Summer 2021

Performance Optimization With An Integrated View Of Compiler And Application Knowledge

Ruiqin Tian
William & Mary - Arts & Sciences, ruiqin.cn@gmail.com

Performance Optimization with an Integrated View of Compiler and Application Knowledge

Ruiqin Tian

Jingning, Gansu, China

Bachelor of Engineering, Northeast Petroleum University, 2012
Master of Science, University of Chinese Academy of Sciences, 2015

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
May 2021

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Ruiqin Tian

Ruiqin Tian

Approved by the Committee, May 2021

Bin Ren

Committee Chair
Bin Ren, Assistant Professor, Computer Science
College of William & Mary

Evgenia Smirni, Professor, Computer Science
College of William & Mary

Weizhen Mao

Weizhen Mao, Professor, Computer Science
College of William & Mary

Pieter Peers, Associate Professor, Computer Science
College of William & Mary

Gokcen Kestor, Computer scientist, HPC group
Pacific Northwest National Lab

ABSTRACT

Compiler optimization is a long-standing research field that enhances program performance with a set of rigorous code analyses and transformations. Traditional compiler optimization focuses on general programs or program structures without considering too much high-level application operations or data structure knowledge. In this thesis, we claim that an integrated view of the application and compiler is helpful to further improve program performance. Particularly, we study integrated optimization opportunities for three kinds of applications: irregular tree-based query processing systems such as B+ tree, security enhancement such as buffer overflow protection, and tensor/matrix-based linear algebra computation.

The performance of B+ tree query processing is important for many applications, such as file systems and databases. Latch-free B+ tree query processing is efficient since the queries are processed in batches without locks. To avoid long latency, the batch size can not be very large. However, modern processors provide opportunities to process larger batches parallel with acceptable latency. From studying real-world data, we find that there are many redundant and unnecessary queries especially when the real-world data is highly skewed. We develop a query sequence transformation framework *Qtrans* to reduce the redundancies in queries by applying classic data-flow analysis to queries. To further confirm the effectiveness, we integrate *Qtrans* into an existing BSP-based B+ tree query processing system, PALM tree. The evaluations show that the throughput can be improved up to 16X.

Heap overflows are still the most common vulnerabilities in C/C++ programs. Common approaches incur high overhead since it checks every memory access. By analyzing dozens of bugs, we find that all heap overflows are related to arrays. We only need to check array-related memory accesses. We propose Prober to efficiently detect and prevent heap overflows. It contains Prober-Static to identify the array-related allocations and Prober-Dynamic to protect objects at runtime. In this thesis, our contributions lie on the Prober-Static side. The key challenge is to correctly identify the array-related allocations. We propose a hybrid method. Some objects can be identified as array-related (or not) by static analysis. For the remaining ones, we instrument the basic allocation type size statically and then determine the real allocation size at runtime. The evaluations show Prober-Static is effective.

Tensor algebra is widely used in many applications, such as machine learning and data analytics. Tensors representing real-world data are usually large and sparse. There are many sparse tensor storage formats, and the kernels are different with varied formats. These different kernels make performance optimization for sparse tensor algebra challenging. We propose a tensor algebra domain-specific language and a compiler to automatically generate kernels for sparse tensor algebra computations, called SPACe. This compiler supports a wide range of sparse tensor formats. To further improve the performance, we integrate the data reordering into SPACe to improve data locality. The evaluations show that the code generated by SPACe outperforms state-of-the-art sparse tensor algebra compilers.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

It is a very exciting experience to pursue my Ph.D. degree in the department of computer science at the College of William and Mary. In the past several years, I gained a lot of help from the professors and the staff members in our department. More specifically, I would like to give my thanks to the following people:

First, I would like to thank my advisor, Prof. Bin Ren, for his generous support and help on my Ph.D. study. I thank him for taking me as his student. He is an open-minded professor who cares about his students' interests. When I told him I am very interested in doing compiler-related research, he gave me many opportunities to explore it. He is also a super nice person who acts not only as an advisor but also as a friend. He gave me a lot of encouragement during these years. I remembered clearly that when I had a baby, he told me that even if you work 6 hours every day, you would still get progress on your projects. These words exactly make me feel confident about finishing my Ph.D. study,

Second, I would like to thank my internship mentor, Dr. Gokcen Kestor, for the extensive guidance during my internship. She always gave me enough details and resources for me to study a new thing, which makes me feel that learning new knowledge is not terrible at all. More importantly, she always gave me trust and encouragement. When I start to handle a new problem, she always says "I trust you." The words make me feel confident. She also taught me how to make our work known to others. It's so lucky to work with her.

Third, I would like to thank our collaborators, Prof. Zhijia Zhao, Prof. Xu Liu, and Prof. Junqiao Qiu on the query redundancy elimination project, Prof. Tongping Liu and Dr. Hongyu Liu on the buffer overflow project, Dr. Luanzheng Guo and Dr. Jiajia Li on the tensor algebra compiler project. Thanks for their help on these projects.

Fourth, I would like to thank my thesis committee members, Prof. Weizhen Mao, Prof. Evgenia Smirni, Prof. Pieter Peers, and Prof. Peter Kemper for their helpful comments on my presentation and thesis. I also thank them for their generous support.

Fifth, I would like to thank our lab members, Zhen Peng, Qihan Wang, Yu Chen, and Wei Niu for sharing great thoughts on group meetings.

Sixth, I would like to thank the staff members in our department, Vanessa Godwin and Dale Hayes, for their support these years. Without their support, my Ph.D. study will not be so smooth.

At last, I would like to thank my family for their constant love and support in all my life. Without their love and support, I will not be who I am today. Special thanks to my husband, Lele Ma, for all his support in the past years.

To my family.

LIST OF TABLES

LIST OF FIGURES

Performance Optimization with an Integrated View of Compiler and Application Knowledge

Chapter 1

Introduction

Performance, which is usually measured by response time, throughput, or resource utilization [130], is one of the key concerns for many applications in various areas, for example, databases [64, 85], parallel file systems [185], online analytical systems [36], security [186, 136, 28, 169], data analysis and mining applications [104, 158, 182], healthcare applications [2, 125], machine learning applications [117, 173], social network analytics [216], natural language processing [24, 145] and many others. These applications require high performance in the form of high throughput, low latency, or efficient memory usage, among others.

Compiler optimization is widely used to improve program performance through a series of optimizing transformations. These optimizations introduce a wide variety of benefits such as execution time reduction [34, 152, 59, 82], memory overhead elimination [33, 201], and/or reduced power consumption [90, 167, 89]. However, traditional compiler optimizations usually focus on analyzing code structures only, such as loop constructs, function calls, isomorphic instructions, and common expressions or sub-expressions. An example of this is loop optimizations, a major kind of compiler optimizations. Loop optimizations usually include loop unrolling, loop fusion, and loop tiling/blocking [13]. These optimizations are general; however, because of their generality, they miss some optimization opportunities due to the lack of high-level application knowledge as well.

1.1 Thesis topic

Application knowledge (or application information) in this thesis refers to multiple aspects of an application, for example, input or output, function operations, data distribution or data storage. If the input of an application is a sequence of queries [74, 126, 56, 86, 206, 143], the query type and operands belong to application knowledge; if the input is a set of data elements, the data pattern, format, and distribution also belong to application knowledge [79, 98, 98, 184, 192, 108, 38].

This thesis argues that it is possible to leverage high-level application knowledge to expose more optimization opportunities to compilers to improve program performance. More specifically, this thesis aims to build an application-compilation integrated view and explore various optimizations that are provided by this integration. In other words, it is impossible to benefit from these optimizations if the application and compilation are treated separately.

1.2 Optimization opportunities

This thesis studies three main applications from various domains: B+ tree-based query processing, buffer overflow protection, and sparse tensor algebra computations. It mainly explores three optimization opportunities: redundant computation elimination, unnecessary computation removing, and efficient parallelism.

Redundant computation elimination corresponds to the classic compiler optimization of *partial redundancy elimination* (PRE). PRE is used to eliminate redundant code in programs. A computational statement is redundant if the same computation is calculated multiple times while the operands of the statement do not change along the path. Eliminating the redundancy computations in the program reduces the number of computations, resulting in performance improvement. Many PRE algorithms have been developed to optimize program performance [132, 57, 147, 32, 148, 101, 26]. As aforementioned, these algorithms consider code-level information only without considering any

application knowledge. Redundancy elimination is also used in storage systems to improve the space utilization [189, 22, 175, 107, 151] and in network communications to reduce data transferred [215]. These strategies leverage the redundancy in data to reduce the storage space or communication overhead. This thesis does not leverage data redundancy but rather targets eliminating redundancy through the use of other kinds of application knowledge, such as the input query of the B+ tree query processing system.

Unnecessary computation removing is an effective way to remove computations that do not affect the final result. In compiler optimizations, unnecessary computation usually has two main forms, redundant computation and dead code in programs. Dead code is code that is executed but whose results are never used [5]. Many dead code elimination approaches have been proposed to improve program performance [100, 23, 203, 78, 141]. These approaches rely on analyzing the programs, i.e. only consider code-level information. This thesis leverages application knowledge to remove the unnecessary computations. For example, it is possible to control the protection scope of buffer overflows by leveraging code patterns in programs.

Parallelism is key to program performance. This thesis mainly considers two types of parallelism, data parallelism and task parallelism. Data parallelism refers to distributing data to different hardware computing resources and computing them in parallel. Task parallelism refers to distributing tasks to different hardware computing resources and executing these tasks in parallel. Data parallelism is often achieved with SIMD (Single Instruction, Multiple Data) units [156, 168], and task parallelism is often achieved with multi-threads. For SIMD data parallel, SIMD utilization plays an important role in performance [156, 72, 159, 157]; for multi-thread task parallel, reducing the synchronization or communication overhead plays a key role in efficient execution [37, 44, 202]. This thesis explores efficient parallelism for B+ tree based query processing system execution and sparse tensor algebra computations.

The three studied applications share common optimization opportunities. Figure 1.1 shows the connections between the optimization opportunities and the applications.

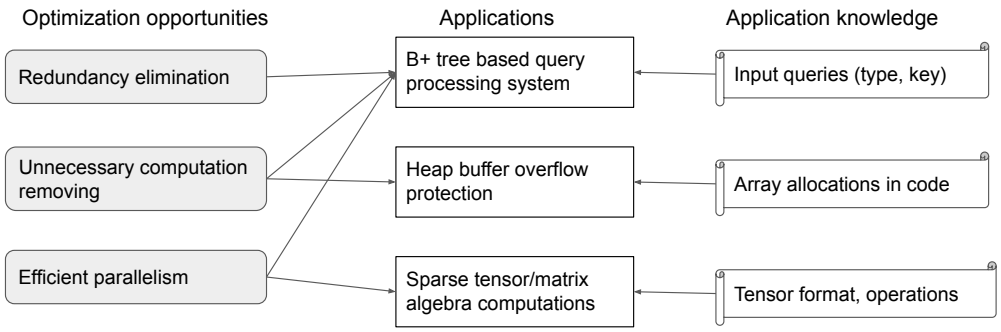


Figure 1.1: Connection between optimizations and applications

For redundancy elimination and unnecessary computation removing, this thesis analyzes input queries in B+ tree query processing systems and identifies many redundant and unnecessary queries (application knowledge). It then applies a compiler optimization, redundancy elimination, to eliminate the redundant and unnecessary queries, thus improving the throughput. Similarly, in buffer overflow protection, this thesis analyzes dozens of heap buffer overflow bugs in C/C++ programs and discovers that all heap overflows are related to arrays (application knowledge). This means that protection of non-array objects is unnecessary for heap buffer overflows. This thesis designs a set of compiler techniques to automatically analyze source code and identify array allocations.

To improve program parallelism in B+ tree query processing system, this thesis analyzes the input queries to guarantee that the queries on the same key (or the same leaf node in B+ tree) are only processed by one thread. It therefore reduces thread conflicts and achieves better thread-level parallelism. Similarly, for sparse tensor algebra computations, the computations on each dimension of output tensor are only processed by one thread thus achieving better thread-level parallelism. Moreover, because the compiler knows the distribution of queries or the computation pattern of tensor computations, it is possible to design effective SIMD optimizations to achieve better SIMD utilization as well.

1.3 Contributions

In this thesis, we explore program optimizations from an integrated view of compiler and application knowledge. As we mentioned above, we study three different types of applications. The contributions in each application are presented in the rest of this section.

1.3.1 Improving B+ tree query processing by reducing redundant queries.

B+ trees are used in a wide range of applications, such as database systems and file systems. Improving the performance of B+ tree processing systems has been thoroughly studied. Most efforts focus on improving concurrency. However, synchronization is still a performance bottleneck in improving concurrency. Latch-free B+ tree query[170] processing is proposed to avoid synchronizations. Queries are collected into batches and each batch is processed by threads parallel under a bulk synchronous parallel (BSP) model. The threads are carefully coordinated so that locks can be avoided. The problem is that the batch size can not be very large to avoid long delays. However, advanced modern processors make it possible to increase the batch size. In this thesis, we find that there will be more optimization opportunities beyond parallelism when the batch-size increases, especially with the highly skewed real-world datasets. We find that there are many redundancies in the queries. To identify and remove the redundant queries, we propose a query sequence analysis and transformation framework - *QSAT* based on applying classic data-flow analysis. For practical use, we implement a one-pass *QSAT*, called *Qtrans*. To evaluate the effectiveness, we integrate *Qtrans* into an existing BSP-based B+ tree query processing system, PALM tree [170]. The evaluation shows that *Qtrans* is effective and efficient, yield up to 16X throughput improvement.

1.3.2 Using compiler static analysis to assist in defending heap buffer overflow.

Heap buffer overflows are still the top vulnerabilities in C/C++ programs. Common approaches often bring too much performance overhead since they check every overflow. Efficient approaches such as Cruiser [211], DoubleTake [121], HeapTherapy [212], iReplayer [119], can not stop the vulnerabilities before overflow happens since they detect buffer overflows after the effect. We propose Prober to overcome these issues. Prober imposes a low overhead and can stop the program before overflow happens. It can also detect both read-based and write-based heap overflows. Prober is based on the key observation that overflows are typically related to arrays. This key observation identifies that we only need to protect array-related objects. Prober is composed of Prober-Static and Prober-Dynamic. Prober-Static is used to identify and instrument the array-related allocations in programs and Prober-Dynamic is for protecting the instrumented array-related objects in run-time. In this thesis, we contribute Prober on the Prober-Static side.

The key challenge of Prober-Static is to correctly identify all the array-related heap objects. On one hand, missing array-related heap objects will lead to no detection of overflows. On the other hand, including unnecessary objects will increase the run-time protection overhead. To this end, Prober-Static uses a hybrid approach. Some objects can be identified as array-related (or not) statically with the compiler. For the remaining ones, we decide in the runtime. We first instrument the size of the basic allocation type statically, then use Prober-Dynamic to determine the real allocation size in run-time. If the real allocation size is multiple times the size of the basic type, the allocation is identified as array-related allocation. Overall, Prober-Static is conservative and it does not miss any array-related allocations. The effectiveness has been evaluated in dozens of real-world heap overflow applications.

1.3.3 Building high-performance compiler for sparse tensor algebra computations.

Tensor algebra is at the core of numerous applications in scientific computing, machine learning, and data analytics, where data is often sparse with most entries as zeros. Achieving high-performance on sparse tensor algebra computations is important. There are many challenges in writing high-performance code for sparse tensor computations. First, the storage format will influence computation performance. There are many storage formats to store the non-zero values in sparse tensors and not a single format is good for all cases. To get high-performance for specific sparse tensors computations, users need to choose the proper format according to the feature of the sparse tensors. Second, optimizing sparse computation is difficult. Sparse tensor computations contain many indirect memory accesses and write dependencies. Besides this, different tensor expressions and different storage formats make the computation kernels different. It is necessary to use different optimizations to solve different performance bottlenecks in different computations kernels. Third, there are many back-end hardware platforms. Different hardware platforms require different code optimizations for high-performance.

To handle some of the above challenges, we propose a compiler-based approach to achieve high-performance on data-intensive sparse tensor computations. We build a sparse tensor compiler, SPACe, based on the Multi-level Intermediate Representation (MLIR) framework, a compiler infrastructure developed by Google to build reusable and extensible compilers. MLIR provides a disciplined, extensible compiler pipeline with gradual and partial lowering. Users can build customized compilers based on MLIR by creating customized domain-specific intermediate representations (IR) and implementing domain-specific optimizations. Since SPACe is built on MLIR infrastructure, it supports different hardware platforms by utilizing the powerful back-end compilation support.

SPACe supports several most common formats, such as Coordinate format, Compressed sparse fiber format and Mode-generic format with our proposed sparse tensor for-

mat attributes, which considers the format attribute of each dimension of tensors. SPACe is implemented as an extension of the MLIR framework. It contains a highly-productive domain-specific language (DSL) that provides high-level programming abstractions for tensor algebra computations. SPACe uses the high-level tensor algebra information, such as tensor expression and tensor formats, to generate the corresponding computation kernels. We also integrate the data reordering optimization into SPACe to further improve the performance. We evaluate the performance of SPACe on massive sparse matrix/tensor datasets. The results show that SPACe can generate more efficient sequential and parallel code compared to state-of-the-art sparse tensor algebra compilers.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces the necessary general background for data-flow analysis, Low-level Virtual Machine (LLVM) compiler infrastructure, and Multi-level Intermediate Representation compiler framework (MLIR) used in this thesis. Chapter 3 shows how we use redundancy elimination techniques in the compiler to improve the performance of B+ tree query processing on many-core processors. Chapter 4 shows how we use compiler static analysis techniques to assist in heap buffer overflows in C/C++ programs effectively and efficiently. Chapter 5 shows how we build the high-performance sparse tensor algebra compiler based on high-level information such as tensor operations and tensor formats. Finally, we conclude the thesis and discuss the future research directions in chapter 6.

Chapter 2

Background

In this section, we provide the necessary background and the compiler frameworks we use in this thesis. The compiler optimizations used in this thesis are mainly based on the data-flow analysis. The compiler frameworks we use are LLVM and MLIR.

2.1 Data-flow analysis

Data-flow analysis is a classic way used by the compiler to infer run-time information statically. In an optimizing compiler, data-flow analysis is mainly used for reasoning about helpful run-time information statically to get more optimization opportunities, and providing logical evidence to prove the correctness of the optimizations at some program points. Programmers can also use data-flow analysis to better understand their programs to improve the programs accordingly. Data-flow analysis is usually conducted by solving a set of equations based on a graphical representation of the program. The output of the data-flow analysis is the possible facts that can happen during run-time [46].

Data-flow analysis has variable forms, such as variable liveness analysis, expression availability analysis, reaching definition analysis and very busy expression analysis. Variable liveness analysis finds *live variables* at program points. A variable v is *live* at point p if and only if there is a path from p to a use of v and there is no redefinition of v in this path. Variable liveness analysis can be used to make global register allocation more

efficient, to detect references to uninitialized variables, and so on. Expression availability analysis discovers the set of available expressions at each program point. It can be used to reason and eliminate global common sub-expressions. Reaching definition analysis finds the set of definitions that reach a block. It can be used to reason about where an operand is defined. An expression is *very busy* at a point if the expression will be guaranteed to be computed at some time in the future. Very busy expression analysis can be used to reduce the number of operations in the whole program. All these analyses play key roles in applying optimizations.

2.2 LLVM compiler infrastructure

LLVM is an open-source compiler infrastructure to support analyses and transformations for any program in all stages, including compile-time, link-time, install-time, run-time, and even idle time between runs. LLVM has five critical features that make it a powerful compiler infrastructure. First, LLVM provides persistent program information during a program's lifetime, which makes it possible to perform code analyses and transformations in all stages. Second, LLVM allows offline code generation. This feature makes it possible to add specific optimizations for performance-critical programs. Third, LLVM gathers user-based profiling information at runtime so that the optimizations can be adapted to the actual users. Fourth, LLVM is language independent so that any language can be compiled. Fifth, LLVM allows whole-program optimizations because it is language independent [110].

LLVM provides the above five features based on two critical parts. First, LLVM provides a low-level, but typed code representation, called LLVM Intermediate Representation (LLVM IR). LLVM IR represents the programs by using an abstract RISC-like three-address instruction set but including higher-level information such as type information, explicit control flow graphs, and data-flow representation. This higher-level information plays an important role in conducting code analysis and optimizations in all

stages. LLVM IR also provides explicit language-independent type information; it contains explicitly typed pointer arithmetic. To this end, LLVM IR also serves as a common representation for code analysis and transformations during the program's lifetime. Second, LLVM uses a compiler design to provide a combination of capabilities. Static compiler front-ends generate codes in LLVM IR and combined them into one LLVM IR code file by LLVM linker. Multiple optimizations can be applied during link-time, including the inter-procedural optimizations. The optimized code will usually be translated into native machine code according to the target given at link-time or install-time. The persistent information and the flexibility of applying optimizations make it possible for LLVM to perform code analysis and optimizations in all the stages [110].

2.3 Multi-level IR compiler framework (MLIR)

MLIR(Multi-level IR) is a compiler infrastructure for building reusable and extensible compilers. MLIR supports the compilation of high-level abstraction and domain-specific constructs and provides a disciplined, extensible compiler pipeline with gradual and partial lowering. The design of MLIR is based on minimal fundamental concepts and most of the IRs in MLIR are fully customized. Users can build domain-specific compilers and customized IRs, as well as combining existing IRs, opting into optimizations and analyses.

The core MLIR concepts include operations, attributes, values, types, dialects, blocks, and regions. An *operation* is the unit of semantics. In MLIR, "instruction", "function" and "module", are all modeled as *operation*s. An *operation* always has a unique opcode. It takes zero or more operands and produces zero or more results. These operands and results are maintained in static single assignment (SSA) form. An *operation* may also have attributes, regions, blocks arguments, and location information as well. An *attribute* provides compile-time static information, such as integer constant values, string data, or a list of constant floating-point values. A *value* is the result of an operation or block arguments, it always has a type defined by the type system. A *type* contains compile-time

semantics for the value. A *dialect* is a set of operations, attributes, and types that are logically grouped and work together. A *region* is attached to an instance of an operation to provide the semantics (e.g., the method of reduction in a reduction operation). A *region* comprises a list of blocks, and a *block* comprises a list of operations [111].

Beyond the built-in IRs in the MLIR system, MLIR users can easily define new customized IRs, such as high-level domain-specific language, dialects, types, operations, analyses, optimizations and transformation passes [111].

Chapter 3

Transforming Query Sequences for High-Throughput B+ Tree Processing on Many-core Processors

The throughput of B+ tree query processing is critical to many databases, file systems, and cloud applications. Based on bulk synchronous parallel (BSP), latch-free B+ tree query processing has shown promise by processing queries in small batches and avoiding the use of locks. As the number of cores on CPUs increases, it becomes possible to process larger batches in parallel without adding any extra delays. In this work, we argue that *as the batch size increases, there will be more optimization opportunities exposed beyond parallelism, especially when the query distributions are highly skewed.* These include the opportunities of avoiding the evaluations of a large ratio of redundant or unnecessary queries.

To rigorously exploit the new opportunities, this work introduces a query sequence analysis and transformation framework – *QTrans*. *QTrans* can systematically reason about the redundancies at a deep level and automatically remove them from the query sequence.

QTrans has interesting resemblances with the classic data-flow analysis and transformation that have been widely used in compilers. To confirm its benefits, this work integrates *QTrans* into an existing BSP-based B+ tree query processing system, PALM tree, to automatically eliminate redundant and unnecessary queries [1]. Evaluation shows that, by transforming the query sequence, *QTrans* can substantially improve the throughput of query processing on both real-world and synthesized datasets, up to 16X.

3.1 Introduction

As a fundamental indexing data structure, B+ trees are widely used in many applications, ranging from database systems and parallel file systems to online analytical processing and data mining [64, 85, 185, 36, 39]. There have been significant efforts on optimizing the performance of B+ trees, with a large portion of work aiming to improve the concurrency [161, 170, 134, 25, 27, 60]. As the memory capacity of modern servers has increased dramatically, in-memory data processing becomes more popular. Without expensive disk I/O operations, the cost of accessing in-memory B+ trees becomes more critical.

To reduce the tree accessing cost, prior work has proposed latch-free B+ tree query processing [170]. Traditionally, B+ tree query processing requires locks (i.e., latches) to ensure the correctness since queries may access the same tree node and if one of them modifies it (e.g., an insertion query), it would cause conflicts. Latch-free B+ tree query processing avoids the use of locks by adopting a bulk synchronous parallel (BSP) model. Basically, it processes the queries batch by batch, with each batch handled by a group of threads in parallel. By coordinating the threads working on the same batch, the use of locks can be totally avoided (see Section 2). To guarantee the quality of service (QoS), the size of a query batch should be carefully bounded to avoid long delays.

Fortunately, as modern processors become increasingly parallel, the size bound of a batch can be dramatically relaxed without incurring extra delays. For example, the latest

[1] Artifact available at: https://doi.org/10.5281/zenodo.1486393

Intel Xeon Phi processors equipped with 64 cores can process 1M queries with time cost at only milliseconds (ms) level. In this work, we argue that as the batch size grows, there will be more optimization opportunities exposed beyond parallelism, which are further compounded by the fact that many real-world queries follow highly skewed distributions. The high level idea is abstractly illustrated by Figure 3.1.

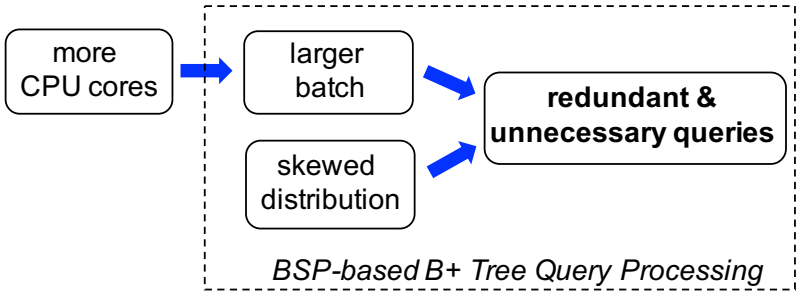


Figure 3.1: New Optimization Opportunities

For example, queries to the locations where taxi drivers stop are highly biased in both the time dimension (e.g., rush hours) and the space dimension (e.g., popular restaurants). As the query batch becomes larger, there will be growing possibilities of redundant queries (e.g., a repeated search of the same location) or unnecessary queries (e.g., a later query "cancel out" the effect of an earlier query).

To identify these "useless" queries, this work proposes a query sequence analysis and transformation framework – *QTrans*, to systematically reason about the relations among queries and exploit optimization opportunities.

QTrans has interesting resemblances with the classic data-flow analysis and transformation, but it targets *query*-level analyses and transformations. Intuitively, *QTrans* treats a query sequence as a "high-level" program, where each query resembles a statement in a regular program. By tracking the queries that "define" values, *QTrans* is able to link search queries to their corresponding defining queries. Based on the analysis, *QTrans* marks all the useful queries in the sequence and sweeps the useless ones, reducing the amount of queries to evaluate. Comparing to a traditional data-flow analysis [46, 4] that iterates over cyclic control flows, *QTrans* only needs to perform acyclic analysis for query

sequences with the most basic types of queries—although the algorithm of redundancy elimination is similar regardless of this difference.

To evaluate its effectiveness, we integrate *QTrans* into an existing BSP-based B+ tree processing system, called PALM tree [170]. The integration is at two levels: *QTrans* for each individual batch (i.e., intra-batch integration), and *QTrans* across batches (i.e., inter-batch integration). To minimize the runtime overhead, we also implement the parallel version of *QTrans* and discuss potential load imbalance issues.

Finally, our evaluation using real-world and synthesized datasets confirms the efficiency and effectiveness of *QTrans*, yielding up to 16X throughput improvement on Intel Xeon Phi processors, with scalability up to all the 64 cores.

In sum, this work makes a four-fold contribution.

- First, this work identifies a class of optimizations for B+ tree query processing, enabled by the increased hardware parallelism and the skewed query distributions.
- It proposes *QTrans*, a rigorous solution to optimizing query sequences, inspired by the conventional data-flow analysis and transformation.
- It integrates *QTrans* into an existing BSP-based B+ tree processing system and the evaluation shows significant throughput improvement.
- The idea of leveraging traditional code optimizations at the query level, in general, could open new opportunities for optimizing query processing systems.

In the following, we will first provide the background on B+ trees and the latch-free query processing (Section 3.2), then discuss the motivation of this work (Section 3.3). After that, we will present *QTrans* (Section 3.4), the integration of *QTrans* into PALM tree (Section 3.5), and the evaluation results (Section 3.6). Finally, we discuss the related work (Section 3.7) and conclude this work (Section 3.8).

3.2 Background

This section introduces B+ trees, its basic types of queries, and the high-level idea of latch-free query evaluation.

3.2.1 B+ Tree and Its Queries

A B+ tree is an N-ary index tree. It consists of internal nodes and leaf nodes. In contrast to B trees, B+ trees only maintain the keys and their associated values in their leaf nodes, and their internal nodes are merely used to hold the comparison keys and pointers for tree traversals. The maximum number of children nodes for internal nodes is specified by the *order* of B+ tree, denoted as b. The actual number of children for internal nodes should be at least $\lceil \frac{b}{2} \rceil$, but no more than b. Figure 3.2 shows an example of a 3-order B+ tree. Each internal node contains comparison keys and pointers to the children nodes. The leaf nodes together hold all the key-value pairs. In the leaf nodes, the numbers represent the keys and the numbers marked with asterisks represent the values of the corresponding keys. For the 3-order B+ tree, each internal node has at least 2 children nodes, but no more than 3.

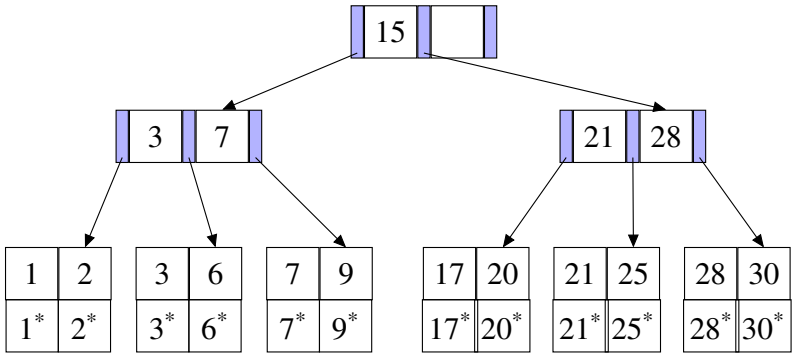


Figure 3.2: A 3-order B+ tree, where key-value pairs are stored only in leaf nodes (i.e., last level).

The structure of B+ tree dynamically evolves as queries to the tree are evaluated. In general, there are three basic types of B+ tree queries: (i) insertion; (ii) search; and (iii) deletion.

Given a B+ tree $\mathcal{T}$, suppose a function FIND$(key_i, \mathcal{T})$ can find the leaf node of key_i if it exists or return *null* otherwise, then the semantics of queries can be described as follows.

- $I(key_i, v_j)$: if FIND$(key_i, \mathcal{T}) \neq null$, then update its value to v_j; otherwise, insert a new entry of (key_i, v_j) into $\mathcal{T}$.
- $S(key_i)$: if FIND$(key_i, \mathcal{T}) \neq null$, return the value of key_i; otherwise, return *null*.
- $D(key_i)$: if FIND$(key_i, \mathcal{T}) \neq null$, then remove the entry (key_i, v_j) from the B+ tree.

Among the three, only $S(key_i)$ returns results; $I(key_i, v_j)$ and $D(key_i)$ only update/-modify the B+ tree. It is important to note that, when multiple queries arrive in a sequence, the order in which the queries are evaluated may affect both the returned results and the tree structure. In other words, there exist dependencies among the queries in general.

3.2.2 Latch-Free Query Evaluation

When there are multiple threads operating on the same B+ tree, it becomes challenging to evaluate the queries efficiently. First, the workload for each thread is too little to benefit from thread-level parallelism [170]; Second, since different queries may access the same node, threads have to lock the nodes (or even subtrees) that they operate, which essentially serializes the computations, wasting hardware parallelism.

A promising solution to the above issues is *latch-free query evaluation* [170]. Basically, it adopts the bulk synchronous parallel (BSP) model and processes queries batch by batch. Threads are coordinated to process the queries in a batch in parallel without any use of locks. Specifically, each query batch is processed in three stages [2], as illustrated in Figure 3.3:

[2]For better illustration, we merged stages 3 and 4 in [170].

Stage-1 ***Partition*** queries to threads evenly; threads then run in parallel to ***find*** the corresponding leaf nodes based on the keys in the queries;

Stage-2 ***Shuffle*** queries based on the leaf nodes such that each thread only handle queries to the same leaf node. ***Evaluate*** queries in parallel, including returning answers to search queries and updating corresponding tuples in the leaf nodes for insert and delete queries;

Stage-3 ***Modify*** tree nodes bottom up:

- ***Update*** tree nodes in parallel and ***collect*** requests for updating the parent nodes (i.e., the upper level);
- ***Shuffle*** modification requests to the parent nodes such that each thread only modifies the same node;
- ***Repeat*** update-shuffle, until the root node is reached and updated as needed.

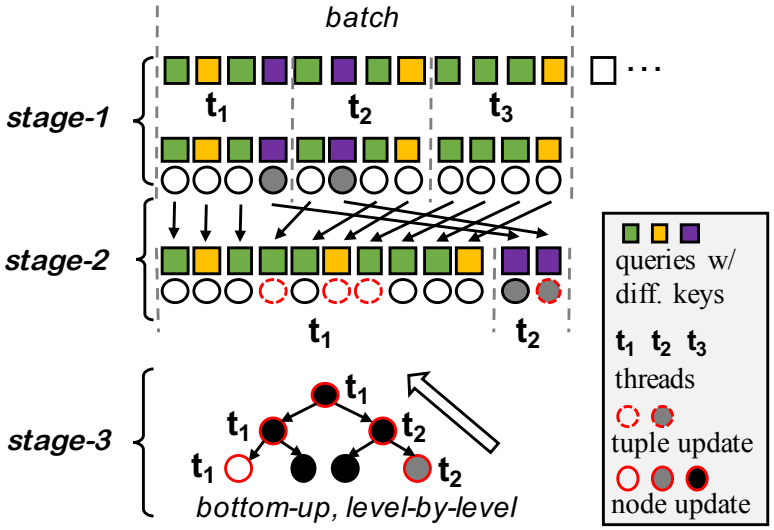


Figure 3.3: Latch-Free Query Evaluation

The shuffling in stages-2 and 3 ensures contention-free operations for each thread, guaranteeing the correctness. Comparing with lock-based schemes, this latch-free scheme

can significantly boost the throughput of query evaluation for B+ trees, up to an order of magnitude [170].

3.3 Motivation

On top of the promises of latch-free query evaluation, we find new opportunities to further improve the efficiency of B+ tree processing, enabled by modern many-core processors and the highly skewed query distributions.

3.3.1 Growing Hardware Parallelism

As the CPU clock frequency has reached a plateau, modern processors have embraced an increase in parallelism to sustain performance gain. For example, the latest Xeon Phi processor, Knights Landing [180], contains 64 cores/256 hyper threads. This massive hardware parallelism enables high processing capacity by allowing a larger pool of threads to run in parallel.

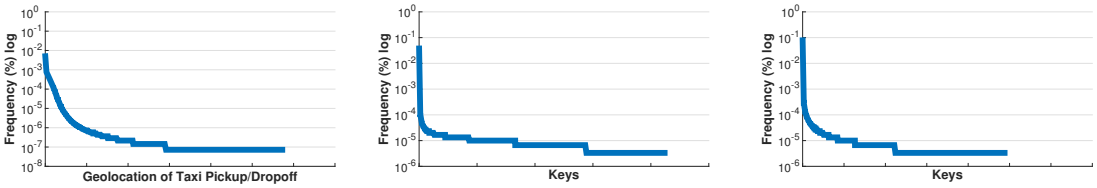
In the context of latch-free B+ tree query processing, the availability of more hardware threads allows the use of larger batch sizes while preserving the processing delay. However, this work argues that the benefits of using larger batches are not limited to the parallelism – as the batches become larger, new optimization opportunities are exposed, especially when the queries are unevenly distributed.

3.3.2 Highly Skewed Query Distribution

We observe that, the query distributions of real-world applications are often highly skewed. Take the taxi data of New York City (NYC) as an example [3]. The geolocations where taxi drivers pick up (or drop off) passengers follow a highly skewed distribution, as shown in Figure 3.4-(a).

The x-axis shows the geolocations and the y-axis indicates the visiting frequencies of

[3] https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2009-01.csv



(a) NYC Taxi (b) YCSB (Zipf Distribution) (c) YCSB (Latest Distribution)

Figure 3.4: Highly Skewed Query Distributions

each geolocation for a period of one month. The top 1000 geolocations out of 4,194,304 (i.e., 0.02%) covers 68.272% of total visits. In this case, the skewed distribution is caused by the fact that some geolocations are much more likely to be visited by taxis, such as shopping malls or popular restaurants.

In fact, skewed distributions frequently appear in other query processing scenarios, such as BigTable [35], Azure [47], Memcached [69], among others. Figures 3.4-(b) and (c) show the key distributions in cloud workloads modeled by Yahoo Cloud Serving Benchmark (YCSB). In these cases, the top 1% keys cover 30% and 56% requests, respectively.

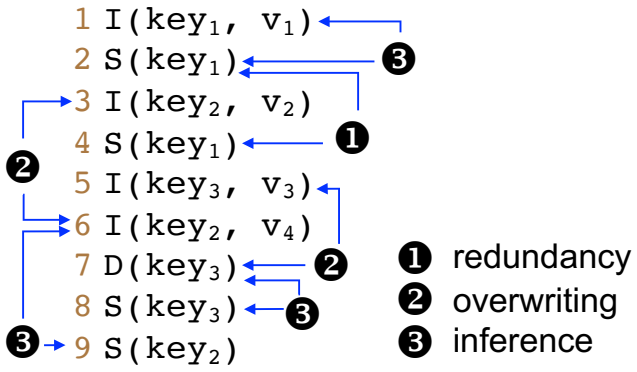


Figure 3.5: Optimization Opportunities

3.3.3 Optimization Opportunities

When the distribution becomes highly skewed, queries with identical keys tend to appear more frequently. This trend not only results in repetitive queries (i.e., query redundancies),

but also queries that might not have to be evaluated.

Next, we use an example query sequence, as shown in Figure 3.5, to illustrate the optimization opportunities, and informally characterize them into three categories.

- ***Query Redundancy*** ❶. One obvious opportunity is for the repeated search queries like queries 2 and 4 in Figure 3.5. Since query 3 does not modify key_1, query 4 should return the same value as query 2. Thus, we only need to evaluate one of them, then forward the return value to the other.

- ***Query Overwriting*** ❷. When two queries operate on the same key and both of them are either insert or delete with no search queries on the same key in between, then the second query may "overwrite" the first query. In another word, the first query becomes unnecessary, such as the overwritten queries 3 and 5 in Figure 3.5.

- ***Query Inference*** ❸. For a search query, by tracing back prior queries in the query sequence, one may find an earlier query carrying the information that the search query needs, thus we may infer its return value without evaluating it, such as query pairs (1, 2), (6, 9), and (7, 8).

In addition, as existing opportunities are exploited, more opportunities might be uncovered. For example, an earlier removal of a search query may enable a new opportunity of query overwriting. As we will show in the evaluation, the above optimization opportunities frequently appear when dealing with both real-world and synthesized datasets.

3.4 Analysis and Transformation

In this section, we present a rigorous way to systematically exploit the new opportunities mentioned above, inspired by the classic data-flow analyses and transformations.

3.4.1 Overview

Basically, we treat the query sequence as a "program", where each "statement" is a B+ tree query. Then the optimization of query sequence follows the typical procedure of a traditional compiler optimization: it first performs an analysis over the query sequence, based on which, it then transforms the query sequence into an optimized version – a new query sequence that is expected to be evaluated more efficiently. We refer to this new optimization scheme as *query sequence analysis and transformation* or QSAT, in short.

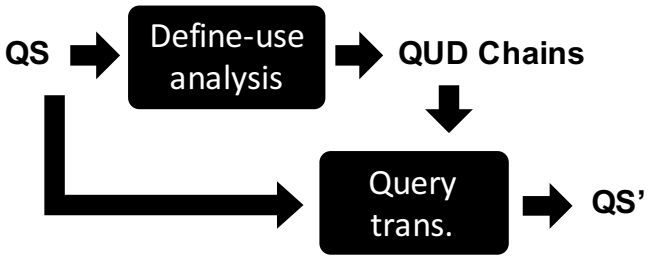


Figure 3.6: Conceptual Workflow of QSAT

Figure 3.6 illustrated the workflow of QSAT. The original query sequence QS is first analyzed to uncover use-define relationships among queries. The output – an intermediate data structure, called QUD chains is then used to guide the query sequence transformation, which yields an optimized query sequence QS'. Next, we present the ideas of QSAT.

3.4.2 Query Sequence Analysis

The goal of query sequence analysis is to uncover the basic define-use relations among the queries, which will be used to facilitate the later transformation. This resembles the classic *reaching-definition analysis* used in compilers [46, 4]. Basically, it examines the queries in the sequence and finds out which queries "define" the "states" of a B+ tree and which queries "use" the "states" correspondingly.

Based on the semantics defined in Section 3.2.1, the queries that define the state are *insert* and *delete* queries, and the queries that use the state are *search* queries. The define-use analysis matches each *search* query with its corresponding defining query (either an

insert or a *delete*) based on the keys that the queries carry.

Example. Figure 3.7-(a) shows the define-use analysis on the running example, where q_i corresponds to the query at line i. Basically, the set e consists of the defining queries that can reach each query. For example, the defining queries q_1, q_6 and q_5 can reach query q_7.

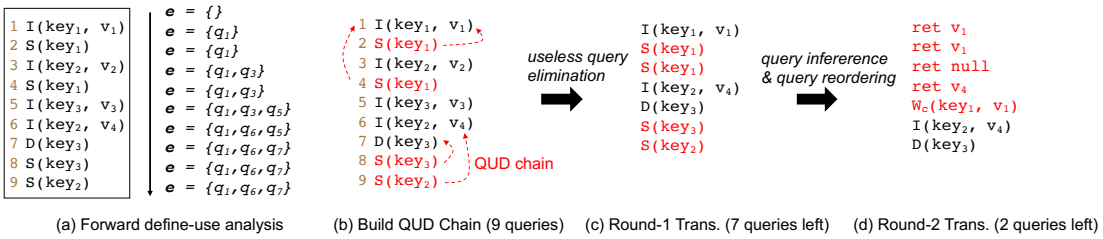


Figure 3.7: Example of Query Sequence Analysis and Transformation (QSAT)

QUD Chain. To represent the results of define-use analysis, we construct a data structure – *query-level use-define chain* (QUD chain). This data structure resembles the UD chain constructed internally by some compilers.

The construction of QUD chains is as follows. Basically, when a use query is met, the construction adds a link from the use query to its corresponding defining query (i.e., the defining query with the same key) if the later exists in current defining query set e. An example of constructed QUD chains is shown in Figures 3.7-(b).

QUD chains capture the dependence relations among the queries in a query sequence. For the query semantics defined in Section 3.2.1, the size of a QUD chain is limited to two queries. However, in general, the length of a QUD chain can go beyond two. QUD chains provide critical information for performing query sequence transformation, as shown next.

3.4.3 Query Sequence Transformation

The purpose of query sequence transformation is to generate an optimized version of query sequence. For clarity, we next describe the transformation with two passes. However, they can be integrated into one pass, as we will show later.

Round-1: Useless Query Elimination. This round is to eliminate queries that do not

Algorithm 1 Useless Query Elimination (Mark-Sweep)

```
1: I = {}        ▷ a list of useful queries
2: QUD()         ▷ QUD(q) returns the defining query of query q
3: for q_i in {q_1 ··· q_n} do
4:     if q_i is a search query then
5:         I.add(q_i)        ▷ mark a search query as "useful"
6:         if QUD(q_i) ≠ ∅ then
7:             I.add( QUD(q_i) )     ▷ mark defining query "useful"
8: return I
```

contribute to the final results of query processing – the returned values of search queries and the key-value pairs stored in the B+ tree. This can be achieved with a *mark-sweep* strategy that has been previously used for garbage collection and dead code elimination.

Algorithm 1 describes the useless query elimination. It first marks all the search queries as useful queries, as they need to return values. Then it traces back the QUD chains to find the corresponding defining queries, and mark them as useful queries as well. Note that the algorithm is customized to QUD chains of length 2, but it can be easily extended to handle QUD chains with arbitrary length.

Example. Figure 3.7-(c) lists the results after useless query elimination. The number of queries drops from 9 to 7. This round explores query overwriting (see Section 3.3.3).

Round-2: Query Inference & Reordering. Besides query overwriting, there are two other optimization opportunities: redundant queries and query inference (see Section 3.3.3). The second round is to explore the latter two.

Basically, for each search query, find its corresponding defining query (if exists), then retrieve the return value and return it. After this optimization, all the search queries with corresponding defining queries (i.e., $\text{QUD}(q_i) \neq \emptyset$) will be eliminated, as Figure 3.7-(d) shown (denoted as `ret` v_i).

Note that, after the optimization, no return operations `ret` v_i depend on any other queries, hence they can be reordered – being moved to the top of the sequence. In this way, the latency of the search queries could be reduced.

An orthogonal optimization is a top-K cache. When the B+ tree is large, performance

can be benefited from putting hot key-value pairs (top K pairs) into a small cache. Thus, when an insert query with a top-K key-value pair is left after round 1, we can transform the query into a cache write operation (e.g., $W_c(key_i, v_j)$ in Figure 3.7-(d)).

Finally, after the two rounds of optimizations, there are only 2 queries left that need to be actually evaluated.

3.4.4 Discussion

Comparison with Classic Data-flow Analysis. Despite the similarities between our define-use analysis and the traditional reaching-definition analysis, there are a couple of critical differences. First, the two analyses work at different granularities. The traditional data-flow analysis performs at the instruction level, while ours is applied at the query level. Each query itself may be implemented by a series of low-level instructions. Second, the traditional data-flow analysis operates on the control-flow graph, which may consist of cycles and take several iterations to converge. By contrast, our analysis works on a sequence of queries which imposes no "backward" control flows.

Potential Extension. Note that the ideas of query sequence analysis and transformation are not limited to the basic query semantics. It may benefit other batch-based query processing systems that may involve more complicated query structures as well. Consider a more advance query $I(key_1, S(key_2))$. The query is to insert/update key_1 with the value drawn from key_2. In this case, the length of a QUD goes beyond 2.

3.5 Integration

To evaluate the proposed analysis and transformation, we integrate *QTrans* into an existing latch-free B+ tree query processing system – PALM tree [170] (see Section 3.2.2). To maximize the benefits, this section also describes a parallel implementation of *QTrans* and optimizations across batches.

Algorithm 2 One-Pass QSAT (for queries of the same key)

```
1: q_o = null        ▷ the last defining query
2: n_s = 0           ▷ number of search queries
3: I = {}            ▷ a list of useful queries & operations
4: for q_i in {q_n ··· q_1} do
5:     if q_i is a search query then
6:         n_s++
7:     else if q_i is an insert or delete query then
8:         if n_s > 0 then
9:             I.add(INFER_AND_RETURN(q_i, n_s))
10:            n_s = 0
11:        if q_o = null then
12:            q_o = q_i
13: if n_s > 0 then
14:    ▷ no defining query for the last n_s queries
15:    I.add(SEARCH_AND_RETURN(q_n, n_s))
16: if q_o = null then
17:    I.add(q_o)
18: return I
```

3.5.1 Parallel Intra-Batch Integration

The *QTrans* described in Section 4 applies optimizations sequentially over the sequence of queries. However, in the actual setting of latch-free B+ tree query processing, queries in a batch are processed in parallel for maximum performance on parallel processors. To match with the intra-batch parallel query processing scheme, we next present a parallel design of *QTrans*.

Given a batch of queries, the parallel *QTrans* creates a pool of threads based on the number of available cores N (part of latch-free query evaluation), then performs the query optimizations in two phases:

- **Phase-I**: First, partition the query batch evenly into N *mini-batches*. Then performs sequential QSAT over different mini-batches in parallel.

- **Phase-II**: Shuffle the queries generated by Phase-I based on the keys. Then let each thread perform a sequential QSAT over queries of the same key.

Figure 3.8 shows the new latch-free query evaluation with the two phases integrated.

After Phase-II, there will be at most one (defining) query left for each key. After applying the parallel *QTrans*, the following steps would be the same as the original latch-free query evaluation (see Figure 3.3).

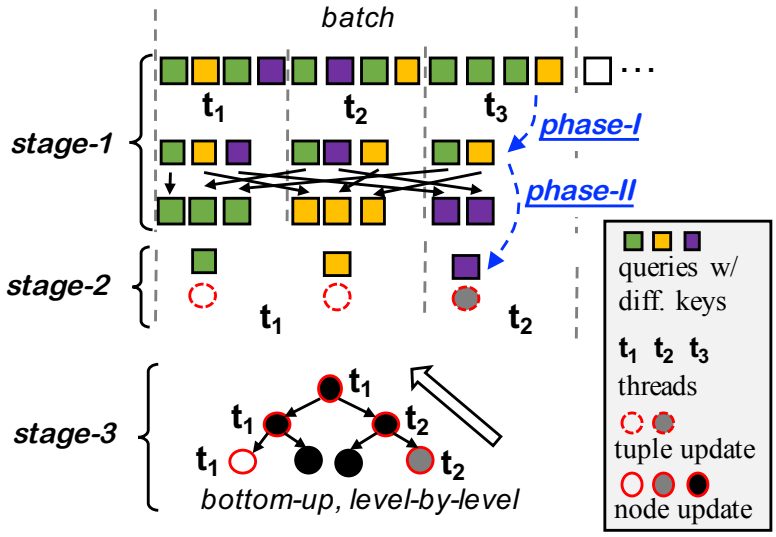


Figure 3.8: Latch-Free Query Evaluation w/ *QTrans*

Advantages. Comparing with the original latch-free query evaluation (Figure 3.3), the new design (Figure 3.8) shows several advantages:

- ***Faster sorting***. In the original design, query sorting is at the batch level. While in the new design, query sorting is only performed at the mini-batch level [4].

- ***Reduced leaf searches***. The original design searches for leaf nodes for every query in the batch; In comparison, the new design only searches for leaf nodes for each distinct key in the batch.

- ***Reduced shuffle overhead***. Both the original and the new designs require to shuffle the queries in Stage 2 and Phase-II. However, in the new design, the shuffle overhead is lower, due to the query reduction in Phase-I.

[4]For generality, query sorting is not shown in Figures 3.3 and 3.8.

Load Balancing. Despite the above advantages, intra-batch optimization may suffer from a workload imbalance at Phase II. After Phase I, the number of remaining queries of different keys might be different. Further, after the query shuffling of Phase II, the number of keys mapped to different leaf nodes might also varies due to the skewed key distribution. Both cases can cause a load imbalance among worker threads. Note that the second case also occurs in the original design of query processing. Here, we address them with a lightweight workload balancing strategy.

Basically, our load balancing method leverages the prefix sum algorithm to calculate the starting query index for each thread, so that the number of queries assigned to each thread can be similar, but not necessarily the same. The assignment should not assign queries with different keys to different threads, which violates the correctness of BSP.

3.5.2 Inter-Batch Optimization

Beside intra-batch optimizations, this work also explores optimization opportunities across batches. However, it is challenging to implement inter-batch QSAT, because the intermediate results of query analysis will grow as more batches are analyzed. For example, a search query's corresponding defining query may appear in a much earlier batch. Keeping tracking all the information will overburden the QSAT, outweighing the benefits.

Instead, we adopt a more scalable strategy that is similar to the alternative solution mentioned in Section 4.4. Basically, it "simulates" the query evaluation at the inter-batch level on a different data structure. In this way, we only need to carry the "state" of key-value pairs from one batch to the next. The key is that the simulation must be faster than the actual query evaluation to bring in potential benefits. We achieve this with a *top-K cache*.

Top-K Cache. This is a small software cache with fixed number of entries – K entries. This design minimizes the costs of read/write operations. The cache can be implemented with a hash table, where the key-value pairs perfectly match with the B+ tree key-value

pairs. As the number of entries is fixed, the hash function can be designed in an efficient way so that hashing conflicts can be minimized or even avoided. The entries in the top-K cache can be pre-populated with training data and periodically updated with testing data using various cache replacement policies (e.g., LRU).

To integrate the inter-batch optimization in the query evaluation system, we place the top-K cache operations in Stage 1 right after Phase II (see Figure 3.8). At this moment, the redundant and useless queries within the batch have been eliminated, hence the cache operations will be reduced to a minimum – only proportional to the number of distinct keys in the batch.

3.6 Evaluation

This section evaluates the efficiency and effectiveness of *QTrans* for optimizing the latch-free B+ query processing.

3.6.1 Methodology

We use an open-source implementation of latch-free B+ query processing system [5] as the baseline, which follows the design of PALM tree querying system [170]. It supports SIMD operations for key search within a tree node. *QTrans* is implemented in C++ language with the use of Pthread for multicore programming and is then integrated into PALM tree, serving as the optimized querying system.

Platform. We evaluate B+ tree query processing on the latest version of Xeon Phi, Knights Landing. Our Xeon Phi is a 64-core 7210 processor, used as a CPU, running at 1.3 GHz with 1M L2 cache shared between two cores, supporting 512-bit AVX512 instructions.

Datasets. To evaluate our query sequence optimization, we build B+ trees based on the unique keys from four synthetic datasets (with configurations the same as those in [170])

[5] `https://github.com/runshenzhu/palmtree`

and two realistic datasets:

- gaussian: the keys of queries follow the classic Gaussian distribution with parameters $\mu = N * 0.5$, $\delta = \mu * 0.5\%$, where N is the range of queries;
- self-similar: the keys follow 80-20 rule, which means 80% queries cover 20% range of queries [76];
- zipf: the keys follow Zipfian [76] with $\theta = 1.0$;
- uniform: the keys are uniformly distributed;
- ycsb: Yahoo! Cloud Service Benchmark (YCSB) [45] that is used to evaluate the performance of cloud systems. It includes Zipfian (ycsb-zipf) and latest (ycsb-latest); Note that zipf and ycsb-zipf are different in terms of the parameter settings.
- taxi: NYC taxi data published by New York City Taxi & Limousine Commission, containing the yellow and green trips in New York City at different time[6].

All key distributions except uniform are skewed. The size of our input queries, the configuration of trees, and the input query distributions are summarized in Table 3.1.

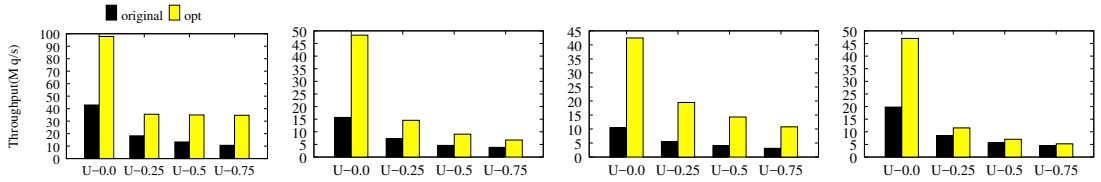
Table 3.1: Dataset configurations

Dataset	#queries	#uniq-key	parameters
Gaussian	100M	50M	$\mu = N * 0.5$, $\delta = \mu * 0.5\%$
Self-similar	100M	50M	80-20 rule
Zipfian	100M	50M	$\theta = 1$
Uniform	100M	50M	/
YCSB-latest	30M	10M	/
YCSB-zipfian	30M	10M	$\theta = 0.99$
Taxi	13.9M	4.1M	/

3.6.2 Performance and Scalability

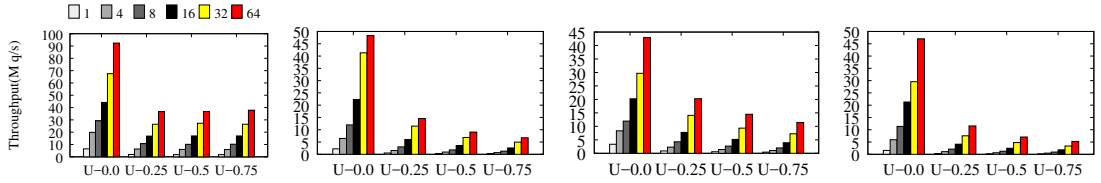
Synthesized Data. Figure 3.9 compares the original B+ tree processing with the one optimized with *QTrans* on four synthetic datasets (i.e., gaussian, self-similar, zipf and

[6] `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`



(a) gaussian (b) self-similar (c) zipf (d) uniform

Figure 3.9: Overall throughput improvement. x-axis: update ratios; y-axis: throughput of queries.

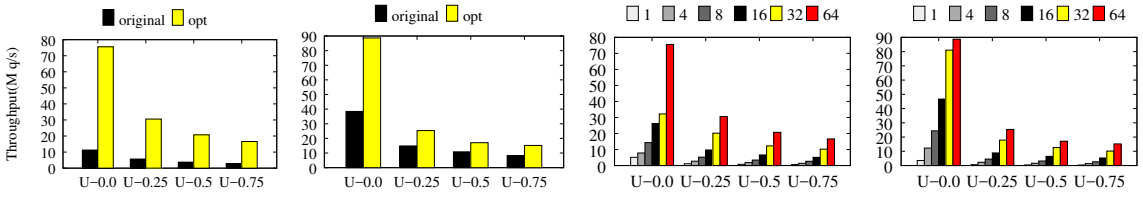


(a) gaussian (b) self-similar (c) zipf (d) uniform

Figure 3.10: Throughput scalability. x-axis: update ratios; y-axis: throughput of queries.

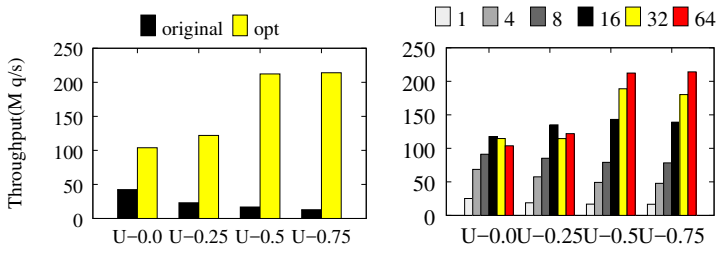
uniform) in terms of throughputs. For each distribution, the update ratio (i.e., the ratio of insert and deletion queries) changes from 0% to 75%. For all distributions, the one with *QTrans* (i.e., opt) shows better throughput, with up to $4.05X$ improvement (occurs on zipf dataset).

Specifically, for all datasets, the throughput improvement is higher when the update ratio is lower. Even for the uniform dataset, the throughput improvement reaches $2.37X$. This is because *QTrans* handles all FIND queries in stage 1, thus avoiding the time consuming stage 2 in the original design. When the update ratio is greater than 0%, for the skewed datasets, such as gaussain, zipf, and self-similar, the throughput improvement



(a) ycsb-latest throughput (b) ycsb-zipf throughput (c) ycsb-latest scalability (d) ycsb-zipf scalability

Figure 3.11: YCSB overall throughput and scalability. x-axis: update ratios; y-axis: throughput of queries.



(a) taxi throughput (b) taxi scalability

Figure 3.12: Taxi throughput and scalability.

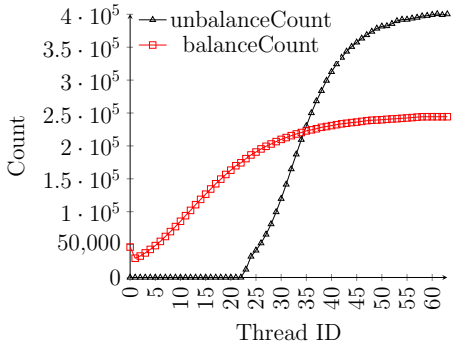
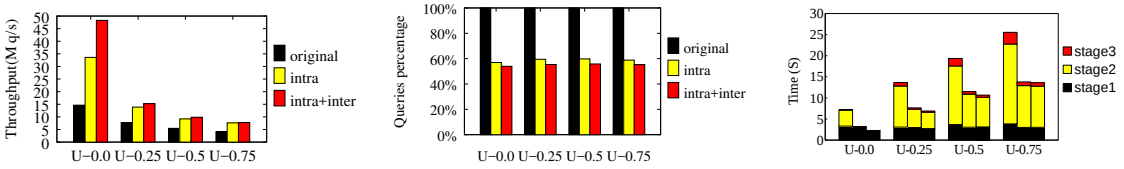


Figure 3.13: `self-similar` (U-0.25) leaf operations

is more significant, from $1.76X$ to $3.59X$. This is because they have higher chances to include queries with identical keys. Interestingly, even for uniform, *QTrans* shows slightly improvement (but much less than other skewed cases) when there are updates, owing to the query transformations.

More specifically, *QTrans* monitors the query types. If no defining queries are found, it will evenly partition the input queries and get rid of the time-consuming workload redistribution. In contrast, such redistribution is always required by the original implementation. Similarly, if the update ratio is low, it only redistributes the update-related queries, leading to better performance.

Realistic Data. Next, we confirm the results with real-world datasets ycsb-latest and ycsb-zipf (Figure 3.11 (a) and (b)), and NYC Taxi dataset taxi (Figure 3.12 (a)). Among the three datasets, *QTrans* optimized version achieves higher improvements on ycsb-latest with a nearly $6.71X$ improvement (U-0), and taxi with a nearly $16.60X$ improvement



(a) throughput (b) query reduction (c) time breakdown

Figure 3.14: `self-similar` throughput analysis, three bars in (c) correspond to bars in (a) and (b)

(U-0.75). In comparison, on ycsb-zipf, the *QTrans* optimized version only achieves 2.31X improvement. Note that the throughput improvements are different between ycsb-zipf and zipf, due to the parameter setting differences. The former is based on the real-world cloud system characterization; while the latter is chosen from prior work for a direct comparison [170].

Scalability. Figures 3.10, 3.11 (c)-(d), and 3.12 (b) report how the throughput of *QTrans* optimized version changes with the number of threads increasing from 1 to 64. Most cases show strong scalability up to 64 threads. Only taxi scales up to 32 threads. This is because taxi has fewer unique keys than other datasets, thus our optimization results in more query reduction. The remaining queries are too few to feed 64 threads. This is proved by its lowest latency in Table 3.2.

3.6.3 Performance Breakdown

To better understand the performance improvements, we perform a case study on the self-similar dataset.

Figure 3.14 compares the intra-batch and inter-batch optimized versions with the original version on throughput, query reduction ratio, and execution time of different stages, with the update ratio ranging from 0% to 75%.

Intra-batch Optimization Benefits. Comparing the original with the one enabled intra-batch optimization (Figure 3.14a), there is a clear throughput improvement.

The improvement is due to two main reasons. First, intra-batch optimization reduces

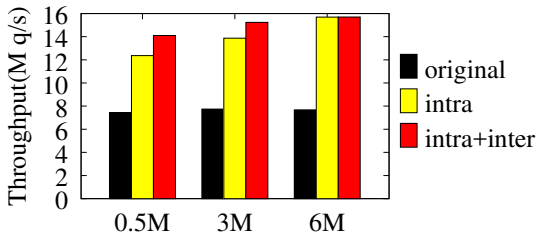


Figure 3.15: `self-similar` (U-0.25) throughput

the number of queries to process, which is reflected by the query reduction ratio, as shown in Figure 3.14b. However, as mentioned earlier, the query reduction may cause the workload imbalance in the later stage for leaf node searching, which can in turn compromises the reduction benefits to a certain degree. To alleviate this, *QTrans* performs a lightweight load balancing with parallel prefix sum (see Section 5.1). This is the second contributor to the throughput improvement.

In addition, we perform a study on the distribution of workload (counts of operations) on the leaf nodes when all 64 threads are employed, as shown in Figure 3.13. The counts are for a whole query sequence. The results demonstrate the efficiency of load balance optimization. However, even with our optimization, it is impossible to achieve a perfect load balance. Because there exists data dependencies among update queries that perform on the same tree node, these modifications will always be processed by the same thread. Since the input query is skewed, the number of queries handled by different threads is also skewed.

Inter-batch Optimization Benefits. The last bar in each sub-figure of Figure 3.14 shows the throughput gain, the task reduction ratio, and the execution time when the inter-batch optimization is applied. In general, the improvement varies because, in some cases, the optimization opportunities have already been explored by the earlier intra-batch optimization, especially for relatively larger batch sizes.

Batch Size Impact. We set batch_size as 0.5M, 3M, and 6M with update ratios of 25% for self-similar distribution, and test the throughput differences under different kinds of

optimizations. As shown in Figure 3.15, the throughput increases as the growing of the batch size, specifically, the benefit from intra-batch redundancy elimination.

Considering the batch size (in Table 3.2) and the absolute throughput after our optimizations together, we observe a strong correlation – a larger batch size leads to a better absolute throughput. In the offline processing case without the latency requirement, we can always select a large batch size to achieve a better throughput. Our work considers a more challenging online processing, and these batch sizes are chosen for a more acceptable latency requirement.

Table 3.2: Latency for each dataset

Dataset	Batch-size	Opt Lat(ms)		Org Lat(ms)	
		U-0.0	U-0.75	U-0.0	U-0.75
Gaussian	5242880	52.26	133.62	122.18	492.09
Self-similar	3145728	65.12	404.88	200.49	818.54
Zipfian	3145728	62.31	253.84	300.37	1011.5
Uniform	2097152	36.50	391.37	105.89	475.99
YCSB-latest	1500000	18.23	112.75	133.16	519.74
YCSB-zipfian	1500000	14.95	99.96	39.02	182.90
Taxi	2081427	14.66	17.65	49.12	161.38

3.6.4 Latency

Table 3.2 reports the latencies for two scenarios: search-only and 75% update, with the corresponding batch size. For comparison, we also report the original PALM tree's latency with identical batch sizes. Even for the largest batch case, we still maintain our search-only latency lower than 50ms and our update latency lower than 400ms. For the three real-world cases, our search-only latencies range from 14.66ms to 18.23ms, and update latencies range from 17.65ms to 112.75ms. This is smaller than 0.5-1s latency maintained in previous buffering method [217]. In addition, we can always trade our high throughput for faster response time by using a smaller batch size, if it is desired.

3.7 Related Work

This section focuses on research related to B+ tree processing, bulk synchronous parallel model, as well as redundancy elimination in traditional compilers.

B+ Trees and Its Optimizations. As a basic data structure, B+ tree has received significant attentions, especially on improving the concurrency by reducing tree contention.

Prior work can be roughly categorized into three groups. The first group contains methods that improve lock performance and designing lock-free trees, in asynchronous processing. For instance, Rodeh designed optimized lock-based B+ trees [161], and later, Braginsky and Petrank proposed a lock-free B+ tree to further improve the performance for high contention cases [25]. More recently, new lock-free tree structures are proposed to address the performance challenges brought by contentions, such as [134, 27, 60]. The second group leverage the Bulk Synchronous Parallel (BSP) model. PALM tree proposed by Sewall et al. [170] is a representative solution. The third group exploits hardware support like Hardware Transactional Memory (HTM), including the red-black tree implemented by Dice et al. [58] and Eunomia proposed by Wang et al. [195]. Based on these techniques, it is also effective to apply lazy tree restructuring [49, 50] to further reduce the contention.

The above methods focus either on improving locking behavior, lock-free policy, or on changing the tree structure. By contrast, this work focuses on exploiting the skewed query distribution, the semantic relations among queries, and the high concurrency provided by modern many-core processors, so it is complementary to all of these existing approaches.

In addition, there are techniques to map B+ trees or other similar index trees on many-core processors or other new architectures. For instance, Fix et al. [70] implemented a B+ tree on GPU, while Daga et al. [53]'s B+ tree is specific for APUs. Kim et al. [91] designed and implemented a fast architecture-sensitive search tree on both CPUs with SIMD units and GPUs. A more recent design of a B+ tree for heterogeneous platforms is given by Shahvarani and Jacobsen [171]. There are also many efforts on improving the cache performance for in-memory trees, such as Cache-sensitive search (CSS) trees [154]

and cache-sensitive B+ trees (CSB+-trees) [155].

Bulk Synchronous Parallel (BSP): The BSP model [190] used in latch-free B+ tree query processing has also been commonly used for many other applications. For example, Pregel [128] and Giraph [42], a well-known graph processing model is based on BSP. Many other graph processing engines or libraries are also directly built on BSP, such as GraphX [188] for distributed clusters and Gunrock [196] for GPUs. Moreover, the BSP model also serves as a design foundation for many successful programming models in big-data and high-performance computing fields, such as MapReduce [55], Spark [210], and Apache Hama [172].

Redundancy Elimination The key idea of this work is to *eliminate redundant and unnecessary queries by transforming the query sequence.* At high level, it shares the objectives with some traditional compiler optimizations, such as *partial redundancy elimination* (PRE) and *memoization*, which are also designed to eliminate unnecessary code in the programs.

Consider the control-flow graph (CFG) of a function in a program. If a computational statement is evaluated again along a certain path, without any of its operands changed in between, the later evaluation would be (partially) redundant and thus will be removed by PRE. Over the past 30 years, many PRE algorithms [132, 57, 148, 101] have been designed to optimize program performance. Another traditional compiler optimization for redundancy elimination is *memorization* [138, 52, 3, 75], which is heavily used for functional programming languages. The basic idea is to cache the results of frequent yet expensive function calls and returning the corresponding cached result when calls with the same inputs appear again.

The above techniques for code optimizations inspire the design of our query sequence analysis and transformation. In addition, redundancy elimination has also been used to improve the space utilization in storage systems [189, 22, 175] and the integration of relational database schema [102].

Finally, there are some compiler optimization techniques being used to optimize SQL queries [8, 11, 10], where the SQL queries are first transformed into imperative programs, then optimized by conventional compiler techniques. By contrast, our techniques directly transform the query sequences without any query-to-code transformations.

3.8 Summary

This chapter targets the critical throughput problem of B+ tree query processing. It, for the first time, points out the new optimization opportunities raised by the growing hardware parallelism and the highly skewed query distributions in real-world B+ tree applications. More specifically, this work identifies three categories of optimization opportunities in the B+ tree query evaluation. To systematically exploit these opportunities, it introduces a novel query sequence analysis and transformation (QSAT) framework, inspired by the conventional code optimizations in compilers. For practical use, this work designs a one-pass QSAT, namely *QTrans*, and integrates it into a latch-free B+ tree query processing system, with parallelization and load balancing supports. Finally, our evaluation confirms the efficiency and effectiveness of *QTrans* on both synthetic and real-world datasets with up to 16X throughput improvement.

Chapter 4

Compiler static analysis assistance in defending heap buffer overflows

Heap-based overflows in C/C++ programs are still not completely solved even after decades of research. We propose Prober, a novel system aiming to detect and prevent heap overflows in production environments. Prober leverages a key observation based on the analysis of dozens of real bugs: all heap overflows are related to arrays. Based on this observation, Prober only focuses on array-related heap objects, instead of all heap objects. Prober utilizes static analysis to label all susceptible call-stacks during the compilation, and then employs the page protection to detect any invalid accesses during the runtime. In addition to this, Prober integrates multiple existing methods together to ensure the efficiency of its detection. Overall, Prober introduces almost negligible performance overhead, with 1.5% on average. Prober not only stops possible attacks in time, but also reports the faulty instructions that could guide bug fixes. Prober is ready for deployment due to its effectiveness and low overhead. In this thesis, my contribution in this work is to use compiler static analysis to effectively identify the array-related heap objects.

4.1 Introduction

C/C++ applications are prone to memory errors, such as buffer overflows (including over-reads/over-writes). Buffer overflows will not only cause a program to crash, but also can be exploited to issue security attacks or cause information leakage [186]. Since it is difficult to expunge all buffer overflows during development phases which are highly dependent on program inputs, significant research has focused on detecting and preventing buffer overflows dynamically. Among them, stack-based overflows can be detected with very low overhead (less than 6.5%) via the shadow stack technique [186]. But heap-based overflows are still unresolved, and they were still ranked as the top 2 vulnerabilities (as shown in Table 4.1).

Table 4.1: Top five vulnerabilities reported in 2018 [51].

Vulnerabilities	DoS	Code Execution	Overflow	XSS	Gain Information
16555	1852	3035	2492	2004	1426

Dynamic detection tools can be further divided into multiple types. The most common approach is to check the overflow before every memory access, which can stop a overflow immediately if a memory access is found to access red zones that are not supposed to be read or written. Existing work, such as Valgrind [136], Dr. Memory [28], and AddressSanitizer [169], employs this approach. However, it imposes high performance overhead. For example, AddressSanitizer still imposes over 40% performance overhead. Efficient approaches such as Cruiser [211], DoubleTake [121], HeapTherapy [212], or iReplayer [119], detect buffer overflows after the effect, typically by checking the evidence of corrupted canaries. However, they cannot detect read-based overflows because reads do not leave any evidence behind. Also, they cannot timely stop security attacks.

We propose a novel system, called Prober, to overcome these issues. Prober imposes low performance and memory overhead so that it can be used in production systems. Second, Prober can detect both read-based and write-based overflows. Third, Prober will stop overflows immediately, eliminating any possibility of memory exploits. Last but not

least, Prober is able to report detailed information to assist bug fixes, e.g., allocation sites and faulty instructions. Prober is based on **a key observation** that separates it from all existing work: *overflowing objects are typically related to arrays*. This observation is based on our analysis on dozens of bugs collected by existing work [208] (as further discussed in Section 4.2.1). We further confirmed that this observation holds for all overflows reported in a randomly-selected period in the CVE database. This observation is also aligned with intuition: for an object not related to an array, there is no need of operating it with error-prone operations, such as pointer arithmetic instructions, string APIs, or loop operations, thus with a low possibility of overflows.

This key observation identifies the type of objects that may have buffer overflows, called as *array-related objects* or *susceptible objects*. Both terms will be utilized interchangeably in the remainder of this chapter. To take advantage of this observation, *Prober proposes to separate array-related objects from normal objects, by placing them into a separate space*. Then Prober employs the page protection to detect overflows, an idea that was initially proposed by Electric Fence [150] but can be seamlessly integrated with this key observation that reduces the scope of detection. Once the vulnerabilities are detected, Prober will immediately stop the execution and any subsequent exploits, and report the faulty instructions precisely.

The key challenge is to correctly identify all array-related heap objects. On the one hand, missing array-related objects will lead to no detection/protection of overflows caused by them, reducing the safety guarantee. On the other hand, if some unnecessary objects were included, it may impose some overhead unnecessarily. To this end, Prober proposes a **hybrid approach** to identify array-related objects. Some objects can be identified as array-related (or not) statically by analyzing the source code as described in Section 4.3, while the remaining ones will be identified in a hybrid way: Prober's static component (Prober-Static) identifies the basic type of such allocations (easier to do), instruments the size of such allocations with the compiler, and its runtime system (Prober-Dynamic) is responsible for determining whether it is an array-related object by the real allocation size.

That is, if the size of an allocation is multiple times of the basic type, then this allocation site is identified as *susceptible allocation site*. Consequently, all future allocations from such sites will be allocated from the protected heap so that all overflowing references can be detected and prevented immediately.

In this thesis, the contribution to Prober lies in Prober-Static side. In its implementation, Prober-Static relies on the LLVM compiler to perform the analysis and instrumentation at the Intermediate Representation (IR) level. Prober proposes to identify array-related allocations based on the allocation function, the definition of the `size` parameter, and the operations of the corresponding object. After that, Prober-Static further labels array-related allocation sites with simple instrumentation, so that Prober-Dynamic will place the corresponding objects in the protected heap. For objects that cannot be identified as array-related ones statically, Prober-Static simply labels the unit size so that Prober-Dynamic can determine its type dynamically. Overall, Prober is over-estimated so that it will not miss any array-related allocations.

To ensure that Prober-Static does not miss any necessary instrumentation, we have confirmed that Prober-Static instruments correctly for all known overflows collected by existing work [208]. Also, we further confirmed that Prober correctly detects and prevents 10 known overflows within real applications. The performance overhead evaluation shows Prober imposes only 1.5% performance overhead. Prober is ready for in-production systems due to its low overhead, timely prevention, and effectiveness.

Overall, in this thesis, the contribution to Prober system is as follows:

- We propose effective way to use compiler static analysis to identify the allocation site of the array-related objects.
- We implement a code transformation pass to successfully instrument the array-related objects based on LLVM.
- We proposes a hybrid mechanism that ensures to identify all array-related allocations. Such a mechanism is based on the allocation function, the definition of `size`

parameter and the operations on the corresponding object, or the combination of the unit size and the requested size.

- The thesis performs extensive evaluation on the effectiveness and performance of Prober, showing that Prober has the potential to be actually employed in the deployment environment.

The remainder of this chapter is organized as follows. Section 4.2 first describes the key observation, the basic idea of Prober, and then describes the attack model of Prober. The implementation details are further described in Section 4.3, and the evaluation is presented in Section 4.4. After that, we discuss Prober's weaknesses in Section 4.5. In the end, Section 4.6 discusses related work, and Section 4.7 concludes.

4.2 Overview

This section first analyzes overflow bugs from an existing study [208], and derives our *key observation*: overflowing objects are all related to arrays. Based on this key observation, it further discusses the basic design and key challenges of Prober in Prober-Static part.

Table 4.2: Analysis on 48 heap overflows collected by [208].

Type	Overflow Reason		Num(#)
Sub-structure overflows	Pointer arithmetic		0
	Loop operation		4
	System call		0
	String API	memcpy	2
		strncpy	3
		strcpy	1
Whole-structure Overflows	Pointer arithmetic		3
	Loop operation		20
	System call		2
	String API	memcpy	6
		strncpy	1
		strncmp	1
		memset	3
		sprintf	1
		memmove	1

4.2.1 Observations on Heap Overflows

One recent work studies 100 "randomly selected bugs within the buffer overflow category from the CVE website" [208]. Based on their description, the study is objective due to random selection, representing the real situation of buffer overflows. Therefore, our analysis was based on these bugs to avoid any bias. Based on our analysis, these 100 overflow bugs include 48 heap overflows, and 52 stack or global buffer overflows. This section focuses on 48 heap overflows, as shown in Table 4.2. We have the following observations.

The first observation is that all of the heap overflows are involved with arrays, either sub-structure or whole-structure overflows. Here, a *whole-structure overflow* is an overflow for which the allocation is an array of structures or basic units (e.g., characters, integers, or words). A *sub-structure overflow* is where object (or allocation) itself is not an array, but the corresponding structure includes one or multiple arrays internally. It is intuitive that array-related objects are prone to overflows. If an allocation is just a structure, every field can be manipulated with a member access operator (e.g., "`->`" or "`.`"), which should not cause the overflow. On the other hand, if an object is related to an array, then it is very likely to employ error-prone operations, such as pointer arithmetic instructions, string APIs, or loop operations.

The second observation is that whole-structure overflows are much more common than sub-structure overflows, consisting of around 79.2% of these bugs (with 38 bugs in total).

The third observation is that overflow bugs can be caused by multiple operations, such as pointer arithmetic instructions, string APIs, loop operations, or system calls, as further shown in Table 4.2. More specifically, 24 out of 48 overflows are related to loops during the iterations, and 19 overflows are related to string APIs. For instance, the `memcpy` function copies more memory than it should. These two categories actually consist of more than 89.5% of these bugs. In addition to these two categories, three overflows are related to pointer arithmetic, and two overflows occur when the `read` system call does not check the

boundary of the buffer. Thus, overflow occurs if programs utilize pointers to access the entry of an array, but without correctly checking its size.

Table 4.3: Heap overflows between 11/01/2018 and 02/15/2019.

Type	Overflow Reason		Num(#)
Sub-structure	Loop operation		4
	String API	sprintf	1
overflows	Pointer arithmetic		5
Whole-structure Overflows	Loop operation		15
	System call		1
	String API	memcpy	4
		strncat	1
		strncpy	1
		memset	2
		snprintf	1
		memmove	2

Confirming Key Observation: In order to further confirm our key observation, we further examined 65 heap overflow bugs reported in the National Vulnerability Database, with the published date between 11/01/2018 to 02/15/2019. Since only 37 bugs out of 65 bugs have a detailed description or have the source code information, we focused on these 37 bugs. Based on our analysis, all of these 37 bugs are array-related, where whole-structure overflows are still the most common types of overflows, with the percentage of 86.4% and a total of 32 bugs.

4.2.2 Basic Idea of Prober

Based on the key observation, Prober focuses only on array-related whole-structure overflows, where around 80% reported heap overflows belong to. Since array-related objects are only a small percentage of all heap objects, the detection overhead can be dramatically reduced as further evaluated in Section 4.4 when using the page-protection mechanism. Prober does not handle sub-structure overflows in this chapter, which will be the future work.

Basically, Prober includes two components, Prober-Static and Prober-Dynamic.

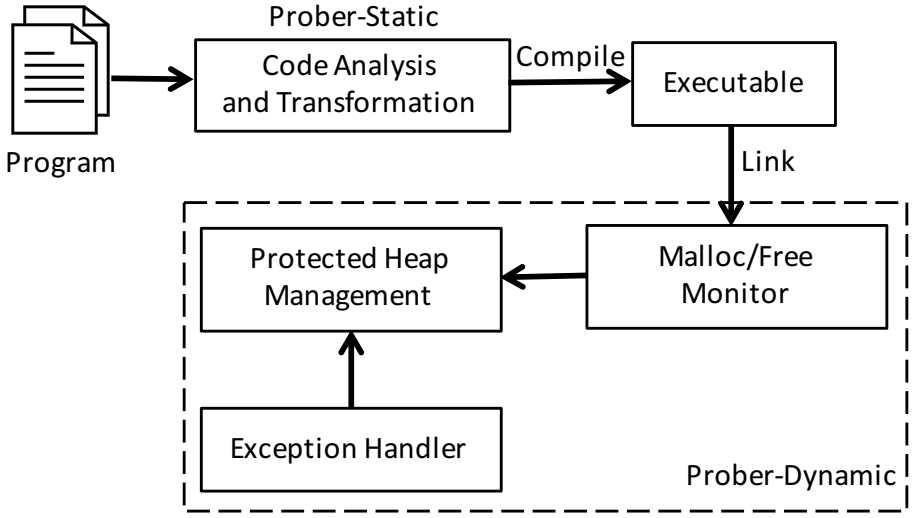


Figure 4.1: Overview of Prober.

Prober-Static is a static compile-time based tool that identifies and labels susceptible memory allocation sites, while Prober-Dynamic performs overflow detection/prevention and determines some array-related allocations on top of the static instrumentation. In this thesis, we only introduce the details of Prober-Static.

4.2.2.1 Prober-Static

Prober-Static performs analysis and instrumentation at the Intermediate Representation (IR) level because of multiple benefits. First, LLVM IR offers multiple built-in functions that can facilitate analysis and instrumentation. For example, *define-use* and *use-define* chains that track the definition and usage of memory allocation, can help determine an array-related allocation. Second, analysis and instrumentation algorithms on LLVM IR are more robust, because many complicated cases (e.g., various macros) at the source code are simplified or merged at the IR level. Third, instrumenting at IR level provides the flexibility of registering the new code transformation pass in an appropriate position of the compilation chain, thus avoiding the possible side-effects to the subsequent analysis and code optimizations (e.g., loop optimizations) that are crucial to the code performance.

Prober-Static analyzes the IR to determine array-related allocations, and marks susceptible allocation sites via the explicit instrumentation. Currently, Prober-Static is registered as a Link Time Optimization (LTO) pass so that it can handle definitions and usages located in multiple C/C++ files.

Research Challenges: The aim of Prober-Static is to design a robust compile-time analysis, which further includes two challenges. First, how to identify memory allocations, given that memory allocations have various forms, e.g., wrapper functions, or function pointers? Second, how to identify array-related memory allocations? Basically, Prober designs a hybrid mechanism to ensure correctness and completeness. If an allocation site can be identified statically, as described in Section 4.3, then it will be labeled explicitly. Otherwise, Prober-Static labels the size of its basic unit, and then relies on its dynamic component to determine array-related allocations.

4.3 Compiler Analysis and Instrumentation

This section describes the detailed design and implementation of Prober-Static.

Prober-Static performs its static analysis and instrumentation on LLVM IR to identify all susceptible allocations, and relies on dynamic confirmation to identify those that cannot be determined statically. Overall, our hybrid approach guarantees a 100% coverage for array-related allocations, which is is conservative and thus does not miss an allocation. Prober-Static is implemented as one Link Time Optimization (LTO) pass because of two major considerations. First, the allocation function may be located inside a wrapper function, but this wrapper function is invoked in another C file, so an inter-module analysis (provided by LTO) is required. Second, placing instrumentation at link-time can effectively avoid complicating or interfering performance-critical compile-time analysis and optimizations (e.g., varied loop optimizations). Prober-Static determines array-related allocations in three steps, as further described in Section 4.3.1.

4.3.1 Identify Susceptible Allocations

Prober-Static analyzes susceptible (or array-related) allocations in the following steps.

Step-I: Identify memory allocation functions: Based on our knowledge, memory allocations are invoked by several APIs and operators in C/C++, such as *new*, *malloc*(), *calloc*(), *realloc*(), *valloc*, *posix_memalign*(), and *memalign*(). But there are multiple situations as described in the following.

Basic Case: Some memory allocation invocations can be directly recognized according to the name in LLVM IR. For example, the `new[]` keyword is translated to `_Znam` in LLVM IR. Similarly, various macro definitions can also be recognized directly in IR level, because they have already been replaced by the preprocessor before being converted to IR.

Special Cases: Prober-Static also handles two more sophisticated but common cases. First, memory allocation is invoked inside a wrapper. For this case, Prober-Static recursively treats all functions in its calling stack as wrappers of memory allocation functions. Second, memory allocation is defined as a function pointer. Fortunately, LLVM translates function pointer calls to indirect calls in its IR, and the function invocation is specified by a load instruction. Listing 4.1 shows a simple example. The definition of `malloc_ptr` requires an additional check to determine whether line 2 is a memory allocation.

Listing 4.1: Memory alloc is defined and called as a fun ptr.

```
1  %4 = load i8* (i64)*, i8* (i64)**@malloc_ptr, align 8
2  %5 = tail call i8* %4(i64 %0)
```

Step-II: Identify array-related allocations: Prober-Static further identifies array-related allocations by the name of functions, the definition of allocation size, and the operations of the corresponding object. Table 4.4 lists multiple examples that cover 36 bugs analyzed in Section 4.2.1. The details of these examples are discussed as follows.

Type I can be identified by the name of memory allocation functions. For example,

`new[]` is known as an operator to allocate an array, and `calloc` allocates an array with multiple objects with the same size. 5 out of 36 cases belong to this simple type.

Type II, III, and IV can be identified by the definition of `size` parameter. If its `size` parameter is defined (or manipulated) by `multiplication`, `addition`, and `strlen` operations, then the corresponding allocation is array-related. We can easily understand this by checking its contradiction. If an allocation is just for a single structure, a `sizeof` operation will be used to compute the `size` parameter, without these operations. Prober-Static employs LLVM's built-in `def-use` and `use-def` chains to assist the analysis on the definition of `size` parameter. 24 out of 36 cases can be analyzed using this method.

Type V can be identified by the operation on corresponding objects. As we know, some APIs, such as `read`, `fread`, `pread`, `readv`, read multiple bytes from the network or a file to the local buffer. Therefore, whenever one object appears as the destination buffer of these system calls, it should be tracked. Based on our analysis, 2 out of 36 cases belong to this type. Similarly, the analysis also requires the support of LLVM's built-in `def-use` and `use-def` analysis.

Type VI requires further analysis, when the `size` parameter of an allocation is a constant integer. For most cases, if the `size` parameter is a constant, the corresponding allocation is an array. But there are some exceptions when analyzing in IR level. For instance, if a statement is like this, $(structS*)malloc(sizeof(structS))$, the `size` parameter is also interpreted as a constant integer in IR level. But this is not an array. To avoid the misidentification, Prober-Static further confirms whether the `size` is equal to the size of the corresponding data type. Although LLVM has some built-in functions to get the size of the object type, it requires some additional analysis to determine the object type. The challenge is to determine this when an allocation returns a void type pointer. Prober-Static adopts a `def-use` or `use-def` analysis to find the definition or the usage of the return value to figure out the object type.

Type VII is more complicated, since the object can be an array in some branches. More specifically, LLVM-IR represents these branches with a PHINode instruction.

Table 4.4: Examples of susceptible allocations.

Type	Example	Explanation	Count
I	... = (**int***) new[5]; ... = (**int***) calloc(5,**sizeof**(**int**));	Memory allocation calls `new[]` or `calloc`.	5
II	size = num * **sizeof**(**struct S**); ... = (**struct S***) malloc(size);	`size` is defined by a `multiply` operation.	13
III	size = size1 + size2; ... = (**struct S***) malloc(size);	`size` is defined by a `add` operation.	10
IV	size = strlen(buffer); ... = (**int***) malloc(size);	`size` is a return value of `strlen()`.	1
V	buffer = malloc(size); read(buffer, 0, size);	Object is operated by array-related syscalls.	2
VI	... = (**int***) malloc(const_value);	`size` is a constant.	3
VII	size = (i > 0 ? **sizeof**(**int**) : 10 * **sizeof**(**int**)); ... = (**int***) malloc(size);	`size` is from a branch that is potentially array-related.	2
SUM			36

Prober-Static tracks all incoming values of this PHINode instruction. If at least one value belongs to **Type II, III, or IV**, this allocation is treated as array-related conservatively.

After the above analysis, Prober-Static will determine most allocations array-related or not and selectively protect the arrays and ignore the ones that are not array-related.

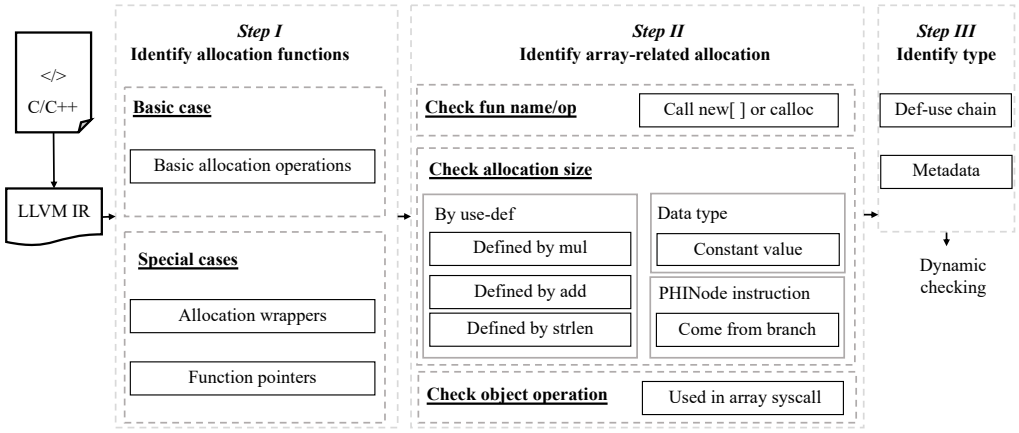


Figure 4.2: Identify susceptible allocations.

Step-III: Identify the object type (and unit size) for memory allocations non-determined: If a statement cannot be determined array-related or not in Step-II, Prober-Static labels the allocated object type (and thus the unit size) so that this allocation can

be determined dynamically by Prober-Dynamic. Prober-Dynamic collects the size of an allocation size and divides it by the unit size[1]. If this result is greater than one, Prober-Dynamic will protect this memory allocation.

Prober-Static mainly employs LLVM's built-in `def-use` chains to find an object's type in its usage site. Prober-Static also relies on the metadata in LLVM IR to find the type information. Listing 4.2 and 4.3 show two examples of finding the object type with `def-use` chains and metadata, respectively. Listing 4.2 illustrates that an explicit casting operation reveals the object type.

Listing 4.2: Identify the object type with a casting

```
1    %1 = call noalias i8* @malloc(i64 70) #4
2    %2 = bitcast i8* %1 to %struct.s*
```

Sometimes, it is difficult to find any obvious usage for a memory allocation, then the metadata information showed in Listing 4.3 also helps to find its object type.

Listing 4.3: Identify the object type with metadata

```
1  %21 = tail call i32 @mbuffer_create(%struct.mbuffer_t* nonnull %20, i64 %19) #7, !dbg
       ↪ !1409
2  !22 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
3  !1393 = !DILocalVariable(name: "r", scope: !1387, file: !137, line: 312, type: !22)
```

It is worth noticing that Prober-Static sets the type size as "1" by default, so even it cannot determine the object type statically via the above analysis, the allocation site will be protected effectively during the runtime. That is, Prober always ensures over-protection.

Summary: Figure 4.2 summarizes Prober-Static's implementation. In Step-I, it checks each LLVM IR instruction according to one basic case and two special cases to identify all

[1]A memory allocation might be used in more than one data types

invocations of allocation functions, allocation function wrappers, and allocation function pointers. In Step-II, only these allocation invocations are further identified based on the following order. First, it checks the function name and operator. Second, it checks the allocation size with use-def, data type, and PHINode instruction information. Finally, it checks whether the operations of the corresponding objects are related to some special system calls. If an allocation meets any of these cases, then it is array-related. Prober-Static labels it explicitly as shown in Section 4.3.2. Otherwise, Prober-Static finds the object type (and type size) with either `def-use` chains or metadata in LLVM IR, and instruments the size of the allocation before the allocation so that Prober-Dynamic will confirm it dynamically.

In real-world applications, pointers and alias variables may complicate this analysis in two aspects. First, an alias pointer points to the protected allocation. However, this will not cause any issue, since Prober detects any access on the protected pages, no matter whether they are accessed via an alias or not. Second, an allocation function contains pointers or alias variables as its size parameter. Prober-Static relies on LLVM's pointer and alias analysis functions to associate these pointers or alias variables to the actual size variable and then performs further analysis. The evaluation in Section 4.4 demonstrates that Prober-Static can successfully identify and instrument array-related allocations for 46 bugs.

Listing 4.4: A LLVM-IR instrumentation example with *new*.

```
1  @specialMalloc = external thread_local global i8, align 1
2  define dso_local i32 @main() #0 {
3    store volatile i8 −1, i8∗ @specialMalloc, align 1
4    %6 = call i8∗ @_Znam(i64 20) #2
5    ret i32 0
6  }
```

Listing 4.5: Equivalent C instrumentation of the *new* example.

```
1  extern __thread volatile bool specialMalloc;
2  int main(){
3      specialMalloc = −1;
4      int* b = new int[5];
5      return 0;
6  }
```

4.3.2 LLVM-IR Instrumentation

After a susceptible allocation site has been identified, a thread-local variable, e.g., *specialMalloc*, will be inserted to mark this site as a susceptible allocation. Here, the *specialMalloc* variable is an integer variable, with the value of "0" by default. This variable is set to "-1" before the allocation site if the allocation is array-related allocation. For instance, the *new* example of **Type I** in Table 4.4 is instrumented as Listing 4.4, where Listing 4.5 shows its equivalent C code for clarification. If a memory allocation is non-determinable statically, *specialMalloc* will be set as the size of the object type. Prober's runtime will determine if it should be protected.

4.4 Experimental Evaluation

We performed the experiments on a two-socket quiescent machine, where each socket is an Intel(R) Xeon(R) Gold 6138 processor with 20 cores. It has 200GB main memory, and 32KB L1, 1024 KB L2 and 28160 KB L3 cache. The experiments were performed on Ubuntu 18.04, installed with Linux-4.15.0 kernel. All applications were compiled with LLVM-8.0, by adding an analysis/instrumentation pass of Prober-Static.

4.4.1 Effectiveness

The effectiveness evaluation includes two parts, 38 bugs included in the existing study [208] and other overflow bugs included in other existing work, such as Bugbench [123], CVE database, or HeapTherapy [212].

4.4.1.1 38 Bugs from the Existing Study

For 38 bugs listed in the existing study, we confirm that Prober correctly instrumented 36 bugs. The remaining two bugs cannot be instrumented due to the invocation of external standard library calls (e.g., libstdc++), which are not analyzed (shared by instrumentation-based approaches). Therefore, Prober's evaluation presents high confidence on the actual overhead, since it could instrument all bugs correctly.

Note that we did not run these buggy applications directly, due to the following reasons. First, these bugs may not include erroneous inputs that are required to exercise them. Second, many of them are not compatible with modern libraries, which requires a significant amount of manual efforts for the compilation. Therefore, we only verify whether the corresponding bugs have been instrumented correctly.

4.4.1.2 Other Real-world Bugs

We performed the effectiveness evaluation on 10 other real-world bugs that are not listed in the existing study [208]. These applications and their specific bug trigger inputs are obtained from Bugbench [123] , CVE database, or HeapTherapy [212]. Among these 10 vulnerable applications, the heartbleed and libtiff-4.0.7 vulnerabilities are caused by buffer over-reads, while others are caused by buffer over-writes. The details of these applications are shown in Table 4.5, where all of these bugs can be detected by AddressSanitizer. Table 4.5 also listed the number of allocation sites that can be identified statically ("Static" column) and dynamically ("Dynamic"). Overall, Prober detects all known overflows without false positives. Upon detection, Prober stops the execution immediately (before the

crashes), and reports the type of an overflow (over-read or over-write), the call path of triggering the overflow, and the allocation site of the corresponding buffer. The evaluation confirms that Prober is able to detect real heap overflows with its proposed instrumentation and runtime system.

Table 4.5: Statically and dynamically identified callsites in buggy applications

Application	Reference	Static #	Dynamic #
bc-1.06	BugBench [123]	43	5
gzip-1.2.4	BugBench [123]	3	1
Heartbleed	CVE-2014-0160 [66]	9314	3941
LibHX-3.4	CVE-2010-2947 [29]	23	15
Libtiff-4.0.1	CVE-2013-4243 [30]	406	75
Libtiff-4.0.7	CVE-2016-10269 [48]	421	104
Memcached-1.4.25	CVE-2016-8706 [187]	80	19
openjpeg-1.3	CVE-2012-3535 [166]	756	201
polymorph-0.4.0	BugBench [123]	1	0
squid-2.3	BugBench [123]	83	175

4.4.1.3 Case Study

Figure 4.3 shows the bug report for the heartbleed vulnerability. Prober identifies that this bug is a buffer over-read problem. The bug report also includes the call stack of the faulty instruction (where the overflow occurs), and the call stack of this object's allocation site.

According to the bug report, the overflow occurs in the *memcpy*() function, which is invoked by the *tls*1_*process_heartbeat* function at line 2586 of `./ssl/t1_lib.c` file. By checking the source code, the corresponding statement is $memcpy(bp, pl, payload)$. According to the attribute of this problem–a buffer over-read problem, it is easy to know that the over-read issue is related to the source of `memcpy`, that is, either *pl* or `payload`. Since *pl* is the starting address of a normal heap object that is allocated at line 770 of `./ssl/s3_both.c` file, then the failure must be caused by `payload`. By examining the source code, we could easily find out that `payload` is computed from the length of the data

```
A buffer over−read problem is detected at:
../glibc/../multiarch/memcpy−avx−unaligned.S:237
../x86_64−linux−gnu/bits/string3.h:53
../openssl−OpenSSL_1_0_1f/ssl/t1_lib.c:2586
../openssl−OpenSSL_1_0_1f/ssl/s3_pkt.c:1092
../openssl−OpenSSL_1_0_1f/ssl/s3_both.c:457
...
../nginx−1.3.9/src/event/ngx_event.c:247
../nginx−1.3.9/src/os/unix/ngx_process_cycle.c:807
...
This object is allocated at:
../openssl−OpenSSL_1_0_1f/ssl/s3_both.c:770
../openssl−OpenSSL_1_0_1f/ssl/s3_pkt.c:949
../openssl−OpenSSL_1_0_1f/ssl/s3_both.c:457
```

Figure 4.3: Bug report for the Heartbleed Problem.

that the server receives from the network. Therefore, via the bug report, programmers can easily reason the root cause of overflow, and fix the problem correspondingly.

4.5 Limitations

Prober focuses on array-related heap overflows, representing over 86% of heap overflows based on our observations (Section 4.2). It cannot detect array-related internal-structure overflows, which is its biggest limitation. However, there is no fundamental reason why this cannot be done. It is possible to arrange the fields of the structure so that array(s) can be placed at the end of the corresponding structure. Adding the support for internal-structure overflows will be our future work.

Prober can only detect overflows landing on the protection page(s). Prober can be configured to change the pages for the protection if necessary. In theory, it is able to detect more errors than existing approaches with redzones, such as AddressSanitizer [169]. It currently cannot detect heap underflows. However, heap underflows cannot do any harm, since they can only land on the non-used area.

Prober only detects overflows when the source code is analyzed and instrumented by Prober-Static. This limitation is also shared by all instrumentation-based tools, e.g.,

EffectiveSan [61] or AddressSanitizer [169]. When an overflowing object is allocated in a library that is not instrumented, Prober cannot detect it. However, different from existing work that detects overflows by checking memory accesses, Prober can detect overflows caused by APIs of a non-instrumented library. This is a significant difference.

4.6 Related Work

We classify existing tools of detecting heap buffer overflow based on the type of approaches.

Static Detection: Many tools utilize static analysis to detect buffer overflow bugs [122, 71, 109, 112]. They only analyze software source code in order to reason which statements could potentially cause buffer overflows. However, some variables (e.g., indirect branches) could not be determined without the execution. Thus, they may generate many false positives or false negatives, which requires further manual efforts to confirm the reported bugs. In contrast, Prober never generates any false positives.

Dynamic detection: Several tools place an inaccessible memory page around every heap object [139, 140, 150, 212], which is similar to Prober. Memory accesses to the protected pages will generate a `SIGSEGV` signal. However, these existing work suffer from a prohibitively high performance overhead by protecting all pages or even probabilistically. Although Prober employs the same mechanism to detect heap buffer overflow, it narrows down heap objects that can potentially result in buffer overflows, which drastically reduces its performance overhead. Also, Prober has a carefully designed runtime system to reduce its overhead.

Static instrumentation-assisted detection: Numerous tools analyze source code to identify necessary instrumentation, which favors sanity checks at runtime [162, 6, 73, 80, 135, 149, 169, 62, 106, 41]. They instrument all memory accesses at compilation phases, and check the validity of accesses at runtime. AddressSantizer [169] is the state-of-art of this type of approaches, which further employs the static analysis to prune out certain unnecessary checks. However, AddressSanitizer still imposes non-negligible performance

overhead. Different from these tools, Prober does not check every memory access, but relies on page protection to detect overflows without checking overhead, if there is no overflow.

Dynamic instrumentation-assisted detection: Many dynamic analysis tools detect memory errors by checking memory accesses during runtime, such as Valgrind's Memcheck tool [136], Dr. Memory [28], Purify [81], Intel Inspector [84], and Sun Discover [144]. Due to the expensive instrumentation and inspection, they typically impose a too high-performance overhead to be employed in a production environment.

Hardware-assisted detection: A few tools rely on new hardware to detect buffer overflows. Intel MPX tries to reduce the overhead of pointer checks by embedding checks into a new hardware [142]. BOGO relies on Intel MPX to provide both spatial and temporal safety [213]. However, the overhead of validating every memory access is too high to be adopted in practice. Sampling-based techniques, such as CSOD [120] and Sampler [174], utilize hardware watchpoints or Performance Monitor Unit (PMU) hardware to monitor a few heap objects at one time or validate a subset of memory accesses. Although they impose low runtime overhead similarly as Prober, they cannot guarantee the same effectiveness as Prober, especially when there are a lot of heap objects. CHERI requires the cooperation of architecture, compiler, and operating system together to enforce memory safety [198], which inevitably increases developers' effort. Prober, which is a dynamically linked library, imposes little manual effort, without changing the underlying OS and requiring new hardware.

Postmortem detection: Some evidence-based tools detect buffer over-writes by appending canaries after each heap object and checks if canaries are corrupted at memory deallocations or epoch ends [211, 121, 119, 212]. Since read operations do not leave evidence, they cannot detect read-based buffer overflow, while Prober can detect both buffer over-reads and buffer over-writes. Also, evidence-based approaches cannot be applied in the security environment, since the attacks may already be issued successfully before performing the detection.

4.7 Summary

This chapter presents a novel system to defend heap overflows. It is based on a key observation obtained from the analysis of 48 real overflow bugs: overflowing objects are typically involved with arrays. Based on this observation, Prober takes a two-phase approach to detect heap overflows: its static component identifies all possible array-related allocations before the compilation, and then instruments the code correspondingly; its dynamic component further intercepts the allocations, and redirects the allocations from susceptible allocation sites to the protected heap in order to detect the overflows with the page protection mechanism. Overall, Prober only imposes around 1.5% performance overhead on average, but without compromising its effectiveness. The low overhead and the high effectiveness makes Prober an always-on approach for the production environment.

Chapter 5

High performance Sparse Tensor Algebra Compiler

Tensor algebra is widely used in many applications, such as scientific computing, machine learning, and data analytics. The tensors representing real-world data are usually large and sparse. There are tens of storage formats designed for sparse matrices and/or tensors and the performance of sparse tensor operations depends on a particular architecture and/or selected sparse format, which makes it challenging to implement and optimize every tensor operation of interest and transfer the code from one architecture to another. We propose a tensor algebra domain-specific language (DSL) and compiler infrastructure to automatically generate kernels for mixed sparse-dense tensor algebra operations, named SPACe. The proposed DSL provides high-level programming abstractions that resemble the familiar Einstein notation to represent tensor algebra operations. The compiler performs code optimizations and transformations for efficient code generation while covering a wide range of tensor storage formats. SPACe compiler also leverages data reordering to improve spatial or temporal locality for better performance. Our results show that the performance of automatically generated kernels outperforms state-of-the-art sparse tensor algebra compiler, with up to 20.92x, 6.39x, and 13.9x performance improvement, for parallel SpMV, SpMM, and TTM over TACO, respectively.

5.1 Introduction

Tensor algebra is at the core of numerous applications in scientific computing, machine learning, and data analytics. Tensors are a generalization of matrices to any number of dimensions, which are often large and sparse. Sparse tensors are used to represent a multifactor or multirelational dataset, and has found numerous applications in data analysis and mining [104, 158, 182] for health care [2, 125], natural language processing [24, 145], machine learning [117, 173], and social network analytics [216], among many others.

Developing optimized kernels for sparse tensor algebra methods is complicated. First, sparse tensors are often stored in a compressed form (indexed data structures) and computational kernels needs to efficiently loop over the nonzero elements of the tensor inputs. Second, iterating over nonzero elements highly depends on the particular storage format employed, hence many algorithms exist to implement the same operation, each targeting a specific format. Finally, applications may use multiple formats concurrently throughout the computation and mix different formats in the same operation to achieve high performance. When tensors with different storage formats are used in the same operation, there are two options: converting one (or both) tensor(s), which is time-consuming especially if the tensor is only used once, or developing an algorithm that can efficiently iterate over both formats simultaneously, which lacks generality and requires different implementations for each combination of tensor formats [17, 118].

The current solutions implement ad hoc high-performance approaches for particular computer architecture and/or format. Most of these algorithms tackle specific problems and domains and conveniently store sparse tensors in a format that exploits the characteristics of the problem. This approach has resulted in tens of different formats [179, 114, 137, 43, 17] to represent sparse tensors. Some of these formats are storage-efficient for specific inputs [43, 193, 105, 17] or evenly nonzero distributions across rows/columns [131, 43]; some are better suited for specific tensor computations, e.g., sparse matrix-vector multiplication [205, 204] versus sparse tensor-matrix multiplica-

tion [178, 17]; others are particularly designed for different computer architectures, such as CPUs [204, 115] versus GPUs [127, 129]. On the other hand, it is infeasible to manually write optimized code for each tensor algebra expressions considering the all possible combinatorial combinations of tensor operations and formats.

To solve the above challenges, we present a sparse tensor algebra compiler, named SPACe, that is agnostic to storage formats: as opposed to a library of sparse tensor methods, where the methods are statically defined, a compiler can automatically and dynamically generate efficient tensor algebra kernel specifically optimized mixed dense and sparse tensor expressions. SPACe Domain-Specific Language (DSL) is a highly-productive language that provides high-level programming abstractions that resemble the familiar Einstein notations [63] to represent tensor operations. SPACe is based on the Multi-Level Intermediate Representation (MLIR) [111] framework recently introduced by Google to building reusable and extensible compiler infrastructures. The key benefit of building on top of MLIR is its built-in performance portability. In the SPACe multi-level Intermediate Representation (IR), domain-specific, application-dependent optimizations are performed at higher levels of the IR stack where operations resemble programming languages' abstractions and can be optimized based on the operations semantics. Generic, architecture-specific optimizations are, instead, performed at lower-levels, where simpler operations are mapped to the memory hierarchy and to the processor's registers.

To enable modular code generation with respect to formats and combinations of formats, we employ four storage format attributes – *dense, compressed unique, compressed non-unique, and singleton* – which are assigned to each tensor dimension [99]. By properly combining those attributes in each dimension, it is possible to express common sparse tensor compressed formats, such as COO, CSR, DCSR, ELLPACK, CSF and Mode-generic. SPACe code generation algorithm analyzes the dimension attributes and produces code to efficiently iterate over the nonzero elements of the input tensors. Since the number of storage format attributes is far lower than all possible combinations of storage formats, the code generation algorithm is greatly simplified and yet can support most of the commonly

used sparse tensor storage formats and arbitrary combinations of those. This approach lets users not only mix and match storage format desired for their applications but also can enable custom formats without modifying the underlying compiler infrastructure. Once the loop form of a computation has been generated at the IR, SPACe either lowers the code for sequential or parallel execution. In the former case, SPACe produces a high-quality LLVM IR (which we show in this work has better loop unrolling and vectorization than an equivalent LLVM IR produced by `clang`); in the latter case, instead, SPACe lowers code to the async dialect for asynchronous task execution based on LLVM co-routines Compared to hand-tuned libraries [115, 19, 205, 114, 179, 137] and source-to-source compilers [99, 97, 95], our approach is more portable, flexible, and adaptable, as emerging architectures and storage formats can be added without re-engineering the computational algorithms. Finally, SPACe employs the state-of-the-art data reordering algorithm [116] to increase spatial and temporal locality on a modern processor.

We evaluated SPACe with 2833 sparse matrices and six tensors from the SuiteSparse Suite Matrix Collection [54], FROSTT Tensor Collection [176] and BIGtensor [87]. Our results show that SPACe can generate efficient code for multi-threaded CPU architectures from high-level descriptions of the algorithms. Compared to state-of-the-art high-productivity tensor algebra languages and compiler, SPACe provides on average 2.29x, up to 6.26x, performance improvements over the TACO compiler for sequential Sparse-Matrix Dense-Matrix (SpMM). We also show that asynchronous task execution outperforms OpenMP parallelization, especially for small input matrices, where runtime overhead is predominant. Our results show up to 6.39x and 13.9x speedup over TACO for SpMM and TTM, respectively. Finally, data reordering achieves up to 3.89x and 7.14x performance improvements for parallel SpMV and SpMM kernels, respectively, over the original SPACe.

To the best of our knowledge, SPACe is the *first* MLIR-based compiler that integrates generic code generation for arbitrary input formats, data reordering, and automatic parallelization within the same framework. SPACe can improve end-user application perfor-

mance while supporting efficient code generation for a wider range of formats specialized for different applications and data characteristics. This chapter makes the following contributions:

- We introduce the SPACe DSL, an intuitive yet powerful and flexible language to implement dense and sparse tensor algebra algorithms;
- We propose an MLIR-based compiler that automatically generates efficient sequential and parallel code for a tensor expression with dense and mixed operands while supporting the important sparse tensor storage formats.
- We integrate the state-of-the-art data reordering algorithm to enhance data locality.
- We provide an exhaustive experimental evaluation and show that SPACe generally outperforms state-of-the-art tensor compiler for both sequential and parallel execution.

5.2 Background and Motivation

There exist various compressed and uncompressed formats to store sparse matrices and tensors in the literature, including COOrdinate (COO), Compressed Sparse Row (CSR), Double Compressed Sparse Row (DCSR), ELLPACK, Compressed Sparse Fiber (CSF), and Mode-Generic [19, 67, 129, 40, 93]. The specific format chosen to represent data in an application generally depends on the expected characteristics of the data itself and how these impact other desired properties, such as performance of a computational kernel or memory footprint (which is particularly important in the case of very large, multi-dimensional tensors).

Each format is important for different reasons. COO [165, 9] is commonly used to store sparse matrices and tensors, such as the Matrix Market exchange format [1] and the FROSTT sparse tensor format [176]. While COO is the most natural format, it is not necessarily the most performant format. CSR [199] is for sparse matrices, which

compresses row indices as pointers to row beginning positions to avoid duplicated storage and increase performance for memory bandwidth-bound computation such as Sparse-Matrix Dense-Vector (SpMV). DCSR [31] further compresses zero rows by adding an extra pointer to nonzero rows based on the CSR format. With an extra level of compression on rows, DCSR is more efficient than CSR for highly sparse (hypersparse) data. The ELLPACK [93] format is efficient for matrices that contain a bounded number of nonzeros per row, such as matrices that represent well-formed meshes. CSF [179] generalizes the DCSR or CSR matrix format to high-order tensors that compresses every dimension. Mode-Generic format [17] is a generic representation of semi-sparse tensors with one or more dense dimensions stored as dense blocks with the coordinates of the blocks stored in COO.

An application might need one or more of these formats based on its needs, which makes it important to support computation with various tensor storage formats and their combinations. The main challenge is that the computational kernel needs to effectively iterate over each sparse input tensor stored in different storage formats. This problem is especially more complicated for expressions that involve multiple operands.

Because of the large number of storage formats and possible combinations, most state-of-the-art sparse tensor libraries support only a few sparse formats (and generally only binary operations) or convert tensors to an internal storage format, thereby potentially losing the performance, memory footprint, or other advantages that a specific format may offer. A compiler, on the other hand, can automatically generate the efficient code for specific input formats and their combinations, increasing flexibility, adaptivity to new formats, and portability to various hardware platforms. To achieve this goal, two important requirements need to be satisfied: 1) a unified way to represent important sparse storage formats (Section 5.4) and 2) an efficient algorithm to generate specific code for a given expression and its particular input formats (Section 5.6).

```
1  def main() {
2      #IndexLabel Definition
3      IndexLabel [a] = [?];
4      IndexLabel [b] = [?];
5      IndexLabel [c] = [32];
6
7      #Tensor Definition
8.     Tensor<double> A([a,b],CSR);   #Tensor<double> A([a,b],{D,CU});
9.     Tensor<double> B([b,c],Dense); #Tensor<double> B([b,c],{D,D});
10     Tensor<double> C([a,c],Dense); #Tensor<double> C([a,c],{D,D});
11
12     #Tensor Readfile Operation
13     A[a,b] = space_read(filename);
14
15    #Tensor Fill Operation
16    B[b,c] = 1.0;
17    C[a,c] = 0.0;
18
19    #Tensor Contraction
20    C[a, c] = A[a,b] * B[b,c];
21 }
```

Figure 5.1: An example `SPACe` program for Sparse Matrix-times-Dense-Matrix operation.

5.3 SPACe Overview

SPACe consists of a DSL for tensor algebra computations, a progressive lowering process to map high-level operations to low-level architectural resources, a series of optimizations performed in the lowering process, and various IR dialects to represent key concepts, operations, and types at each level of the multi-level IR. This section reviews the key characteristics of our compiler framework. SPACe is based on the MLIR framework [111], a compiler infrastructure to build reusable and extensible compilers and IRs. MLIR supports the compilation of high-level abstractions and domain-specific constructs and provides a disciplined, extensible compiler pipeline with gradual and partial lowering. Users can build domain-specific compilers and customized IRs (called *dialect*), as well as combining existing IRs, opting into optimizations and analysis.

Our previous work focuses on *dense* high-dimensional tensor contractions. The compiler reformulates tensor contractions as a sequence of transpose and matrix-matrix multiplication operations, then generates efficient code by several code optimizations (e.g.,

loop tiling, micro kernel). The detailed description of previous work and its performance results for important tensor expressions from the Northwest Chemistry framework (NWChem) [191] can be found in [133]. This work, instead, focuses on *sparse* tensor algebra.

Figure 5.1 shows an example SPACe program for an SpMM operation. The `IndexLabel` operation defines an index label. It can assign the size of the index with a scalar number. If the size is unknown in static time, then a question mark (?) is used (Lines 3-5). The `Tensor` operation defines new tensors (Lines 8-10); the SpMM operation is defined at Line 20. In particular, the matrix `A` is stored in the CSR format while the matrix `B` and the result matrix `C` are dense. Note that there is no specific operation for SpMM at language level, nor the programmer needs to explicitly state the format of each input tensor while contracting the two tensors. SPACe atomically derives the specific operation from the format of input tensors and the index labels. SPACe internally annotates each tensor with storage format attributes, devises the storage formats used in the contraction, and properly passes this information down to the IR stack when lowering the code. SPACe can generate the appropriate code according to the input tensor storage formats (Section 5.6.2).

The code generation in SPACe follows a progressive lowering approach where optimizations are applied at different levels. Figure 5.2 shows the compilation pipeline of SPACe, where our contributions are annotated by the dashed box. Users express their computation in a high-level tensor algebra DSL (Section 5.5). First, the SPACe DSL is lowered to a Sparse Tensor Algebra (TA) IR, the first dialect in the SPACe IR stack. The language operators, types, and structures are first mapped to an abstract syntax tree and then to the TA dialect. The TA dialect contains domain-specific concepts, such as multi-dimensional tensors, contractions, and tensor expressions. Our compiler framework applies high-level optimizations and transformation leveraging semantics information carried from the DSL. For example, SPACe tracks the input tensors' definitions and annotates each tensor with storage format attributes on each dimension, based on the index label definitions.

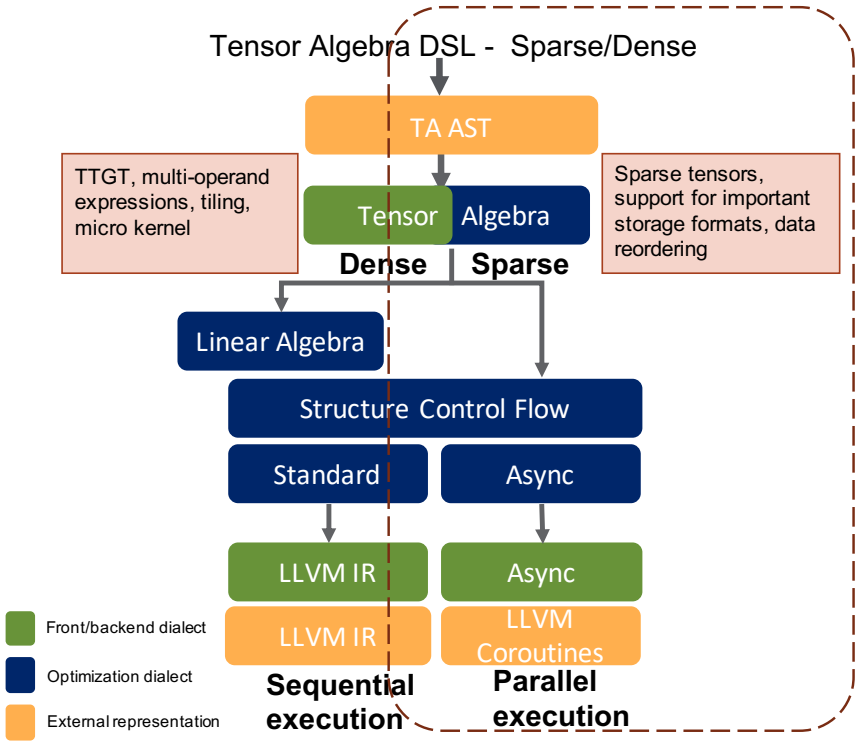
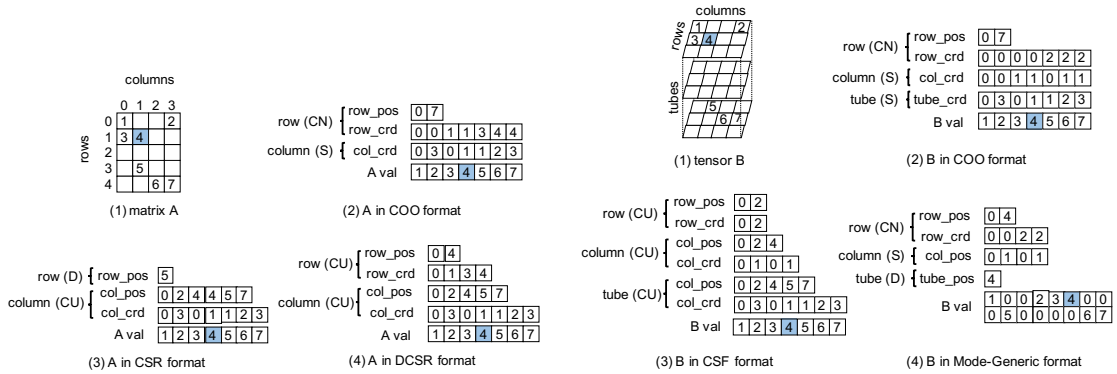


Figure 5.2: SPACe execution flow and compilation pipeline

Next, our compiler lowers the Tensor Algebra (TA) IR code to lower levels of the IR stack, which follows different paths depending on the operation and input formats. Dense tensor algebra operations are lowered first to the linear algebra dialect, then to the Structured Control Flow (SCF) dialect, and finally to the standard dialect. Sparse linear algebra operations are lowered to the SCF dialect which is a loop represented in the MLIR framework. At this point, SPACe employs generic optimizations during the lowering steps but also considers additional information about the final target architecture. For CPU execution, the code is lowered to the Low-Level Virtual Machine (LLVM) dialect for sequential execution and the async dialect to models asynchronous execution at a higher-level and then to proper LLVM IR for final assembly and linking.



(a) Sparse Matrix (b) Sparse Tensor

Figure 5.3: Example matrix and tensor represented in different formats. Each format is a combination of the storage format attributes.

5.4 Tensor Storage Format

As reported in Section 5.2, to support multiple sparse storage formats a compiler needs a uniform way to represent each tensor in memory. This internal storage formats need to preserve the characteristics of the original format, e.g., data compression or performance for specific sparse patterns, while allowing a unified algorithm to generate efficient code for each computational expression. SPACe defines a set of storage format attributes for each dimension to represent various sparse tensor formats. Code generation is then based on each dimension's storage format attributes rather than the whole format, which greatly reduces the number of formats and combinations that a compiler needs to support. Importantly, SPACe does not convert the original data layout into a different storage format. Instead, the storage format attributes are used to compose meta-data information that describes the original format, i.e., the data layout of the original format is preserved in memory and retains the original characteristics (compression, locality, etc.).

Representing every tensor dimension separately has been shown to be an effective way to *generalize* tensor storage formats and support efficient code generation [98]. Representing each dimension independently makes it easier to manage, adapt, and convert formats and to generate computational kernels uniformly. SPACe defines the following four storage

format attributes borrowed from [179, 98, 114]:

Dense (D). This dimension is in the "dense" format, i.e., all coordinates in this dimension will be accessed during the computations. For this format, we only use one scalar number stored in the `pos` array to represent the size of this dimension, such as the `row` dimension in Figure 5.3a(3).

Compressed_Unique (CU). This dimension is in a "compressed unique" format, i.e., the coordinates of nonzero elements in this dimension are compressed, and only the unique (no duplication) ones are stored in the array `crd`. It uses another array `pos` to store the start position of each unique coordinate, such as the `row` dimension Figure 5.3a(4), where the elements 1 and 2 are in the same row, but only one row coordinate is stored in `row_crd` array.

Compressed_Nonunique (CN). This dimension is in a "compressed non-unique" format, i.e., all the coordinates of nonzero elements will be recorded in `crd` array, and every coordinate in the `crd` array will be accessed one by one. CN then stores the start and the length of the `crd` array to the `pos` array, such as the `row` dimension Figure 5.3a(2), where all the row coordinates of the nonzeros are stored in `row_crd` array, `row_pos` only stores the start and the length of the `row_crd` array.

Singleton (S). The dimension is in a "singleton" format, i.e., all the nonzero coordinates are recorded to the array `crd` without any other information, such as the `column` dimension Figure 5.3a(2), only the column coordinates of the nonzeros are stored in `row_crd` array.

Internally, each tensor dimension is described by two arrays, a position (`pos`) and a coordinate (`crd`) array. **D** only uses the `pos` array to store the size of the dimension; the compressed storage format attributes **CU** and **CN** use both `pos` and `crd` arrays to store the nonzero coordinates and their positions; **S** only uses the `crd` array to store the nonzero coordinates in the dimension.

Furthermore, Figure 5.3 shows two examples that store a sparse matrix and a sparse tensor, respectively, in three formats (COO, CSR, and DCSR) with the representation of varied storage format attributes combinations. By properly combining the tensor storage format attributes, SPACe can represent the important sparse storage formats, including COO, CSR, DCSR, BCSR, CSB, ELLPACK, CSF and Mode-generic, in a uniform way, while retaining each format's characteristics.

5.5 SPACe Language Definition

SPACe provides a high-level Tensor Algebra DSL that increases portability and productivity by allowing scientists to reason about their algorithms implementation in their familiar notation and syntax. Specifically, SPACe DSL allows scientists 1) to express concepts and operations in a form that closely resembles their familiar notations and 2) to convey domain-specific information to the compiler for better program optimization. For example, our language represents Einstein mathematical notation and provides users with an interface to express tensor algebra semantics. The same SPACe program can be lowered to different architectures, and the lowering steps can follow different optimizations and lowering algorithms, allowing SPACe to produce high-quality code for target architectures without excessive burden on the programmer (see Section 5.6). This work extends the SPACe tensor algebra language to support sparse tensor algebra operations and syntax, the storage formats described in the previous sections.

Furthermore, we extend SPACe to support dynamic data types. As discussed above, Figure 5.1 shows an example of a SPACe program. In the SPACe language, a tensor object refers to a multi-dimensional array of arithmetic values that can be accessed by indices. Range-based index label constructs (`IndexLabel`) represent the range of indices expressed through a scalar, a range, or a range with increment. Index labels can be used both for constructing a tensor or for representing a tensor operation. Different from the original SPACe compiler [133], `IndexLabel`s can now be defined as *static* or *dynamic*.

Static `IndexLabel`s explicitly state the size of the dimension (Line 5) while dynamic `IndexLabel`s (Lines 3 and 4) only indicate that there exists a dimension, but the size will be determined later on during the execution of the program. Dynamic and static index labels differ in that dynamic index labels indicate an unknown size through a question mark (`?`) operator while static index labels explicitly state the size of the dimension through a scalar value.

A tensor is constructed by defined static or dynamic index labels and by declaring the sparsity of each dimension, according to the internal storage format described in the previous section. In Figure 5.1 tensor `A` is stored in CSR format, while tensors `B` and `C` are stored in dense format. Note that SPACe provides convenient notation to represent the most common tensor storage format, avoiding the need to specify the storage format for each dimension, as described in the comments at Lines 8-10. Internally, however, SPACe reasons in terms of sparsity on each dimension when generating code.

In the example SPACe program in Figure 5.1, the tensor `A`, `B`, and `C` are initialized with a tensor file by `space_read()`, the constant value `1.0`, and the constant value `0.0`, respectively. The function `space_read()` first reads a tensor from the file in `COO` format and then converts it to our internal storage format (see Section 5.4) to represent `CSR`. We implement `space_read()` as a *runtime function*, and it can be called in the SPACe program directly.

The last line in the program performs the SpMM operation. However, users need not explicitly state that the operation is an SpMM but can simply use the common tensor contraction `*` operator. SPACe will infer that the operator refers to an SpMM operation from the storage format of the input tensors, in this case, a sparse matrix and a dense matrix, and will generate the proper code to iterate over the specific storage format through rules generated from the definition of storage format attributes. Also, note that SPACe employs index labels to determine the type of operation to perform. For example, the `*` operator refers to a tensor contraction if the contraction indices are adjacent or to element-wise operation otherwise. In Figure 5.1, the index label `b` is used as contraction indices between

```
1  #map0 = affine_map<(d0, d1, d2) -> (d0, d1)>
2  #map1 = affine_map<(d0, d1, d2) -> (d1, d2)>
3  #map2 = affine_map<(d0, d1, d2) -> (d0, d2)>
4  module {
5    func @main() {
6      %c0 = constant 0 : index
7      %c1 = constant 1 : index
8      %c32 = constant 32 : index
9      %a = "ta.index_label_dynamic"(%c0, %c1) : (index, index) -> !ta.range
10     %b = "ta.index_label_dynamic"(%c0, %c1) : (index, index) -> !ta.range
11     %c = "ta.index_label_static"(%c0, %c32, %c1) : (index, index, index) -> !ta.range
12     %A = "ta.tensor_decl"(%a, %b) {format = ["D", "CU"]} : (!ta.range, !ta.range) -> tensor<?x?xf64>
13     %B = "ta.tensor_decl"(%b, %c) {format = ["D", "D"]} : (!ta.range, !ta.range) -> tensor<?x32xf64>
14     %C = "ta.tensor_decl"(%a, %c) {format = ["D", "D"]} : (!ta.range, !ta.range) -> tensor<?x32xf64>
15     %labeledA = "ta.labeled_tensor"(%A, %a, %b) : (tensor<?x?xf64>, !ta.range, !ta.range) -> tensor<?x?xf64>
16     %read_data = "ta.generic_call"() {callee = @space_read,  filename = "dataset.mtx"} : () -> tensor<*xf64>
17     %setop = "ta.set_op"(%labeledA, %read_data) {__beta__= 0.000000e+00 : f64} :
                 (tensor<?x?xf64>, tensor<*xf64>) -> tensor<?x?xf64>
18     "ta.fill"(%B) {value = 1.0 : f64} : (tensor<?x32xf64>) -> ()
19     "ta.fill"(%C) {value = 0.0 : f64} : (tensor<?x32xf64>) -> ()
20     "ta.tc"(%A, %B, %C) {alpha = 1..000000e+00 : f64, beta = 0..000000e+00 : f64,
              format = [["D", "CU"], ["D", "D"], ["D", "D"]],  indexing_maps = [#map0, #map1, #map2]} :
              (tensor<?x?xf64>, tensor<?x32xf64>, tensor<?x32xf64>) -> ()
21     "ta.return"() : () -> ()
```

Figure 5.4: Generated sparse tensor algebra dialect for SpMM operation

`A` and `B` (adjacent or internal indices), thus the operator `*` refers to a tensor contraction. Therefore, *SPACe can not only support tensor contraction but are generally applicable to many other operations as well.* In conclusion, the SPACe TA language simplifies writing tensor algebra program by supporting common programming paradigms and enables users to express high-level concepts in their preferred notation.

5.6 Compilation Pipeline

We introduce the sparse tensor algebra dialect in MLIR to support mix dense/sparse tensor algebra computations with a wide range of storage formats. We use format attributes to represent each dimension's sparsity format in a uniform way in the proposed TA IR. SPACe compiler generates efficient code based on the represented format attribute per dimension. This section describes the compiler framework, which consists of two main parts: 1) a sparse MLIR TA dialect to represent tensor storage formats and operations, and 2) code generation algorithms to generate efficient serial and parallel code starting from the proposed TA DSL.

5.6.1 Sparse Tensor Algebra Dialect

SPACe supports a uniform tensor storage format based on the attributes described in Section 5.4 and the tensor algebra operations supported in our DSL. Figure 5.4 shows the generated tensor algebra IR for the SpMM program in Listing 5.1. The rest of this section details the various operation in the sparse TA dialect.

Static/Dynamic Index Labels. The sparse tensor algebra dialect supports two types of index labels, static and dynamic. If the dimension size of the index is known at compile-time, SPACe uses `ta.`*index_label_static* to represent the index label. It has three operands, which represent the start, end, and step value on this index. `ta.`*index_label_dynamic* is used to represent the index label when the dimension size is unknown at compile time. `ta.`*index_label_dynamic* has two operands, the start, and step value on this index. The end value on this index will be known at runtime.

Sparse Tensor Declaration. In the sparse tensor algebra dialect, the tensor is declared with `ta.`*tensor_decl* operation. The operands of `ta.`*tensor_decl* are the index labels of the tensor. It can contain an arbitrary number of operands, which means it can declare arbitrary dimensional tensor. The `ta.`*tensor_decl* operation also contains storage format attributes of the tensor in each dimension for sparse tensors.

Sparse Tensor Operations. The sparse TA dialect also defines the tensor algebra operations supported by SPACe. For example, the tensor contraction `ta.tc` operation for an SpMM computation (shown in line 20 of Figure 5.4) takes two input tensors and computes the result of the contraction. The first and second operands (%A and %B) are input tensors, and the third operand (%C) is the output tensor. "ta.sptensor<tensor<?×i32>, tensor<?×i32>, tensor<?×i32>, tensor<?×i32>, tensor<?×f64>>" is the data type for %A, while "tensor<?×32×f64>" is the data type for %B, and "tensor<?×32×f64>" is the data type for %C. "-> ()" represents the return type which is `void`.

We introduce a `formats` attribute to extend the original `ta.tc` to provide the storage format information of each input tensor. In line 20 of Figure 5.4, the first tensor is in

```
%A = "ta.sptensor_construct"(%A1pos, %A1crd, %A2pos, %A2crd, %Aval) :
      (tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xi32>,
      tensor<?xf64>)  ->
      (!ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>,
      tensor<?xi32>, tensor<?xf64>>)
```

Figure 5.5: Sparse tensor data structure construction operation

the CSR format, while the second and third are all Dense tensors. The code in the figure shows that each input tensors is associated with its storage format information. We also introduce `indexing_maps` to `ta.tc` to represent the indices of each tensor. The `indexing_maps` helps propagate indices information along with the lowering stack. The tensor expression and the storage format information will be further propagated down to the lower level of the IR to provide the format attribute in each dimension when generating the computational code.

Sparse Tensor Data Type. As described in Section 5.4, a tensor T consists of k dimensions d_i for $0 \leq i \leq k-1$, where every dimension d_i is associated with a uniform storage attribute $a_i \in \{\mathbf{D}, \mathbf{CU}, \mathbf{CN}, \mathbf{S}\}$. SPACe associates two arrays `crd` and `pos` to each dimension to describe the storage format (meta-data). In the TA dialect, we define a sparse tensor as a `struct` data structure, which contains the nonzero indices in each dimension and their values.

Figure 5.5 shows how a 2D sparse matrix is represented in our TA dialect. In Figure 5.5, `ta.sptensor_construct` is the function to construct the sparse tensor struct, which is implemented as an operation in the TA dialect. The `sptensor_construct` operation takes the `pos` and `crd` arrays in each dimension (%A1pos, %A1crd, %A2pos, %A2crd) and the nonzero values (%AVal) as input, and returns a `ta.sptensor` type data structure that represents a sparse tensor in the TA dialect. The tensor types within `ta.sptensor` represent the `pos` and `crd` arrays corresponding to each dimension of the tensor itself (see Section 5.4). In the `ta.sptensor` structure, the type of %A1pos, %A1crd, %A2pos, %A2crd are tensor<?×i32>, the type of %Aval is tensor<?×f64>.

5.6.2 Sparse Code Generation Algorithm

SPACe lowers the code from high-level SPACe DSL language to low-level machine code in multiple lowering steps.

DSL Lowering. The first step in our compilation pipeline consists of lowering the high-level SPACe DSL into the sparse TA dialect. Figure 5.4 shows the TA dialect corresponding to the SPACe code presented in Figure 5.1. In Figure 5.4, "`ta.`" represents the tensor algebra dialect. The `indexLabel` operation in SPACe DSL will be lowered either into a `ta.index_label_static` operation or a `ta.index_label_dynamic` operation (e.g., Lines 9-11 in Figure 5.4) based on whether the size of the dimension represented by the index label is known or unknown at compile time. The `ta.index_Label` operation has three parameters (%A, %a, and %b), which are the start, the end, and the iteration step values in the dimension represented by the index label. The `IndexLabel` at Lines 3-4 of Figure 5.1 has an unknown size, so it will be lowered into the `ta.index_label_dynamic` operation, which only contains the start value of the dimension. The dimension size will be inferred during the runtime.

Progressive Lowering. Next, the sparse TA dialect is further translated to lower MLIR dialects. We describe this lowering process in two parts, early lowering and late lowering .

First, in the early lowering step, SPACe lowers all the operations in the sparse TA dialect, except the `ta.tc` operation. In particular, the `ta.tensor_decl` operation, which declares a tensor, is lowered into `alloc` and `tensor_load` operations, which are standard dialect operations in `std` dialect for dense. For sparse tensors, `ta.tensor_decl` operations are lowered into more, a composition of `alloc` and `tensor_load` operations for `pos` and `crd` arrays to store the coordinates of nonzeros in each dimension, and `val` array to store nonzero values. These coordinates of nonzeros are later used by `ta.sptensor_construct` operation (Figure 5.5) to construct a sparse tensor. To fill the `pos`, `crd`, and `val` arrays, the `ta.generic_call` operation is invocated to to call the `space_read()` function.

The `ta.generic_call` operation is then lowered to the `call` operations in the MLIR `std` dialect. The `ta.fill` operation initializes dense tensors with identical values. The `ta.fill` operation will be lowered into the `fill` operation in the MLIR *linalg* dialect. The `ta.return` operation returns the function, and is lowered into `return` operation in the MLIR *std* dialect. The `ta.index_label_dynamic` operations is lowered into the `ta.index_label_static` operation when the index label is identified from the input file.

Second, in the late lowering step, `ta.tc` operations are lowered into the MLIR `scf` (structure control flow) dialect operations. Figure 5.7 describes the lowering algorithm to `ta.tc` with an example mix sparse dense tensor contraction operation, where a sparse tensor A times a dense tensor B, and the output can be either sparse or dense. The algorithm takes `ta.tc` as input, and automatically generates the computational kernel code of a combination of `scf` and `std` dialects. `ta.tc` is the sparse tensor algebra dialect of the tensor contraction operation presented at Line 20 in Figure 5.1. As shown at Line 20 in Figure 5.4 `ta.tc` operation is lowered based on the code generation algorithm in Figure 5.7.

Figure 5.7 shows SPACe's code generation algorithm that consists of three key steps. This algorithm is general, applicable to varied tensor algebra operations, and can generate arbitrary index permutations. Moreover, in contrast to TACO, SPACe can generate sparse output. Take tensor expression $C_{ik} = A_{ij} * B_{jk}$ as an example, and assume the format of A is [**D**, **CU**], B is [**D**, **D**] and C is [**D**, **D**], respectively. The basic idea of this code generation is as follows:

Step-I (Line 1 to Line 3) collects both index information as well as the format attribute of each index. The above sample tensor expression has three indices ($allIndices = \{i, j, k\}$). The order of these indices matters, and is decided by tensor access orders. The format attribute of each index is decided by the usage of this index. If this index appears in dense input tensors only, its format attribute is **D**; otherwise, the format attribute is decided by the corresponding dimension of the sparse tensor. For the above sample tensor expression, the format attribute of index i is **D** and j is **CU** (both decided

by sparse input tensor A), and k is **D** (decided by dense input tensor B), respectively. After collecting this information, this algorithm defines three index variables ($vIdx_A$, $vIdx_B$ and $vIdx_C$) to access the value array of tensor A, B and C, respectively (Line 3).

Step-II (Line 4 to Line 19) iterates each index to generate loop structure code (as the algorithm line starting with "emit" shows). It leverages the aforementioned definition of each storage format attribute to find nonzero coordinates in each dimension via `pos` and `crd` arrays (e.g., d_pos and d_crd in the algorithm). Table 5.1 shows the sample loop code in C language for each format attribute. Besides generating loop structure code for each index, this step also updates three index variables ($vIdx_T$, $T \in \{A, B, C\}$) that will be used for the inner-most computation. If the format attribute of an index (e.g., d) is D, i.e., d only appears in dense tensors, then $vIdx_T = vIdx_T \times d\_SIZE + arg$, where T denotes all dense tensors that contain index d, arg is the coordinate on index d (i.e., the argument of the generated loop for index d), and d_SIZE is index d's dimension size. If the format attribute of index d is sparse (e.g., CU), this step handles sparse tensors and dense tensors separately. For sparse tensors T that contain index d, $vIdx_T = arg$, where arg is still the argument of the generated loop for index d. For dense tensors T that contain index d, $vIdx_T = vIdx_T + d\_crd[arg]$, where d_crd is the `crd` array of index d, and $d\_crd[arg]$ is the coordinate.

Step-III (Line 20) generates the inner-most computation code to load values from $A[vIdx_A]$ and $B[vIdx_B]$, compute their product, and update $C[vIdx_C]$, after step-II generates $vIdx_T$ for tensor T ($T \in \{A, B, C\}$).

Table 5.1: Generated code to access nonzeros coordinates

Attr	Corresponding code
D	`for i from 0 to pos[0] { ... }`
CU	`for i from pos[m] to pos[m+1]{ idx = crd[i];}` (m: The argument of the upper level loop. m is 0 when the dimension is the first dimension of the tensor)
CN	`for i from pos[0] to pos[1]{idx = crd[i];}`
S	`idx = crd[m];`

```
1  %A1SIZE_i32 = load %A1pos[%c0] : memref<?xi32>                          } 6-7
2  %A1SIZE = index_cast %A1SIZE_i32 : i32 to index                         }
3  scf.for %i = %c0 to %A1SIZE step %c1 {                                  }
4     %next_i = addi %i, %c1 : index                                       } 8-9
5     %A2pos_start_i32 = load %A2pos[%i] : memref<?xi32>                   }
6     %A2pos_start = index_cast %A2pos_start_i32 : i32 to index            }
7     %A2pos_end_i32 = load %A2pos[%next_i] : memref<?xi32>                }
8     %A2pos_end = index_cast %A2pos_end_i32 : i32 to index                }
9     scf.for %arg1 =  %A2pos_start to %A2pos_end step %c1 {               }
10       %j_i32 = load %A2crd[%arg1] : memref<?xi32>                       }
11       %j = index_cast %j_i32 : i32 to index                             }
12       scf.for %k = %c0 to %c32 step %c1 {                               } 6-7
13          %Avalue = load %Aval[%arg1] : memref<?xf64>                    } 20
14          %Bvalue = load %B[%j, %k] : memref<?x32xf64>                   }
15          %product = mulf %Avalue, %Bvalue : f64                         }
16          %Cvalue_old = load %C[%i, %k] : memref<?x32xf64>               }
17          %Cvalue = addf %Cvalue_old, %product : f64                     }
18          store %Cvalue, %C[%i, %k] : memref<?x32xf64>                   }
19 }}}
```

Figure 5.6: Lowered `scf` dialect code example for SpMM in the CSR format. The right side numbers represent line numbers in Algorithm 5.7

5.6.3 Parallel Code Generation

For sequential execution SPACe lowers the `scf` dialect to the `llvm IR` dialect and then to proper LLVM IR for assembly and linking. For parallel execution, instead, the `scf` dialect is lowered to the `async` dialect (See Figure 5.2). In details, we developed a pass to lower `scf.for` loops to `scf.parallel` loops and the latter to the `async` dialect. The `async` dialect encapsulates the semantics of an asynchronous task-based parallel runtime in which computational tasks are spawned and asynchronously executed by parallel worker threads. Currently, MLIR supports a task continuation stealing approach (like Cilk [21]) in which the control is returned to the parent task after spawning. The dialect provides semantics primitives to synchronize the execution of tasks. SPACe lowers those asynchronous tasks execution primitives to LLVM co-routines in LLVM IR, which is then passed to the assembler and linker to create a binary. As Figure 5.8d shows, the MLIR asynchronous

```
# TensorExpr e.g. Cik=Aij*Bjk; Format e.g.A[D, CU], B[D, D], C[D, D]
CodeGen(TensorExpr, Format)     :
1.  Generate iteration graph iGraph with the tensor expression information
2   Extract format attr for each index, then put them into formats
    # Define variables to store coordinates in the value array of each tensor
3.  Value-Indices vIdx_A, vIdx_B, vIdx_C = 0

    # Generate for loops based on the format (iterate over all indices in the iteration graph)
4.  for d in iGraph do
5.    switch(formats(d))
6.      case D:  emit-for(arg = 0 to d_pos[0])
7.                 vIdx_T = vIdx_T * d_SIZE + arg          # T ∈ {all tensors that contain index d}
          # m is 0 when d is the first index in the input tensor;
          # Otherwise, m is the argument of the upper-level loop
8.      case CU: emit-for(arg = d_pos[m] to d_pos[m + 1])
9.               emit-load(d_crd, arg)
10               vIdx_T = arg       # T ∈ {all sparse tensors that contain index d}
11.              vIdx_T = vIdx_T * d_SIZE + d_crd[arg]   # T ∈ {all dense tensors that contain index d}
12.     case CN: emit-for(arg = d_pos[0] to d_pos[1])
13.              emit-load(d_crd, arg)
14               vIdx_T = arg      # T ∈ {all sparse tensors that contain index d}
15               vIdx_T = vIdx_T * d_SIZE + d_crd[arg]  # T ∈ {all dense tensors that contain index d}
16.     case S: arg = argument of upper-level loop
17.              emit-load(d_crd, arg)
18.              vIdx_T += 0          # T ∈ {all sparse tensors that contain index d}
19.              vIdx_T = vIdx_T * d_SIZE + d_crd[arg]     # T ∈ {all dense tensors that contain index d}

    # Generate the loop body of the innermost loop
20  emit operations to do computation for C[vIdx_C] += A[vIdx_A] *  B[vIdx_B]
    # Generate load, mul, store operations
```

Figure 5.7: Sparse code generation algorithm

runtime introduces relatively low overhead during execution, which improves performance, especially for small computations.

5.7 Data Reordering

The distribution of the nonzero entries in sparse matrices/tensors can significantly affect the performance of sparse matrix/tensor algebra computations. Reordering [88, 116] is the de facto technique to optimize the memory access pattern caused by uneven data distribution. Different from existing compiler frameworks [97, 94] which apply reordering to iterations, we apply reordering to matrices and tensors to optimize their memory access patterns.

We borrow from the reordering algorithm presented in [116] (`LexiOrder`), extended it to support sparse matrices, and implemented it in the SPACe runtime

(`tensor_reorder()`). The `LexiOrder` algorithm is built on top of the doubly lexical ordering algorithm [124, 146] with some optimization techniques to advance its overall efficiency and availability in some corner cases. The basic idea of the `LexiOrder` algorithm is to sort a specific dimension (either rows or columns for matrices) in an iteration using the doubly lexical ordering algorithm and sort all dimensions in turn across iterations. The algorithm's objective is to cluster all nonzero entries around the diagonal to increase spatial and temporal locality.

5.8 Evaluation

In this section we evaluate SPACe against state-of-the-art high-level compiler frameworks and DSL for dense and sparse tensor algebra. Specifically, we compare our results against TACO [98], a tensor algebra compiler that performs automatic source-to-source transformation from TACO DSL to sequential C++, Parallel OpenMP, and data-parallel CUDA. For brevity, we evaluated the performance of selected benchmarks with a single storage format – matrices (CSR) and tensors (CSF), though our compiler can operate on other formats as well. All results reported are the average of 25 runs.

5.8.1 Experimentation Setup

We performed our experiments on a compute node equipped with two Intel Xeon Gold 6126 sockets running at 2.60GHz. Each CPU socket consists of 12 processing core (for a total of 24 cores). The system features 192 GB of DRAM memory. We compiled SPACe, TACO, and all the benchmarks with $-$`O3` and `clang 12.0` and use the most recent MLIR version at the time of writing this manuscript.

We use as input datasets 2833 matrices and six tensors of different sizes and shapes chosen from the SuiteSparse Matrix Collection [54], the FROSTT Tensor Collection [177], and BIGtensor [87]. The SuiteSparse Matrix Collection is a growing dataset of sparse matrices in real-world applications. The dataset is widely used in the numerical linear

algebra community for performance evaluation. The FROSTT Tensor Collection is a composition of open-source sparse tensor datasets from various data sources that are difficult to collect. The BIGtensor dataset is a tensor database that contains large-scale tensors for large-scale tensor analysis. Our input datasets represent the most important HPC domains in scientific computing, including chemistry, structural engineering, various linear solvers, computer graphics and vision, and molecular dynamics. We provide the description of the six tensors in Table 5.2.

Name	Size	Nonzeros	Domain
NELL-1	2,902,330 x 2,143,368 x 25,495,389	143599552	Natural Language Processing
NELL-2	12,092 x 9184 x 28,818	76879419	Natural Language Processing
delicious-3d	532,924 x 17,262,471 x 2,480,308	140,126,181	Tags from Delicious website
flickr-3d	319,686 x 28,153,045 x 1,607,191	112,890,310	Tages from Flickr website
vast-2015-mc1-3d	165,427 x 11,374 x 2	26,021,854	Theme park attend event
Freebase-music[87]	23,344,784 x 223,344,784 x 166	99,546,551	Entries related with music in Freebase

Table 5.2: Description of sparse tensors

5.8.2 Sparse Tensor Operations

We define the sparse tensor operations considered in SPACe below.

SpMV. The Sparse Matrix-times-Vector (SpMV or SpMSpV), $\mathbf{y} = \mathbf{X} \times \mathbf{v}$, is the multiplication of a sparse matrix $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2}$ with a dense vector $\mathbf{v} \in \mathbb{R}^{I_2}$. $y_{i_1} = \sum_{i_2=1}^{I_2} x_{i_1 i_2} v_{i_2}$.

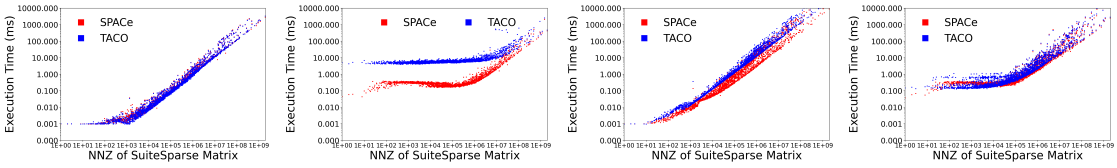
SpMM. The Sparse Matrix-times-Matrix (SpMM or SpGEMM), $\mathbf{Y} = \mathbf{X} \times \mathbf{U}$, is the multiplication of a sparse matrix $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2}$ with a dense matrix $\mathbf{U} \in \mathbb{R}^{I_2 \times R}$. $y_{i_1 r} = \sum_{i_2=1}^{I_2} x_{i_1 i_2} u_{i_2 r}$.

SpTTV. The Sparse Tensor-Times-Vector (SpTTV) [14] in mode n, $\mathcal{Y} = \mathcal{X} \times_n \mathbf{v}$, is the multiplication of a sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ with a dense vector $\mathbf{v} \in \mathbb{R}^{I_n}$, along mode n. Given $n = 1$, $y_{i_2 i_3} = \sum_{i_1=1}^{I_1} x_{i_1 i_2 i_3} v_{i_n}$. This results in a two-dimensional $I_2 \times I_3$ tensor which has one less dimension.

SpTTM. The Sparse Tensor-Times-Matrix (SpTTM) [103, 14] in mode n, denoted by $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}$, is the multiplication of a sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ with a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, along mode n. Mode-1 TTM results in a $R \times I_2 \times I_3$ tensor, and its operation is defined as $y_{r \cdots i_2 i_3} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 i_3} u_{i_n r}$. Also, note that R is typically much smaller than I_n in low-rank decompositions, typically $R < 100$.

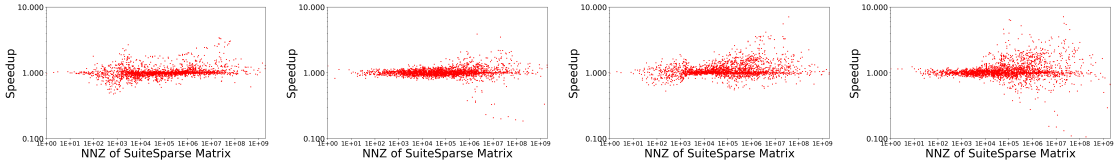
SpMV and SpMM widely appear in applications from scientific computing, such as direct or iterative solvers [113, 200], to data intensive domains [214], graph analytics [115]. SpTTV and SpTTM are computational kernels of popular tensor decompositions, such as the Tucker decomposition [103, 209, 173] , tensor power method [7, 197], for a variety of applications, including (social network, electrical grid) data analytics, numerical simulation, machine learning.

5.8.3 Performance Evaluation



(a) sequential SpMV (b) parallel SpMV (c) sequential SpMM (d) parallel SpMM

Figure 5.8: Performance comparison with TACO on CPU.



(a) seq SpMV-lexi (b) para SpMV-lexi (c) seq SpMM-lexi (d) para SpMM-lexi

Figure 5.9: Performance of Lexi ordering

SpMV and SpMM. We measured the performance of SPACe and TACO while running SpMV and SpMM with each of the 2833 matrices for sequential and parallel execution. We present the experimental results in Figure 5.8, where SPACe and TACO

are represented in red and blue dots respectively. In the plot, the x-axis represents a matrix (2,833 matrices, ordered by increasing number of nonzeros) and the y-axis execution time (lower is better). As we can see from the plots, SPACe achieves better performance than TACO on sequential SpMM (Figure 5.8c and parallel SpMV (Figure 5.8b, and comparable performance on sequential SpMV and parallel SpMM (Figures 5.8a and 5.8d, respectively). For sequential execution, SPACe outperforms TACO by up to 6.26x for SpMM (average 2.29x) and by up to 2.14x for SpMV (average 0.94x). A comparison of SPACe and TACO generated LLVM IR codes shows that SPACe results in more optimized code with better SIMD (or vectorization) utilization and loop unrolling. For both SpMV and SpMM, take SpMM as an example. The utilization of many SIMD instructions in TACO is only half of that in SPACe (e.g., TACO only uses 2 lanes while SPACe uses 4 lanes). SPACe unrolls multiple loops by 8 while TACO unrolls them by 2. Although the generated LLVM IR for both SpMV and SpMM show similar differences, the effect of better vectorization and loop unrolling are more evident for larger computation (SpMM). These results highlight one of the major goals of MLIR and MLIR-based compilers: by leveraging higher-level semantics information and progressive lowering steps, it is possible to produce a more aggressive and higher-quality LLVM IR that, eventually, results in higher performance and resource utilization.

For parallel SpMV, SPACe achieves an average of 20.92x speedup over TACO. Especially for small matrices, SPACe outperforms TACO by a significant margin, however, after further inspection, we realized that this performance difference is due to the overhead introduced by the underlying parallel runtime. SPACe uses an asynchronous task-based programming model based on LLVM co-routines while TACO leverages OpenMP. For small computation, LLVM co-routines introduce less overhead than OpenMP threading (which is beneficial for larger parallel regions). As we can see from Figure 5.8d, when there is enough computation for each OpenMP thread, the runtime overhead is amortized and both SPACe and TACO perform similarly.

Reordering. By reordering data in memory, SPACe attempts to increase spatial and

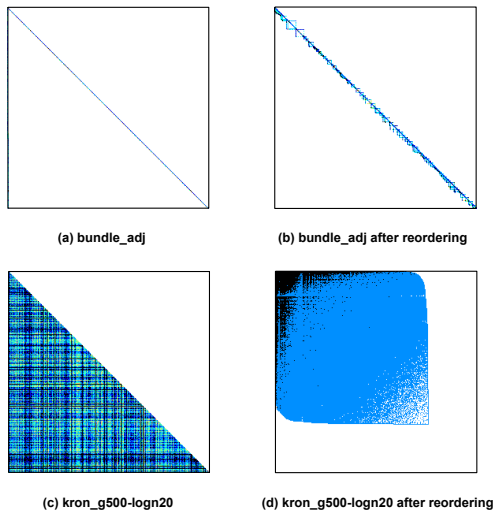
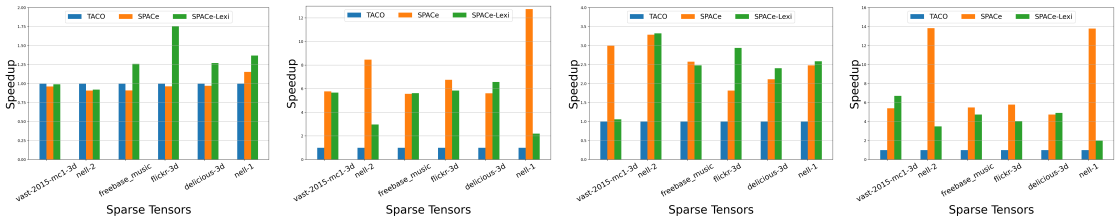


Figure 5.10: Visualization comparison of matrices with and without reordering

temporal locality to achieve higher performance. The plots in Figure 5.9 show SPACe performance when reordering data compared to original case (no reordering). Figure 5.9 shows that, indeed, in many cases there is significant advantage of reordering data, with up to 3.41x (average 1.04x), 3.89x (average 1.03x), 7.12x (average 1.12x), and 7.14x (average 1.13x) for SpMV sequential, SpMV parallel, SpMM sequential, and SpMM parallel, respectively. However, we also note that there might be significant performance degradation, especially for parallel execution. We further analyzed the reasons for this disparity and identified load imbalance as the primary source of performance degradation. Our reordering algorithm attempts to cluster nonzeros on the top-left corner of sparse matrices. In an ideal case, after reordering the nonzeros are distributed around the matrix diagonal.

Figure 5.10(a) and (b) shows a case in which reordering results in high performance improvements. In this case, the nonzero elements originally around the first column are distributed around the diagonal. Figure 5.10(c) and (d), instead, shows a case in which reordering reduces performance. In this case, the nonzeros are clustered around the top-left corner, thus threads that operate on the top rows have more work to perform compared to threads that operate on the bottom rows, which results in load imbalance and performance

degradation.



(a) sequential TTV (b) parallel TTV (c) sequential TTM (d) parallel TTM

Figure 5.11: Performance of tensor operations

TTV and TTM. We also compare SPACe with TACO on TTV and TTM with six sparse tensors on CPU and multi-threads and with reordering optimization on and off. Figure 5.11 illustrates the experimental results. TACO does not generate parallel code if the output tensor is stored in sparse format, even if instructed to do so, thus the results in the Figure for parallel execution are with respect to sequential execution of the TACO benchmarks. For sequential TTV, SPACe performs comparably to TACO. With reordering, SPACe achieves better performance on four out of six sparse tensors. For parallel TTV, SPACe performs significantly better than TACO with up to 12.5× and on average 8× speedup. With reordering, SPACe's performance is degraded on five of six sparse tensors except for delicious-3d. As for the case of SpMV and SpMM, we observed similar load imbalance issues. For sequential TTM, SPACe performs better than TACO with up to 3.3× and on average 2.53× speedup. With reordering, SPACe achieves better performance on three out of six sparse tensors. For parallel TTM, SPACe performs significantly better than TACO with up to 13.9× and on average 8.13× speedup. With reordering, SPACe's performance is degraded on five of six sparse tensors except for vast-2015-mc1-3d.

Our results show that reordering tensors have a significant (positive or negative) impact on performance, more than for matrices. One possible reason is that the `LexiOrder` algorithm reorders all dimensions of data simultaneously, which means the data locality is the best when accessing all the dimensions in conjunction, as in conjunction. The

sparse tensor operation MTTKRP [116] follows this behavior to gain a good performance speedup. However, this does not mean that the indices in every dimension get good locality when accessing the vector or matrix in TTV or TTM, potentially leading to low performance. We will investigate alternative reordering algorithms and adaptive methods in future work.

5.9 Related Work

Compiler for Tensor Algebra. Compiler techniques have been used to drive irregular computation in tensor algebra [83, 98, 15, 92, 183]. TCE [83] is a compiler optimization framework that focuses on dense tensor contraction operations in quantum chemistry. TTC [183] is a compiler framework that carries out a composition of high-performance tensor transpose strategies for GPUs. TACO [98] is a compiler that generates code for given tensor algebra expressions and used as a higher-level domain-specific language for tensor algebra. Kim et al. [92] use similar compiler techniques for high-performance tensor contractions but focus on its application on Graphics Processing Unit (GPU)s. Different from existing works, we develop a high-performance sparse tensor algebra compiler using MLIR, which supports both serial and parallel code generation and enables better portability and adaptability.

Domain-specific Libraries for Tensor Algebra. There have been a collection of tensor algebra libraries developed [77, 153, 194, 164, 163, 65, 68, 160, 181]. FLAME [77] is a library aiming for the derivation and implementation of tensor algebra operations on CPUs. Later, serial linear algebra libraries are extended to run on distributed parallel systems [153, 164, 163, 65]. On the other hand, these libraries are extended to support sparse tensor algebra operations using different sparse tensor formats [68, 160, 181]. Tensor algebra libraries favor scientific computing and are widely utilized in scientific application development. By contrast, SPACe transparently implements tensor algebra algorithms per se and can compile most types of sparse tensor formats and automatically generate

efficient code.

Tensor Algebra Optimization. Plenty of work [12, 19, 16, 105, 179, 207, 114, 20] leverage reordering to optimize tensor algebra with respect to distinct tensor formats for different tensor operations and heterogeneous architectures. Kjolstad et al. [96, 99] reorder loops of tensor algebra computations to improve the data locality. Smith et al. [179] use reordering to enable high-performance tensor factorization operations. Yang et al. [207] identify an efficient memory access pattern for high-performance SpMM operations through merge-based load balancing and row-major coalesced memory access. Other works, such as [127, 114, 43, 18], to name a few, design high-performance algorithms considering computer architecture characteristics using techniques like register blocking, cache blocking, and reordering. SPACe.

5.10 Summary

In this work, we presented a high-performance sparse tensor algebra compiler, called SPACe, and a high-productive DSL to support next-generation tensor operations. Our DSL enables high-level programming abstractions that resemble the familiar Einstein notation to express tensor algebra operations. SPACe is based on the MLIR framework, which allows us to build portable, adaptable, and extensible compilers. SPACe provides an effective and efficient code generation which supports most tensor storage formats through an internal storage format based on four dimension attributes and a novel code generation algorithm. Furthermore, we incorporate a data reordering algorithm to increase the data locality. The evaluation results reveal that SPACe outperforms competing for baseline sparse tensor algebra compiler TACO with up to 20.92x, 6.39x, and 13.9x performance improvement for SpMV, SpMM, and TTM computations respectively. In future work, we plan to extend SPACe to support heterogeneous architectures and to explore alternatives reordering schemes that better adapt to the sparsity patterns observed in scientific and engineering input sets.

Chapter 6

Conclusions and Future Work

6.1 Summary of Dissertation Contributions

In this thesis, we apply compiler optimizations to improve the performance of applications by combining application knowledge. Our contributions lie in the following three areas:

Improving B+ tree query processing by reducing redundant queries. We observe that there are many redundant and unnecessary queries in BSP based latch-free B+ tree query processing systems. We propose a novel query sequence analysis and transformation framework, *QSAT*, to identify the redundant and unnecessary queries. *QSAT* is inspired by classical data-flow analysis, which is often used in compilers for code optimizations. Practically, we implement a one-pass *QSAT*, called *Qtrans*. We integrate *Qtrans* into an existing latch-free B+ tree query processing system, the throughput improvement is up to 16X.

Using compiler static analysis to assist in defending heap buffer overflow. We observe that all the heap overflows in C/C++ programs are related to arrays. We develop a compiler approach to automatically identify and instrument the array-related allocations to effectively reduce protected objects in run-time. We implement a transformation pass in LLVM to instrument the identified array-related allocations. Our evaluations on massive real heap overflow applications show that our implementation is effective enough to identify

the array-related heap allocations.

Building high-performance compiler for sparse tensor algebra computations. We build a high-performance sparse tensor algebra compiler based on the high-level tensor algebra information, such as tensor expressions and tensor formats. We develop a high-productive domain-specific language for tensor algebra computations. We propose an internal storage format and an automatic code generation algorithm to generate the computation kernels for the given tensor operations and formats. We integrate data reordering optimizations in our compiler to improve the data locality. The evaluations show that the performance of our compiler is comparable with state-of-the-art sparse tensor algebra compilers.

6.2 Future Research Direction

Our future work targets improving the sparse tensor algebra compiler, mainly from the generality and performance side.

To improve generality, we will evaluate our compiler on multiple GPUs, and target to support more storage formats for sparse tensors and more tensor algebra operations. MLIR supports automatic generation of code for different hardware platforms, and we will evaluate the performance of our compiler on GPUs. Besides the formats we have already supported in our compiler, there are many high-performance storage formats for specific cases. We will support more sparse storage formats in our compiler. We will also support more tensor algebra operations, such as sparse tensor times sparse tensor.

With respect to performance improvement, we will explore more optimization opportunities. For example, no one storage format is good for all cases. We will explore an automatic way to choose the optimal format according to the sparsity patterns of the input tensors. We will also utilize the features of different hardware platforms to improve the performance of our compiler on different hardware platforms.

Bibliography

[1] National institute of standards and technology. `http://math.nist.gov/MatrixMarket/formats.html`, 2013.

[2] Evrim Acar, Canan Aykut-Bingol, Haluk Bingol, Rasmus Bro, and Bülent Yener. Multiway analysis of epilepsy tensors. *Bioinformatics*, 23(13):i10–i18, 2007.

[3] Umut A Acar, Guy E Blelloch, and Robert Harper. *Selective Memoization*, volume 38. ACM, 2003.

[4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools, 2006.

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[7] Animashree Anandkumar, Rong Ge, and Majid Janzamin. Analyzing tensor power method dynamics in overcomplete regime. *The Journal of Machine Learning Research*, 18(1):752–791, 2017.

[8] Henrique Andrade, Suresh Aryangat, Tahsin Kurc, Joel Saltz, and Alan Sussman. Efficient execution of multi-query data analysis batches using compiler optimization strategies. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 509–523. Springer, 2003.

[9] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M Tsai, and Weichung Wang. Load-balancing sparse matrix vector product kernels on gpus. *ACM Transactions on Parallel Computing (TOPC)*, 7(1):1–26, 2020.

[10] Suresh Aryangat. *Optimizing the Execution of Batches of Data Analysis Queries*. PhD thesis, 2004.

[11] Suresh Aryangat, Henrique Andrade, and Alan Sussman. Time and space optimization for processing groups of multi-dimensional scientific queries. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 95–105. ACM, 2004.

[12] Alexander A Auer, Gerald Baumgartner, David E Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, et al. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 2006.

[13] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[14] Brett W. Bader and Tamara G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007.

[15] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[16] Muthu Baskaran, Benoit Meister, and Richard Lethin. Low-overhead load-balanced scheduling for sparse tensor computations. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014.

[17] Muthu Baskaran, Benoît Meister, Nicolas Vasilache, and Richard Lethin. Efficient and scalable computations with sparse tensors. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–6. IEEE, 2012.

[18] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[19] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–11, 2009.

[20] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. ICS '93, page 416–424, New York, NY, USA, 1993. Association for Computing Machinery.

[21] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995.

[22] Deepak R Bobbarjung, Suresh Jagannathan, and Cezary Dubnicki. Improving duplicate elimination in storage systems. *ACM Transactions on Storage (TOS)*, 2(4):424–448, 2006.

[23] Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 159–170, 1997.

[24] Guillaume Bouchard, Jason Naradowsky, Sebastian Riedel, Tim Rocktäschel, and Andreas Vlachos. Matrix and tensor factorization methods for natural language processing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing: Tutorial Abstracts*, pages 16–18, 2015.

[25] Anastasia Braginsky and Erez Petrank. A Lock-Free B+ tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 58–67. ACM, 2012.

[26] Preston Briggs and Keith D Cooper. Effective partial redundancy elimination. *ACM SIGPLAN Notices*, 29(6):159–170, 1994.

[27] Trevor Brown, Faith Ellen, and Eric Ruppert. A General Technique for Non-Blocking Trees. In *ACM SIGPLAN Notices (PPoPP)*, volume 49, pages 329–342. ACM, 2014.

[28] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[29] Bugzilla. "libhx: buffer overrun in hx_split()". https://bugzilla.redhat.com/show_bug.cgi?id=625866, 2010.

[30] Bugzilla. "libtiff (gif2tiff): possible heapbased buffer overflow in readgifimage()". http://bugzilla.maptools.org/show_bug.cgi?id=2451, 2013.

[31] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2008.

[32] Qiong Cai and Jingling Xue. Optimal and efficient speculation-based partial redundancy elimination. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 91–102. IEEE, 2003.

[33] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *ACM SIGPLAN Notices*, 29(11):252–262, 1994.

[34] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEEE, 2007.

[35] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[36] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[37] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. *ACM SIGARCH Computer Architecture News*, 18(2SI):239–248, 1990.

[38] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[39] Shimin Chen, Phillip B Gibbons, Todd C Mowry, and Gary Valentin. Fractal Prefetching B+-trees: Optimizing both Cache and Disk Performance. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2002.

[40] Yuedan Chen, Guoqing Xiao, M Tamer Özsu, Chubo Liu, Albert Y Zomaya, and Tao Li. aesptv: An adaptive and efficient framework for sparse tensor-vector product kernel on a high-performance computing platform. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2329–2345, 2020.

[41] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. Detecting memory errors at runtime with source-level instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 341–351, New York, NY, USA, 2019. ACM.

[42] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-Scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[43] Jee W Choi, Amik Singh, and Richard W Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *ACM sigplan notices*, 45(5):115–126, 2010.

[44] Nikos P Chrisochoides and Florian Sukup. *Task parallel implementation of the Bowyer-Watson algorithm*, volume 235. Cornell Theory Center, Cornell University, 1996.

[45] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Pro-*

ceedings of the 1st ACM symposium on Cloud computing, pages 143–154. ACM, 2010.

[46] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Elsevier, 2011.

[47] Marshall Copeland, Julian Soh, Anthony Puca, Mike Manning, and David Gollob. Microsoft azure and cloud computing. In *Microsoft Azure*, pages 3–26. Springer, 2015.

[48] The MITRE Corporation. Cve-2016-10269. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10269, 2016.

[49] Tyler Crain, Vincent Gramoli, and Michel Raynal. A Contention-Friendly Binary Search Tree. In *European Conference on Parallel Processing (Euro-Par)*, pages 229–240. Springer, 2013.

[50] Tyler Crain, Vincent Gramoli, and Michel Raynal. A Fast Contention-Friendly Binary Search Tree. *Parallel Processing Letters*, 26(03):1650015, 2016.

[51] CVEdetails. Vulnerabilities by type. https://www.cvedetails.com/vulnerabilities-by-types.php, 2019.

[52] Amarildo T Da Costa, Felipe MG França, et al. The Dynamic Trace Memoization Reuse Technique. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 92–99. IEEE, 2000.

[53] Mayank Daga and Mark Nutter. Exploiting Coarse-Grained Parallelism in B+ Tree Searches on an APU. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 240–247. IEEE, 2012.

[54] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[55] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[56] David J DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. Split query processing in polybase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1255–1266, 2013.

[57] Dhananjay M. Dhamdhere. Practical Adaption of the Global Optimization Algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):291–294, 1991.

[58] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early Experience with a Commercial Hardware Transactional Memory Implementation. 2009.

[59] Johannes Doerfert and Hal Finkel. Compiler optimizations for openmp. In *International Workshop on OpenMP*, pages 113–127. Springer, 2018.

[60] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. *ACM SIGPLAN Notices (PPoPP)*, 49(8):343–356, 2014.

[61] Gregory J. Duck and Roland H. C. Yap. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 181–195, New York, NY, USA, 2018. ACM.

[62] Gregory J. Duck and Roland H. C. Yap. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 181–195, New York, NY, USA, 2018. ACM.

[63] Albert Einstein. Die grundlage der allgemeinen relativitätstheorie. In *Das Relativitätsprinzip*. Springer, 1923.

[64] Ramez Elmasri. *Fundamentals of Database Systems*. Pearson Education India, 2008.

[65] Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I Krylov. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of computational chemistry*, 2013.

[66] Exploit. "openssl heartbeat poc with starttls support". https://gist.github.com/takeshixx/10107280, 2014.

[67] Xiaowen Feng, Hai Jin, Ran Zheng, Kan Hu, Jingxiang Zeng, and Zhiyuan Shao. Optimization of sparse matrix-vector multiplication with variant csr on gpus. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 165–172. IEEE, 2011.

[68] Matthew Fishman, Steven R White, and E Miles Stoudenmire. The itensor software library for tensor network calculations. *arXiv preprint arXiv:2007.14822*, 2020.

[69] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[70] Jordan Fix, Andrew Wilkes, and Kevin Skadron. Accelerating Braided B+ Tree Searches on a GPU with CUDA. In *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*, 2011.

[71] Micro Focus. Fortify static code analyzer. https://www.ndm.net/sast/hp-fortify, 2019. last visited: 02/08/2019.

[72] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W Ueberhuber. Efficient utilization of simd extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005.

[73] Frank Ch. Eigler. *Mudflap: pointer use checking for C/C++*. Red Hat Inc., 2003.

[74] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *IEEE Pervasive computing*, 3(1):46–55, 2004.

[75] Alexander G Gounares, Ying Li, Charles D Garrett, and Michael D Noakes. Selecting Functions for Memoization Analysis, 2013. US Patent App. 13/671,828.

[76] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. Quickly generating billion-record synthetic databases. In *ACM Sigmod Record*, volume 23, pages 243–252. ACM, 1994.

[77] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 2001.

[78] Rajiv Gupta, DA Benson, and Jesse Zhixi Fang. Path profile guided partial dead code elimination using predication. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113. IEEE, 1997.

[79] Liyang Hao, Siqi Liang, Jinmian Ye, and Zenglin Xu. Tensord: A tensor decomposition library in tensorflow. *Neurocomputing*, 318:196–200, 2018.

[80] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, 2012. ACM.

[81] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, Berkeley, Califonia, USA, 1992. USENIX Association.

[82] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for fortran d on mimd distributed-memory machines. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 86–100. IEEE, 1991.

[83] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 2003.

[84] Intel Corporation. Intel inspector xe 2013. http://software.intel.com/en-us/intel-inspector-xe, 2012.

[85] Hosagrahar V Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. iDistance: An Adaptive B+-tree based Indexing Method for Nearest Neighbor Search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[86] Christian S Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient b+-tree based indexing of moving objects. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 768–779, 2004.

[87] Inah Jeon, Evangelos E. Papalexakis, U Kang, and Christos Faloutsos. Haten2: Billion-scale tensor decompositions. In *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[88] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 376–388, 2020.

[89] Mahmut Kandemir, N Vijaykrishnan, and Mary Jane Irwin. Compiler optimizations for low power systems. In *Power aware computing*, pages 191–210. Springer, 2002.

[90] Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Wu Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th Annual Design Automation Conference*, pages 304–307, 2000.

[91] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[92] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and Ponnuswamy Sadayappan. A code generator for high-performance tensor contractions on gpus. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019.

[93] David R Kincaid, Thomas C Oppe, and David M Young. Itpackv 2d user's guide. Technical report, Texas Univ., Austin, TX (USA). Center for Numerical Analysis, 1989.

[94] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. Optimizing indirect memory references with milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 299–312, 2016.

[95] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. pages 180–192, 2019.

[96] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 180–192. IEEE, 2019.

[97] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 943–948, Oct 2017.

[98] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[99] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[100] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *ACM SIGPLAN Notices*, 29(6):147–158, 1994.

[101] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy Code Motion. *ACM SIGPLAN Notices*, 39(4):460–472, 2004.

[102] Jae Jin Koh. Relational database schema integration by overlay and redundancy elimination methods. In *Strategic Technology, 2007. IFOST 2007. International Forum on*, pages 610–614. IEEE, 2007.

[103] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[104] Tamara G Kolda and Jimeng Sun. Scalable tensor decompositions for multi-aspect data mining. In *2008 Eighth IEEE international conference on data mining*, pages 363–372. IEEE, 2008.

[105] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Csx: an extended compression format for spmv on shared memory systems. *ACM SIGPLAN Notices*, 46(8):247–256, 2011.

[106] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 22:1–22:14, New York, NY, USA, 2018. ACM.

[107] Purushottam Kulkarni, Fred Douglis, Jason D LaVoie, and John M Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2004.

[108] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a {PC}. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 31–46, 2012.

[109] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[110] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[111] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasi-

lache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[112] Wei Le and Mary Lou Soffa. Marple: A demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 272–282, New York, NY, USA, 2008. ACM.

[113] Benjamin C Lee, Richard W Vuduc, James W Demmel, and Katherine A Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 169–176. IEEE, 2004.

[114] Jiajia Li, Jimeng Sun, and Richard Vuduc. Hicoo: hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 238–252. IEEE, 2018.

[115] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 117–126, 2013.

[116] Jiajia Li, Bora Uçar, Ümit V Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. Efficient and effective sparse tensor reordering. In *Proceedings of the ACM International Conference on Supercomputing*, pages 227–237, 2019.

[117] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment*, 10(12):1933–1936, 2017.

[118] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. A unified optimization approach for sparse tensor operations on gpus. In *2017 IEEE international conference on cluster computing (CLUSTER)*, pages 47–57. IEEE, 2017.

[119] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. ireplayer: In-situ and identical record-and-replay for multithreaded applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 344–358, New York, NY, USA, 2018. ACM.

[120] Hongyu Liu, Sam Silvestro, Xiaoyin Wang, Lide Duan, and Tongping Liu. Csod: Context-sensitive overflow detection. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 50–60. IEEE Press, 2019.

[121] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 911–922, New York, NY, USA, 2016. ACM.

[122] Checkmarx Ltd. Checkmarx. https://www.checkmarx.com, 2019. last visited: 02/08/2019.

[123] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, Chicago, IL, USA, 2005.

[124] Anna Lubiw. Doubly lexical orderings of matrices. *SIAM Journal on Computing*, 16(5):854–879, 1987.

[125] Yuan Luo, Fei Wang, and Peter Szolovits. Tensor factorization toward precision medicine. *Briefings in bioinformatics*, 18(3):511–514, 2017.

[126] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[127] Marco Maggioni and Tanya Berger-Wolf. Adell: An adaptive warp-balancing ell format for efficient sparse matrix-vector multiplication on gpus. In *2013 42nd international conference on parallel processing*, pages 11–20. IEEE, 2013.

[128] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[129] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 678–689. IEEE, 2016.

[130] Ian Molyneaux. *The art of application performance testing: from strategy to tools*. " O'Reilly Media, Inc.", 2014.

[131] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 111–125. Springer, 2010.

[132] Etienne Morel and Claude Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[133] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. Comet: A domain-specific compilation of high-performance computational chemistry. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'20)*. Springer.

[134] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-Free Binary Search Trees. In *ACM SIGPLAN Notices (PPoPP)*, volume 49, pages 317–328. ACM, 2014.

[135] George C. Necula Necula, McPeak Scott, and Weimer Westley. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*, pages 128–139, New York, NY, United States, 2002. Association for Computing Machinery.

[136] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[137] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and Ponnuswamy Sadayappan. An efficient mixed-mode representation of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–25, 2019.

[138] Peter Norvig. Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[139] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 1–11, New York, NY, USA, 2007. ACM Press.

[140] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2009)*, pages 397–407, New York, NY, USA, 2009. ACM.

[141] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. An extensible approach for taming the challenges of javascript dead code elimination. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 291–401. IEEE, 2018.

[142] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):28:1–28:30, June 2018.

[143] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu. Lazy-update b+-tree for flash devices. In *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, pages 323–328. IEEE, 2009.

[144] Oracle Corporation. Sun memory error discovery tool (discover). http://docs.oracle.com/cd/E18659_01/html/821-1784/gentextid-302.html, 2011.

[145] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[146] Robert Paige and Robert E Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[147] Vineeth Kumar Paleri, YN Srikant, and Priti Shankar. A simple algorithm for partial redundancy elimination. *ACM Sigplan Notices*, 33(12):35–43, 1998.

[148] Vineeth Kumar Paleri, YN Srikant, and Priti Shankar. Partial Redundancy Elimination: A Simple, Pragmatic, and Provably Correct Algorithm. *Science of Computer Programming*, 48(1):1–20, 2003.

[149] parasoft Company. *C and C++ Memory Debugging*, 2013.

[150] Bruce Perens. Electric fence. `https://linux.softpedia.com/get/Programming/Debuggers/Electric-Fence-3305.shtml`, 2005.

[151] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2004.

[152] Constantine D Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 37(8):991–1004, 1988.

[153] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, 2013.

[154] Jun Rao and Kenneth A Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, volume 99, pages 78–89, 1999.

[155] Jun Rao and Kenneth A Ross. Making B+-Trees Cache Conscious in Main Memory. In *ACM SIGMOD Record*, volume 29, pages 475–486. ACM, 2000.

[156] Bin Ren, Gagan Agrawal, James R Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. Simd parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

[157] Bin Ren, Shruthi Balakrishna, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. Extracting simd parallelism from recursive task-parallel programs. *ACM Transactions on Parallel Computing (TOPC)*, 6(4):1–37, 2019.

[158] Steffen Rendle, Leandro Balby Marinho, Alexandros Nanopoulos, and Lars Schmidt-Thieme. Learning optimal ranking with tensor factorization for tag recommendation. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 727–736, 2009.

[159] Minsoo Rhu and Mattan Erez. Maximizing simd resource utilization in gpgpus with simd lane permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 356–367, 2013.

[160] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning. *arXiv preprint arXiv:1905.01330*, 2019.

[161] Ohad Rodeh. B-trees, Shadowing, and Clones. *ACM Transactions on Storage (TOS)*, 3(4):2, 2008.

[162] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, San Diego, California, USA, 2004. The Internet Society.

[163] Martin D Schatz, Tze Meng Low, Robert A van de Geijn, and Tamara G Kolda. Exploiting symmetry in tensors for high performance: Multiplication with symmetric tensors. *SIAM Journal on Scientific Computing*, 2014.

[164] Martin D Schatz, Robert A Van de Geijn, and Jack Poulson. Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing*, 2016.

[165] Naser Sedaghati, Te Mu, Louis-Noël Pouchet, Srinivasan Parthasarathy, and P Sadayappan. Automatic selection of sparse matrix representation on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 99–108, 2015.

[166] Kurt Seifried. "cve request: Heap-based buffer overflow in openjpeg". https://seclists.org/oss-sec/2012/q3/300, 2012.

[167] John S Seng and Dean M Tullsen. The effect of compiler optimizations on pentium 4 power consumption. In *Seventh Workshop on Interaction Between Compilers and Computer Architectures, 2003. INTERACT-7 2003. Proceedings.*, pages 51–56. IEEE, 2003.

[168] Binanda Sengupta and Abhijit Das. Use of simd-based data parallelism to speed up sieving in integer-factoring algorithms. *Applied Mathematics and Computation*, 293:204–217, 2017.

[169] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[170] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *Proc. VLDB Endowment*, 4(11):795–806, 2011.

[171] Amirhesam Shahvarani and Hans-Arno Jacobsen. A Hybrid B+-Tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1523–1538. ACM, 2016.

[172] Kamran Siddique, Zahid Akhtar, Edward J Yoon, Young-Sik Jeong, Dipankar Dasgupta, and Yangwoo Kim. Apache Hama: An Emerging Bulk Synchronous Parallel Computing Framework for Big Data Applications. *IEEE Access*, 4:8879–8887, 2016.

[173] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, and Christos Faloutsos. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582, 2017.

[174] Sam Silvestro, Hongyu Liu, Tong Zhang, Changhee Jung, Dongyoon Lee, and Tongping Liu. Sampler: Pmu-based sampling to detect memory errors latent in production software. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 231–244. IEEE, 2018.

[175] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: Shrinking the petacube. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 464–475. ACM, 2002.

[176] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[177] Shaden Smith, Jee W Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. Frostt: The formidable repository of open sparse tensors and tools, 2017.

[178] Shaden Smith and George Karypis. Accelerating the tucker decomposition with compressed sparse tensors. In *European Conference on Parallel Processing*, pages 653–668. Springer, 2017.

[179] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 61–70. IEEE, 2015.

[180] Avinash Sodani. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi Processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24. IEEE, 2015.

[181] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 2014.

[182] Qingquan Song, Hancheng Ge, James Caverlee, and Xia Hu. Tensor completion algorithms in big data analytics. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13(1):1–48, 2019.

[183] Paul Springer, Aravind Sankaran, and Paolo Bientinesi. Ttc: A tensor transposition compiler for multiple architectures. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2016.

[184] Paul Springer, Tong Su, and Paolo Bientinesi. Hptt: A high-performance tensor transposition c++ library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 56–62, 2017.

[185] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *USENIX Annual Technical Conference*, volume 15, 1996.

[186] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, USA, 2013. IEEE Computer Society.

[187] Talos. "memcached server sasl autentication remote code execution vulnerability". https://www.talosintelligence.com/reports/TALOS-2016-0221/, 2016.

[188] Serafettin Tasci and Murat Demirbas. Giraphx: Parallel yet Serializable Large-Scale Graph Processing. In *European Conference on Parallel Processing*, pages 458–469. Springer, 2013.

[189] Andrej Tolic and Andrej Brodnik. Deduplication in unstructured-data storage systems. *Elektroteh Vestn*, 82(5):233, 2015.

[190] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[191] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 2010.

[192] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 071. IOP Publishing, 2005.

[193] Richard W Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *International Conference on High Performance Computing and Communications*, pages 807–816. Springer, 2005.

[194] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014.

[195] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. Eunomia: Scaling Concurrent Search Trees under Contention Using HTM. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 385–399. ACM, 2017.

[196] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 11. ACM, 2016.

[197] Yining Wang, Hsiao-Yu Tung, Alexander Smola, and Animashree Anandkumar. Fast and guaranteed tensor decomposition via sketching. *arXiv preprint arXiv:1506.04448*, 2015.

[198] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, May 2015.

[199] James B White and Ponnuswamy Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings Fourth International Conference on High-Performance Computing*, pages 66–71. IEEE, 1997.

[200] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[201] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, et al. The suif compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford University Technical Report No. CSL-TR-94-620, 1994.

[202] Ming Wu and Xiao-Feng Li. Task-pushing: a scalable parallel gc marking algorithm without synchronization operations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10. IEEE, 2007.

[203] Hongwei Xi. Dead code elimination through dependent types. In *International Symposium on Practical Aspects of Declarative Languages*, pages 228–242. Springer, 1999.

[204] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. Cvr: Efficient vectorization of spmv on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 149–162, 2018.

[205] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaspmv: yet another spmv framework on gpus. *Acm Sigplan Notices*, 49(8):107–118, 2014.

[206] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. Harmonia: a high throughput b+ tree for gpus. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, pages 133–144, 2019.

[207] Carl Yang, Aydin Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*. Springer, 2018.

[208] T. Ye, L. Zhang, L. Wang, and X. Li. An empirical study on detecting and fixing buffer overflow bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 91–101, April 2016.

[209] Tatsuya Yokota and Andrzej Cichocki. Multilinear tensor rank estimation via sparse tucker decomposition. In *2014 Joint 7th International Conference on Soft Computing and Intelligent Systems (SCIS) and 15th International Symposium on Advanced Intelligent Systems (ISIS)*, pages 478–483. IEEE, 2014.

[210] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[211] Qiang Zeng, Dinghao Wu, and Peng Liu. Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 367–377, New York, NY, USA, 2011. ACM.

[212] Qiang Zeng, Mingyi Zhao, and Peng Liu. Heaptherapy: An efficient end-to-end solution against heap buffer overflows. In *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 485–496, Washington, DC, USA, 2015. IEEE Computer Society.

[213] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-*

Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, pages 631–644, New York, NY, USA, 2019. ACM.

[214] Xianyi Zhang, Yunquan Zhang, Xiangzheng Sun, Fangfang Liu, Shengfei Liu, Yuxin Tang, and Yucheng Li. Automatic performance tuning of spmv on gpgpu. *HPC Asia, Kaohsiung, Taiwan, China*, pages 173–179, 2009.

[215] Yan Zhang and Nirwan Ansari. On protocol-independent data redundancy elimination. *IEEE Communications Surveys & Tutorials*, 16(1):455–472, 2013.

[216] Yin Zhang, Min Chen, Shiwen Mao, Long Hu, and Victor CM Leung. Cap: Community activity prediction based on big data analysis. *Ieee Network*, 28(4):52–57, 2014.

[217] Jingren Zhou and Kenneth A Ross. Buffering Accesses to Memory-Resident Index Structures. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 405–416. VLDB Endowment, 2003.

VITA

Ruiqin Tian

Ruiqin Tian is a Ph.D. candidate in the Department of Computer Science at the College of William & Mary advised by Prof. Bin Ren. Her research interests are compiler optimizations for high-performance computing, compiler analysis and runtime optimizations. Her Ph.D. research has been published in CGO 2019, ASE 2020, and LCPC 2020. Before joining William & Mary, she received her B.Eng degree from Northeast Petroleum University in 2012 and an M.Sc degree from the University of Chinese Academy of Sciences in 2015. She has been working as a PhD research intern at Pacific Northwest National Lab since Feb 2020.