td>
<td>6 × 4</td>
<td>8.4GB</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2000 × 6000</td>
<td>18 × 4</td>
<td>6.4GB</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>8000 × 7000</td>
<td>6 × 4</td>
<td>10.0GB</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>2000 × 7000</td>
<td>18 × 4</td>
<td>7.6GB</td>
</tr>
</tbody>
</table>

in memory. Plan 2 in addition merges the two loop nests and shares the read of A. Plan 3 shares the I/O to B and D instead of C and E. Each plan works the same way for both configurations, except with different size parameters.

Figure 5.5 and 5.6 summarize the plan spaces and characteristics of the selected plans. Figure 5.5(a) and Figure 5.6(a) clearly illustrate that different matrix size configurations have dramatic impact on plan cost and optimality. This observation highlights the need for automatic and systematic optimization, because code manually optimized based on expert knowledge or past experience has fragile performance. Even if one knows the best plan for the current problem, a slight change in memory cap or problem size can easily render the current solution inappropriate. The plans shown in Figure 5.5(b) and Figure 5.6(b) exemplify this observation. Plan 2 has the lowest I/O cost under Configuration A, but is suboptimal under Configuration B, where Plan 3 is the best. Comparing predicted and actual I/O times, we find the average error to be merely 0.6%. Even though matrix multiplication is traditionally considered CPU-dominant, the I/O and CPU time breakdown here actually reveals that, for big data, I/O is equally (if not more) expensive than CPU; optimizing I/O therefore provides good overall performance improvement.
5.6.3 Linear Regression: A Complete Program

We next test RIOTShare with a commonly used statistical method—linear regression. Suppose we want to fit a linear model for a set of response variables \( y = (y_1, \ldots, y_k) \) from \( m \) predictor variables \( x = (x_1, \ldots, x_m) \), i.e., \( y_j = x'_i \beta_j + \epsilon_j \), where \( \epsilon_j \sim N(0, \sigma^2_j) \). Suppose \( n \) i.i.d. observations are collected: \( \{y_i, x_i\}_{i=1}^n \). Using the ordinary least square method, we can estimate the coefficient vectors \( \beta_j \), which are column-combined to form a matrix \( \beta \), simultaneously by \( \hat{\beta} = (X'X)^{-1}X'Y \), where \( X \) is formed by row-stacking \( x_i \)'s and \( Y \) by row-stacking \( y_i \)'s. After obtaining \( \hat{\beta} \), we further compute the Residual Sum of Squares: \( RSS(Y_j - X \hat{\beta}_j) = \sum_{i=1}^n (y_{ji} - x'_i \hat{\beta}_j)^2 \). Written in matrix form, this program has 7 steps (statements): \( U = X'X; V = X'Y; W = U^{-1}; \hat{\beta} = WV; \hat{Y} = X \hat{\beta}; E = Y - \hat{Y}; R = RSS(E) \). Normally the number of response and predictor variables \( k \) and \( m \) are small but the number of observations \( n \) can be very large. Below we consider a case where \( k = 400, m = 4000 \) and \( n = 1.5 \times 10^6 \). The following table summarizes the detailed size configuration of the matrices involved.

We have implemented the above linear regression program for optimization by RIOTShare. The input program has a sequence of 7 loop nests, one for each step. Following the design of BLAS, matrix transpose is not modeled as a separate operator,
Table 5.5: Matrix sizes in the linear regression experiment.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Block size</th>
<th># Blocks</th>
<th>Total size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{X}$</td>
<td>60000 × 4000</td>
<td>25 × 1</td>
<td>44.7GB</td>
</tr>
<tr>
<td>$Y, \hat{Y}, E$</td>
<td>60000 × 400</td>
<td>25 × 1</td>
<td>4.5GB</td>
</tr>
<tr>
<td>$U, W$</td>
<td>4000 × 4000</td>
<td>1 × 1</td>
<td>122.1MB</td>
</tr>
<tr>
<td>$V, \hat{\beta}$</td>
<td>4000 × 400</td>
<td>1 × 1</td>
<td>12.2MB</td>
</tr>
</tbody>
</table>

but as a flag passed to operations such as matrix multiplication.

Figure 5.7(a) plots all the plans generated by the optimizer. The best plan (bottom-right, Plan 2) uses only 6.0% more memory than the original unoptimized plan (top-left, Plan 0), but saves I/O time by 43.8%. This improvement comes from sharing the reads of $X$ for the two out-of-core matrix multiplications and eliminating the materialization of intermediate results. Figure 5.7 plots the predicted and actual running time of the two plans together with another Plan 1, which merely keeps $U$ and $V$ in memory during the multiplication. Again, the prediction is highly accurate, with maximum error 2.3%. In terms of total running time, the best plan gives 27.0% improvement over the original plan.

5.7 Conclusion

Big data analytics are often I/O-intensive. In this chapter, we have presented RIOTShare, for representing and optimizing I/O patterns in such tasks. Building on the polyhedral model, RIOTShare strikes the balance between feasibility and flexibility of representation and optimization, by exploring the middle ground between the high-level, database-style operator-based query optimization and low-level, compiler-style loop-based code optimization. Experiments show that RIOTShare produces accurate estimates for the I/O costs of candidate plans, and finds nontrivial plans with significant I/O improvement under memory constraints.
The I/O sharing optimization discussed in Chapter 5 does not consider array layouts as one dimension of its optimization space; instead, layouts of arrays are assumed to be predetermined by the user or some other software component. However, the choice of array layouts can dramatically affect a program’s overall I/O performance, and thus needs to be determined in a cost-based manner. In this chapter, we focus on the commonly used block-based layout and show how it can be co-optimized with I/O sharing and yield further I/O savings.

6.1 Introduction

We have seen in Chapter 5 the importance of I/O sharing optimization in achieving high I/O performance. However, so far we have not concerned ourselves with the problem of deciding optimal layouts for arrays involved in a given program. When searching for the best I/O sharing plan, the RIOTShare optimizer assumes predetermined layouts for all arrays. Although this approach helps cleanly formulate and solve the I/O sharing problem, it can be improved by further incorporating the array
layout factor, an important dimension of optimization space, into a unified framework to produce potentially more I/O-efficient execution plans.

In this chapter, we tackle the problem of jointly optimizing I/O sharing and array layouts. Many advanced array layouts have been designed, such as those based on advanced space-filling curves [Sag94]. However, the most widely used layout is still the block-based, which is adopted by most out-of-core numerical analysis algorithms (e.g., [Tol99b, RT01]) and linear algebra libraries (e.g., [TG96, CDPW92]). Note that the row-major and column-major layouts are special cases of the block-based layout. In this chapter, we focus on the block-based layout only, which is also consistent with Chapter 5’s assumption.

In previous chapters we have already mentioned that many operators require specific block layouts of their operands in order to achieve optimal I/O efficiency. Let us again take the fundamental matrix multiplication operator as an example to see the importance of layout choices.

**Example 8.** Consider the number of reads and writes of matrix elements as a simplified measure of I/O cost. The program listed below computes two matrix multiplications: $C = AB$ and $E = DA$. Matrices $A$, $B$ and $D$ are initially on disk and $C$ and $E$ need to be on disk, too, when the program finishes execution. Suppose the input matrices $D$, $A$, and $B$ have sizes $p \times q$, $q \times r$, and $r \times s$, respectively. Further suppose we have the opportunity of storing the input matrices using the block-based layout before the program is executed, and have the flexibility of choosing proper blocking factors for them. For this example, we require that once chosen, the blocking factors cannot change throughout program execution (i.e., $A$ should have the same layout at $s_1$ and $s_2$). Let the blocking factors of $D$, $A$, and $B$ be $w \times x$, $x \times y$, and $y \times z$, respectively. As a result, $C \ (q \times s)$ has blocking factor $x \times z$ and $E \ (p \times r)$ has blocking factor $w \times y$. The following C-style code shows an implementation of
the two matrix multiplications in sequence.\footnote{For this example we assume each dimension can be perfectly blocked without remainders; in practice, various techniques, such as rounding and padding, can be used to address the imperfect division issue.} In the following (and the rest of this chapter), each array access (such as $C[i,j]$ below) represents a block access, not an element access.

\begin{verbatim}
for (i=0; i<q/x; ++i)
    for (j=0; j<s/z; ++j) {
        for (k=0; k<r/y; ++k) {
            C[i,j] += A[i,k] * B[k,j]; // s1 
        }
    }
for (i=0; i<p/w; ++i)
    for (j=0; j<r/y; ++j) {
        for (k=0; k<q/x; ++k) {
            E[i,j] += D[i,k] * A[k,j]; // s2 
        }
    }
\end{verbatim}

Let us first consider the number of matrix elements accessed during the execution of the first loop nest ($C = AB$). If we regard each array access as an I/O, it is easy to see that $A$ is read $s/z$ times, $B$ is read $q/x$ times, and $C$ is read $r/y - 1$ times\footnote{$C[i,j]$ does not need to be read from disk when $k = 0$; instead, it is directly initialized with 0.} and written $r/y$ times. Thus the read and write costs, in terms of the number of elements, are:

\[
R_1 = \frac{qrs}{z} + \frac{qrs}{x} + \frac{qrs}{y} - qs, \quad W_1 = \frac{qrs}{y}.
\]

Suppose we are given a fixed amount of memory, which can hold $M$ matrix elements. The optimal blocking factors can be solved by minimizing $R_1 + W_1$ subject to $xy + yz + xz \leq M$. Solving this optimization problem by hand gives

\[
x = z = \sqrt{(\sqrt{5} - 1)M}, \quad y = \frac{3 - \sqrt{5}}{2} \sqrt{(\sqrt{5} + 2)M}.
\]

If we consider the second loop nest ($E = DA$) in isolation, it can be similarly
shown that the I/O cost is \( R_2 + W_2 \), where

\[
R_2 = \frac{pqr}{y} + \frac{pqr}{w} + \frac{pqr}{x} - pr, \quad W_2 = \frac{pqr}{x},
\]

which is minimized when

\[
w = y = \sqrt{(\sqrt{5} - 1)M}, \quad x = \frac{3 - \sqrt{5}}{2} \sqrt{(\sqrt{5} + 2)M}.
\]

Combining the two operations, the globally optimal blocking factors are even more complicated to solve, and are generally different from either solution above.

Example 8 shows that blocking factor solutions for individual operations do not directly carry over to composite programs; in fact, they may conflict each other. Given a particular cost model (which can be more complex than the one used in Example 8), every program needs to be custom-analyzed. Even for a single program, the blocking factor analysis needs to be combined with different ways of program transformations. For instance, the code in Example 8 does not take advantage of any I/O sharing opportunities. If some sharing opportunities are realized, e.g., \( s_1WC \rightarrow s_1RC \) and \( s_1WC \rightarrow s_1WC \) (for explanation of the notation, refer to Section 5.4.3), the objective function will change and it may require another set of values for the blocking factors to minimize the new objective function. Which subset of I/O sharing opportunities to realize and what blocking factors to choose are two inseparable problems. It is necessary to perform automatic, cost-based, joint optimization of sharing opportunities and blocking factors, preferably under a unified mathematical framework.

Specifically, the optimizer should provide the following features:

1. In addition to I/O sharing, the search space must include array blocking scheme as a dimension. An optimal execution plan should be specified in terms of both factors.
2. The cost model must account for array blocking factors, in a way that makes optimization tractable. Specifically, I/O cost must be automatically derivable as a function of blocking factors by a mechanical procedure, and ideally must be numerically solvable by one, too. Just as the I/O sharing optimizer in Chapter 5, the polyhedral description of a program (i.e., iteration domains, array accesses, and original schedule) should be the only necessary input to the optimizer. Analysis such as that in Example 8 should be automatically derived. Contrary to I/O sharing optimization, we cannot afford to enumerate blocking factors the same way we enumerate combinations of I/O sharing opportunities, even with effective pruning, because blocking factors have a much wider domain. Therefore, mathematical structure of the cost function, if any, must be exploited to make the search efficient.

3. Conflicts among blocking factors of the same array must be handled. In Example 8, matrix $A$ appears both as the left operand of a matrix multiplication and also as the right operand of another. Each operator requires dramatically different blocking factors of $A$. The optimizer must reconcile such conflicts and make cost-based decisions with a global view.

4. As a result of possible incompatible layout requirements, the optimizer must consider layout conversion as one way of conflict resolution. For example, if two operators do not have I/O sharing opportunities, or have some which the optimizer decides not to activate, their conflict on an array’s blocking factors can be resolved by explicitly inserting a layout conversion operation between them.

In this chapter we propose an optimizer that jointly considers I/O sharing and array layouts in finding the I/O-optimal plan for an input program. The optimizer builds on RIOTShare and thus inherits its capability of I/O sharing optimization.
In addition, it considers block-based array layouts as a new dimension of the search space, by directly incorporating blocking factors as variables into the global cost function. The optimizer takes a two-stage approach. First a schedule is found for each feasible subset of I/O sharing opportunities. Optimal array blocking factors are then found under each schedule. Independent layout choices are allowed for the same array in different statements. Determining the optimal blocking factors under a particular schedule, subject to the memory capacity constraint, is modeled as a complementary geometric program, for which efficient approximation algorithms exist. By exploring a richer search space, the new optimizer is able to produce more efficient plans, as demonstrated by experiments.

Section 6.2 reviews related work. Section 6.3 gives an overview of the optimization problem, the proposed two-stage approach, and the blocking model. The two stages of optimization are presented in details in Section 6.4 and Section 6.5. We then present experiment results in Section 6.6 and conclude the chapter in Section 6.7.

6.2 Related Work

In compiler optimization, tiling, also called blocking or chunking, is a common technique to increase data locality. It is important to distinguish two types of tiling—iteration space tiling and data space tiling. Iteration space tiling divides the given iteration space into tiles to achieve certain objectives, such as minimizing the volume of data communicated at tile boundaries and data reuse distance [BHRS08], minimizing the length of the longest path of dependent tiles [HCF99], or minimizing the size of data accessed by tiles [BF03] (with a very coarse estimate), etc. Much of this line of work, as well as some data space tiling work, focuses on the cache level and is thus inevitably heuristic, in either cost estimation or tile size capping. This is because cache is hardware-managed and its behavior depends on the machine’s runtime state. Cache associativity further complicates the problem. In contrast,
we directly optimize the total amount of data reuse, because the memory-level (as opposed to cache-level) data reuse can be precisely characterized.

Data space tiling, on the other hand, focuses on first partitioning the data (array) space into tiles and then organize computation accordingly. Park et al. [PHP03] have studied the impact of in-memory block-based tiling (as well as Morton layout) on cache and TLB performances. They only consider selected individual linear algebra algorithms, without principled method for choosing the best layout for complex, multi-operation programs. Kadayif and Kandemir [KK05] have applied data space tiling in the context of Scratch Pad Memories (SPMs), a type of software-managed cache memory. They propose to process each tile sequentially, by executing the loop iterations that access the elements in a given tile together while taking data dependences into account. This approach can be viewed as a rigid and limited form of I/O sharing compared to our approach.

Array restructuring that is more general than rectilinear tiling has also been applied at the main memory level for better data locality. Leung and Zahorjan [LZ95] propose to permute array elements with invertible mappings to make accesses exhibit spatial locality. However, this approach is heuristic and local, because spatial locality for individual loop nests and arrays are not unified into a global cost model.

From a database perspective, chunking has been adopted as an effective approach to managing multidimensional array data, for example in raster image database applications [Bau94] and Multidimensional Online Analytical Processing (MOLAP) [ORS07]. RasDaMan [Bau94] divides arrays into rectangular chunks and uses spatial indexing for fast access of chunks. We focus less on ad hoc region-based retrieval, but more on whole-matrix operations with more predictable but specific access patterns. The more recent SciDB [SBL+09, Bro10, SBW11] stores array data in potentially overlapping chunks. However, both RasDaMan and ScidB do not support cost-based automatic tile shape selection. Sarawagi and Stonebraker [SS94] propose to use the
metric of average number of blocks fetched to determine the optimal chunk shape, and also use the metric of average number of disk seeks to determine a good reordering of array axes when laying out the chunks on disk. Otoo et al. [ORS07] improve the chunk shape optimization with a better cost model. Both works require a chunk fit within a disk block. In contrast, we define chunks (blocks) to be a logical unit much larger than disk blocks. Chunks in our setting represent the unit of consecutive, batched I/O. Therefore, chunks are not constrained like disk blocks by a predetermined fixed size, but by the fact that chunks simultaneously required for any step of computation cannot exceed the amount of physical memory.

Geometric programming [DPZ67] has been used to formulate the iteration space tile size selection problem in restricted scenarios. Renganarayana and Rajopadhye [RR04] have studied the problem of selecting optimal tile size for a single fully permutable, perfectly nested, rectangular loop with uniform dependences. They have later generalized the approach to be applicable to a variety of tiling models [RR08]. We in this chapter tackle the problem of array tiling (blocking) optimization in conjunction with I/O sharing optimization. There are two key differences. First, I/O sharing complicates the cost formula and makes the optimization problem not directly describable under the geometric programming framework. Second, we optimize across multiple operators (loop nests) which may access the same array and have different blocking requirements.

Geometric programming [DPZ67] is a powerful mathematical tool for a class of optimization problems. Our optimizer is built on the solid foundation of geometric programming and its extensions. Due to the complexity of I/O sharing and blocking co-optimization, our problem cannot be directly formulated as a standard geometric program, but instead falls under the category of complementary geometric programs, which can be efficiently approximated by an iterative procedure [Duf70, AW70].
6.3 Overview

As I/O sharing and array layout jointly affect overall I/O efficiency, our optimizer considers both factors. Our basic assumption is that layout options of arrays, including appearances of the same array in different statements, are free variables unless specified otherwise. The optimizer first enumerates feasible combinations of I/O sharing opportunities, much like the way described in Section 5.5.3. A selected combination of I/O sharing opportunity implies certain equality constraints on the layouts. The optimizer enforces such constraints, together with those inherently required by individual statements or operators (which can contain multiple statements), and keep other array layouts still as free variables. This is distinct from pure I/O sharing optimization, where all array layouts are fixed and all constraints are implicitly satisfied. At the end of this optimization stage, a schedule is produced for every legal combination of I/O sharing opportunities.

The obtained schedules so far do not specify concrete values for the array layout variables yet. However, using the same procedures as in Section 5.5.4, we can still derive formulas for the memory requirement and I/O cost of any given schedule. Because an array can appear in multiple statements, there are additional intricacies in the formulation of the cost function. For such an array, we can require a homogeneous layout across different statements and solve for a globally optimal layout, or allow each statement to have independent layouts for the array and perform dynamic layout conversion, or even mix the two approaches (i.e., enforcing homogeneous layouts for certain appearances while converting for others). To this end, the optimizer uses a greedy algorithm to consider different layout conversion options and select an I/O-efficient scheme for all arrays across all statements. Each layout conversion scheme, together with the current schedule, completely determines mathematical formulas for the memory requirement and the total I/O cost. The optimizer then formulates
the optimization problem as a complementary geometric program and employs an efficient approximation algorithm to solve for the optimal array layouts. The globally best solution can then be selected among the solutions for all the schedules found in the first stage.

It is important to note that breaking the optimization into two stages as described above is merely a way to decompose the problem; it does not introduce an approximation, because the first stage does not discard any feasible combination of sharing opportunities. Also, the two stages are not independent of each other. On the one hand, the first stage by itself cannot select an optimal schedule purely based on I/O sharing opportunities, because the cost measure of a schedule depends on the yet undetermined array layouts. On the other hand, the second stage needs to put layout optimization under the context of a specific schedule. Thus, the two-stage approach accounts for the two interrelated factors.

6.3.1 Blocking Model

At any given time, an array is only allowed to have a single layout (i.e., two or more copies with different layouts are not allowed), so that we do not have to deal with data consistency issues. Because of this and layout preference differences of operators, we introduce the option of layout conversion. Furthermore, because it is difficult to model the cost of random accesses caused by incompatible layout, we always ensure the compatibility between layout and accesses by converting array layout when necessary.

In the following we focus on the block-based layout only. An array is divided into equal-size blocks, with necessary padding as determined by the chosen block size. We use the term blocking factor to refer to a block’s size in each dimension. On top of the polyhedral model introduced in Section 5.4.1, we introduce the following blocking model.
Recall that a program consists of one or more operators and each operator consists of one or more statements, each of which is enclosed by a static-control loop nest. For example, the matrix multiplication operator has a single statement, which is enclosed in a three-level nested loop. For any array $A$ and any statement $s$ that accesses the array, we require the same layout of $A$ throughout the execution of all instances of $s$. This is a reasonable requirement because a statement is always written assuming a particular layout for each of the arrays it accesses; otherwise, a block access such as $A[i, j]$ will be undefined across all iteration instances. Across the boundary of completely separated statements, however, the blocking factor of a array is allowed to change. All operators and programs should be written with this blocking model in mind.

Based on the blocking model, we further enforce the no-interleaving rule: if two statements have different blocking factors for an array that they both access, no instances of the two statements may interleave in time. Instances of statements $s$ and $s'$ do not interleave under a schedule $\Theta = \{\Theta_s \mid s \in \mathcal{S}\}$ if and only if $\forall (\vec{x}, \vec{x}') \in \mathcal{D}_s \times \mathcal{D}_{s'}$, $\Theta_s \vec{x} < \Theta_{s'} \vec{x}'$ or $\forall (\vec{x}, \vec{x}') \in \mathcal{D}_s \times \mathcal{D}_{s'}$, $\Theta_s \vec{x} > \Theta_{s'} \vec{x}'$. Without the no-interleaving rule, the layout of the commonly accessed array needs to be converted back and forth for the interleaved statement instances, which incurs too much overhead.

**Notation** For an array $A$, let $m_A$ be its dimensionality. For example, for a matrix $A$, $m_A = 2$. We use a vector $\alpha^A$ of $m_A$ components to denote its size in each dimension. Note that each $\alpha^A$ is a constant vector known as input to the optimizer. We use vector $\beta^{A,s}$ to denote the blocking factor of array $A$ when accessed in statement $s$, and the following vector to denote the number of blocks in each dimension: $\gamma^{A,s} = (\alpha_1^A / \beta_1^{A,s}, \ldots, \alpha_{m_A}^A / \beta_{m_A}^{A,s})$. Strictly speaking, the components of $\gamma^{A,s}$ should be enclosed by the ceiling function. To simplify cost modeling and optimization, we shall ignore the integer rounding problem and treat both $\beta^{A,s}$ and $\gamma^{A,s}$ as vectors
of real-number variables. In the end the solution can be rounded to integers. This approach reduces optimization complexity but still produces high-quality plans, as shown in Section 6.6.

6.3.2 Blocking Factor Equality Constraints

Just like pure I/O sharing optimization, the input to our optimizer is the polyhedral representation of a given program. Specifically, a program is specified as a sequence of operator invocations, where the polyhedral representations of the operators are already provided, by either the system (for common, built-in operators) or the user (for custom operators).

In addition, we require equality constraints on blocking factors as part of operator specifications. Such constraints can represent either 1) conditions mandatory for the correct execution of operators, or 2) manual hints set by the operator implementer. For example, in Example 8, statement $s_1$ performs matrix multiplication; so the following equality constraints must be present:

$$
\beta_{1,s_1}^{C,s_1} = \beta_{1,s_1}^{A,s_1}, \quad \beta_{2,s_1}^{C,s_1} = \beta_{2,s_1}^{B,s_1}, \quad \beta_{2,s_1}^{A,s_1} = \beta_{1,s_1}^{B,s_1}.
$$

These constraints are necessary for the correct execution of the matrix multiplication. If the developer wishes, he can also implement a special square-block matrix multiplication operator with the following additional constraints:

$$
\beta_{1,s_1}^{C,s_1} = \beta_{2,s_1}^{C,s_1}, \quad \beta_{2,s_1}^{A,s_1} = \beta_{2,s_1}^{A,s_1}.
$$

In this case, the equality constraints are manual hints set by the developer. Currently, it is the library developer’s responsibility to explicitly specify all blocking factor constraints; in the future, type-1 constraints may be automatically extracted from the operator implementation itself.

Formally, blocking factor equality constraints can be defined as follows.
Definition 6 (Blocking Factor Equality Constraint and Relation). A blocking factor equality constraint is a pair of blocking factor variables: \((\beta_{i,s}^{A,s}, \beta_{j,s'}^{A,s'})\).

A blocking factor equality relation \(\mathcal{E}\) of a given program is a set of blocking factor equality constraints. Specifically, it is a binary relation on the set of all blocking factor variables. Any \((\beta_{i,s}^{A,s}, \beta_{j,s'}^{A,s'}) \in \mathcal{E}\) means that blocking factors \(\beta_{i,s}^{A,s}\) and \(\beta_{j,s'}^{A,s'}\) must always take equal values.

Given an input program consisting of multiple operator invocations, its initial blocking factor equality relation, denoted by \(\hat{\mathcal{E}}\), can be statically computed by taking all blocking factor equality constraints (pairs of blocking factor variables) from the specifications of its constituent operators.

It is important to note that a pair of variables in a blocking factor equality relation represents an equality constraint on the two variables; it describes not a particular value assignment of the variables under a plan, but the general requirement that the two variables must always take the same value under any plan.

6.4 Stage One: I/O Sharing Enumeration

The first stage of optimization takes the polyhedral representation of a program as input and first automatically computes all dependences and I/O sharing opportunities. Feasible combinations of I/O sharing opportunities are enumerated, again using the Apriori property (Lemma 2) for effective pruning. During enumeration, a schedule is computed for each feasible combination. Although the high-level approach resembles that of pure I/O sharing optimization in Section 5.5, the key difference here is that blocking factors are not fixed constants but variables. This seemingly minor change has a significant impact: the definitions of co-accesses (Definition 3), data dependences (Definition 4), and sharing opportunities (Definition 5) do not hold any more. This is because the same array may have heterogeneous layouts throughout a program.
and the co-access relationship cannot be defined on differently shaped blocks which contain different array elements. Below we show how to address this issue and find schedules for feasible combinations of I/O sharing opportunities in this new setting.

6.4.1 Identifying I/O Sharing Opportunities and Dependences

Given an input program, we first need to know the set of all dependences and I/O sharing opportunities. However, under the heterogeneous blocking factor setting, dependences and I/O sharing opportunities are not properly defined. To see the problem, suppose there are two accesses \( a = \langle s, t, A, \Phi \rangle \) and \( a' = \langle s', t', A, \Phi' \rangle \) to the same array \( A \). For a dependence or a sharing opportunity to be defined (Definitions 4 and 5), the extent polyhedron of the co-access \( a \rightarrow a' \) must be nonempty, i.e., \( P(a \rightarrow a') \neq \emptyset \). The polyhedron, by Definition 3, consists of pairs of iteration vectors \((\vec{x}, \vec{x}')\) where \( \vec{x} \in D_s, \vec{x}' \in D_{s'}, \Phi \vec{x} = \Phi' \vec{x}' \), and \( \Theta_s \vec{x} < \Theta_{s'} \vec{x}' \) (\( \Theta_s \) and \( \Theta_{s'} \) represent the original schedules of \( s \) and \( s' \)). One of the defining criteria, \( \Phi \vec{x} = \Phi' \vec{x}' \), requires that the pair of statement instances must access the same array block. However, if \( A \) is allowed to have different blocking factors for \( s \) and \( s' \), i.e., \( \beta^{A,s} \neq \beta^{A,s'} \), then \( \Phi \vec{x} = \Phi' \vec{x}' \) only means identical logical block index; the underlying array elements covered by the blocks can be different. Fortunately, this problem can be “patched” as follows.

**Sharing Opportunities** We keep the definition of sharing opportunities and compute the set of all sharing opportunities \( \mathcal{O} \) the same way as before, as if the layout of any array is homogeneous throughout the input program. When a sharing opportunity \( a \rightarrow a' \) is selected for realization, we enforce a blocking factor equality constraint \( \beta_{a,A,s}^{a',A,s'} = \beta_{a'A,s}^{a'a',s} \) because, as we have seen, homogeneous blocking factor is a prerequisite for sharing. With such constraint, the sharing opportunity’s polyhedron \( P(a \rightarrow a') \) is well defined.
Dependences

We keep the definition of dependences and compute the set of all dependences \( D \) the same way as before, as if the layout of any array is homogeneous throughout the input program. In addition, for a dependence \( a \rightarrow a' \):

- **Case 1.** If \( a.s \) and \( a'.s \) must have the same blocking factor for \( a.A \), its polyhedron \( P(a \rightarrow a') \) is well defined and no special handling is needed.

- **Case 2.** Otherwise, we introduce a new dependence whose polyhedron subsumes \( P(a \rightarrow a') \). Specifically, we let the new dependence’s polyhedron be \( D_{a.s} \times D_{a'.s} \), so that satisfying the new dependence requires *all* instances of \( a.s \) precede *all* instances of \( a'.s \), i.e., \( a.s \) and \( a'.s \) are not interleaved. This allows blocking factors \( \beta^{a.A.a.s} \) and \( \beta^{a'.A.a'.s} \) to be different. Also, since its polyhedron is subsumed, the original dependence \( a \rightarrow a' \) can be ignored and thus the heterogeneous blocking factor problem disappears. Note that introducing the new dependences does not affect the amount of I/O because it does not hinder any activated sharing opportunity or impose additional constraints on the blocking factors.

An immediate question is: how can we determine if a dependence \( a \rightarrow a' \) falls under Case 1 or Case 2? The answer depends on the sharing opportunity subset \( Q \) selected for realization. We first use Algorithm 6 to find \( I(Q) \), the set of pairs of statements that are bound to interleave given that must be realized, and then check if \( (a.s, a'.s) \in I(Q) \). If true, Case 1 holds (by the no-interleaving rule); otherwise, Case 2 holds. Algorithm 6 works by checking if completely separating two statements conflicts with the selected subset of sharing opportunities and all dependences (on the outermost loop level, i.e., the first schedule dimension). If yes, then the two statements must interleave.

As a concrete example, consider the program shown in Figure 6.1. Suppose the optimizer is trying to find a schedule that realizes the sharing opportunity subset \( Q = \{s_2WB \rightarrow s_3RB\} \). First of all, by Algorithm 6, \( (s_2, s_3) \in I(Q) \) and
$(s_1, s_2) \in I(Q)$, intuitively because the sharing opportunity requires the interleaving of $s_2$ and $s_3$, and dependences $s_1WA \rightarrow s_2RA$ and $s_2WB \rightarrow s_1RB$ require the interleaving of $s_1$ and $s_2$. Algorithm 6 also finds that $(s_1, s_3) \in I(Q)$, as the complete separation of $s_1$ and $s_3$ would conflict with the aforementioned sharing opportunity and dependences.

**Algorithm 6: InterleavedPairs($S, D, Q$)**

**Input:** Statement set $S$, dependence set $D$, activated sharing opportunity set $Q$

**Output:** Set of statement pairs which are bound to interleave

1. $I \leftarrow \emptyset$
2. $X \leftarrow$ the polyhedron containing all integer points
3. **foreach** $a \rightarrow a' \in Q$ that is non-self **do**
   4. $X \leftarrow X \cap \{ \theta \mid \forall (\vec{x}, \vec{x}') \in P(a \rightarrow a'), \theta_{a',s}, \vec{x}' - \theta_{a,s} \vec{x} = 0 \}$
4. // Mandatory constraints imposed by dependences
5. **foreach** $a \rightarrow a' \in D$ that is non-self **do**
6. $X \leftarrow X \cap \{ \theta \mid \forall (\vec{x}, \vec{x}') \in P(a \rightarrow a'), \theta_{a',s}, \vec{x}' - \theta_{a,s} \vec{x} \geq 0 \}$
7. **foreach** $(s, s') \in S \times S$ where $s.ID < s'.ID$ **do**
   8. $X^+ \leftarrow \{ \theta \mid \forall (\vec{x}, \vec{x}') \in D_s \times D_{s'}, \theta_{s,s'}, \vec{x}' - \theta_{s,s} \vec{x} \geq 1 \}$
   9. $X^- \leftarrow \{ \theta \mid \forall (\vec{x}, \vec{x}') \in D_s \times D_{s'}, \theta_{s,s'}, \vec{x}' - \theta_{s,s} \vec{x} \leq -1 \}$
10. **if** $X \cap X^+ = \emptyset$ and $X \cap X^- = \emptyset$ **then**
11. $I \leftarrow I \cup \{(s, s'), (s', s)\}$
12. **return** $I$

We use $\hat{D}(Q)$ to denote the updated set of dependences, i.e., the union of $D$ and the set of new dependences introduced by Case 2, which is a function of $Q$.

### 6.4.2 Deriving Constraints

Recall that in pure I/O sharing optimization, given a subset of sharing opportunities $Q$ selected for realization, there are three types of constraints imposed on a schedule: dimensionality constraints, dependence constraints, and sharing opportunity constraints. Here we still have to handle these three types of constraints, in exactly the same way as discussed in Section 5.5.2, except that now the dependence set is
\( \hat{\mathcal{D}}(\mathcal{Q}) \) instead of the static \( \mathcal{D} \).

In addition to the three types of constraints above, we have to consider a new type—no-interleaving constraints.

**No-Interleaving (Separation) Constraints** If two statements \( s \) and \( s' \) satisfy the following two conditions, their instances must not interleave: 1) no sharing opportunity or dependence exists between \( s \) and \( s' \); 2) \((s, s') \notin \mathcal{I}(\mathcal{Q})\) (because \( \mathcal{I}(\mathcal{Q}) \) is symmetric, it also means \((s', s) \notin \mathcal{I}(\mathcal{Q})\)). This essentially says if two statements have the option of interleaving or not interleaving, we always require them not to interleave.

First of all, this constraint does not affect the amount of I/O because it does not hinder any activated sharing opportunity or impose additional constraints on the blocking factors. Furthermore, this constraint is actually a necessary condition for allowing heterogeneous blocking factors. Specifically, if the instances of two statements interleave under a new schedule but do not have to, the opportunity of assigning different blocking factors for any shared array accessed by the two statements is missed.

Continuing the previous example on the code shown in Figure 6.1, because there are no dependences or activated sharing opportunities between \( s_3 \) and \( s_4 \) and \((s_3, s_4) \notin \mathcal{I}(\mathcal{Q})\), we enforce the no-interleaving constraint between \( s_3 \) and \( s_4 \).

We use Algorithm 7 to find all pairs of statements for which the no-interleaving constraint should be enforced. Lines 1 finds the set of statement pairs whose instances must interleave. Line 2 then computes the set of completely separable statement pairs. Lines 3 and 4 exclude from it statement pairs between which a dependence exists. Finally, Line 5 treats the statement pairs unordered and collapses the result set.

For each statement pair \((s, s')\) returned by Algorithm 7, imposing the no-interleaving constraint on the schedule means \((\forall (\vec{x}, \vec{x'}) \in \mathbb{D}_s \times \mathbb{D}_{s'}, \Theta_s \vec{x} < \Theta_{s'} \vec{x'}) \lor (\forall (\vec{x}, \vec{x'}) \in \hat{\mathcal{D}}(\mathcal{Q}))\).
for (i=1; i<n1; ++i) {
    A[i] = B[i] + B[i-1]; // s1
    B[i] = A[i] * 2; // s2
}
for (j=0; j<n2; ++j) {
    D[i] += B[i] + C[i]; // s3
}
for (p=0; p<n3; ++p) {
    for (q=0; q<n4; ++q) {
        F[p,0] += E[p,q] * D[q]; // s4
    }
}

Figure 6.1: Example program with interleaving.

Algorithm 7: NonInterleavedPairs(S, D, Q)

Input: Statement set S, dependence set D, activated sharing opportunity set Q

Output: Set of statement pairs for which no-interleaving constraints should be applied

1 \( I(Q) \leftarrow \text{InterleavedPairs}(S, D, Q) \)
   // Completely separable statement pairs
2 \( \mathcal{P} \leftarrow \{(s,s') \in S \times S, s \neq s'\} \cup I(Q) \)
3 \( \mathcal{R} \leftarrow \{(a.s,a'.s) \in D, a.s \neq a'.s\} \cap \mathcal{P} \)
4 \( \mathcal{P} \leftarrow \mathcal{P} \setminus \{(s',s) \in \mathcal{R} \}
   // \mathcal{P} \text{ is symmetric; keep only half of it}
5 \text{return } \{(s,s') \in \mathcal{P}, s.ID < s'.ID\}

\( D_s \times D_{s'}, \Theta_s \bar{x} > \Theta_{s'} \bar{x}' \). This is similar to how dependence constraints are satisfied when the source and the target statements are allowed to have different blocking factors (Case 2), except that here either statement can precede the other. As before, the affine form of the Farkas Lemma can be used to translate the above constraint into a linear constraint on the schedule’s coefficients; for details, see Section 5.5.2.

6.4.3 Search Algorithm

The goal of this optimization stage is to enumerate feasible combinations of sharing opportunities and find a legal schedule for each, subject to dimensionality, dependence,
sharing opportunity, and no-interleaving constraints. At this stage we leave array blocking factors as variables, whose optimal values will be determined at the next stage of optimization.

The Apriori property introduced in Lemma 2 still holds and thus Algorithm 2 can still be used to efficiently enumerate candidate sets of sharing opportunities. However, the FindSchedule procedure, which attempts to find a schedule that realizes all the sharing opportunities in the candidate while satisfying all dimensionality and dependence constraints, needs to be modified to account for the new no-interleaving constraints. The new algorithm is shown in Algorithm 8. Line 2 augments the dependence set as discussed in Section 6.4.1. Lines 1 and 17–21 enforce the no-interleaving constraints.

When this stage of optimization completes, a set of legal schedules are returned, one for each feasible combination of sharing opportunities. The next stage will determine the optimal array blocking factors for each schedule and choose the globally optimal plan for execution.

6.5 Stage Two: Blocking Factors Optimization

The schedules obtained from the first stage of optimization each characterize a way of reorganizing the temporal execution order of statement instances so that certain I/O sharing opportunities can be realized, subject to various program legality constraints. Such schedules specify only the execution order of statement instances; array blocking factors are yet to be determined. Consequently, schedules cannot be compared in terms of I/O performance or memory requirement yet, as they are functions parameterized by symbolic variables, i.e., array blocking factors and array sizes measured in blocks. For example, for a statement $s_1$ which performs matrix multiplication $C = AB$, there are 12 symbolic variables: blocking factors $\beta_1^{A,s_1} = \beta_1^{C,s_1}, \beta_2^{A,s_1} = \beta_1^{B,s_1}$, and $\beta_2^{B,s_1} = \beta_2^{C,s_1}$, and array sizes measured in blocks $\gamma_1^{A,s_1} = \gamma_1^{C,s_1}, \gamma_2^{A,s_1} = \gamma_1^{B,s_1}$, and
Algorithm 8: FindSchedule*(Q, D).

Input: Sharing opportunity set Q, dependence set D

1 \( \mathcal{P} \leftarrow \text{NonInterleavedPairs}(S, D, Q) \)
2 \( D \leftarrow D(Q) \)
3 \( n \leftarrow |S| \), the number of statements
4 \( \hat{d} \leftarrow \max_{s \in S} d_s \)
5 \( Q_{sw} \leftarrow \text{self sharing opportunities of types } W \rightarrow R, W \rightarrow W \)
6 \( Q_{sr} \leftarrow \text{self sharing opportunities of type } R \rightarrow R \)
7 \( Q_{nw} \leftarrow \text{non-self sharing opportunities of types } W \rightarrow R, W \rightarrow W \)
8 \( Q_{nr} \leftarrow \text{non-self sharing opportunities of type } R \rightarrow R \)
9 \( k_1, \ldots, k_n \leftarrow 0; \Theta_1, \ldots, \Theta_n \leftarrow \emptyset \)
10 Let \( \theta^d \) denote \((\theta^d_1, \ldots, \theta^d_n)\), the \( d \)-th dimension of schedules

11 for \( d \leftarrow 1 \) to \( \hat{d} \) do

12 // Initialize the space of schedules for dimension \( d \)
13 \( X_d \leftarrow \text{the polyhedron containing all integer points} \)
14 // Weakly satisfy remaining dependence constraints
15 foreach \( a \rightarrow a' \in D \) do
16 16 \( X_d \leftarrow X_d \cap \{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{P}(a \rightarrow a'), \theta^d_{a,s,\vec{x}} - \theta^d_{a,s,\vec{x}'} \geq 0 \} \)
17 // Sharing opportunity constraints
18 \( X_d \leftarrow \text{SatisfySharingOpportunities}(X_d, Q_{sw}, Q_{sr}, Q_{nw}, Q_{nr}) \)
19 // Dimensionality constraints
20 \( X_d \leftarrow \text{SatisfyDimensionality}(X_d, n, d, k_1, \ldots, k_n, \Theta_1, \ldots, \Theta_n) \)
21 // No-interleaving constraints
22 foreach \((s, s') \in \mathcal{P}\) do
23 23 \( T \leftarrow \{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{D}_s \times \mathbb{D}_{s'}, \theta^d_{s,\vec{x}} - \theta^d_{s',\vec{x}'} < 0 \} \)
24 24 \( T \leftarrow T \cup \{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{D}_s \times \mathbb{D}_{s'}, \theta^d_{s,\vec{x}} - \theta^d_{s',\vec{x}'} > 0 \} \)
25 25 \( X_d \leftarrow X_d \cap T \)
26 26 \( \mathcal{P} \leftarrow \mathcal{P}\setminus\{(s, s')\} \)
27 // Strongly satisfy remaining dependence constraints
28 foreach \( a \rightarrow a' \in D \) do
29 29 \( T \leftarrow \{ \theta^d \mid \forall (\vec{x}, \vec{x}') \in \mathbb{P}(a \rightarrow a'), \theta^d_{a,s,\vec{x}} - \theta^d_{a,s,\vec{x}'} \geq 0 \} \)
30 if \( X_d \cap T \neq \emptyset \) then
31 31 \( X_d \leftarrow X_d \cap T \)
32 32 \( D \leftarrow D\setminus\{a \rightarrow a'\} \)
33 if \( X_d = \emptyset \) then return \( \emptyset \)
34 \( \theta^d_1, \ldots, \theta^d_n \leftarrow \text{sample a point from } X_d \)
35 foreach \( i \leftarrow 1 \) to \( n \) do \( \Theta_i \leftarrow \Theta_i \cup \{\theta^d_i\} \)
36 Find constants for the last dimensions of \( \Theta_1, \ldots, \Theta_n \)
37 return \( \Theta = \{\Theta_1, \ldots, \Theta_n\} \)
In this section, we present a cost-based search algorithm for finding the optimal blocking factors given a schedule that realizes sharing opportunity subset \( Q \). The globally I/O-optimal plan can then be selected from the solutions for the schedules of all feasible \( Q \)'s.

Note that no corresponding optimization problem was solved for pure I/O sharing optimization (Section 5.5), as all blocking factors are predetermined values in that setting.

6.5.1 Search Space

Given an input program, we assign a free blocking factor (vector) variable for each statement-array pair. In other words, if a statement \( s \) accesses array \( A \), then we have a blocking factor variable \( \beta^{A,s} \). Note that if an array appears in multiple statements, it would have independent blocking factors for each different appearance. Let \( V \) be the set of all statement-array pairs for the given program and \( \beta \) be the concatenation of \( \beta^{A,s} \) variable vectors for all \((A, s) \in V \). Similarly, based on the notation established in Section 6.3.1, let \( \alpha \) be the concatenation of all the \( \alpha^A \) constant vectors (\( \alpha^A \) is duplicated for each statement accessing \( A \)) and \( \gamma \) be the concatenation of all the \( \gamma^{A,s} \) variable vectors. Initially, the space for blocking factors and array sizes measured in blocks is:

\[
U = \{ \beta, \gamma \mid \beta \odot \gamma = \alpha, \; \beta \geq 1, \; \gamma \geq 1 \},
\]

where \( \odot \) denotes component-wise vector product. Note that we have relaxed the integer requirement. Strictly speaking, all variables have to be integers and we have to deal with the intricacy of imperfect blocking. However, a strict integer formulation makes the problem we are about to formulate significantly harder [BKVKH07]. Following a commonly used heuristic method [BKVKH07], we relax the integer requirement and round the obtained solution in the end.

Given the schedule for a subset of sharing opportunities \( Q \), we compute the
**current blocking factor equality relation**, \( \hat{E}(Q) \), by augmenting the initial blocking factor equality relation \( \check{E} \) with pairs of blocking factor variables of statements that are bound to interleave:

\[
\hat{E}(Q) \leftarrow \check{E} \cup \{ (\beta_i^a.A,a.s, \beta_i^{a'.A,a'.s}) \mid \text{share an array must interleave for all dims} \} 
\]

Since each pair of blocking factor variables in \( \hat{E}(Q) \) must be equal, the final blocking factor search space \( S \) can be defined as:

\[
S = U \cap \{ \beta, \gamma \mid \forall (\beta', \beta'') \in \hat{E}(Q), \beta = \beta' \}. 
\]

**Layout Conversion** It is important to realize that for a given schedule, blocking factors by themselves do not completely specify an executable plan. To see the problem, suppose two completely separated statements \( s_1 \) and \( s_2 \) both access array \( A \) and have independent blocking factor variables \( \beta^{A,s_1} \) and \( \beta^{A,s_2} \). If the optimizer chooses different values for \( \beta^{A,s_1} \) and \( \beta^{A,s_2} \), at least one of the statements will have accesses that are incompatible with the array layout if no layout conversion is performed. Because it is difficult to model the cost of random accesses caused by incompatible layout, we always ensure the compatibility between layout and accesses by converting array layout when necessary. In the context of block-based layout, which is the focus of this work, it means prior to the execution of a statement \( s \), for any array \( A \) accessed by \( s \), \( A \) must have a blocked-based layout with the same blocking factor as specified by \( \beta^{A,s} \).

It now becomes clear that the search space must include another dimension: layout conversion choices. That is, we consider the option of allowing or forbidding the layout conversion of an array between completely separated statements that both access the array. Formally, given a schedule, for an array \( A \), construct a
Figure 6.2: Graphs for finding layout conversion decision points.

graph $G_A = (V_A, E_A)$ as follows. Let the vertex set $V_A$ contain a vertex $v_s$ for each statement $s$ that accesses $A$. Add to $E_A$ an edge between two vertices $v_s$ and $v_{s'}$ if the search space $S$ in (6.3) specifies that $\beta^{A,s} \Rightarrow \beta^{A,s'}$. By the construction of the schedule, each connected component in the obtained graph $G_A$ represents a subset of statements whose instances interleave and whose blocking factors for $A$ must match. Now perform edge contraction for all the edges in $G_A$ so that each connected component is contracted to a single vertex. Let $V'_A$ be the new vertex set. For any $v \in V'_A$, denote by $p(v)$ the set of original vertices which have been contracted into $v$. Add a directed edge $(u, v)$ to $E'_A$ if there exist $x \in p(u)$ and $y \in p(v)$ such that $x$ precedes $y$ under the given schedule and no other statement is scheduled in between.

Note that by construction, $G'_A$ is always a “chain”. In the resulted directed graph $G'_A = (V'_A, E'_A)$, each vertex represents a consecutive block of statements which share the same blocking factor for $A$, and each edge represents a layout conversion decision point, with the direction of the edge denoting the execution order under the given schedule. As a concrete example, for the schedule shown in Figure 6.1, the graphs $G_B$ and $G'_B$ for array $B$ are shown in Figure 6.2.

Based on the above graph definition, the search space can be further refined as follows. For any array $A$ accessed in the program, for any edge $(u, v)$ in $G'_A$ consider two options:
• Allow a layout conversion of $A$ between $u$ and $v$. In this case, for any $s \in p(u), s' \in p(v)$, $\beta^{A,s}$ and $\beta^{A,s'}$ stand as independent variables as they were before.

• Forbid a layout conversion of $A$ between $u$ and $v$. In this case, it is required that $\beta^{A,s} = \beta^{A,s'}$ for any $s \in p(u), s' \in p(v)$.

6.5.2 Search Algorithm

Given a fixed schedule, we show how to search for the optimal blocking factors in the search space described previously in Section 6.5.1. We first enumerate layout conversion options, as they directly affect the final blocking factor search space. For each layout conversion configuration, we consider all constraints on the blocking factors and the constraint on the total memory consumption, and cast the I/O optimization problem into a complementary geometric program [AW70], for which an approximate solution can be efficiently computed. The obtained solution is only for the particular layout conversion configuration under the given schedule. The globally best solution can then be selected from these local solutions.

Enumerating Layout Conversion Options

As discussed in Section 6.5.1, each edge in a layout conversion decision point graph $G'_A = (V'_A, E'_A)$ represents an option of allowing or forbidding a layout conversion. Thus, the total number of cases we need to consider is $O(2^{\sum_A |E'_A|})$. It may be infeasible to completely enumerate all the cases. Below we propose a greedy algorithm (Algorithm 9) which reduces the search space size to $O(\sum_A |E'_A|)$.

In Algorithm 9 we process arrays in order of increasing size. This is based on the heuristic that the blocking schemes of larger arrays are likely to influence the I/O performance more than smaller arrays. For each array $A$, we consider the layout conversion decision points in scheduled execution order (i.e., topological order in the graph $G'_A$) and decide which option is better: allowing or forbidding a layout
Algorithm 9: DecideLayoutConversions($\mathcal{X}', S$) for a given schedule

**Input:** Array set $\mathcal{X}'$, blocking factor search space $S$ as defined in (6.3)

**Output:** Optimal cost and the corresponding blocking factors

1. $\hat{c} \leftarrow \infty$
2. sort $\mathcal{X}'$ by array size
3. foreach $A \in \mathcal{X}'$ do
   4. foreach $(u, v) \in E'_A$ in topological order do
      5. $s \leftarrow$ choose a statement from $p(u)$
      6. $s' \leftarrow$ choose a statement from $p(v)$
      7. $c_1, \beta_1 \leftarrow$ OptimizeBlockingFactors($S$)
      8. $c_2, \beta_2 \leftarrow$ OptimizeBlockingFactors($S \cap \{\beta, \gamma \mid \beta^{A,s} = \beta^{A,s'}\}$)
      9. if $c_1 > c_2$ then
         10. $S \leftarrow S \cap \{\beta, \gamma \mid \beta^{A,s} = \beta^{A,s'}\}$
         11. if $\hat{c} > c_2$ then $\hat{c} \leftarrow c_2, \hat{\beta} \leftarrow \beta_2$
      else
         12. if $\hat{c} > c_1$ then $\hat{c} \leftarrow c_1, \hat{\beta} \leftarrow \beta_1$

14. return $\hat{c}$ and $\hat{\beta}$, best solution for the given schedule

conversion for $A$. If the latter is better, we remember the constraint for this decision point (Line 10); otherwise, there is no additional constraint to be kept. Finally, we return the best cost and corresponding blocking factors for the given schedule.

Note that other heuristics can be used to guide the order in which layout conversion decision points are processed. For example, we can proceed in iterations and in each iteration decide the layout conversion that results in the most I/O benefit given the decisions already made. In practice, we find the heuristic implemented in Algorithm 9 works well.

*Optimizing Blocking Factors*

**Optimization Objective** We employ a cost model that accounts for four factors of I/O costs: the total read volume in bytes $R$, the total write volume in bytes $W$, the number of block I/O requests $B$, and the cost of layout conversion operations $T$. Note that $R$, $W$, and $B$ account for the program execution only, excluding layout
conversion operations. The total I/O cost, measured by time, can be modeled as a linear function of the factors: 

\[ C = w_R R + w_W W + w_B B + T. \]

Because blocks are the unit of I/O and array layouts always match accesses, the weights \( w_R \) and \( w_W \) directly correspond to the inverse of sustained sequential read and write rates, respectively, while \( w_B \) directly corresponds the average time required by a random disk seek. The layout conversion cost \( T \) requires modeling on its own, as we shall discuss below. By applying the same procedure as described in Section 5.5.4, we can compute \( R \), \( W \), and \( B \) in the form of polynomials in \( \beta \) and \( \gamma \).

We allow the use of different layout conversion algorithms, as long as the I/O cost \( T \) can be modeled by a posynomial function in \( \beta \) and \( \gamma \) (we will discuss posynomials shortly). For example, one way to perform layout conversion is to treat it as a sorting problem and use an algorithm such as external merge sort, whose I/O cost is a constant multiple of array size. For experiments in Section 6.6, we assume a simple conversion cost model: converting the layout of an array is accomplished by a pass of read and a pass of write of the array. The results show that this model is sufficient for making optimization decisions. Under this model, \( T \) can be expressed as a polynomial of \( \beta \) and \( \gamma \). Specifically, for each layout conversion of array \( A \) from blocking factor \( \beta^{A,s} \) to \( \beta^{A,s'} \), we add a polynomial to \( T \):

\[
(w_R + w_W) \prod_{i=1}^{m_A} \alpha_i^A + w_B \left( \prod_{i=1}^{m_A} \gamma_i^{A,s} + \prod_{i=1}^{m_A} \gamma_i^{A,s'} \right). \quad (6.4)
\]

The first term represents the element-level cost (which is actually a constant) while

\[ \text{To be exact, the results are piecewise quasi-polynomials, polynomial expressions with coefficients that may depend periodically on the values of the variables. They come up in the counting of integer points in a polyhedron. For most linear algebra operations, operand matrices are traversed in regular ways and the resulting quasi-polynomials are actually polynomials. For other unusual operations, quasi-polynomials can be well approximated by polynomials [MV08]. Henceforth, we treat expressions which may be actual piecewise quasi-polynomials as simply polynomials.}\]

\[ \text{The value of the log term (number of passes) in the I/O cost formula for external merge sort can be completely determined given the array size, memory size, and disk block size.}\]
the second term represents the block-level cost. Overall, the total I/O cost \( C \) can be expressed as a polynomial of \( \beta \) and \( \gamma \), which we write as \( C(\beta, \gamma) \).

Note that \( R \), \( W \), and \( B \) are all piecewise polynomials, because they depend on the configuration of the \( \beta \), \( \gamma \) parameters. For example, consider the program in Example 8, but with two sharing opportunities realized: \( s_1RA \rightarrow s_2RA \) and \( s_1RB \rightarrow s_1RB \). First note that we must have \( \beta^{A,s_1} = \beta^{A,s_2} \), or equivalently, \( \gamma^{A,s_1} = \gamma^{A,s_2} \). Our optimizer is able to compute \( R \) as a piecewise polynomial. One case, when \( \gamma^{A,s_1} \geq 2 \), \( \gamma^{A,s_1} \geq 2 \), is

\[
R = \gamma^{A,s_1} \alpha^{D,s_2} \alpha^{A,s_1} \\
+ (\gamma^{A,s_1} - 1) \alpha^{D,s_2} \alpha^{A,s_1} \\
+ \gamma^{D,s_2} \alpha^{A,s_1} \alpha^{A,s_1} \\
+ (\gamma^{A,s_1} - 1) \alpha^{A,s_1} \alpha^{B,s_1} \\
+ \alpha^{A,s_1} \alpha^{B,s_1}
\]

reading \( D \) in \( s_2 \)
reading \( E \) in \( s_2 \)
reading \( A \) in \( s_1 \) and \( s_2 \) (shared) \( \quad (6.5) \)
reading \( C \) in \( s_1 \)
reading \( B \) in \( s_1 \) (shared).

The \(-1\) terms in (6.5) are because the blocks of \( C \) and \( E \) initialized in memory in the beginning, not read from disk.

**Memory Constraint** We are given a fixed amount of physical memory \( M \). At any point during the execution of a schedule, the memory usage must not exceed \( M \). Since the memory is used to hold array blocks, the memory requirement at time \( \tau \) is a function of \( \tau \), \( \beta \) and \( \gamma \), which we denote by \( m(\tau, \beta, \gamma) \). The memory requirement of the entire schedule is then defined as \( M(\beta, \gamma) = \max_{\tau} m(\tau, \beta, \gamma) \). As shown in Section 5.5.4, \( M(\beta, \gamma) \) can be computed as a piecewise expression, with each piece expressed as the maximum of a set of polynomials in \( \beta \) and \( \gamma \). Continuing the previous example, for the program in Example 8 with two realized sharing opportunities, \( s_1RA \rightarrow s_2RA \) and \( s_1RB \rightarrow s_1RB \), our optimizer obtains the memory...
The expression (6.6) can be understood as follows. First of all, the realization of \( s_1RA \rightarrow s_2RA \) requires merging the two loop nests, while the realization of \( s_1RB \rightarrow s_1RB \) requires that consecutive iterations of the innermost loop access the same \( B[k,j] \) block, which means the original \( i \) loop around \( s_1 \) must be moved to the innermost level. Because the sharing of \( A \)'s blocks, the original \( k \) loop around \( s_2 \) must also be moved to the innermost level so that the two loop nests can be merged. To compute the memory requirement, first note that for \( s_1 \) and \( s_2 \) to successfully execute, the two input blocks and the output block must co-exist in memory for each iteration. This contributes to the common terms for the two cases. The sharing opportunity \( s_1RA \rightarrow s_2RA \) requires the shared \( A \) block to be kept in memory. However, because the sharing happens only within the innermost level of the merged loop, it does not cause additional memory usage. The sharing opportunity \( s_1RB \rightarrow s_1RB \) requires the reuse of \( B \)'s block between consecutive iterations of the innermost loop. Because the execution of \( s_1 \) and \( s_2 \) is interleaved, the shared \( B \)'s block adds additional memory usage to \( s_2 \)'s iteration instances. The additional memory size equals \( B \)'s block size, which is \( \beta_1^{R,s_1} \beta_2^{R,s_1} = \beta_2^{A,s_1} \beta_2^{R,s_1} \). Of course, this is true only if \( s_1RB \rightarrow s_1RB \) exists, i.e., \( \beta_2^{A,s_1} \geq 2 \). Thus, the memory requirement is expressed in two cases as shown in (6.6). Also, the above analysis is correct no matter which statement precedes the other in the innermost loop. The detailed cases are illustrated in Figure 6.3.

The above memory constraint caps the memory usage of statement instances only; how about the memory usage of layout conversion operations? First note that layout conversion and I/O sharing never co-occur. This is because layout conversion...
occurs only at operator boundaries, i.e., after a loop nest completely finishes and before another starts, while I/O sharing is realized only within the innermost level of a loop nest (for non-self sharing opportunities) or across consecutive iterations of the innermost level of a loop nest (for self sharing opportunities). This means memory can be separately capped for statement execution (as above) and layout conversion. We can then perform layout conversion operations one at a time, by calling the conversion algorithm (e.g., external merge sort) with the given memory cap as parameter, so that memory usage is implicitly capped and no additional memory constraint is necessary.

**A Geometric Programming Formulation** We next cast the problem of finding the \((\beta, \gamma) \in S\) that minimizes \(C(\beta, \gamma)\), the I/O cost of a schedule (with layout
conversion decisions made), subject to the memory constraint \( M(\beta, \gamma) < \underline{M} \), as a complementary geometric program (CGP) [AW70], an extension of geometric program (GP) [DPZ67]. We first introduce some definitions and notations.

**Definition 7 (Monomials).** Let \( x_1, \ldots, x_n \in \mathbb{R}^+ \) be \( n \) real positive variables. A real valued function \( g \) of the vector \( x = (x_1, \ldots, x_n) \), with the form

\[
g(x) = cx_1^{a_1}x_2^{a_2} \cdots x_n^{a_n},
\]

where \( c \in \mathbb{R}^+ \) and \( a_i \in \mathbb{R} \), is a monomial of the variables \( x_1, \ldots, x_n \).

Examples of monomials include \( 3 \), \( x \), and \( 1.5x^2y^{-1.2} \). Monomials are closed under multiplication and division.

**Definition 8 (Posynomials).** Let \( x_1, \ldots, x_n \in \mathbb{R}^+ \) be \( n \) real positive variables. A real valued function \( f \) of the vector \( x = (x_1, \ldots, x_n) \), with the form

\[
f(x) = \sum_{k=1}^{K} g_k(x) = \sum_{k=1}^{K} c_k x_1^{a_{1k}}x_2^{a_{2k}} \cdots x_n^{a_{nk}},
\]

where \( g_1(x), \ldots, g_K(x) \), are monomials, is a posynomial of the variables \( x_1, \ldots, x_n \).

Examples of posynomials include \( x \), \( x^2 + 2xy + y^2 \), and \( 1/x + 1/y \). Posynomials are closed under addition and multiplication. In addition, dividing a posynomial by a monomial results in another posynomial.

A geometric program in standard form is an optimization problem in the following form:

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) \leq 1, \quad i = 1, \ldots, p, \\
& \quad g_i(x) = 1, \quad i = 1, \ldots, m,
\end{align*}
\]

(6.7)

where \( f_0 \) and \( f_1, \ldots, f_p \) are posynomials and \( g_1, \ldots, g_m \) are monomials. It is implicitly required that all components of \( x \) are positive.
A GP can be converted into a convex optimization problem and solved efficiently using the barrier-based interior-point method in polynomial time [NN94]. For a recent and detailed introduction, refer to Boyd and Vandenberghe’s book [BV04].

Next we show how the blocking factor optimization problem can be cast into a GP. First consider the constraints. The search space $S$ defined in (6.3) entails size constraints $\beta \circ \gamma = \alpha$ and blocking factor equality constraints, which can all be characterized by monomial equalities, and also $\beta, \gamma \geq 1$, which can be characterized by posynomial inequalities. The layout conversion decisions entail blocking factor equalities only, which can also be written as monomial equalities. The memory constraint needs to be handled together with the I/O cost optimization objective, as they are both piecewise expressions.

Recall that the memory requirement $M(\beta, \gamma)$ is a piecewise expression, whose definition varies for different subsets of the variable domain. For each subdomain, it is the maximum of one or more polynomials, as shown in (6.6). Similarly, the cost function $C(\beta, \gamma)$ is also a piecewise expression, whose definition varies for different subsets of the variable domain. For each subdomain, it is a polynomial. To optimize a piecewise objective subject to piecewise constraints, we break the problem into pieces based on a common partition of the variable domain and solve a subproblem for each subdomain. The final solution can be obtained by comparing the solutions for each subproblem and choosing the one that minimizes the objective. Specifically, let $D_M$ and $D_C$ be the set of subdomains of $M(\beta, \gamma)$ and $C(\beta, \gamma)$, respectively. For each $D \in \{X \cap Y \mid X \in D_M, Y \in D_C\}$, we minimize $C(\beta, \gamma) |_{D}$ subject to $M(\beta, \gamma) |_{D} \leq M$, $(\beta, \gamma) \in D$, and the search space constraints mentioned in the previous paragraph, where $|_{D}$ denotes the definition of the piecewise expression on subdomain $D$. Such defined subproblems do not involve piecewise expressions any more. The objective becomes a polynomial while the memory requirement becomes a maximum of one or more polynomials. For the memory constraint, we can replace
max\{f_1(\beta, \gamma), \ldots, f_L(\beta, \gamma)\} \leq M \text{ by } L \text{ constraints: } f_1(\beta, \gamma) \leq M, \ldots, f_L(\beta, \gamma) \leq M.

It is easy to see that each \(f_i(\beta, \gamma)\) is a posynomial, because by definition, the memory requirement is always the sum of a set of array blocks’ sizes. It only remains to turn the objective \(C(\beta, \gamma) \mid_D\) into a posynomial and the constraints defining the subdomain \(D\) into standard-form GP constraints.

**Handling the Subdomain** The subdomain we consider here for an optimization subproblem is obtained by intersecting a subdomain of \(C(\beta, \gamma)\) and a subdomain of \(M(\beta, \gamma)\). The computation process described in Section 5.5.4, which uses basic operations implemented in [Ver10], ensures that each subdomain of \(C(\beta, \gamma)\) and \(M(\beta, \gamma)\) is actually a polyhedron. Thus, the intersected subdomain is also a polyhedron. This means each constraint defining the subdomain must take the form

\[
c_1^T \beta + c_2^T \gamma + c_3 \cdot 1 \geq 0, \tag{6.8}
\]

where \(c_1, c_2\) are constant vectors and \(c_3\) is a constant scalar. Group the terms on the left-hand side with positive and negative coefficients into two separate polynomials, \(f^+\) and \(f^-\), so that (6.8) can be rewritten as \(f^+ + f^- \geq 0\). There are two cases based on the number of terms in \(f^-\).

- **Case 1.** If \(f^-\) has no terms, the constraint trivially holds and can be ignored as all variables are positive already.

- **Case 2.** If \(f^-\) has at least one term, then there must be at least a term in \(f^+\), too; otherwise the left-hand side would have a negative value. Putting the positive and negative terms on different sides gives \(-f^- \leq f^+\). Note that both sides are posynomials. If \(f^+\) is actually a monomial, multiplying \(1/f^+\) to both sides gives a posynomial constraint as required by the GP standard form: \(-f^-/f^+ \leq 1\). If \(f^+\) is not a monomial, but a posynomial with at least two monomial terms, we cannot directly cast it into a standard-form constraint.
Fortunately, there is an iterative method to cope with this kind of constraint, which we will discuss shortly.

**Handling the Objective** In order to cast our optimization problem into a GP, the optimization objective must be a posynomial. However, the objective may contain negative monomial terms. To understand how this can happen, let us first revisit the linear sharing model introduced in Section 5.5.1. With any program schedule, every statement instance is mapped to a specific execution time, which defines a linear ordering of all statement instances. I/O sharing only happens between consecutive accesses to the same array block in time order. Restricting sharing to only consecutive accesses avoids the problem of over-counting reuses. For \( R \rightarrow R \) and \( W \rightarrow R \) sharing opportunities, I/O is saved for the target (read) accesses, while for \( W \rightarrow W \) sharing opportunities, I/O is saved for the source (write) accesses.

We examine the components of the cost objective one by one. It is obvious from (6.4) that the layout conversion cost \( T \) is a posynomial. The other cost components, \( R \), \( W \), and \( B \), are similar in nature and below we show how to handle \( R \). \( R \) can be decomposed as follows. For each array \( A \) and each statement \( s \) that reads \( A \), compute the read volume of \( A \) by \( s \) in the original program, minus the amount saved by the realized sharing opportunities, and denote the result by \( r(A, s, \beta, \gamma) \), or simply \( r(A, s) \). It follows that \( R = \sum_{A,s} r(A, s) \). Note that each \( r(A, s) \) is guaranteed to evaluate to a positive value, because the shared amount by definition can never exceed the original amount. However, it may not be a posynomial, due to the subtraction operation. Let us again group the terms in \( r(A, s) \) according to their signs and rewrite it as \( r(A, s) = r^+(A, s) + r^-(A, s) \), where \( r^+(A, s) \) contains all terms with positive coefficients and \( r^-(A, s) \) contains those with negative coefficients. There are two cases based on the number of terms in \( r^-(A, s) \).

- **Case 1**. If \( r^-(A, s) \) has no terms, \( r(A, s) \) is a posynomial. No further treatment
is necessary.

- **Case 2.** If $r^-(A, s)$ has at least one term, then there must be at least a term in $r^+(A, s)$, too; otherwise $r(A, s)$ would have a negative value. Introduce a new variable $t_{A,s}$ and let $t_{A,s} = r(A, s) = r^+(A, s) - (-r^-(A, s))$. Replace $r(A, s)$ with $t_{A,s}$, a posynomial, in the objective. We now need to ensure the constraint $t_{A,s} = r^+(A, s) - (-r^-(A, s))$. However, doing so would result in a posynomial equality constraint, which does not fit in the GP framework. Instead, we replace it with an equivalent constraint $t_{A,s} \geq r^+(A, s) - (-r^-(A, s))$. The equality is guaranteed to be achieved as long as the original problem has an optimal solution, because $t_{A,s}$ appears as a single term in the objective function and does not appear in any other constraints, which means $t_{A,s}$ will be minimized. The new constraint can be rewritten as \( r^+(A, s)/(t_{A,s} + (-r^-(A, s))) \leq 1 \). Because the denominator is a posynomial with more than one terms, the constraint cannot be cast into GP standard form. This situation is the same as the one that occurs in Case 2) of handling subdomains.

The program we obtain from the above translation process is a standard GP as in (6.7) with additional constraints of the form

\[
h_i(x)/d_i(x) \leq 1, \quad i = 1, \ldots, s, \tag{6.9}
\]

where $h_i(x)$ and $d_i(x)$ are posynomials and $d_i(x)$ has at least two monomial terms. Such a program is a *complementary geometric program* (CGP) [AW70], and can be solved by an iterative procedure, where each iteration approximates the rational function on the left-hand side of (6.9) with a posynomial and solves the resulted standard GP [AW70, Duf70], as described below.

Let the denominator posynomial in (6.9) be $d_i(x) = \sum_j u_{ij}(x)$, where each $u_{ij}(x)$
is a monomial. By the geometric inequality,

\[ d_i(x) = \sum_j u_{ij}(x) \geq \prod_j \left( \frac{u_{ij}(x)}{\delta_j} \right)^{\delta_j}, \]

for any \( \delta_j \geq 0 \) and \( \sum_j \delta_j = 1 \) (assuming \( (u_{ij}(x)/\delta_j)^{\delta_j} = 1 \) if \( \delta_j = 0 \)). A condensation of (6.9) is

\[ h_i(x) \prod_j \left( \frac{u_{ij}(x)}{\delta_j} \right)^{-\delta_j} \leq 1, \quad i = 1, \ldots, s, \tag{6.10} \]

which is now in standard GP form. Replacing constraints (6.9) in the CGP with (6.10) gives a regular GP, whose optimal solution is a conservative estimate of the original CGP’s, and can be solved efficiently. Initially, choose arbitrary values for the weights \( \delta_j \) and condense the original CGP. Once an approximate solution \( x' \) is obtained, form a new condensed program with new weights defined by \( \delta_j = u_{ij}(x')/d_i(x') \). Since \( x' \) also satisfies the new condensed program, the solution for the new condensed program will be no worse than \( x' \). Thus we can continue this iterative process and obtain successively improved estimates for the optimal solution of the original CGP.

As shown in [AW70] and commented in [BKVKH07], generally speaking, the above method may not converge (but often does), and may converge to a local optimal. Nevertheless, it is a widely useful method for solving CGPs that arise from real-world engineering optimization problems. In practice, we have not observed non-convergence for the linear algebra programs we consider. To cope with potential non-convergence, we can terminate the search after some prescribed number of iterations. We having built an iterative CGP solver on top of the CVXOPT\(^5\) Python package, which implements a regular GP solver.

Local solutions for individual subdomains are compared to produce the best solution for a given schedule and a layout conversion scheme. The globally best

\(^5\) http://abel.ee.ucla.edu/cvxopt/
solution can then be selected from solutions for different combinations of schedules and layout conversion schemes. The output of the second optimization stage is thus a schedule satisfying a particular set of sharing opportunities, with a particular layout conversion scheme and a set of values for the blocking factors, which together minimize the total I/O cost and satisfy the various constraints considered above.

6.6 Experiments

Setup and Storage Scheme The experiment setup and the storage scheme are the same as described in Section 5.6. The only difference is that now we model the cost of disk seeks.\(^6\) We have performed benchmarks on our disk and found that the average seek time is \(w_B = 16\text{ms}\). Upon the execution of a plan, all input arrays are prepared in the same layout as specified by the plan.

6.6.1 Two Matrix Multiplications (1)

We first test our optimizer with a program consisting of two matrix multiplications: \(C = AB\), \(E = AD\). This is the same program as tested in Section 5.6.2. In Section 5.6.2, only I/O sharing is optimized, whereas in this section both I/O sharing and array layouts are optimized. For easy comparison, we use the same matrix sizes as Configuration A given in Table 5.4, but leave the blocking factors as variables. We also set the memory cap to 954MB, the same as the maximum amount of memory required by any plan for the same program in Section 5.6.2 (to have a common ground for comparison). There are 9 sharing opportunities in the program, as already shown in Table 5.3.

Our optimizer produces 40 plans for this input program. Figure 6.4(a) summarizes the predicted performance of all plans. The best plan and the worst plan’s I/O costs

\(^6\) In Section 5.6 blocking factors are predetermined and large enough so that the total seek overhead can be ignored. Here, blocking factors are variables, so we must guard against small blocks for operators such as matrix addition whose I/O volume alone is independent of blocking factors.
can vary by a factor of 3. Note that all plans fully utilize the given memory, so the comparison is fair.

To demonstrate the accuracy of our cost model and also the ratio of I/O cost to CPU cost, we select the following four plans for execution (for this purpose, it does not matter which plans are chosen). Plan 0 enables no sharing opportunities, Plan 35 enables o1–o3, Plan 38 enables o1 plus o5–o8, and Plan 39 enables o5–o8. Intuitively, Plan 39 uses two separate loop nests to accumulate $C$ and $E$ blocks in memory. Plan 38 in addition merges the two loop nests and shares the read of $A$. Plan 35 shares the I/O to $B$ and $D$ instead of $C$ and $E$. Note that these plans are numbered 0, 3, 2, and 1 in the previous experiment (see Figure 5.5 in Section 5.6.2).

![Figure 6.4: Two matrix multiplications $C = AB$; $E = AD$: performance of plans.](image)

Figure 6.4(b) shows the characteristics of the four selected plans and Table 6.1 shows the matrix blocking factors chosen by the plans. First note that our optimizer predicts I/O cost with very high accuracy; the average error is only 3.6%. It is also interesting to note that although Plan 38 enables one more sharing opportunity, o1 ($s_1RA \rightarrow s_2RA$), its I/O cost is not lower but higher (by 7 seconds) than Plan 39’s. This is because realizing o1 requires keeping the shared blocks of $C$ and $E$ simultaneously in memory, reducing the amount of memory available for the actual multiplication.
Figure 6.5: Two matrix multiplications $C = AB$; $E = AD$: impact of joint optimization.

### Table 6.1: Matrix blocking factors of selected plans for $C = AB$; $E = AD$.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Plan 0</th>
<th>Plan 35</th>
<th>Plan 38</th>
<th>Plan 39</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$ in $s_1$</td>
<td>$5101 \times 9701$</td>
<td>$521 \times 16550$</td>
<td>$10571 \times 622$</td>
<td>$10750 \times 439$</td>
</tr>
<tr>
<td>$A$ in $s_2$</td>
<td>$5101 \times 9701$</td>
<td>$521 \times 16550$</td>
<td>$10571 \times 622$</td>
<td>$10750 \times 439$</td>
</tr>
<tr>
<td>$B$</td>
<td>$9701 \times 5101$</td>
<td>$16550 \times 3462$</td>
<td>$622 \times 5441$</td>
<td>$439 \times 10750$</td>
</tr>
<tr>
<td>$D$</td>
<td>$9701 \times 5101$</td>
<td>$16550 \times 3462$</td>
<td>$622 \times 5441$</td>
<td>$439 \times 10750$</td>
</tr>
</tbody>
</table>

**Impact of Joint Optimization** To better illustrate the importance of joint optimization of I/O sharing and array layouts, we plot the plans from Figure 5.5 and plans produced here side by side in Figure 6.5(a). Plans in each group realize the *same* subset of sharing opportunities, but have significantly different I/O costs. This is because the plans from Figure 5.5 have manually picked array layouts whereas here the optimizer actively searches for the layouts that work best in conjunction with the selected sharing opportunities. For all four plans, the joint optimization results in more efficient plans, with a speedup factor up to 1.9. A more important observation is that the optimal set of sharing opportunities can vary depending on if blocking factor optimization is enabled or not. Without blocking factor optimization, sharing opportunities $o1$ plus $o5$–$o8$ (as picked by Plan 38) are optimal, while sharing opportunities $o5$–$o8$ alone (as picked by Plan 39) is far from optimal. However, with
blocking factor optimization, the cost of Plan 39 dramatically decreases, making sharing opportunities o5–o8 the optimal choice.

So far we have fixed the memory cap at 954MB. We next vary the memory cap and compare the best plans chosen by the two approaches above. Figure 6.5(b) shows the result. With a smaller memory cap, pure I/O sharing optimization (with predetermined blocking factors) chooses o0, o1, and o4 as the best sharing opportunity subset. Any other subset that leads to better I/O efficiency requires more memory space and are rejected. In contrast, the joint optimizer is able to scale down arrays’ blocking factors to fit the smaller memory while selecting the same subset of sharing opportunities as the one selected when the memory cap is 954MB. This automatic scaling also happens when the memory cap is increased. With predetermined blocking factors, a larger memory cap cannot directly bring performance improvement when no more I/O sharing opportunities can be exploited.

The above experiments show that I/O sharing and array layouts are two tightly coupled factors which cannot be separately optimized. Also, hand-picking layouts usually lead to sub-optimal performance: blocking factors that are too small may under-utilize the memory; blocking factors that are too large may prevent the most beneficial I/O sharing opportunities due to memory constraints. The optimal choice is best left to be made by an automatic optimizer like ours.

6.6.2 Two Matrix Multiplications (2)

We next test our optimizer on the program shown in Example 8. It also consists of two matrix multiplications, but the order of operands are different from that in the previous experiment. Here A serves as the left operand of the first multiplication and the right operand of the second. Table 6.2 summarizes the sizes of the matrices involved. Memory usage is capped at 954MB.

Figure 6.6(a) plots the predicted I/O cost of all 39 plans generated by the optimizer.
Table 6.2: Matrix sizes for $C = AB; E = DA$.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimensions</th>
<th>Total size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$48000 \times 48000$</td>
<td>17.2GB</td>
</tr>
<tr>
<td>$B$</td>
<td>$48000 \times 30000$</td>
<td>10.8GB</td>
</tr>
<tr>
<td>$C$</td>
<td>$48000 \times 30000$</td>
<td>10.8GB</td>
</tr>
<tr>
<td>$D$</td>
<td>$42000 \times 48000$</td>
<td>15.0GB</td>
</tr>
<tr>
<td>$E$</td>
<td>$42000 \times 48000$</td>
<td>15.0GB</td>
</tr>
</tbody>
</table>

Plans are numbered in order of decreasing predicted I/O cost. Although the plans all fully utilize the given memory cap, their I/O cost differs dramatically, up to a factor of 3. Figure 6.6(b) plots the actual and predicted performance of three selected plans. Plan 5 enables no sharing opportunities, and sets blocking factors $\beta^{A,s_1} = \beta^{D,s_2} = (5032, 9856)$, $\beta^{A,s_2} = \beta^{B,s_1} = (9856, 5032)$. Plan 32 enables four sharing opportunities: $s_1RA \rightarrow s_2RA$, $s_2RD \rightarrow s_2RD$, $s_1WC \rightarrow s_1RC$, $s_1WC \rightarrow s_1WC$, and sets blocking factors $\beta^{A,s_1} = \beta^{A,s_2} = (14552, 533)$, $\beta^{B,s_1} = (533, 3273)$, $\beta^{D,s_2} = (4582, 14552)$. Plan 38 enables a different set of four sharing opportunities: $s_1WC \rightarrow s_1RC$, $s_1WC \rightarrow s_1WC$, $s_2WE \rightarrow s_2RE$, $s_2WE \rightarrow s_2WE$, and sets blocking factors $\beta^{A,s_1} = \beta^{D,s_2} = (10728, 439)$, $\beta^{A,s_2} = \beta^{B,s_1} = (439, 10728)$. Both Plan 5 and Plan 38 convert the layout of $A$ between the two matrix multiplications. Again, the prediction is highly accurate; the average error is 3.8%.

One interesting observation is that although Plan 5 enables no sharing opportunity, it is not the worst plan; Plan 0, which enables $s_1RA \rightarrow s_2RA$ and $s_2RA \rightarrow s_2RA$, is. This is because some sharing opportunities, such as $s_1RA \rightarrow s_2RA$, can place restrictions on the choices of blocking factors and lead to suboptimal overall performance. This is also the reason why the best plan in the experiment, Plan 38, does not enable $s_1RA \rightarrow s_2RA$. 

167
6.6.3 Linear Regression: A Complete Program

We next test the linear regression program previously tested in pure I/O sharing optimization (Section 5.6.3). We use the same matrix size setup as before, but leave their blocking factors undetermined. The memory cap is set to 2148MB, the maximum amount of memory used by any plan in Section 5.6.3 (to have a common ground for comparison). With joint optimization of I/O sharing and matrix layouts, the plans considered by the optimizer are plotted in Figure 6.7(a). Again, each point represents a plan satisfying a particular subset of sharing opportunities, with the best
blocking factors chosen for the matrices. Although all utilizing the given memory cap, the I/O costs of the best plan and the worse plan vary by a factor of almost 2. In Figure 6.7(b), we show the predicted and actual performance of three selected plans. Plan 0 enables no I/O sharing opportunities. Plan 271 keeps $U$ and $V$ in memory. Plan 1727 in addition shares the reads of $X$ for the first two matrix multiplications and eliminates the materialization of all intermediate results. The blocking factors chosen by the three plans are listed in the table below.

The same three plans have been examined in Section 5.6.3, with hand-picked blocking factors. To show the effectiveness of cost-based blocking factor optimization, we compare the two sets of results, with and without blocking factor optimization, in Figure 6.8. The figure shows that the hand-picked blocking factors (“predetermined layout (a)”) happen to be optimal for all three plans, as the I/O costs they result in
are about the same as the results obtained by blocking factor optimization ("optimized
layout"). For the non-optimal Plan 0 and 271, our optimizer actually picks blocking
factors that result in slightly worse performance (not noticeable in the plot). There
are mainly two reasons for this. First, integer approximation introduces some errors
in cost estimation. Second, when a matrix block has size (e.g., $63636 \times 4000$) that
does not align with the disk block size (1000 double values in our setting), more disk
blocks need to be accessed at matrix block boundaries. However, as shown in this
experiment, the errors are negligible for large arrays.

Had we manually chosen a different set of blocking factors in Section 5.6.3, the
comparison to automatic blocking factor optimization would be different. For example,
suppose we enlarge the row dimension of each block of the input matrices $X$ and
$Y$ (see Table 5.5) by 25 times and shrink the column dimension of each block by 25
times, so that the total size of each block of $X$ and $Y$ remains the same. The blocking
factors of the remaining matrices are implied. Note that this choice of blocking
factors, corresponds to clustering data by variables instead of observations. With this
predetermined layout, we run pure I/O sharing optimization and plot selected plans in
Figure 6.8 as "predetermined layout (b)". Again, plans are renumbered so that plans
from different experiments with the same number activate the same subset of sharing
opportunities. For this layout configuration, the best plan happens to activate the
same subset of sharing opportunities as Plan 1727 does (thus renumbered to 1727).
Because of the non-optimal choice of blocking factors, overall the I/O performance is
vastly worse than our automatic, cost-based solution. This demonstrates the necessity
of blocking factor optimization.

6.7 Conclusion

I/O sharing and array layouts are two important factors that can greatly affect a
program’s overall I/O performance. In this chapter we have presented a cost-based
optimizer for both factors. By taking a two-stage approach, the optimizer makes search tractable while considering a rich plan space. By exploiting the structure of the optimization problem, we are able to cast the blocking factor selection problem into a known mathematical framework and efficiently solve it. Experiments show that our optimizer produces accurate estimates for the I/O costs of candidate plans, and finds plans with equal or less I/O costs than a pure I/O sharing optimizer.
Recent advances in science and technology have enabled collection of massive amounts of data in various application domains, such as science and business. Harnessing these large, high-resolution datasets can provide deep understandings of scientific problems as well as predictive insights for business organizations. However, the exponential growth in the amount of data has also created an urgent and difficult technical challenge. Many existing data analysis tools still assume that datasets fit in main memory; when applied to massive datasets, they becomes unacceptably inefficient due to excessive I/O operations. The open-source statistical computing environment, R, is one such example.

In this dissertation, we have proposed RIOT, a framework for transparent and efficient I/O for statistical analysis of massive array data. In particular, we have focused on array-centric programs involving block-based array accesses describable by nested loops, and provided novel solutions at the language, storage, and program optimization levels.

RIOT keeps R’s language-level design and makes it completely transparent to users how efficient I/O is supported. In Chapter 3 we have described the mechanisms
RIOT uses to interface the host language environment and the out-of-core data processing backend. This approach not only achieves transparency, but also enables high-level optimizations. A proof-of-concept RIOT implementation using a relational database as the backend shows RIOT’s advantages over plain R and conventional databases.

Chapter 3 also reveals the need for an efficient array-oriented storage and processing engine. To this end, we have proposed LAB-tree as a solution for storing arrays. LAB-tree uses linearization to provide flexible array layouts, and automatically adapts to varying sparsity across parts of an array and over time. In Chapter 4 we have discussed the design of LAB-tree, and how it unique splitting strategies and update batching policies help it achieve significantly higher I/O efficiency than existing solutions.

I/O operations in a program can often be rescheduled to create opportunities of I/O sharing (data reuse). In Chapter 5 we have introduced an I/O sharing optimizer for RIOT based on this observation. Building on the polyhedral model, the optimizer strikes the balance between feasibility and flexibility of representation and optimization. It considers a rich space of plans and is able to accurately determine their legality, I/O costs, and memory requirements.

In Chapter 6 we further treat array layouts as a dimension of the search space and jointly optimize I/O sharing and array layouts. The extended optimizer incorporates both I/O sharing and array layouts into a unified cost model, and selected the best plan by cost-based search. Experiments show that the optimizer is able to find plans with even higher I/O efficiency than if I/O sharing is optimized alone.

With its language-level transparency, efficient array storage support, and effective I/O sharing and array layout optimization, RIOT is a holistic solution for I/O-efficient statistical computing. Based on the contributions made in this dissertation, there are many opportunities for future work. For example, since the polyhedral model is
quite general, it would be interesting to extend the applicability of RIOT to programs
involving a mix of array operations and database- or MapReduce-style operations.
In a distributed setting, inter-node communication is as important as I/O. The
techniques developed in RIOT may be applied to optimize communication and data
partitioning as well.
Appendix A

I/O Lower Bounds

A.1 I/O Lower Bound for Matrix Multiplication

Problem: Given two matrices $A(m \times l)$ and $B(l \times n)$, compute their product $C = AB$. The available memory can hold $M$ scalar numbers. Suppose $\min\{ml, ln, mn\} \gg M$. $A$ and $B$ initially reside on disk. Each disk block can store $B$ numbers. Assuming any algorithm requiring $\Theta(lmn)$ scalar multiplications can be used, give an optimal schedule that minimizes the amount of I/O in terms of disk blocks read/written.

Solution: We first give a lower bound for the amount of I/O and then give a schedule that achieves this lower bound.

At any time, the memory contains elements from $A$, $B$ or $C$. During $M$ number of I/Os, the number of distinct elements from the three matrices must be $\leq 2M$ (elements initially in memory plus newly fetched ones). Let the number of distinct, active elements from $A$, $B$, and $C$ that contribute to the matrix multiplication be $a$, $b$ and $c$, respectively. An active element is one that either participates in element multiplication (elements in $A$ and $B$), or gets assigned (elements in $C$). Further suppose that the $a$ elements from $A$ are taken from $U$ rows, each having $u_i$ elements,
where \( i = 1, \ldots, U \). Similarly, suppose that the \( b \) elements from \( \mathbf{B} \) are taken from \( V \) columns, each having \( v_j \) elements, where \( j = 1, \ldots, V \). Note that if active elements from row \( i \) of \( \mathbf{A} \) and column \( j \) of \( \mathbf{B} \) are multiplied and contribute to \( \mathbf{C}[i, j] \), then \( \mathbf{C}[i, j] \) must appear in memory. Thus we have

\[
a = \sum_{1 \leq i \leq U} u_i \\
b = \sum_{1 \leq j \leq V} v_j \\
c \geq UV \\
a + b + c \leq 2M.
\]

Now consider the number of scalar multiplication operations, which we denote by \( Z \), that can be performed with the above constraints. Note that \( \mathbf{A}[h, i] \) and \( \mathbf{B}[j, k] \) are multiplied if and only if \( i = j \). Thus, for each row \( i \) of \( \mathbf{A} \) and each column \( j \) of \( \mathbf{B} \), the maximum number of scalar multiplications is \( \min(u_i, v_j) \). Therefore,

\[
Z = \sum_{1 \leq i \leq U} \sum_{1 \leq j \leq V} \min(u_i, v_j).
\]

We want to maximize \( Z \) in order to minimize the amount of I/Os. Without loss of generality, let us assume \( u_1 \leq u_2 \leq \cdots \leq u_U \) and \( v_1 \leq v_2 \leq \cdots \leq v_V \). Let all \( u_i \)'s that are greater than \( v_1 \) and less than \( (v_1 + v_V)/2 \) be \( D = \{u_d, \ldots, u_{d+r}\} \), and all \( u_i \)'s that are between \( (v_1 + v_V)/2 \) and \( v_V \) be \( E = \{u_e, \ldots, u_{e+s}\} \). Let all \( u_i \)'s greater than \( v_V \) be \( F = \{u_f, \ldots, u_{f+l}\} \). Now consider the following change: We remove \( v_1 \) and \( v_V \) and introduce two identical numbers \( (v_1 + v_V)/2 \). There are still \( V \) numbers, with their sum unchanged. Consider the resulting \( Z \). All elements in \( D \) now contribute to \( Z \) once because of the change from \( v_1 \) to \( (v_1 + v_V)/2 \). All elements in \( E \) have their contribution to \( Z \) decremented by one because of the change from \( v_V \) to \( (v_1 + v_V)/2 \).
So the net change in $Z$ is

$$\Delta Z = \sum_{u \in D} u - \sum_{u \in E} u - v_1(|D| + |E| + |F|) - v_V|F|$$

$$+ 2\frac{v_1 + v_V}{2}(|E| + |F|)$$

$$= \left( \sum_{u \in D} u - v_1|D| \right) + \left( v_V|E| - \sum_{u \in E} u \right)$$

$$= \sum_{d \leq i \leq d+r} (u_i - v_1) + \sum_{e \leq i \leq e+s} (v_V - u_i)$$

$$\geq 0.$$  

If we repeat this operation, i.e., replacing the smallest and the largest numbers with two copies of their mean, we can eventually make all $v_i$’s equal. The same procedure can be done to $u_i$’s. Thus, when $Z$ achieves its maximum, we have $u_1 = \cdots = u_U = a/U$ and $v_1 = \cdots = v_V = b/V$. Now consider maximizing $Z$ under this condition. Let $u = a/U$ and $v = b/V$. Without loss of generality, we assume $u \leq v$, or equivalently $aV \leq bU$, in which case $Z$ becomes

$$Z = UV \cdot u. \quad (A.1)$$

We want to maximize (A.1) subject to:

$$u \leq v$$

$$uU + vV + UV \leq 2M$$

Let $f = UVu + \alpha(v - u) + \beta(2M - uU - vV - UV).$ Using Lagrange multiplier, we
know $Z$ gets its maximum when $\nabla f = 0$; that is:

\[
\frac{\partial f}{\partial u} = UV - \alpha - \beta U = 0 \tag{A.2}
\]

\[
\frac{\partial f}{\partial v} = \alpha - \beta V = 0 \tag{A.3}
\]

\[
\frac{\partial f}{\partial U} = uV - \beta u - \beta V = 0 \tag{A.4}
\]

\[
\frac{\partial f}{\partial V} = uU - \beta v - \beta U = 0 \tag{A.5}
\]

\[
\frac{\partial f}{\partial \alpha} = v - u = 0 \tag{A.6}
\]

\[
\frac{\partial f}{\partial \beta} = 2M - uU - vV - UV = 0 \tag{A.7}
\]

By (A.4), (A.5), and (A.6) we have $(u - \beta)V = (u - \beta)U$. Note that we cannot have $u = \beta$, because plugging it into (A.4) would yield $u = \beta = 0$. So we must have $U = V$. Now combining with (A.2) and (A.3), we get $U = 2\beta$. Plugging this into (A.4), we get $u = U$. Coming back to (A.7), we have $a = uU = \frac{2}{3}M$. We conclude that $Z \leq (\sqrt{2M/3})^3$.

Recall that $Z$ is the number of scalar multiplications performed during $M$ elements of I/Os. Because the algorithm for computing $C$ requires a total of $lnn$ scalar multiplications, the total number of element I/Os should be at least

\[
\frac{lnn}{\left(\sqrt[3]{\frac{2M}{3}}\right)}M = \Theta \left(\frac{lnn}{\sqrt{M}}\right).
\]

In the best case, we can service these element I/Os with $\Theta \left(\frac{lnn}{BV\sqrt{M}}\right)$ number of block I/Os, which is the lower bound.

**Tightness** The above lower bound is asymptotically tight, as can be shown by the following two schedules. However, to the best of our knowledge, we are not aware of
a schedule that achieves the bound with the exact given constant. In practice, any
schedule with a constant close enough to $(\sqrt{2/3})^{-3} \approx 1.84$ can be considered good
enough. Below we introduce two schedules that are asymptotically optimal but with
different constants.

Schedule 1 The proof above gives a natural schedule that achieves the lower
bound asymptotically. The condition that yields the minimum is $a = b = c$ and
$U = u_1 = \cdots = u_U = V = v_1 = \cdots = v_V$. We can divide $A$, $B$ and $C$ into submatrices
of size $p \times p$, where $p = \sqrt{M/3}$. The memory can hold exactly three such submatrices.
For each block $C_{i,j}$ in $C$, we perform the block matrix algorithm by loading and
multiplying pairs of submatrices from $A$ and $B$ in turn—{$A_{i,1}, B_{1,j}, \ldots, A_{i,k}, B_{k,j}$}—
to compute $C_{i,j}$. For each $C_{i,j}$, we read in $\frac{2p^2 l}{B} \frac{1}{p}$ blocks, and write out the result,
which has $\frac{p^2}{B}$ blocks. There are $\frac{mn}{p^2}$ submatrices in $C$. So the total number of I/Os
(blocks) is

$$
\left( \frac{2p^2 l}{B} \frac{1}{p} + \frac{p^2}{B} \right) \frac{mn}{p^2} = \frac{2\sqrt{3}lmn}{B\sqrt{M}} + \frac{mn}{B}.
$$

The constant for the dominating term is $2\sqrt{3} \approx 3.46$.

Schedule 2 Another feasible schedule is to divide $A$, $B$ and $C$ into submatrices of
size $p \times 1$, $1 \times p$, and $p \times p$, respectively. To fully utilize memory, we have $p^2 + 2p = M$,
or $p = \sqrt{M + 1} - 1$. Using the same execution method as in Schedule 1, the total
number of I/Os (blocks) is

$$
\frac{2lmn}{(\sqrt{M + 1} - 1)B} + \frac{mn}{B}.
$$

The constant for the dominating term is 2, better than Schedule 1’s.
A.2 I/O Lower Bound for a Chain of Matrix Multiplications

**Problem:** Given $n$ matrices $A_i (d_i \times d_{i+1})$, $i = 1, \ldots, n$, compute their product $C = A_1 A_2 \cdots A_n$. The available memory can hold $M$ scalar numbers. Suppose any matrix involved in the computation of $C$ has size $\gg M$. Let $N$ be the number of scalar multiplications performed in order to compute $C$. All input matrices and the result matrix reside on disk. Each disk block can store $B$ numbers. Give an optimal schedule that minimizes the amount of I/O in terms of disk blocks read/written.

**Solution:** We first bound the number of scalar multiplications that can happen during $M$ elements of I/Os. During $M$ elements of I/Os, the number of distinct elements that appear in memory must be $\leq 2M$. Define an *active* matrix multiplication to be $Z = XY$, where $X$, $Y$, and $Z$ can be input matrices or intermediate results, and some elements of $X$ and $Y$ are in memory and produce some elements of $Z$. There could be multiple active matrix multiplications in progress. Suppose $m_1, \ldots, m_k$ memory resource is allocated to each active matrix multiplication, so that $m_1 + \cdots + m_k \leq 2M$. According to the proof in Appendix A.1, at most

$$\left( \frac{2m_1}{3} \right)^{\frac{3}{2}} + \cdots + \left( \frac{2m_k}{3} \right)^{\frac{3}{2}} \leq \left( \frac{2m_1}{3} + \cdots + \frac{2m_k}{3} \right)^{\frac{3}{2}} = \left( \frac{4M}{3} \right)^{\frac{3}{2}}$$

multiplications can happen during $M$ number of I/Os.

It follows that the minimum number of element I/Os for computing $C$ is

$$\frac{N}{\left( \frac{4M}{3} \right)^{\frac{3}{2}}} M = \Theta \left( \frac{N}{\sqrt[3]{M}} \right).$$

Therefore, the I/O lower bound in terms of blocks is $\Theta \left( \frac{N}{B\sqrt[3]{M}} \right)$.

The above reasoning also indicates that the lower bound is achieved by doing one active matrix multiplication at a time, and by applying the optimal schedule in
Appendix A.1 to each matrix multiplication. As an example, suppose we want to compute $C = A_1A_2A_3$, and among all possible ways of parenthesizations, $A_1(A_2A_3)$ takes the minimum number of scalar multiplications. The optimal I/O performance is attained by first using any of the two schedules in Appendix A.1 to compute $T = A_2A_3$ and materialize $T$. Following that, $C = A_1T$ is similarly computed.
Appendix B

Theorems for Splitting Strategies and Update Batching Policies

Theorem 3. No deterministic online local splitting strategy has a competitive ratio less than 2.

Proof. Given any splitting strategy $\Sigma$, we construct an insertion sequence that results in a tree with overall density of at most $1/2$, i.e., with at least twice the number of leaves produced by the optimal offline algorithm. We start by inserting any $\kappa + 1$ records, causing the first split. There are two cases. 1) If any of the two result leaves has a range with length no greater than $\kappa$, we mark both nodes inactive; 2) otherwise, we mark the result leaf with fewer records inactive and the other one active. After the split, we pick any active leaf and keep inserting records into it until the next split. The process is repeated until there is no active leaf left.

In the end, all leaves are inactive. Those generated by Case 2 all have density no greater than $1/2$. Each inactive leaf generated by Case 1 is paired with exactly one other inactive leaf. These two leaves are resulted from splitting a leaf and have not be inserted into since. Thus, their combined density is (arbitrarily close to) $1/2$. 

182
Therefore, the overall density of the tree is at most 1/2.

**Lemma 4.** Every leaf produced by a no-dead-space splitting strategy must have a range whose length is divisible by \( \kappa \); i.e., all splitting points picked are multiples of \( \kappa \).

**Proof.** For brevity, if a leaf has a range length not divisible by \( \kappa \), we call the leaf *misaligned*. No matter how we choose the splitting point, if the original node is misaligned, at least one of the resulting leaves after the split is misaligned. If a splitting strategy ever produces any misaligned leaf that is not full, we keep inserting into it, and continue inserting into any misaligned leaf subsequently generated, until a misaligned leaf \( \ell \) with range length less than \( \kappa \) is produced. At this point, no future insertion sequence can eliminate the dead space in \( \ell \). Therefore, the splitting strategy does not have the no-dead-space property.

For an array with range \([0, m\kappa]\) for some integer \( m \), all splitting points picked by a no-dead-space splitting strategy must be multiples of \( \kappa \) because the range of the first leaf starts with 0.

**Theorem 5.** Any no-dead-space splitting strategy has a competitive ratio of at least 3.

**Proof.** Given any no-dead-space splitting strategy, we construct an insertion sequence that results in at least three times the number of leaves produced by the optimal offline algorithm. Suppose the tree's key domain spans range \([0, m\kappa]\). We call each interval \([i\kappa, (i+1)\kappa)\) a *unit interval*; there are \( m \) unit intervals. Without loss of generality, assume \( \kappa + 1 = 3k \) where \( k \in \mathbb{N} \).

Starting with an empty tree, insert \( k \) records each into the first (0-th), last ((\( m-1 \))-th), and middle ((\( \frac{m-1}{2} \))-th) unit intervals, causing the first split. By Lemma 4, records in the same unit interval will never be separated. It is thus clear that after the first split, one leaf contains \( k \) records and the other contains \( 2k \). We leave the smaller
leaf intact. We call the larger leaf $\ell$. The range of $\ell$ covers $\frac{m+1}{2}$ unit intervals, the first and last of which contain $k$ records each. We then insert $k$ records into the unit interval right in the middle of $\ell$, resulting in the second split with the configuration as the first, except that total range is halved. This process can be repeated recursively until the larger leaf contains only 2 unit intervals. In the end, all except one leaf contain $k = \frac{m+1}{3}$ records each. Therefore, the number of leaves is (arbitrarily close to) three times that of the optimal offline algorithm.

**Theorem 6.** Split-aligned has a competitive ratio of 3.

**Proof.** Given any insertion sequence on an initially empty tree $T$, we construct a split tree $S$, which captures the history of node splits. We maintain a bijection $f$ between $T$’s leaves (including those that were leaves at one point but were later split and thus do not exist in $T$ any more) and $S$’s nodes by the following procedure: Initially $T$ contains only an empty leaf $\ell$ and $S$ contains a single node, $f(\ell)$; whenever a leaf $\ell$ of $T$ splits into $\ell_1$ and $\ell_2$, we create two new nodes $f(\ell_1)$ and $f(\ell_2)$ in $S$ and add them as $f(\ell)$’s left and right children, respectively. By construction, for any node $x$ in $T$, $x$ is a leaf in $T$ iff $f(x)$ is a leaf in $S$. To simplify notation, we will use $x$ to mean $f(x)$ when there is no confusion. For any node $x$ in $S$, we denote its parent in $S$ by $p(x)$, and its sibling in $S$ by $s(x)$. In both $T$ and $S$, if $x$ is a leaf, we denote the leaf to its left by $\alpha(x)$, and the leaf to its right by $\beta(x)$.

In the following, we show that all leaves of $T$ (or $S$) can be put into groups so that each group has density at least $\frac{1}{3}$. Therefore, for any insertion sequence, the number of leaves generated by the split-aligned is at most three times of that generated by the optimal offline strategy, establishing a competitive ratio of 3. The tightness is given by Theorem 5.

We group leaves of $S$ as follows:

- **Case A.** For any leaf $x$ in $S$ whose sibling is also a leaf, we put $x$ and its sibling
into one group. Such a group has at least $\kappa + 1$ records, so the density is at least $\frac{1}{2}$.

• **Case B.** For any leaf $x$ in $S$ whose sibling is not a leaf, if $\rho(x) \geq \frac{1}{3}$, we put $x$ in one group by itself.

• **Case C.** For any leaf $x$ in $S$ whose sibling is not a leaf, if $\rho(x) < \frac{1}{3}$, we put $x$ to an existing group. Specifically, if $x$ is the left (or right) child of its parent in $S$, we add it to a group to its right (or left, respectively).

It remains to be shown that in Case C above, adding $x$ to an existing group never decrease the group’s density to below $\frac{1}{3}$.

Without loss of generality, assume $x$ is a left child. Consider the point when $p(x)$ was split into $x$ and $s(x)$. Suppose $p(x)$ has range $[hk, jk)$ and the splitting point was $ik$ ($h < i < j$). Let $\varrho$ be the density of $x$ right after this split; i.e., inside $p(x)$ before the split, interval $[hk, ik)$ contained $\varrho$ records. Clearly, $\varrho \leq \rho(x) < \frac{1}{3}$. We define the companion interval of $x$, denoted $C(x)$, to be the unit interval $[ik, (i + 1)\kappa)$, i.e., one that is adjacent to $x$’s range and anchored to the splitting point. Note that $C(x)$ must contain the $\varrho$-$\kappa$-th to the $((1 - \varrho)\kappa - 1)$-th records in $p(x)$ before the split. Had there existed a possible splitting point $i'\kappa > ik$ between the $\varrho$-$\kappa$-th and the $((1 - \varrho)\kappa - 1)$-th records of $p(x)$, $i'\kappa$ would have been a better splitting point than $ik$ according to split-aligned, as two resulting leaves would have been more balanced in their numbers of records. As a unit interval, $C(x)$ will never be split, so $C(x)$ always contains at least $(1 - 2\varrho)\kappa > \frac{1}{3}\kappa$ records. Figure B.1 illustrates the definition of companion interval.

After the split, $s(x)$ contained $C(x)$. In $S$, $\beta(x)$ is $s(x)$’s left-most descendant, which must contain $C(x)$ as its left-most unit interval; thus, $\beta(x)$ has more than $\frac{1}{3}\kappa$ records. We have two cases.
**Case 1.** If $s(\beta(x))$ is not a leaf, then $\beta(x)$ is in a group $G$ just by itself (Case B); we simply add $x$ to $G$. The other nodes that can possibly join $G$ are $\beta(\beta(x))$ and $\alpha(x)$, but we argue that they will not. As a left child, $\beta(\beta(x))$ can only join the group to its right. If $\alpha(x)$ is a right child, it can only join the group to its left. If $\alpha(x)$ is a left child, then $x$ must be $s(\alpha(x))'$s leftmost descendant and thus contain $C(\alpha(x))$. However, for $\alpha(x)$ to be added to an existing group, we must have $\rho(\alpha(x)) < \frac{1}{3}$ and $C(\alpha(x))$ must contain more than $\frac{1}{3} \kappa$ records, contradicting the fact that $\rho(x) < \frac{1}{3}$. Therefore, the density of $G$, which contains only $x$ and $\beta(\beta(x))$, is at least $\frac{\rho + 1 - 2\rho \kappa}{2\kappa} > \frac{1}{3}$.

**Case 2.** If $s(\beta(x))$ is a leaf, then $\beta(x)$ and $s(\beta(x))$ are already in a group $G$ (Case A). We add $x$ to $G$ and claim that no other node can join $G$. Let $y = \beta(x)$ and $z = s(y) = \beta(y)$; i.e., $G$ contains $x$, $y$, and $z$. The other nodes that can possibly join $G$ are $\alpha(x)$ and $\beta(z)$. As a right child, $\alpha(x)$ cannot join a group to its right. Now consider $\beta(z)$. If $\beta(z)$ is a left child, it will never be added to a group to its left. If $\beta(z)$ is a right child, there are two cases based on how long the path from $s(x)$ to $y$ is. Note that this path consists of left branches only.

- **Case 2a.** If the path length is at least 2, then $\beta(z)$ is the right child of $u = p(p(z))$ (Figure B.2(a)). Consider the time when $u$ split. Obviously $x$ was created before $u$, so $u$ contained $C(x)$. For $\beta(z)$ to be split off, $\beta(z)$ had to contain no fewer records than $C(x)$; otherwise the split would not be the most balanced one. Therefore, $\rho(\beta(z)) > \frac{1}{3}$, and hence $\beta(z)$ will not be added to any existing group.

- **Case 2b.** If the path length from $s(x)$ to $y$ is 1, then $p(y) = p(z) = s(x)$. In this case, $\beta(z)$ is found by traversing up the tree from $z$ until the first left branch is taken, to $u = p(\beta(z))$, and then taking the sibling right branch down (Figure B.2(b)). It is obvious that $\beta(z)$ was created before $x$. Note that all nodes along the path from $s(\beta(z))$ to $z$ contain $C(\beta(z))$ as their rightmost
unit interval. Consider $p(x)$ on this path, which contained at least three unit intervals (because of its three descendants). In order for $\beta(z)$ to be added to $G$, we must have $\rho(\beta(z)) < \frac{1}{3}$. In that case, however, $C(\beta(z))$, or $p(x)$’s rightmost unit interval, would contain more than $\frac{1}{3}\kappa$ records, contradicting the fact that $x$, with fewer than $\frac{1}{3}\kappa$ records, was split off $p(x)$. Therefore, $\beta(z)$ will not be added to any existing group.

To conclude Case 2, $G$ contains $x$, $y$, and $z$, and has density at least $\frac{\rho_0 + \kappa}{3\kappa} > \frac{1}{3}$.  

Lemma 7. Any flushing policy is $O(K)$-competitive.

Proof. Consider any page $P$. OPT has to flush $P$ at least once per $K$ requests for $P$. Any policy flushes $P$ at most once per request for $P$.  

Theorem 8. ALL is $\Omega(\sqrt{K})$-competitive.
Proof. Let $R = \lfloor \sqrt{K} \rfloor$. We construct a request sequence consisting of $R/2$ phases, each with $K$ requests. Each such phase has one request for each of pages $P_1, P_2, \ldots, P_R$, plus $K - R$ requests for page $P_0$. ALL incurs a cost of $R + 1$ per phase.

On the other hand, a better policy $\Pi$ would keep all requests for $P_1, \ldots, P_R$ until the end of the last phase, and in the meantime, flush $P_0$ as needed. The number of requests for $P_1, \ldots, P_R$ in the buffer increases up to $(R/2)R \leq K/2$, so at least $K/2$ space is always available for buffering $P_0$ requests. Therefore, $\Pi$ needs to flush $P_0$ at most twice per phase, and flush $P_1, \ldots, P_R$ once at the end of the $R/2$ phases. Therefore, $\Pi$’s amortized cost per phase is at most $2 + \frac{R}{R/2} = 4$. 

Definition 9 (c-recency). A flushing policy is $c$-recent if it has the following property: If there has been no request for page $P$ among the past $cK$ requests, then no request for $P$ is currently buffered.

Theorem 9. Any $c$-recent flushing policy is $\Omega(\sqrt{K}/c)$-competitive.

Proof. Consider the request sequence from the proof of Theorem 8. We modify it as follows. For each phase, we add $cK$ requests for page $P_0$ at the end of the phase. These new requests would force any $c$-recent algorithm to flush $P_1, P_2, \ldots, P_R$, incurring a cost of $R$ per phase for these pages.

On the other hand, a better policy $\Pi$ would keep flushing $P_0$ as needed, incurring at most $2c$ flushes of $P_0$ per phase since at least $K/2$ space is available for buffering $P_0$ requests.

Corollary 10. LRU is $\Omega(\sqrt{K})$-competitive.

Proof. By Theorem 9, it suffices to show that LRU is 1-recient. For any page $P$ currently buffered, consider the $K$ requests immediately following the most recent request for $P$. If none of these $K$ requests are for $P$, LRU must flush them later than $P$. $P$ cannot be buffered after these $K$ requests because that would require the buffer to hold $P$ plus the $K$ requests, exceeding its capacity.
Theorem 11. ALL is $O(\sqrt{K} \log K)$-competitive.

Proof. Divide the request sequence into phases of length $K$. Without loss of generality, assume that the total number of requests is a multiple of $K$. Suppose there are $m$ pages. Let $m$-dimensional vector $r^{(t)}$ denote the collection of requests in Phase $t$, where the $i$-th component of the vector, denoted $r_i^{(t)}$, specifies the number of requests for $P_i$.

We consider the behavior of a policy $\Pi$. $\Pi$ has a buffer of size $2K$ and mimics OPT as follows. Let vector $s^{(t)}$ denote the state of OPT’s buffer at the beginning of Phase $t$, where $s_i^{(t)}$ specifies the number of requests for $P_i$ buffered by OPT at that time. $\Pi$ buffers all these requests (using at most $K$ space) throughout Phase $t$, together with all requests (using $K$ space) received during Phase $t$. At the end of Phase $t$, $\Pi$ flushes whatever requests that OPT has flushed during Phase $t$. Clearly, $\Pi$ and OPT incur the same cost over the entire request sequence.

Define a potential function over the current buffer state $s$ as: $\Phi(s) = \sum_{i=1}^{m} \ln(1 + s_i)$. Let $\Delta$ be the total potential increase due to incoming requests over the course of executing $\Pi$ on the entire input sequence. Each flush of $\Pi$ involves at most $2K$ requests and lowers the potential by at most $\ln(2K + 1)$. The potential is 0 at the beginning and the end of the entire request sequence. Therefore, $\Delta/C^{OPT} = O(\log K)$, where $C^{OPT}$ is the same as the total number of flushes by $\Pi$.

Consider the change in potential in Phase $t$. We divide the requests in this phase into two groups:

- Requests for cold pages, where a page $P_i$ is cold in Phase $t$ if $r_i^{(t)} \geq 1$ and

$$\ln \left( \frac{1 + s_i^{(t)} + r_i^{(t)}}{1 + s_i^{(t)}} \right) < \frac{1}{\sqrt{K}};$$

i.e., the total potential increase in Phase $t$ due to $P_i$ requests is less than $1/\sqrt{K}$.
• Requests for *hot pages*, where the total potential increase in Phase $t$ due to requests to each hot page is at least $1/\sqrt{K}$.

Let $C^{\text{ALL}}_t$ denote the cost of ALL incurred in Phase $t$. Clearly, $C^{\text{ALL}}_t = q_c + q_h$, where $q_c$ is the number of cold pages and $q_h$ is the number of hot pages in Phase $t$. Note that for a cold page $P_i$,

$$\frac{1}{\sqrt{K}} > \ln \left( \frac{1 + s_i^{(t)} + r_i^{(t)}}{1 + s_i^{(t)}} \right) = \ln \left( 1 + \frac{r_i^{(t)}}{1 + s_i^{(t)}} \right)$$

$$> \frac{r_i^{(t)}}{1 + s_i^{(t)}} \left( 1 + \frac{r_i^{(t)}}{1 + s_i^{(t)}} \right) = \frac{r_i^{(t)}}{1 + s_i^{(t)} + r_i^{(t)}},$$

which implies $s_i^{(t)} > (\sqrt{K} - 1)r_i^{(t)} - 1$. It follows that $q_c < \frac{K}{\sqrt{K} - 2}$, because

$$K \geq \sum_{P_i \text{ is cold}} s_i^{(t)} > (\sqrt{K} - 1) \sum_{P_i \text{ is cold}} r_i^{(t)} - q_c \geq (\sqrt{K} - 2)q_c.$$

Let $\Delta_t$ denote the total potential increase due to incoming requests in Phase $t$. We have $\Delta_t \geq q_h/\sqrt{K}$. At the same time, note that there exists at least one page $P_j$ with $r_j^{(t)} \geq \max(1, s_j^{(t)})$ (otherwise, $\sum_{i=1}^m r_i^{(t)} < \sum_{i=1}^m s_i^{(t)} = K$, a contradiction), so

$$\Delta_t \geq \ln \left( \frac{1 + s_j^{(t)} + r_j^{(t)}}{1 + s_j^{(t)}} \right) \geq \ln \left( 1 + \frac{r_j^{(t)}}{1 + s_j^{(t)} + r_j^{(t)}} \right)$$

$$\geq \ln 1.5 > 0.4.$$

Therefore,

$$\frac{C^{\text{ALL}}_t}{\Delta_t} \leq \frac{q_c + q_h}{(0.4 + q_h/\sqrt{K})/2} = 2\sqrt{K} \cdot \frac{q_c + q_h}{0.4\sqrt{K} + q_h}$$

$$< 2\sqrt{K} \cdot \frac{\frac{K}{\sqrt{K} - 2} + q_h}{0.4\sqrt{K} + q_h} = O(\sqrt{K}).$$

Finally, let $C^{\text{ALL}}$ denote the cost of ALL over the entire request sequence. $C^{\text{ALL}}/\Delta = (\sum_t C^{\text{ALL}}_t)/(\sum_t \Delta_t) = O(\sqrt{K})$. We have already shown $\Delta/C^{\text{OPT}} = O(\log K)$, so $C^{\text{ALL}}/C^{\text{OPT}} = O(\sqrt{K} \log K)$. \hfill \square
Theorem 12. LP is $\Omega(K)$-competitive.

Proof. Consider the request sequence

$$P_1, P_2, P_3, \ldots, P_{K-2}, P_0, P_0, P_0^*,$$

where $P_0^*$ denotes repeating $P_0$ requests. After the $K$-th request (the second $P_0$), the buffer contains two $P_0$ requests and one request for every other page, so LP flushes $P_0$. Subsequently, LP incurs one unit of cost every two new $P_0$ requests.

A better policy $\Pi$ would be to first flush $P_1, P_2, \ldots, P_{K-2}$ after the $K$-th request. Subsequently, $\Pi$ would buffer $P_0$ and flush when needed, incurring one unit of cost every $K$ new $P_0$ requests.

Theorem 13. SP is $\Omega(K)$-competitive.

Proof. Let $K = 3k + 1$. Consider the request sequence

$$P_1, P_1, P_2, P_2, P_2, \ldots, P_k, P_k, P_k, P_0, P_0^*.$$

After the $K$-th request (the first $P_0$), the buffer contains one $P_0$ request and three requests for every other page, so SP flushes $P_0$. Subsequently, SP incurs one unit of cost for each new $P_0$ request.

A better policy $\Pi$ would be to first flush $P_1, P_2, \ldots, P_k$ after the $K$-th request. Subsequently, $\Pi$ would buffer $P_0$ and flush when needed, incurring one unit of cost every $K$ new $P_0$ requests.

Lemma 14. If OPT is given a buffer of size $K/c$ (where $c > 1$) instead of $K$, its cost increases by at most a factor of $2c[\log K]$.

Proof. Given the behavior of OPT using a buffer of size $K$, we design a policy $\Pi$ using a buffer of size $K/c$ as follows. Suppose OPT flushes $x$ requests for a page $P$. We divide the period between this flush and the previous flush of $P$ into phases, according to how many $P$ requests have been buffered by OPT:
• The first phase is when this number is within \([1, 2^\sigma]\), where \(\sigma = \lfloor \log c \rfloor + 1\);

• In each subsequent phase this number is within \([2^i + 1, 2^{i+1}]\), for \(\sigma \leq i \leq \lfloor \log x \rfloor - 1\).

For the first phase, \(\Pi\) would reserve no buffer space for \(P\), and simply flush every request for \(P\) immediately; there are at most \(2^\sigma \leq 2c\) flushes. For the phase corresponding to \([2^i + 1, 2^{i+1}]\), \(\Pi\) would reserve \([(2^i + 1)/c]\) buffer space for \(P\), and flush \(P\) whenever the reserved capacity is reached or at the end of the phase; there are at most \(\left\lfloor \frac{2^{i+1} - 2^i}{(2^i + 1)/c} \right\rfloor < 2c\) flushes because \(i \geq \sigma\). Since the number of phases is at most \(\lfloor \log x \rfloor \leq \lfloor \log K \rfloor\), \(\Pi\) does at most \(2c[\log K]\) flushes for each flush of \(P\) by OPT.

Finally, it is easy to see that \(\Pi\) never uses more than \(K/c\) space. During each phase, OPT spends more space on \(P\) than it does at the beginning of the phase, while \(\Pi\) uses no more than \(1/c\) of that amount. Therefore, \(\Pi\) uses no more than \(1/c\) of the space used by OPT at any time. \(\square\)

**Theorem 15.** \(LG\) is \(O(\log^3 K)\)-competitive.

*Proof.* Divide the request sequence into \(R = \lfloor \log K \rfloor + 1\) subsequences, one for each group in \(LG\). Subsequence \(S_i\) contains all requests that are flushed by \(LG\) as part of Group \(i\).

For each \(i \in [0, R)\), our first step is to compare \(C_i^{LG}\), the number of flushes of Group \(i\) by \(LG\), against \(C_i^{OPT}\), the number of flushes incurred by running OPT on \(S_i\) with a buffer of size \(\lfloor K/3R \rfloor\). Divide \(S_i\) into phases separated by flushes of Group \(i\) by \(LG\). Consider any such phase. Let \(r\) denote the number of requests in this phase. We have \(r \geq K/R\) because there are \(R\) groups and \(LG\) always flushes the largest group. Let \(q\) denote the number of distinct pages requested in this phase. \(C_i^{LG} = q\). OPT has to

\[\text{If } i \geq \log c + 1, \text{ then } 2^i > 2c - 1, \text{ and we can show } \left\lfloor \frac{2^{i+1} - 2^i}{(2^i + 1)/c} \right\rfloor < \frac{2^{i+1} - 2^i}{(2^i + 1)/c} + 1 < 2c \text{ (details omitted). If } \lfloor \log c \rfloor + 1 = \sigma \leq i < \log c + 1, \text{ we have } 2^i \in (c, 2c), \text{ so } \lfloor (2^i + 1)/c \rfloor \geq 1 \text{ and the inequality follows.}\]
flush a page $P$ at least once in this phase if OPT is unable to buffer all requests to $P$ in this phase. Recall from the definition of Group $i$ that the number of requests per page is in the range $[2^i, 2^{i+1})$. With $\lfloor K/3R \rfloor$ space, OPT can buffer all requests for no more than $\lfloor K/3R \rfloor / 2^i$ pages. Therefore,

$$\frac{C_i^{\text{OPT}}}{C_i^{\text{LG}}} \geq q - \frac{\lfloor K/3R \rfloor / 2^i}{q}$$

$$\geq 1 - \frac{(\frac{r}{3})/2^i}{q} \quad \text{by } r \geq \frac{K}{R}$$

$$\geq 1 - \frac{(\frac{r}{3})/2^i}{r/(2^{i+1} - 1)} \quad \text{by } r \leq (2^{i+1} - 1)q$$

$$\geq 1/3.$$ 

Let $C^{\text{LG}}$ denote the total cost of LG, and let $C^{\text{OPT}}$ denote the cost of running OPT with a buffer of size $K$ over the entire request sequence. Next, we will show that $C_i^{\text{OPT}}/C^{\text{OPT}} = O(R^2)$, so

$$\frac{C_i^{\text{LG}}}{C^{\text{OPT}}} = \frac{\sum_{i=0}^{R-1} C_i^{\text{LG}}}{C^{\text{OPT}}} < 3 \sum_{i=0}^{R-1} \frac{C_i^{\text{OPT}}}{C^{\text{OPT}}} = \sum_{i=0}^{R-1} O(R^2) = O(R^3),$$

completing the proof. To this end, note that $C_i^{\text{OPT}_K}$, the cost of OPT on $S_i$ with $K$ space, must be no more than $C^{\text{OPT}}$, where OPT runs on a strictly bigger sequence. Therefore,

$$\frac{C_i^{\text{OPT}}}{C^{\text{OPT}} < \frac{C_i^{\text{OPT}}}{C_i^{\text{OPT}_K}}} = O(R[\log K]) = O(R^2)$$

by Lemma 14. □
Bibliography


Biography

Yi Zhang was born in September, 1984 in Liyang, Jiangsu, China. He graduated from Jiangsu Liyang Middle School in 2002 and then spent four years at Tsinghua University, Beijing, China, where he graduated summa cum laude with a B.E. degree. He has been a Ph.D. student in the Department of Computer Science since 2006. During his Ph.D. study, he has interned at Spirent Communications in Raleigh, North Carolina, USA and also the esteemed Microsoft Jim Gray Systems Lab in Madison, Wisconsin, USA. He is interested in data management and optimization for data-intensive statistical/scientific computing. He has also worked on energy-efficient data processing techniques in wireless sensor networks.

As of March 2012, his publications include:

