Improving Main-memory Database System Performance through Cooperative Multitasking

Thèse N° 9712

Présentée le 25 octobre 2019

à la Faculté informatique et communications
Laboratoire de systèmes et applications de traitement de données massives
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Georgios PSAROPOULOS

Acceptée sur proposition du jury

Prof. A. Argyraki, présidente du jury
Prof. A. Ailamaki, directrice de thèse
Prof. J. Teubner, rapporteur
Dr T. Willhalm, rapporteur
Prof. J. Larus, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

To those who made it possible…

Contents

List of Figures

List of Tables

Acknowledgements

This PhD has been an adventure, the successful completion of which would not have been possible without the help and support of my advisor, my colleagues, my friends, and my family.

First and foremost, I would like to thank my advisor, Prof. Anastasia Ailamaki. When I came to Natassa's group as a third year PhD student she made sure I find a research topic that would not only advance the state of the art but also be feasible in the given timeframe. Thanks to the confidence she put in me, I was able to drive my thesis from the conception of the general idea of interleaved execution to the publication of my papers and writing of the present dissertation. At the same time, she was always available to provide guidance and support on any matter, be it research-related, or about personal life and career development.

Second, I would like to thank my SAP supervisors, Dr. Norman May and Dr. Thomas Legler. While Natassa ensured my work has academic merit, Norman and Thomas helped me frame a thesis with business value. Norman's discrete but effective supervision allowed me to experiment with many ideas, while his feedback substantially improved my papers and thesis. And Thomas brought a more applied perspective to our discussions, while being essential in understanding and working with SAP HANA.

I would also like to thank Prof. James Larus, Prof. Jens Teubner, and Dr. Thomas Willhalm for serving in my thesis committee and contributing with their valuable comments; as well as Prof. Katerina Argyraki who presided over my oral exam.

Next, I would like to thank my academic families. On the EPFL side, I am thankful to Onur, Stavros, Cansu, Djordje, and Javier for teaching how to create a decent set of slides and how to run experiments. Many thanks are also due to Danica and Manos for introducing me to database research and brainstorming with me on potential research topics, as well as to Iraklis for helping me to settle in Germany and at the new environment of SAP. Special thanks go to Eleni and Mira, whose office armchair I occupied numerous afternoons while at EPFL, enjoying their company and getting informed about the social events in Lausanne. Further, I am thankful to all the amazing people I had the privilege to coincide with while at DIAS lab: Angelos, Aunn, Ben, Bikash, Srinivas, Cesar, Darius, Diane, Dimitra, Erika, Foteini, Lionel, Matt, Odysseas, Panos, Periklis, Raja, Renata, Satya, Sharareh, Stella, Tahir, Utku, and Viktor.

On the SAP side, I would like to thank the SAP HANA Database Campus team: Arne, Axel, David, Elena, Florian, Frank, Ismail, Lucas, Marcus, Max, Michael (senior), Michael (junior), Robert, Robin, Thomas, Tiemo, and Stefan. Besides being valuable colleagues at work, these people were also my budddies for lunch, dinner, board game, movie watching, hiking, and many other activities. Particular mentions go to Ismail and Stefan, who were good friends and the most critical readers of my papers, to Frank, Max, and Florian, with whom I had many casual dinners and outings, and to Arne, for the great team culture he has created in the Campus; I am extra gratefult to Stefan for translating the abstract of this dissertation in German. Finally, thanks goes to Thomas and Roman, who were my go-to people regarding everything about Intel hardware.

Apart from my academic families, I would also like to thank my friends in Lausanne and Heidelberg that brightened the days during the past six years. Special thanks go to Dimitris, Giorgos, who stood with me in both good and bad times, to Mario, Arash, and Eleni, who substantially facilitated my time between Lausanne and Heidelberg, to Kostas and Ioanna, for making me feel as a member of their family, to Marios and Abdel, for being great neighbors in Bahnstadt.

Last but not the least, my deepest gratitude goes to my parents Triantafyllos, Eleni, and my siblings, Marialena and Iakovos, for their unconditional love and support.

Lausanne, 10 October 2019 Georgios Psaropoulos

Abstract

Database systems access memory either sequentially or randomly. Contrary to sequential access and despite the extensive efforts of computer architects, compiler writers, and system builders, random access to data larger than the processor cache has been synonymous to inefficient execution. Especially in the big data era, data processing is memory bound, and accesses to DRAM and non-volatile memory each take several tens or hundreds of nanoseconds respectively, posing a great challenge to current processors. Due to the mismatch between the way humans write code and the way processors execute this code, workload execution stalls on main memory access, instead of executing the other parallel work that typically exists in big data workloads.

This thesis establishes cooperative multitasking as the principal way to hide memory latency in operations that consist of parallel tasks. We first systematize cooperative multitasking presenting an analogue of Amdahl's law for latency hiding. More importantly, we then introduce *interleaving with coroutines*, a general-purpose and practical technique to interleave the execution of parallel tasks within one thread interleaved execution and thereby hide memory access. This form of cooperative multitasking enables significant performance improvements for analytical and transactional use cases that suffer from unavoidable memory accesses, such as index joins and tuple reconstruction, as well as concurrent `GET` and `PUT` operations in key-value stores. Given enough parallel work, sufficient hardware support for concurrent memory accesses, and a low-overhead mechanism to switch between tasks, interleaved execution renders important database operations oblivious to memory latency and makes their execution behave as if all accessed data resides in the processor cache.

Keywords. Main memory databases, random memory access, pointer chasing, memory latency, data cache miss, memory stalls, cooperative multitasking, interleaved execution, coroutines, C++ coroutines, latency-bound operation, software hyperthreading.

Zusammenfassung

Datenbankmanagementsysteme greifen auf Daten im Speicher entweder sequenziell oder zufällig zu. Doch trotz großer Anstrengungen von Computer-Architekten, Kompilerentwicklern und Systemingenieuren ist der zufällige Speicherzugriff im Vergleich zum sequentiellen Zugriff nach wie vor kostspielig und ineffizient, sobald die Daten größer als der Cache eines Prozessors sind. Im Zeitalter von Big Data hängt die Geschwindigkeit der Datenverarbeitung maßgeblich von der Zugriffsgeschwindigkeit des Speichers ab. So dauern Zugriffe auf DRAM oder nichtflüchtigen Speicher mehrere zehn beziehungsweise hunderte von Nanosekunden. Auf Grund der unterschiedlichen Art und Weise wie Entwickler Programmcode schreiben und dieser dann von Rechnern ausgeführt wird, wird daher zur Programmausführung viel Zeit mit Warten auf den Speicher verschwendet. Dabei könnte in dieser Zeit auch andere Arbeit parallel ausgeführt werden.

Diese Dissertation führt kooperatives Multitasking als wesentliche Möglichkeit zum Verbergen von Speicherlatenzen innerhalb von parallelen Anwendungen und Aufgaben ein. Zunächst systematisieren wir das Konzept von kooperativem Multitasking und präsentieren eine Analogie zum Amdahl'schen Gesetz für das Verbergen von Speicherlatenzen. Im Anschluss stellen wir *Verschränkung mit Koroutinen* vor, eine allgemeine und praktikable Technik zum Verschränken von parallelen Ausführungseinheiten innerhalb der Ausführung eines einzelnen Threads, um Speicherlatenzen zu verbergen. Diese Form von kooperativem Multitasking ermöglicht signifikante Leistungsverbesserungen für analytische und transaktionale Anwendungsfälle, die durch eine Vielzahl von unvermeidbaren Speicherzugriffen verlangsamt werden. Dazu zählen beispielsweise der Index Join und die Tupelrekonstruktion sowie nebenläufige `GET` und `PUT` Operationen einer Schlüssel-Werte-Datenbank. Sofern genügend parallele Arbeit zur Verfügung steht und die Hardware parallele Speicherzugriffe sowie einen Mechanismus für effiziente Kontextwechsel bereitstellt, macht die verschränkte Ausführung zentrale Datenbankoperationen unabhängig von der Speicherlatenz. Dies führt dazu, dass Operationen so ausgeführt werden können, als wären alle notwendigen Daten bereits im Cache des Prozessors verfügbar.

1 Introduction

For more than a decade, the database industry has been undergoing a shift towards main memory systems, targeting faster workload execution. This shift from out-of-core to main memory data management, while eliminating the previous main execution bottleneck, i.e., disk I/O, leaves main memory as the slow component in the database system and the next frontier in the quest of ever-increasing workload performance.

Fact is, the execution of both analytical and transactional workloads on modern database systems is inefficient [9, 30, 56] due to instruction and data cache misses that lead to execution stalls. Instruction cache misses can be avoided with code generation that reduces the instruction footprint of a workload [25]; whereas data cache misses are eliminated with data structures optimized for the cache hierarchy [27, 32, 51, 55, 61], or hidden with techniques like out-of-order execution and software prefetching. This thesis studies unavoidable data cache misses that cause accesses to main memory, whose latency is too high to hide with the aforementioned techniques.

1.1 The cost of indirection

Indirection constitutes the main source of data cache misses and the associated stalls. Database systems rely on indirection for efficient random data access that avoids full table scans. They typically employ index structures like B+-trees and hash-tables, or algorithms like binary search to locate the desired records in sublinear time. Although these access methods avoid touching all data, they exhibit memory access patterns that cannot be modeled and predicted by hardware. Such memory accesses incur data cache misses with latencies up to a couple of hundred cycles, resulting in the observed inefficiency. Naturally, significant investments have been made towards cache-friendly data structures that minimize—but not eliminate—the number of cache misses. The remaining cache misses are treated with software prefetching, which hides memory latency by overlapping memory access with computation. However, index structure traversals involve sequences of memory accesses where the previous access determines the following one, presenting a dependency chain that reduces the potential of overlapping access with computation.

Data cache misses affect database workloads, owing to the pervasive use of index structures. Along with table scans, index-heavy operators like index and hash joins or group-bys dominate analytical (OLAP) workloads. When the indexes involved outsize the cache, these operations compound the latency problem by comprising a multitude of lookups. These lookups do not depend on each other but are executed sequentially, resulting execution profiles where most execution time consists of memory stalls. At the same time, transactional (OLTP) workloads concurrently locate and update multiple records using point queries that are facilitated by indexes. In both analytical and transactional workloads, the use of indirection implies cache misses and long memory stalls that are difficult to mitigate.

Many important database operations, such as the aforementioned joins and concurrent point queries, penalize the overall workload execution with their cache misses. Still, they comprise enough work which could in principle be executed while asynchronously fetching data, thereby hiding the latency of main memory accesses. Instead, with the current system implementations, the latency is exposed to the runtime, rendering these operations latency-bound.

1.2 How humans write code and how hardware runs it

The difficulty in hiding memory latency is rooted in the way humans think and write code. Humans think in terms of coherent sequences of ideas, in which one idea leads to the next one forming trains of thought. Furthermore, the human mind can cope with vast amounts of knowledge only thanks to abstraction, which allows to focus on the right level of detail by encoding away entire notions and ideas into single terms. These two characteristics of how people think are reflected in how humans write code: the trains of thought are expressed as individual tasks with dependency chains, whereas abstraction techniques are easily identified in structured programming as the functions, classes, and modules that comprise a program.

Considering how hardware architectures have been designed to hide latency, we see a discrepancy between human-written code and hardware mechanisms. Current general-purpose processor architectures employ superscalar pipelines and complex out-of-order engines to maximize the number of instructions executed per time unit, yet they are effective only on tasks with enough instruction-level parallelism. Compilers try to increase the instruction-level parallelism in the generated assembly by reordering instructions; however, the compiler can prove the correctness of this reordering only within function bodies, limiting the reordering scope. Consequently, the way humans write code implies low instruction-level parallelism, which limits the capability of the hardware to hide latency, thus leading to memory stalls.

One approach to reconcile code and its execution by hardware is to manually rewrite the code, reordering instructions to achieve the ideal execution schedule that hides latency with independent work. However, tearing down abstractions and breaking dependency chains means the code no longer resembles a coherent thought process, which makes it hard to read and reason about. Most of the time and especially in production environments, the performance gains are not worth the

unmaintainable codebase that would occur from such rewrites.

We need a principled way to change the order in which instructions enter the processor pipeline, while preserving the logic and structure of the code these instructions correspond to. Changing execution order while preserving logic and structure requires to separate the two by employing the proper abstraction. This abstraction must keep each individual task intact, and allow to cooperatively multitask, switching between different tasks upon cache misses in order to find work to execute.

The current state of the art in multitasking is lacking when the goal is to hide the latency of main memory access. On the hardware side, simultaneous multithreading has proven ineffective as mainstream processors offer only coarse-grained control over context switching (via OS threading) and support only a limited number of hardware contexts (2 in Intel, 8 in IBM Power) which often comprise insufficient work to completely hide memory latency. On the software side, the existing techniques comprise compiler optimizations (e.g., strip mining, loop distribution) that automatically reorder code without programmer effort, and principled ways to rewrite code (e.g., group prefetching, asynchronous memory access chaining), while preserving some notion of individual tasks; still, the software techniques proposed to this day either apply only to limited code patterns, or increase code complexity to a prohibitive extent. As a result, multitasking remains a niche approach outside of the standard toolbox of latency hiding techniques.

1.3 Thesis statement and contributions

This thesis addresses the inefficiency problem main memory accesses impose to database system implementations and aims to improve workload performance by eliminating processor stalls.

Thesis Statement

When database systems access main memory, the processor wastes valuable execution cycles waiting for data to be fetched instead of executing work from the many other parallel tasks that comprise a database workload. Cooperative multitasking—implemented efficiently leveraging language and compiler support—minimizes processor stalls by overlapping memory access and computation across parallel tasks, thereby drastically improving workload performance.

Our work establishes cooperative multitasking as the principal way to hide memory latency in latency-bound operations that comprise parallel tasks. Using one processor core, a group of tasks is normally executed sequentially, one task after the other. In this scenario, when a data cache miss occurs in a task, the processor pipeline stalls, waiting for the requested data to be fetched from memory, instead of executing work from the other tasks. With multitasking, we can interleave task execution, so that a task is suspended upon a cache miss and control of execution is transferred to another task, and in general as many tasks as necessary to retrieve

enough instructions to execute while fetching data. This way, memory latency is in effect hidden and the processor does not stall.

We make the following key contributions:

- We identify index joins and tuple reconstruction in column stores to be fundamental database operations that are latency-bound and comprise parallel tasks.
- We devise a practical form of cooperative multitasking that employs coroutines to interleave task execution and hide main memory latency. *Interleaving with coroutines* is the core technical contribution of this thesis, and we assess its effectiveness through microbenchmarks, as well as end-to-end experiments in two systems, SAP HANA and Silo. In the microbenchmarks, our technique offers similar performance improvements to prior software techniques, but, unlike them, it results into code that resembles the original code structure, and is thus easy to implement and maintain. The end-to-end experiments provide evidence that interleaving with coroutines applies also to tasks with complex codepaths, for which the prior techniques would be impractical or even infeasible to use.
- We introduce a performance model that describes how task interleaving works, determines the optimal number of tasks to interleave given the latency to hide, the task code, and overhead of the interleaving technique. In addition, we provide a formula that estimates the speedup to expect if we hide latency, serving as the analogue of Amdahl's law for interleaved execution. With our model, we fill a gap in the literature, systematizing cooperative multitasking as an approach to hide any kind of latency.

The aforementioned contributions, serve as a platform to show the following key insights:

- Memory latency can be hidden, so long as there is work for the processor to execute. In many latency-bound database operations, this work can be found in parallel tasks.
- Interleaved execution renders latency-bound operations either compute- or bandwidth-bound, depending on the overhead introduced by the interleaving technique and the number of concurrent memory requests generated by the size of the task group that comprises the necessary amount of work. Provided with enough parallel work, sufficient number of outstanding memory requests supported by hardware, and a low-overhead implementation of interleaved execution, any latency-bound operation can operate with predictable performance, as if the entire working set fits into the cache.
- Interleaved execution improves the overall execution throughput of a group of tasks, at the cost of higher clock time for individual tasks, when the work interleaved is more than the latency to hide. In case individual task clock time matters, e.g., when the tasks correspond to transactions, we can trade off throughput to meet service level objectives.

- The coroutine construct is the proper abstraction to separate the order in which instructions from a group of tasks reach the processor pipeline, from the logic and structure of the code in each of these tasks.

1.4 Thesis outline

This thesis is organized as follows: Chapter 2 introduces the essential background for the topics discussed in this thesis. Chapter 3 systematizes interleaved execution, introducing an analytical model which helps to assess the optimal configuration parameter values and estimate the resulting speedup. In the same chapter, we present how to use coroutines to interleave parallel tasks, covering cases from simple lookups to one index to complex codepaths in production database systems involving accesses to multiple different data structures. Further, Chapter 4 demonstrates a thorough performance analysis of interleaving with coroutines (a) compared to prior software-based techniques and simultaneous multithreading, (b) scaling with the number of cores, and (c) for both DRAM and NVM latency. Chapter 4 presents the impact of interleaving on the performance of analytical and transactional workloads. Finally, Chapter 5 summarizes takeways, describes other application areas for interleaving, and lays out directions for future work.

Chapters 3 and 4 incorporate content published in [47, 48, 49]. Moreover, the effectiveness and practicality of our coroutine-based approach has been independently corroborated by [24, 28, 45].

2 Preliminaries

In this chapter, we provide the reader with preliminary information, necessary to follow the rest of this thesis. We first discuss the computer architecture elements that are essential to understand the results of our performance analyses. Then, we describe the SAP HANA column store, on which we base the main prototype we use in our end-to-end evaluation. Further, we categorize the traditional software mitigations for main memory latency, explaining where multitasking fits in the presented categorization. Finally, we introduce the running example of this thesis, the binary search implementation on which we apply and analyse our latency-hiding technique, interleaving with coroutines.

2.1 Elements of computer architecture and performance analysis

In this section, we describe the essentials of modern computer microarchitecture for understanding our performance analyses using the Top-Down Microarchitectural Analysis Method (TMAM) by Intel. The interested reader can find more information on computer architecture textbooks, e.g., [19], and the processor manuals provided by vendors such as Intel [23] and IBM [21].

2.1.1 The execution pipeline

A simplified model for the execution pipeline of a processor core comprises the following two major components:

- The *Front-end*, which fetches program instructions and decodes them into one or more micro-ops (μops) that are then *issued* to the Back-end.

- The *Back-end*, which *executes* a μop on an available execution unit, as soon as the operands of the μop become available. μops that complete execution *retire* after writing their results to registers or memory, as specified be the instruction.

In this thesis, the semantic difference between instructions and μops is not important, so we will be using the former term.

To increase the number of instructions they retire per second (*IPC*), modern core pipeline designs employ techniques, such as superscalar execution, out-of-order execution, speculative execution, and simultaneous multithreading.

Superscalar execution leverages the multiple execution units (8 in the Intel Skylake microarchitecture) per each core by issuing, executing, and retiring multiple instructions in parallel per cycle (up to 4 on the Intel Haswell microarchitecture). Of course, the actual number of instructions issued, executed, and retired in parallel depends on the Front-end supplying enough instructions, and the Back-end having the resources to execute them.

Out-of-order execution executes instructions whose operands are available without waiting for previous instructions to retire. However, out-of-order execution is transparent to the outer world, so instructions retire in the order they were issued. The *reorder buffer* facilitates in-order retire by tracking the issued but non-retired instructions. New instructions can keep being issued so long as there is space in the reorder buffer; instructions with long latency, such as loads that fetch data from main memory are typical examples when the reorder buffer becomes full, blocking execution.

Speculative execution decodes and executes instructions before determining if the control flow path they belong to should be executed. Branch speculation is the most common form of speculative execution that works by taking a snapshot of the architectural state and speculatively executing one of the two branches until the branch condition is evaluated [17]. In case of conditions that involve fetching data from memory, speculation allows instruction execution to continue until either the reorder buffer becomes full, or, in tight loops with branches, the maximum number of snapshots are already taken—whichever happens first. On correct prediction of the branch, the respective snapshot is dropped and the speculatively executed work is retired; otherwise, the results of speculative execution are dropped and the snapshot becomes the starting point from where execution continues. Besides to wasted work in the Back-end, bad speculation translates also into problems in the Front-end, since fetching and decoding the instructions of the correct branch may lead to instruction cache misses. Most of the time, however, branch conditions are predictable and speculation improves performance.

Simultaneous multithreading (SMT) executes multiple sequences of instructions on the same processor core, switching among them upon stalls to maximize the use of core resources. Each instruction sequence corresponds to a different operating system thread and is mapped to a distinct hardware context. Core resources, such as the register file and the reorder buffer, are statically spit or dynamically shared among the hardware contexts. Intel Xeon implementations exhibit 2 hardware contexts per core, whereas IBM POWER implementations have 4 or 8 contexts per core, depending on the model. *Hyperthreading* is Intel terminology for SMT, and we use the two terms interchangeably.

From these 4 hardware techniques to increase IPC, superscalar execution leverages the *instruction-level parallelism* (*ILP*) available in the instruction sequence that reaches the core, whereas the other 3 try to extract ILP from a wider instruction scope. All 4 techniques are essentially hiding short latencies, but due to physical and design limitations cannot hide the long latency of memory access. This means, if 4 instructions can be executed per cycle with superscalar execution, a latency of 75 ns (300 cycles at 4 GHz) translates into a wasted execution potential of 1200 instructions.

2.1.2 Memory and caches

In this thesis, we focus on main memory databases, hence the relevant parts of the memory hierarchy are main memory and processor caches. Assuming basic familiarity with DRAM and the cache hierarchy [14], here we elaborate on (a) how caches handle cache misses, (b) address translation, and (c) *non-volatile memory*.

Cache miss handling

Processor caches in current Intel Xeon and IBM POWER microarchitectures do not block upon cache misses, but continue to process subsequent memory requests while the cache miss is being served by the other parts of the cache hierarchy. The *Miss Status Handling Registers (MSHR)* handle the state of the outstanding cache misses, enabling the cache to accept more memory requests, both hits and misses. The number of available MSHRs determines how many outstanding cache misses are supported per cache. Each private L1 cache on recent Intel Xeon processors has 10 MSHRs, whereas IBM POWER9 exhibits 8 MSHRs per L1 cache. MSHRs exist also on the other levels of the cache hierarchy, but more importantly in the memory controllers, which handle the memory requests that miss the last level cache and require access to main memory.

The reason we discuss MSHRs here is because L1 MSHRs become a performance bottleneck when we interleave code that is heavily latency bound. In the rest of this thesis, we will refer to L1 MSHRs with their Intel name: *Line Fill Buffers (LFB)*.

Non-volatile memory (NVM)

Main memory can be either volatile or non-volatile. Non-Volatile Memory (NVM), also called Storage-Class Memory or Persistent Memory, is a class of memory technologies that combine the low latency (although higher) and byte-addressability of DRAM with the non-volatility and large capacity of storage media. Examples of NVM include Phase-Change Memory (PCM) [37], Magnetic RAM (MRAM) [13], and Resistive RAM (RRAM) [18]. In the race to bring NVM to market, Intel recently announced the commercial availability of its Optane DC Memory technology in the DIMM form factor [6]: Optane DC Persistent Memory Modules (PMM) embed

up to 512 GB NVM, i.e., double the capacity of the largest DRAM DIMMs available today (256 GB), enabling computer systems with more main memory per socket. However, Optane DC PMM (and NVM in general) exhibits higher access latency and lower bandwidth in comparison to DRAM, with writes being slower than reads. The latency and bandwidth of Optane DC PMM can be masked with the *Memory* mode of operation, which leverages DRAM as a cache and has no memory persistency. For persistency, applications can use the *App Direct* mode: the PMM is exposed as a storage device that mapped to the address space of a process and subsequently accessed like DRAM, with ordinary load and store instructions—still, a special programming model is necessary to ensure that stores are persisted in case of power outage. The two modes are detailed in [23], while our work employs the *App Direct* mode.

Table 2.1 compares the characteristics[1] of DRAM, Optane DC PMM in the *App Direct* mode, and a state-of-the-art SSD. We measured the characteristics of DRAM and Optane DC PMM on a system with a second generation Intel Xeon Scalable processor, codenamed Cascade Lake, using the Intel Memory Latency Checker [1]. As the table indicates, while noticeably faster than SSDs, Optane DC PMM has 4× higher latency and 10× lower random read bandwidth compared to DRAM.

Table 2.1 – Characteristics of DDR4 DRAM, Intel Optane DC PMM, and a Samsung 983 ZET SSD.

	DRAM	PMM	SSD
Read Latency	73 ns	300 ns	15 µs
Seq. Read BW	110 GB/s	36 GB/s	3.4 GB/s
Rand. Read BW	100 GB/s	10 GB/s	3.0 GB/s
Byte-addressable	Yes	Yes	No

2.1.3 Top-down microarchitectural analysis

To analyse the microarchitectural behavior of the code studied in this thesis, we use Intel's Top-Down Microarchitectural Analysis Method (TMAM) [23] as implemented in the Intel VTune Amplifier.

TMAM uses *pipeline slots* to abstract the hardware resources necessary to execute one µop and assumes there are four available slots per cycle and per core. In each cycle, a pipeline slot is either filled with a µop, or remains empty (*stalled*) due to a stall caused by either the *Front-end* or the *Back-end*. The *Front-end* may not be able to provide the *Back-end* with a µop due to, e.g., instruction cache misses; whereas the *Back-end* may be unable to accept a µop from the *Front-end* due to data cache misses (*Memory*) or unavailable execution units (*Core*). In the absence of stalls, the slot can either retire (*Retirement*) or execute non-useful work due to *Bad Speculation*.

[1] We do not discuss write operations and endurance; this work focuses on read operations, whereas endurance is expected to be several years thanks to the *wear leveling* that Optane DC PMMs embed.

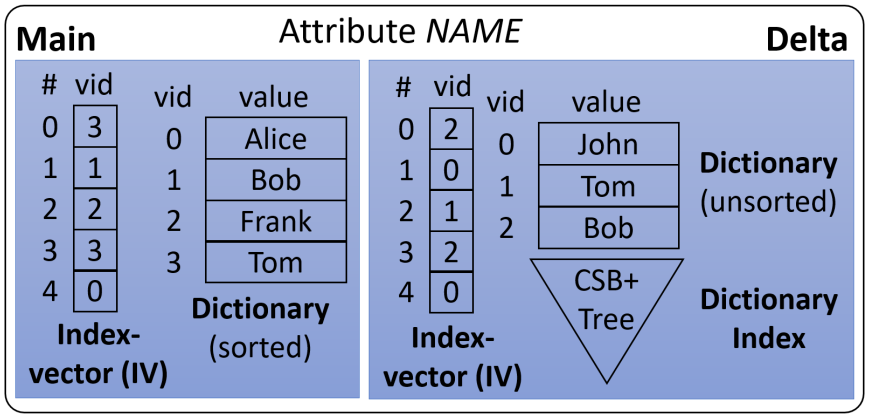


Figure 2.1 – The physical representation of an attribute `NAME` in the SAP HANA column store.

2.2 The SAP HANA column store

The main system we use in the end-to-end evaluation of this thesis is a prototype based on SAP HANA, a commercial main memory database system that supports both analytical and transactional workloads on the same data copy. In this section, we provide a brief overview of the SAP HANA column store, while the interested reader can find more information about the overall system architecture in [8, 16].

To efficiently supports ACID transactions and analytical queries on the same data copy, the physical representation of table attributes in the SAP HANA column store consists of the two fragments: the Main and the Delta, as depicted in Figure 2.1. The Main comprises read-only data in a compressed form optimized for space and analytics, whereas the Delta holds the changes effected on the data by transactions; the Delta is regularly merged into the Main in the background, and upon explicit request by the user or the administrator. This separation of read-only and update-friendly data facilitates the use of NVM [8], with Main and Delta respectively placed on NVM and DRAM.

Both Main and Delta fragments employ dictionary encoding [16, 33, 35, 46, 50], a common compresssion technique that assigns a code to each distinct data value and stores the resulting collection of codes for each column, along with the code-value mapping, i.e., the dictionary. In the HANA nomenclature, codes and the code collections are respectively known as value ids and index vectors. In the Main, the value ids are assigned in an order-preserving manner—the dictionary is sorted—, whereas they are assigned in a first-come-first-served way in the Delta; the Main dictionary is thus a sorted array, whereas the Delta dictionary is an unsorted array, indexed by a cache-conscious B^+-tree (CSB^+-tree) [51] that provides a sorted view of the values. Main dictionaries can be further compressed with compressions schemes that depend on the attribute datatype.

Note that dictionary encoding and compression in general introduce one or more levels of indirection in the physical representation of attributes. The indirection reduces space requirements for datasets with redundancy, at the cost of additional cache misses when the involved data structures outsize the cache. Fetching the value that corresponds to a row in a column, as well as finding the rows that contain a specified value are pointer-chasing operations and thus latency-bound. Both operations are the respective building blocks for the tuple reconstruction and the index joins we study in this thesis, the two key database use cases where multitasking effectively hides access to main memory.

2.3 Traditional memory latency mitigations

Given the hundrends of instructions that can be retired in the 73 ns it takes to fetch a cacheline from main memory, the literature is full of software techniques that deal with cache misses. Based on how they affect the number of cache misses and the incurred penalty, these techniques fall in one of the following categories:

- *Eliminate cache misses* by increasing spatial and temporal locality. Locality is increased by (a) eliminating indirection with cache-conscious data structure designs, e.g., CSB$^+$-tree [51]; (b) matching the data layout to the access pattern of the algorithm, i.e, store data that are accessed together in contiguous space; or (c) reorganizing memory accesses to increase locality, e.g., with array [32] and tree blocking [27, 61]. However, unless data access becomes sequential, not all cache misses can be avoided; for sufficiently large data structures, main memory accesses are inevitable.

- *Reduce the cache miss penalty* by scheduling independent instructions to execute after a load; this approach increases instruction-level parallelism and leads to more effective out-of-order execution. To reduce the penalty of accessing main memory, a non-blocking load is introduced early enough, allowing independent instructions to execute while fetching data. This is achieved by *prefetching* within a sequence of instructions, or by exploiting *simultaneous multithreading* with helper threads that prefetch data [60]. Still, in pointer-chasing code, one memory access depends on the previous one and there are few independent instructions in-between, so the benefit of these techniques is minimal.

- *Hide the cache miss penalty* by overlapping independent memory accesses. The memory system can serve several memory requests in parallel (10 in current Intel CPUs) and exploiting this *memory-level parallelism* increases memory throughput. Overlapping, though, requires independent memory accesses which are scarce in pointer-chasing code.

The techniques in the first category mitigate memory latency by avoiding memory access (*latency avoidance* techniques), whereas the other two categories tolerate latency (*latency tolerance* techniques). In this thesis, we assume all these techniques have already been applied to the data

structures we study; the observed memory latency cannot be avoided and cannot be tolerated within individual tasks, thus motivating for multitasking.

2.4 Running example: binary search

In this section, we describe binary search, a fundamental search algorithm that plays a significant role in the SAP HANA column store and database systems in general. In the rest of this thesis, we employ binary search as the main example of a latency-bound operation.

Searching for an element in sorted array is a basic operation of computer science that attracts even today the attention of researchers who aspire to extract the best possible performance [54, 59]. The established algorithm to perform this operation is binary search. The binary search algorithm splits the search array into two roughly equal subarrays, compares the element in the middle to the searched element, and, so long the array contains more than one element, recursively calls itself on the left or right subarray depending on the comparison result. In the lower bound variants of binary search, the single element in the array of the last recursive call is either the searched element if it appears in the range, or the first element that is larger than the searched element.

In Listing 2.1, we depict an iterative C++ implementation of the lower bound variant of binary search. The `lower_bound` function we present here is used in our microbenchmarks as the `Baseline` implementation we compare against. `lower_bound` accepts two arguments, the sorted array `table` and the `value` to search for. The body of the **`while`** loop in lines 4–11 retrieves the value `v` from the `probe` position and continues the search to the left or right subarray respectively if `v` is less than `value` or not (lines 7–9). The loop runs until the `size` of the occuring subarray becomes less or equal to 1, and the **`if`** statement outside the loop ensures the lower bound for the searched element is returned by incrementing the low variable in case after the loop `size == 1` and `table[low] < value`.

```
1  int lower_bound(vector<int>& table, int value) {
2    int size = table.size();
3    int low = 0;
4    while(size > 1) {
5      int probe = low + size / 2;
6      int v = table[probe];
7      if(v < value) {
8        low = probe;
9      }
10     size -= size / 2;
11   }
12   if(size == 1 && table[low] < value) {
13     low++;
14   }
15   return low;
```

```
16 }
```

Listing 2.1 – Binary search.

In this code, the array access in line 6 is effectively a random access that cannot be predicted by hardware, incurring a cache miss when the array outsizes the cache. Given the work each loop iteration comprises does not suffice to hide main memory latency, the cache miss translates into memory stalls. As a result, although the sorted array exhibiting no pointers, binary search is similar to pointer chasing in terms of microarchitectural behavior.

Moreover, the code the compiler generates for the **`if`** statement in lines 7–9 plays a major role in terms of microarchitectural behavior. At each iteration, the search continues *with the same probability* in either the left or the right subarray, depending on the result of the comparison in line 7. If the compiler generates a conditional branch, the assembly looks as follows:

```
1   ...
2   cmp     $v, $value
3   jl      .LBL
4   mov     $low, $probe
5 .LBL:
6   ...
```

In this case, the processor predicts which of the two alternative control flow paths will be chosen and executes it speculatively. Given both alternatives have the same probability, the prediction is wrong 50% of the time, so the speculatively executed instructions have to be rolled back. As we show in Figure 4.3, bad speculation dominates execution for small array sizes, but becomes less significant as the array size increases. This bad speculation can be avoided by using a predicated instruction, such as the x86 `cmov`. In this case, the corresponding assembly looks as follows:

```
1   ...
2   cmp     $v, $value
3   cmovge  $low, $probe
4   ...
```

When main memory access dominates execution, runtimes are worse compared to speculative execution. This means, in the absence of cache misses, predication is preferable since there is little latency to hide, but in case data to be fetched from main memory, speculated execution is better. Our focus in this thesis is hiding memory latency with multitasking, so we ensure our `Baseline` is compiled with predication.

3 Interleaved Execution

Leveraging task parallelism to hide memory latency is an old idea. However, despite the extensive research works by computer architects, system builders, language designers, and compiler writers, the literature provides neither a model to estimate the performance benefits of hiding latency, nor a practical methodology to interleave parallel tasks with cache misses in the general case. The fact is, production systems waste processor cycles on cache misses because 1. hardware and compiler limitations prohibit automatic task interleaving, and 2. existing techniques that involve the programmer produce unmaintainable code and are thus avoided in practice.

In this chapter, we first systematize interleaved execution, introducing an analytical model based on which one can decide how many tasks to interleave and estimate how much performance can be improved by hiding memory latency. Then, we introduce *interleaving with coroutines*, a generally applicable approach to interleave that is based on another old idea, i.e., coroutines.

3.1 Analytical model

In this section, we expand on the idea of interleaved task execution with a performance model that helps to determine the cases for which interleaved execution makes sense and to estimate the performance benefits over sequential execution.

We study the general case of multiple parallel tasks, some or all of which exhibit memory access patterns that the hardware cannot identify. Our objective is to overlap memory accesses from one task with computation from the others, keeping the processor pipeline filled with instructions instead of incurring memory stalls. We interleave the execution of a group of tasks, switching to a different task each time a load stalls on a cache miss.

In Figure 3.1, we illustrate this *interleaved execution* through an example with three parallel tasks, A, B, and C. For simplicity, all tasks execute the same code, which causes three cache misses. These cache misses effectively split each task into four computation stages of duration $T_{compute}$. With sequential execution, the three tasks run one after the other (we do not depict the execution

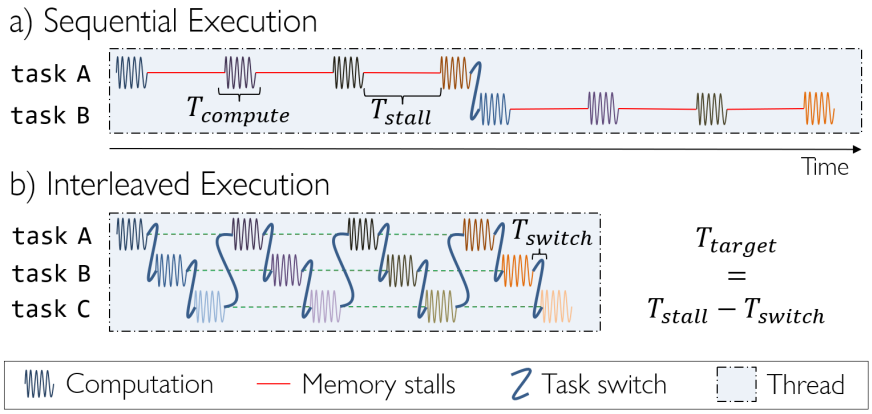


Figure 3.1 – Sequential vs interleaved execution.

of the third task for space reasons). After task A completes its execution, control switches to task B, and then to task C. Every cache miss incurs a main memory access of duration T_{stall}. Contrary to sequential execution, interleaved execution switches from one task to another at each memory access, incurring an overhead T_{switch} that overlaps with T_{stall} and decreases the effective stalls to $T_{target} = T_{stall} - T_{switch}$. In the example of Figure 3.1, when task A causes a cache miss during its first stage, execution switches to the first stage of task B, then to the first stage of task C, and then back to task A.

We generalize the example of Figure 3.1 to model a group of G parallel tasks, where each task i has distinct values for the $T_{i,compute}$, $T_{i,switch}$, and $T_{i,target}$ parameters. To eliminate $T_{i,target}$, i.e., the pipeline slots in which the processor executes no instructions, we need to select a group size G such that:

$$T_{i,target} \leq \sum_{j \in [1..G]}^{j \neq i} (T_{j,compute} + T_{j,switch}) \tag{3.1}$$

For identical model parameters across task, we can drop the indices:

$$\begin{aligned} & T_{target} \leq (G-1) \times (T_{compute} + T_{switch}) \\ \Leftrightarrow\ & G \geq \frac{T_{target}}{T_{compute} + T_{switch}} + 1 \end{aligned} \tag{3.2}$$

Formula 3.2 estimates the optimal G, i.e., the minimum group size for which stalls are eliminated. Interleaving more tasks does not further improve performance since there are no stalls left, i.e., the processor if fully utilized. To the contrary, larger values for G can harm performance by leading to cache conflicts and additional cache misses—see Section 4.2.3. We should also note here the implicit assumption that the hardware supports G outstanding memory requests at a time; if the group of interleaved tasks issues more memory requests than the hardware-supported limit, we should expect few to no performance benefits.

For a group of G parallel tasks with identical model parameters, we can also calculate the speedup thanks to interleaved execution. The total time required to sequentially execute the group is $T_{sequential} = G \times (T_{compute} + T_{stall})$. With interleaved execution, the necessary time becomes $T_{interleaved} = G \times (T_{compute} + T_{switch}) + T_{remaining\ stalls}$, where $T_{remaining\ stalls}$ is difficult to model and depends on the model parameters and the group size. The ratio of these two quantities determines the speedup:

$$\begin{aligned} \text{Speedup} &= \frac{T_{sequential}}{T_{interleaved}} = \frac{G \times (T_{compute} + T_{stall})}{G \times (T_{compute} + T_{switch}) + T_{remaining\ stalls}} \\ &= \frac{T_{compute} + T_{stall}}{T_{compute} + T_{switch} + \frac{T_{remaining\ stalls}}{G}} \end{aligned} \tag{3.3}$$

If $T_{remaining\ stalls} = 0$, i.e., all stalls have been eliminated, we get the maximum speedup:

$$\begin{aligned} \text{Speedup}_{\max} &= \frac{T_{compute} + T_{stall}}{T_{compute} + T_{switch}} = \frac{1}{\frac{T_{compute}+T_{switch}}{T_{compute}+T_{stall}}} = \frac{1}{\frac{T_{compute}}{T_{compute}+T_{stall}} + \frac{T_{switch}}{T_{compute}+T_{stall}}} \\ &= \frac{1}{\%_{compute} + \%_{switch}} \\ \text{where } \%_{compute} &= \frac{T_{compute}}{T_{compute} + T_{stall}} = \frac{T_{compute}}{T_{sequential}} \\ \text{and } \%_{switch} &= \frac{T_{compute}}{T_{compute} + T_{switch}} = \frac{T_{switch}}{T_{sequential}}. \end{aligned} \tag{3.4}$$

Formula 3.4 indicates that speedup is determined by the stalls observed in sequential execution and the cost of switching from one task to another. In Figure 3.2, we depict the ideal speedup as a function of $\%_{stall}$ for different values of $\%_{switch}$. The lower the overhead added to the total execution by the task switch mechanism, the better the speedup. In the ideal case of instant switching between tasks, i.e., $T_{switch} = 0$, the speedup $\text{Speedup}_{\max}$ depends only on $\%_{compute}$. In this case, for T_{stall} that corresponds to 90% of the sequential runtime, we can expect 10× better performance. As T_{switch} increases, the speedup decreases, while T_{switch} values larger than T_{stall} lead to a slowdown. In essence, we are replacing T_{stall} with T_{switch}, so interleaved execution makes sense only if $T_{stall} >> T_{switch}$.

Based on the observations above and the experimental analysis of Section 4.2, in Section 5.1, we provide some general optimization guidelines for loops that suffer from unavoidable cache misses.

Takeaway. Our analytical model can determine whether interleaved execution makes sense or not: in analogy to Amdahl's law and parallel execution, Formula 3.4 provides an ideal upper bound for the speedup we achieve with interleaved execution, whereas Formula 3.2 helps to pick the optimal group size.

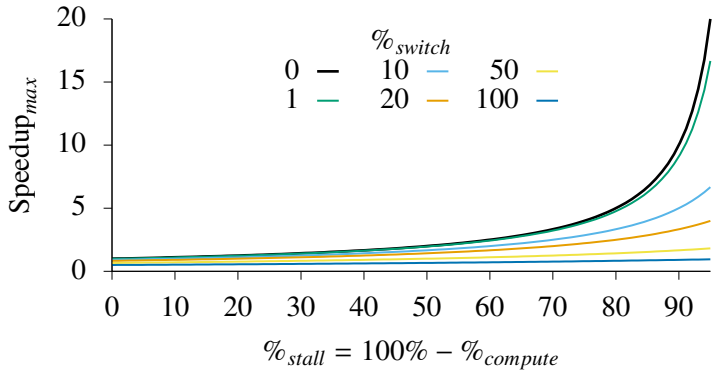


Figure 3.2 – Maximum speedup of interleaved execution according to Formula 3.4.

3.2 Implementing interleaved execution

In this section, we describe how to effectively implement interleaved execution using coroutines, which is the core contribution of this thesis. In principle, we can use any cooperative multitasking technique to implement interleaved execution. However, as we explain in Section 3.1, the mechanism to switch between tasks needs to have much lower overhead than the latency we want to hide. In addition, the task switch mechanism should not cause additional cache misses.

These two restriction preclude implementations that encode concurrently running tasks as operating system (OS) threads and cooperatively multitask with thread synchronization. Preemptive multithreading has non-negligible synchronization overhead, context switching involves system calls and takes several thousand cycles to complete, while switching among full-blown thread stacks likely thrashes the cache and the TLB. With OS threads, the overheads eclipse any memory stalls we want to avoid.

Here we present *interleaving with coroutines*, our practical technique to interleave tasks with low overhead. After briefly introducing the coroutine construct and the relevant support in current programming languages, we describe C++ implementations based on (a) the C++20 language feature and (b) the `Boost` library.

3.2.1 The coroutine construct

A coroutine is a control abstraction that generalizes subroutines [42]. A subroutine starts its execution upon *invocation* by a caller and can only run to *completion*, at which point the control of execution returns to the caller. The coroutine construct augments this lifetime with *suspension* and *resumption*: a coroutine can suspend its execution and return control before its completion; the suspended coroutine can be resumed at a later point, continuing its execution from the suspension point onward. To resume a coroutine, one has to use the coroutine handle that is returned to the caller at the first suspension.

Although coroutines were introduced in 1963 [12], mainstream programming languages did not support them until recently, except for restricted generator constructs in languages like C# [2] and Python [5]. The advantages of coroutines gave rise to library solutions, e.g., `Boost.ASIO` and `Boost.Coroutine`[1], which rely on tricks like Duff's device[2], or OS support like Windows fibers [53] and `ucontext_t` on POSIX systems [26]. Asynchronous programming and its rise in popularity brought coroutines to the spotlight as a general control abstraction for expressing asynchronous computations without requiring the use of callbacks or explicit state machines. Languages like C#, Python, Scala and Javascript have adopted coroutine-like `await` constructs, while C++ is expected to support coroutines as a language feature in the C++20 standard—at the time of writing, coroutine support in C++ has the form of a technical specification [4], which is implemented by the Visual C++ and Clang compilers.

Naturally, database implementations have also picked up coroutines to simplify asynchronous I/O [52], but this thesis pioneers the use of coroutines for hiding memory latency.

3.2.2 Interleaving with C++ coroutines

Coroutines can yield control in the middle of their execution and be later resumed. This ability makes them candidates for implementing interleaved execution, as already remarked by Kocberber et al. [29]. An efficient implementation needs (a) a suspension/resumption mechanism that consumes a few tens of cycles at most, and (b) a space footprint that does not thrash the cache. C++20 coroutines satisfy these requirements, being compiled into assembly code that resembles a state machine. In a sequence of transformation steps, the compiler splits the body of a coroutine into distinct stages which are determined by the suspension/resumption points. These stages correspond to the state machine stages that a programmer would otherwise had to derive manually; in the coroutine case, however, the compiler performs the transformation, taking also care of preserving the necessary state across suspension/resumption. The compiler identifies which variables to preserve and stores them on the heap area of the address space of the process, in a dedicated coroutine frame that is analogous to the memory required to store the state of a manually-derived state machine. Beside these variables, the coroutine frame contains also the resume address and some register values; these are stored during suspension and restored upon resumption, adding an overhead of two function calls. An interested reader can find more information about C++ coroutines in the current draft of the language standard [7] and in Lewis Baker's excellent blog: https://lewissbaker.github.io/.

Binary search as a coroutine

In Listing 3.1, we use the example of binary search to illustrate how to transform any C++ function to a coroutine that supports interleaved execution. Calling `lower_bound_co()` creates a coroutine instance and returns a synthetic `resumable<int>` object (line 2), that keeps a private

[1] http://www.boost.org/doc/libs

[2] http://www.lysator.liu.se/c/duffs-device.html

handle to the coroutine instance. The class `resumable<T>` exposes the following methods:

- `resume()` resumes the execution of a suspended coroutine.
- `done()` returns **`true`** if the coroutine finished, and **`false`** otherwise.
- `result()` provides access to the result value of the original function, which has type `T`. This method is not defined if `T == void`, i.e., there is no result value.

Listing 3.1 – Binary search coroutine.

```
1  template<bool suspend>
2  resumable<int> lower_bound_co(
3    vector<int>& table, int value
4  ) {
5    int size = table.size();
6    int low = 0;
7    while(size > 1) {
8      int probe = low + size / 2;
9      int v = co_await load<suspend>(table[probe]);
10     if(v < value) {
11       low = probe;
12     }
13     size -= size / 2;
14   }
15   if(size == 1 && table[low] < value) {
16     low++;
17   }
18   co_return low;
19 }
```

Lines 5–18 are identical to the original sequential implementation in Listing 2.1, except for lines 9 and 18. In line 9, `table[probe]`, i.e., the array dereference that likely causes a cache miss for large `array` sizes, is replaced with **`co_await`** `load<suspend>(array[probe])`. Depending on the template parameter `suspend`, this **`co_await`** expression either loads `array[probe]` immediately, or issues a prefetch to `array[probe]`, suspends the execution of `lower_bound_co` and loads the `array[probe]` value from the cache upon resumption. Leveraging the extension points of the coroutine feature in C++, we have defined `load` as a library function that allows to succinctly express the desired behavior. Alternatively, line 9 can be replaced with the following lines of code:

```
1  if constexpr(suspend) {
2    prefetch(table[probe]);
3    co_await suspend_always{};
```

```
4 }
5 int v = table[load];
```

The `co_await suspend_always{}` statement suspends the execution of the coroutine, returning control to its caller. Furthermore, the condition of the **`if constexpr`** statement is guaranteed to be evaluated at compile time, enabling the compiler to generate optimal assembly code depending on the `suspend` parameter. We carefully decide the value for `suspend`, relying on profiling to identify cache misses, and introduce suspensions only when a memory dereference incurs a cache miss most of the time—coroutine suspension and resumption has a non-negligible cost that translates into slowdowns in case of cache hits. Ideally, we would make a precise decision at runtime based on cache contexts, but we lack the necessary hardware support for informing memory operations [20]. Finally, in line 18, **`co_return`** replaces the **`return`** keyword, and makes the return value of the coroutine available for access through the `resumable<int>` object.

CSB$^+$-tree lookup as a coroutine

Any function can be converted into a coroutine that supports interleaved execution, following the same methodology as we did with binary search. Here we discuss a second example, a CSB$^+$-tree lookup that adheres to the original proposal of Rao et al. [51].

Listing 3.2 – CSB$^+$-tree lookup coroutine.

```
1  template<bool suspend>
2  resumable<int> tree_lookup_co(
3    tree_t& tree, int value
4  ) {
5    node_t* node = tree.root;
6    while(node->level > 0) {
7      vector<int>& keys = node->keys;
8      resumable<int> h = lower_bound_co<false>(keys, value);
9      int child = h.result();
10     node_t& node = co_await
         load_node<suspend>(node->children[child]);
11   }
12   vector<int>& keys = node->keys;
13   resumable<int> h = lower_bound_co<false>(keys, value);
14   vector<int>& records = node->records;
15   co_return records[h.result()];
16 }
```

In Listing 3.2, we depict the coroutine implementation for the CSB$^+$-tree lookup. For simplicity, we assume a cached root node; for all other inner nodes, we use the `load_node` function in line 10

to prefetch the cache lines of the appropriate child node, suspend, and return a reference to that child node upon resumption. Note that for the binary search within nodes, we use the coroutine of Listing 3.1 without suspension (`lower_bound_co` in line 8 is called with `suspend == false`); the node prefetch brings the `keys` to the cache, so the binary search causes no cache misses. Moreover, a leaf node differs from an inner node since the result of the binary search is used to fetch the searched value from the `valueList` instead of a child node; this value is the result returned in line 15.

Sequential and interleaved execution

An index `lookup`, such as the `lower_bound_co` and the `tree_lookup_co` described above, can be executed with or without suspension, depending on the *scheduler*, i.e., the code implementing the execution policy for the lookup sequence. In Listing 3.3, we present two schedulers:

`run_sequential` performs the lookups one after the other (lines 4–8). The coroutines are called with `suspend == false`, so they do not suspend. The only difference to an ordinary function is that we retrieve the result through the handle.

`run_interleaved` scheduler initializes a group of G coroutines, specifying `suspend == true`, and maintains a buffer of coroutine `handles` (line 14). `G` is a parameter provided by the caller (line 11) and should be selected according to Formula 3.2 to eliminate memory stalls without introducing unnecessary overhead (see Section 4.2.3 for how to calculate the necessary parameters). Moreover, since `lower_bound_co` execution suspends, the `while` loop over the buffer resumes unfinished binary searches (line 24), or retrieves the results from the finished binary search (lines 26–27) and starts new ones (lines 28–34).

Either one of the two schedulers can be selected depending on the probability of cache misses in the lookup and the number of values to look for; in cases like the node search in Listing 3.1, or when there is no other work to interleave with, sequential execution is better as it incurs no overhead. Since the schedulers are agnostic to the coroutine implementation, they can be used with any task.

Finally, it is important to emphasize the distinction between these coroutine schedulers and OS thread schedulers. Coroutine schedulers are normal functions that run on the thread which called them. All coroutines created by these schedulers run, suspend, resume, and complete their execution on the thread of their scheduler. By running multiple threads, each with its own coroutine scheduler, we achieve *multithreaded interleaved execution*.

Performance considerations for C++ coroutines

The described way of interleaving with coroutines relies on an optimizing compiler to generate assembly that (a) in interleaved execution, recycles coroutines from completed lookups for

subsequent coroutine calls (line 30 in Listing 3.3), and (b) in sequential execution, allocates no coroutine frame, since it is not necessary for non-suspending code (line 5 in Listing 3.3).

Listing 3.3 – Sequential and interleaved schedulers.

```
1  void run_sequential(
2    vector<int>& table, vector<int>& values, vector<int>& output
3  ) {
4    for(int v: values) {
5      resumable<int> handle = lookup<false>(index, value);
6      int low = handle.result();
7      output.push_back(low);
8    }
9  }
10
11 void run_interleaved(int G,
12   vector<int>& table, vector<int>& values, vector<int>& output
13 ) {
14   vector<resumable<int>> handles{G};
15   for(int i = 0; i < G; i++) {
16     int value = values[i];
17     handles[i] = lower_bound_co<true>(table, value);
18   }
19   int not_done = G;
20   int i = G;
21   while(not_done > 0) {
22     for(resumable<int>& handle: handles){
23       if(!handle.done()){
24         handle.resume();
25       } else {
26         int low = handle.result();
27         output.push_back(low);
28         if(i < values.size()) {
29           int value = values[i];
30           handle = lower_bound_co(table, value);
31           i++;
32         } else {
33           not_done--;
34         }
35       }
36     }
37   }
38 }
```

These optimizations avoid unnecessary overhead by eliding unnecessary frame allocations and the instructions necessary to manage the lifetime of each coroutine. At the time of writing, no compiler applied these optimizations reliably, so we apply them manually in separate implementations for sequential and interleaved execution. As compiler support for coroutines matures, these optimizations will become unnecessary.

Listing 3.4 – Manual coroutine elision for interleaved binary searches.

```
1  using vec_it = vector<int>::iterator;
2
3  resumable<void> lower_bound_co(
4    vector<int>& table,
5    vec_it& current, vec_it end,
6    vector<int>& output, int& not_done
7  ) {
8    while(current < end) {
9      int value = *current++;
10     int size = table.size();
11     int low = 0;
12     while(size > 1) {
13       int probe = low + size / 2;
14       int v = co_await load<true>(table[probe]);
15       if(v < value) {
16         low = probe;
17       }
18       size -= size / 2;
19     }
20     if(size == 1 && table[low] < value) {
21       low++;
22     }
23     output.push_back(low);
24   }
25   not_done--;
26 }
```

In Listing 3.4, we demonstrate the implementation of `lower_bound_co` optimized for interleaved execution—for sequential execution, we use the implementation described in Section 2.4. `lower_bound_co` executes multiple binary searches so long as the `current` iterator, which is shared among all coroutines, has not reached the `end` of the `values` vector (line 5). The `low` values are directly stored in `output` (line 23), so there is no return value and the template argument of `resumable` is **`void`**. If `end` is reached, the coroutine finishes its execution after decrementing `not_done` (line 25), which is also shared. These changes simplify the scheduler implementations, since `lower_bound_co` iterates over `values` and signals its completion through `not_done`.

`run_sequential` calls `lower_bound_co` and ignores the return value, whereas the **`while`** loop in `run_interleaved` only resumes each of the `handles` so long as `handle.done()` returns **`true`** and `not_done` has a positive value.

3.2.3 Interleaving with library-based coroutines

As already mentioned, libraries like `Boost.Coroutine` provide robust coroutine implementations without special compiler support. These implementations, however, have considerable switching cost. Here, we study `Boost.Context`, a lightweight library that provides primitives for *one-shot continuations* [15, 31], which are used by other `Boost` libraries as a building block for implementing coroutines.

We implement interleaved execution with *one-shot continuations* instead of proper coroutines to avoid abstraction overheads and thus assess the minimum overhead library-based coroutines can have. In the case of `Boost.Context`, a *continuation* represents a suspended computation as a data structure which, contrary to the coroutines of Section 3.2.2, has no special compiler support and consists of a stack, a set of CPU register contents, and a program counter. Being *one-shot* means the computation can be resumed only once.

Listings 3.5 and 3.6 illustrate the resulting code that resembles the structure of the coroutine implementation, albeit with some key differences:

- Resuming a continuation (Listing 3.5, line 14) suspends the current execution context. The suspended context is passed to the resumed context as a new continuation that gets stored in `continuation` for later resumption.
- The **`callcc`** (*call with current continuation*) function creates a new context: **`callcc`** takes one function argument (Listing 3.6, lines 27–29) that accepts a continuation and returns the same or another continuation, suspends the current execution context and allocates a new stack on which it starts its function argument called with a continuation containing the remaining computation.
- Since the function argument of **`callcc`** returns a continuation, we store `low` directly to the `output` vector which is passed by reference.
- The decision to interleave or not depends on the validity of the `continuation` value.

Performance considerations for library-based coroutines

In addition to the changes described in Section 3.2.2, our actual implementation with **`callcc`** directly resumes the next continuation from the `conts` vector, eliding two unnecessary context switches, to and from the scheduling context, which introduce significant overhead: given that the switch logic is library code, it has no knowledge of what architectural state needs to be saved, so

Listing 3.5 – Binary search with `Boost.Context`

```
using cont_t = boost::context::continuation;

template<bool suspend>
cont_t lower_bound_ctx(
  vector<int>& table, int value, vector<int>& output,
  cont_t&& continuation
) {
  int size = table.size();
  int low = 0;
  while(size > 1) {
    int probe = low + size / 2;
    if constexpr(suspend) {
      prefetch(table[probe]);
      continuation = continuation.resume();
    }
    int v = table[probe];
    if(v < value){
      low = probe;
    }
    size -= size / 2;
  }
  if(size == 1 && table[low] < value) {
    low++;
  }
  output.push_back(low);
  return move(continuation);
}
```

Listing 3.6 – Binary search with `Boost.Context`

```
void run_sequential(
  vector<int>& table, vector<int>& values, vector<int>& output
){
  for(int v: values) {
    lower_bound_ctx<false>(index, value, output, dummy_cont);
  }
}

void run_interleaved(int G,
  vector<int>& table, vector<int>& values, vector<int>& output
) {
  vector<cont_t> conts{G};
  for(int i = 0; i < G; i++){
    int value = values[i];
    conts[i] = callcc([&](cont_t&& continuation) {
      return lower_bound_ctx<true>(table, value, output,
        move(continuation));
    });
  }
  int not_done = G;
  int i = G;
  while(not_done > 0) {
    for(cont_t& cont: conts) {
      if(cont){
        cont = cont.resume();
      } else {
        if(i < values.size()) {
          cont = callcc([&](cont_t&& continuation) {
            return lower_bound_ctx(table, value, output,
              move(continuation));
          });
          i++;
        } else {
          not_done--;
        }
    }
  }
}
```

it copies all registers out to the suspended stack and in from the resumed stack. We further reduce the number of instructions involved in a switch by compiling a version that does not copy vector or floating point registers, which are not used in our code. Moreover, we allocate small stacks (still large enough to store the state of `lower_bound_ctx`) to reduce memory requirements and avoid the TLB misses that appear if we use the default stack size, which is larger than a memory page. In principle, both the switch code and the stack size could be automatically optimized by the compiler, which knows what state to save and can assess, in the absence of recursion, how much stack space is needed.

3.2.4 Interleaving arbitrary tasks

Seizing opportunities for interleaved execution in production codebases might not be as easy as in the simple examples we considered so far. Good software engineering practices dictate modularity, i.e., factoring out repeated functionality in separate functions, and abstracting behavior into classes. As a result, the distance between the cache misses and the loop we need to interleave can be several frames in a call stack. Moverover, each parallel task can be executing a different code path, not only in terms of simple control flow divergence, but also with regards to diverse behaviors enabled by compile and runtime polymorphism.

Tuple reconstruction is a concrete example exhibiting the aforementioned properties that complicate interleaving. In the context of a column store like the one of SAP HANA, tuple reconstruction is a loop over a requested set of columns that retrieves the attribute values for a particular row. Column stores often employ dictionary encoding, under which a column consists of a dictionary and a data vector, each having a different physical representation that depends on the stored datatype and the compression scheme used. Retrieving a value from such a column in a production-grade system involves a long call chain that differs from column to column, depending mainly on the data type and the compression scheme used in the dictionary and data vector implementations.

The deep call stacks in conjunction with the multitude of different code paths substantially differentiate tuple reconstruction from the previous examples of multiple lookups to the same data structure, which involve one lookup implementation with few nested function calls. The same few nested function calls can be manually inlined, resulting into one function that can be converted into a `resumable<T>` coroutine as explained in Section 3.2.2. However, the deeper the call stack, the less likely is that a developer tears down the abstraction tower by inlining everything. In addition, inlining is not an option when runtime polymorphism is used. For instance, the columns in the materialization example above are objects with a common interface but different implementations; the code for each column lookup is determined at runtime, so inlining is precluded.

To address these shortcomings of interleaving with `resumable<T>` coroutines, Jonathan et al. [24] suggest a different coroutine type: `task<T>` keeping track of the caller and resumes it upon completion. With this behavior, `task<T>` supports coroutine composition and thus

facilitates the conversion of legthy call stacks into suspendable coroutine chains. When the leaf coroutine in a call chain is suspended, control returns to the scheduler, which, in turn, resumes the next coroutine. We use a variation of the interleaving proposal of Jonathan et al. that resumes the next coroutine upon suspension without returning to the caller. We achieve that by defining our `load` to accept an additional parameter, a lookup `context` that comprises the following:

- An optional `id` that identifies each column lookup and facilitates debugging.
- A low-overhead `allocator` for coroutine frames: given the size of all coroutine frames in a column lookup is bounded, we avoid the unnecessary overheads of a general purpose allocator by using a private-per-lookup, append-only allocator with preallocated memory. By reseting the allocator when the lookup finishes, we can reuse it in the next lookup.
- A reference to the coroutine scheduler. This reference enables `load` to directly resume the next lookup with a tail call.

In Listing 3.7 we depict how to interleave a simplified version of tuple reconstruction. To reconstruct the tuple that corresponds to a given `key`, we first look for the matching `row` in the `KEY` column of table `TBL` (line 2). Then we iterate over all columns (lines 4–7) using a custom `for_each` function that implements a `run_interleaved` scheduler. In each column, we retrieve the attribute value of the desired `row` by calling `get` with the provided `context` (which is different per iteration) and the `row` as arguments (line 6); we store the retrieved value in the appropriate `tuple` position (`tuple[col.id]`). We should note here that `context` simplifies the implementation of `root_task`, which was originally required to keep track of the current suspended leaf coroutine [24].

Listing 3.7 – Interleaving tuple reconstruction.

```
1  tuple_t reconstruct(int key) {
2    int row = TBL.columns[KEY].find(key);
3    tuple_t tuple;
4    for_each(G, TBL.columns.begin(), TBL.columns.end(),
5      [&] (context_t& context, column_t& col) -> root_task {
6        tuple[col.id] = co_await col.get(context, row);
7    });
8    return tuple;
9  }
10
11 task<value_t> column_t::get(
12   context_t& context, int id
13 ) {
14   int code = co_await load(context, codes[id]);
15   co_return co_await dict.decode(context, code);
16 }
```

We already mentioned that each column implementation is different depending on the datatype and the compression scheme of the column. In Listing 3.7, we also present an implementation of the `column_t::get` method for a dictionary-encoded column: we first `load` the encoded row value from a `codes` array (line 14) and then `decode` it using the dictionary `dict` (line 15). The array access causes one cache miss, so we use the `load` to prefetch, suspend, and load, whereas `decode` causes one or more cache misses depending on the `dict` implementation. Despite its simplicity, this example is representative of the code changes necessary also for production-level column implementations, such as the ones of SAP HANA—the only difference is the number of functions we need to convert into `tasks`.

Listing 3.8 – Example of coroutine elision.

```
1 task<int> f_coroutine(context_t& context) {
2   co_return transform(co_await g(context));
3 }
4
5 wrapped<int> f_function(context_t& context) {
6   return wrapped<int>{g(context), transform};
7 }
```

More performance considerations for C++ coroutines

A column lookup comprises many small functions that are inlined by the compiler. Essential to good performance is a compiler that inlines also the `task` counterparts of these functions, eliding a plethora of small coroutine allocations [57] and the associated instructions that manage coroutine lifetime.

At the time of writing, no compiler can reliably inline tasks, so we systematically replace each task that **`co_await`**s one nested task, with an ordinary function that has no **`co_await`** nor **`co_return`** in its body. Consider the example in Listing 3.8: `f_coroutine` passes the result of the **`co_await`** expression to a `transform` function before returning it (line 2). We convert `f_coroutine` to `f_function` by wrapping the `task<int>` returned from `g` along with the function `transform` in a `wrapped<int>` object. This object is not a separate coroutine, but a wrapper that can participate in a **`co_await`** expression, with the distinctive property of transforming the result of the wrapped coroutine before returning it. One variation of this pattern has no result transformation, in which case the nested coroutine can be immediately returned without a wrapper. In other cases, there are two or more code branches that **`co_await`** different nested coroutines and apply distinct result transformations each; to unify the code branches, we define the wrapper to accept not only different transformations but also coroutine types other than `task`. Furthermore, to ensure a task does not outlive its parameters—a likely case when we elide coroutines—we move call parameters either to the nested coroutine or the wrapper. These considerations increase implementation complexity, but, as compiler support for coroutines matures, we expect coroutine elision to become a job for the compiler and not the programmer.

3.2.5 Code complexity

A glance over the interleaved implementations presented in this section indicates that interleaving with coroutines is easy to implement and maintain. We quantify the complexity of our technique through the example of binary search, comparing to the original sequential implementation, as well as to *group prefetching* and *asynchronous memory access chaining*, i.e., the prior state of the art that we present in detail in Section 3.3.1. In particular, we study the following implementations:

- `Baseline` resembles the sequential code in Listing 2.1.
- `Coro-U` uses coroutines and supports both sequential and interleaved execution in the same code, resembling Figures 3.1 and 3.3.
- `Coro-S` uses coroutines but has different implementations for sequential and interleaving execution, incorporating the optimizations described in Section 3.2.2.
- `Cont-U` uses the `continuation`s of the `Boost.Context` library and supports both sequential and interleaved execution in the same code, resembling Figures 3.5 and 3.6.
- `Cont-S` uses the `continuation`s of the `Boost.Context` library but has different implementations for sequential and interleaving execution, incorporating the optimizations described in Section 3.2.3.
- `GP` implements *group prefetching*, resembling the code in Listing 3.9.
- `AMAC` implements *asynchronous memory access chaining*, resembling the code in Listing 3.10.

In Table 3.1, we juxtapose the lines of code (LoC) that are different between each interleaving implementation and the original sequential code (*Diff-to-Original*), as well as the total LoC one has to maintain per lookup algorithm, e.g., binary search, to support both sequential and interleaved execution (*Total Code Footprint*). The first metric hints to the implementation complexity, while the second one to maintainability; for both metrics, lower values are better. `Coro-U` and `Cont-U` require the least modifications/additions (3 and 7 LoC) to the original code, while they have the smallest code footprints (14 and 19 LoC) thanks to the unified codepath; all other implementations have separate codepaths for each mode of execution and, thus, have two implementations for the same lookup algorithm. The code of `Coro-S` and `Cont-S` is nonetheless significantly smaller than that of `GP` and `AMAC`.

Why not use essential or cyclomatic complexity?

Contrary to the two LoC metrics we use above, standard metrics like essential and cyclomatic complexity [40] reflect code properties that are not useful in determining which technique is easier

Table 3.1 – Implementation complexity and code footprint of interleaving techniques. The two `Coro` variants differ the least from the `Baseline` (13 LoC) and require the least amount of code to support both sequential and interleaved execution.

Technique	GP	AMAC	Coro-U	Coro-S	Cont-U	Cont-S
Interleaved	26	69	13	18	18	21
Diff-to-original	18	64	3	7	7	10
Total Code Footprint	39	82	14	31	19	34

to implement and maintain. Essential complexity assesses how structured the code is, examining the entry and exit points of each control flow structure used in the code; depending on their state, coroutines are entered and exited at different points, which means they have high essential complexity although they are arguably easy to understand. Moreover, cyclomatic complexity is a property of the control flow graph, which is almost identical for `AMAC` and `Coro` since they are both state machines with the same states; consequently, `AMAC` and `Coro` have similar cyclomatic complexity, despite the little resemblance between them, in analogy to how a **`switch`** statement and the equivalent sequence of **`if`** statements have the same cyclomatic complexity. For these reasons, we do not consider these two metrics.

3.3 Related work

In this section, we expand on the prior state of the art for implementing interleaved execution in software. Moreover, we present the two (at the time of writing) follow-up works that independently corroborate the value of interleaving with coroutines.

3.3.1 Prior work

Prior proposals for interleaved execution fall into two categories: proposals that rely on the hardware [36] and the compiler [43] to automatically interleave parallel tasks, and proposals that require from the programmers to manually apply loop optimizations that should have been automatically applied by the compiler [10], or convert their code into a state machine [29]. However, mainstream processor architectures and compilers have yet to automate task interleaving, while the development and maintenance costs of the manual techniques prohibit their use in production code.

Here, we focus on the two works of the second category, which are found in the database literature. The first, by Chen et al. [10], applies interleaved execution to hash joins and particularly to lookups on hash tables with bucket lists. The second, by Kocberber et al. [29] expands the application area to cover also lookups on other pointer-based data structures, such as trees and skip-lists.

Group Prefetching (*GP*)

Chen et al. [10] proposes to exploit the task parallelism across subsequent tuples in hash joins by manually applying the *strip mining* and *loop distribution* transformations, that a general-purpose compiler cannot consider due to lack of dependency information. Chen et al. propose two techniques, *group prefetching (GP)* and *software-pipelined prefetching (SPP)*, which transform a fixed chain of N memory accesses into sequences of N+1 computation stages separated by prefetches. *GP* executes each stage in a loop over the whole group of tasks before moving to the next stage; whereas *SPP* executes a different task at each stage in a pipeline fashion.

Both *GP* and *SPP* interleave tasks with a fixed number of stages, yet the code of an index lookup is a loop with a dynamic number of iterations that depends on the index size, meaning that neither technique can be applied in its vanilla form. However, in the absence of early exits, all lookups perform the same number of loop iterations and each iteration has a fixed number of stages. These two properties enable us to share the loop among a group of tasks and statically couple their execution, generalizing *GP* to support pointer chains whose length is only known at runtime. This generalization of *GP* is a minor contribution of the present thesis.

In Listing 3.9, we present the generalized `GP` applied to binary search. To derive this implementation, we decompose the binary search loop (lines 4–11 in Listing 2.1) into a prefetch and a load stage (lines 12–15 and 16–22 in Listing 3.9). The loop is shared among a group of *G* tasks. Each task in the group corresponds to one binary search and each computation stage is executed for the whole group before proceeding to the next stage. As the loop is shared, fewer state variables have to be maintained for each task, reducing the executed instructions. However, the tasks are statically coupled and have to execute the same instructions, precluding the cases described in Section 3.2.4, where cache misses occur in nested function calls and the tasks executed rely on runtime information.

Asynchronous Memory Access Chaining (AMAC)

Kocberber et al. [29] considered as a limitation the coupling imposed by group prefetching to the otherwise independent tasks. This coupling complicates cases where each tasks follows a different code path. To decouple the progress of different tasks, Kocberber et al. proposed *asynchronous memory access chaining (AMAC)*, a technique that encodes traversals of pointer-intensive data structures as finite state machines. The traversal code is manually rewritten to resemble a state machine, enabling each task in a group of traversals to progress independently from the others.

In Listing 3.10, we illustrate binary searches on a sorted dictionary interleaved with *AMAC*. The state machine code is a **`switch`** statement (line 14–44) with one **`case`** for each stage. Every task executes the stages defined by the same switch statement, but each task corresponds to a dedicated state machine, whose state is stored in a `circural_buffer` (line 11) and retrieved

Listing 3.9 – Binary search with *GP*.

```
void lower_bound_gp(
  int G,
  vector<int>& table, vector<int>& values, vector<int>& output
) {
  vector<int> lows(G);
  for(int start = 0; start < values.size(); start += G) {
    int size = table.size();
    for(int low: lows) {
      low = 0;
    }
    while(size > 1) {
      for(int i = 0; i < G; i++) {
        int probe = lows[i] + size/2;
        prefetch(table[probe]);
      }
      for(int i = 0; i < G; i++) {
        int probe = lows[i] + size/2;
        int v = table[probe];
        if(v < values[start + i]) {
          lows[i] = probe;
        }
      }
      size -= size/2;
    }
    for(int low: lows) {
      if(size == 1 && table[low] < value) {
        low++;
      }
      output.push_back(low);
    }
  }
}
```

Listing 3.10 – Binary search with *AMAC*.

```
enum stage { A, B, C, Done };
struct state { int value; int size; int low; stage st = A; };
struct circular_buffer {
  /* other members */
  state& load_next_state() { ... }
};

void lower_bound_amac(int G,
  vector<int>& table, vector<int>& values, vector<int>& output
) {
  circular_buffer bf(G); int not_done = G; int index = 0;
  while(not_done > 0) {
    auto& [value, size, low, st] = b_f.load_next_state();
    switch (st){
    case A: //Initialization
      if(index < values.size()) {
        value = values[index++];
        size = table.size();
        low = 0;
        st = B;
      } else {
        not_done--;
        st = Done;
      }
      break;
    case B: //Prefetch
      if(size > 1) {
        int probe = low + size / 2;
        prefetch(table[probe]);
        size -= size / 2;
        st = C;
      } else {
        if(size == 1 && table[low] < value) { low++; }
        output.push_back(low);
        st = A;
      }
      break;
    case C: //Access
      int probe = low + size / 2;
      int v = table[probe];
      if(v < value) { low = probe; };
      st = B;
    case Done: //No more values to look for
    }
  }
}
```

by calling its `load_next_state()` method[3] (line 13) in a round-robin fashion until all state machines reach the `Done` stage. The **`switch`** statement for binary search has the following stages:

- `A` is responsible for starting the lookup for the next value if there is one, setting the stage to `B` (lines 16–21); if there are no more values to look for, stage `A` decrements the number of active state machines and sets the stage to `Done` (lines 21–24).
- `B` executes either the prefetch stage of an iteration of binary search and sets the stage to `C` (lines 27–32), or performs the last check, stores the result and sets the stage to `A` (lines 32–36).
- `C` performs the load stage of the binary search iteration and sets the stage to `B` (lines 39–42).
- `Done` skips to the next state machine.

The **`switch`** examines the current stage of a task and decides which **`case`** to execute. The conversion of tasks to state machines enables each task to proceed independent of the others, but doing so incurs a significant development and maintenance cost that precludes the adoption of *AMAC* in large codebases. Particularly when cache misses occur at arbitrary depths in the call stack and need to be surfaced to the function that iterates over the columns, *AMAC* dictates to convert all functions into state machines, thereby increasing code complexity to extreme levels.

3.3.2 Follow-up work

At the time of writing, there are two follow-up works that leverage coroutines to implement interleaved execution.

The first, by Jonathan et al. [24], evaluates the technique presented in this thesis, interleaving with C++20 coroutines, on code that is prone to cache misses. The studied data structures range from hash tables and sorted arrays to the more complex Masstree and Bw-tree. Besides confirming the usability benefits of coroutines in juxtaposition to *GP* and *AMAC*, Jonathan et al. propose `task<T>`, a coroutine type that enables coroutine composition and thereby facilitates interleaving entire call stacks, a capability that is essential for the codebase of the Masstree and the BW-tree.

The second, by Kyrianski et al. [28], proposes a domain-specific language (DSL) that supports coroutines, and validates interleaved execution for sorted arrays, binary trees, hash tables, and skip-lists. With full control over the coroutine implementation, this approach achieves best performance by supporting both static interleaved execution similar to *GP* and dynamic like *AMAC*. However, the proposed DSL has little resemblance to ordinary C++ code hampering adoption. Using the frontend of native C++ coroutines with the code generation scheme of this proposal would enable best performance in intuitive C++ code.

[3] Note that we get references to the variables comprising a state using a *structured binding declaration* from C++17.

3.4 Summary

Interleaved execution is a universal way to hide latency. In this chapter, we introduced a simple analytical model that provides an analogue to Amdahl's law for estimating the speedup of interleaved execution, and determines how many tasks need to be interleaved, assuming (a) enough parallel tasks in the workload, (b) hardware support for memory parallelism, and (c) a sufficiently fast mechanism to switch between tasks. Furthermore, we presented two implementations of task interleaving that rely on coroutines. Interleaving with coroutines, whether compiler- or library-based, has two key properties:

1. The lookup logic is kept separate from the execution policy, enabling a single codepath to be configured for sequential or interleaved execution;
2. The coroutine code is mostly the sequential code with a modified function signature and suspending loads for addresses that cause cache misses.

Thanks to these properties, coroutines are practical to use, incurring only minor development and maintenance costs that strongly contrast the prior state of the art.

4 Performance Analysis & Applications

Interleaved execution promises to eliminate memory stalls, improve processor utilization and thereby the overall performance of database systems. The goal of this chapter is to demonstrate that *interleaving with coroutines* not only constitutes a practical and generally applicable way to implement interleaved execution, but also exhibits the execution characteristics that are essential for significant performance improvements. Through microbenchmarks implementing binary searches on sorted arrays stored in DRAM, we first compare interleaving with coroutines to non-interleaved execution and the prior state of the art in terms of performance and microarchitectural behavior. Secondly, we employ analytical and transactional use cases found in database systems with large codebases to assess the practical aspect and effectiveness of our technique for large database systems. Finally, we study the effectiveness of interleaving with coroutines in the case of accesses to non-volatile memory instead of the usual DRAM.

4.1 Experimental setup

For the experiments of this chapter, we have used four different systems. The processor, operating system, and compiler details of the DRAM-only systems used in Sections 4.2 and 4.3 are detailed in Table 4.1, while Table 4.2 contains the corresponding information for the NVM-equipped system used in Section 4.4.

Thread-to-core mapping. For our single-threaded measurements, we pin our microbenchmarks on one core, and have simultaneous multithreading (SMT) disabled in all experiments, but the related ones in Sections 4.2.5, 4.2.6, and 4.2.7 to simplify the interpretation of the results [39]. For our multi-threaded measurements, we pin the microbenchmarks to one socket, and increase the cores used from one to the number of cores on the socket, depending on how many threads we want to use. All 4 systems have two or more sockets, and in our evaluation we avoid external interference by executing our experiments on one socket, having all other processes migrated to other sockets. Finally, unless otherwise mentioned, we run experiments with disabled performance scaling (turbo mode off).

System Identifier	*Windows*	*Linux*	*Power*
Processor	Xeon E5-2660v3	Xeon E5-2683v4	POWER9 SO
Architecture	Haswell	Broadwell	Sforza
Technology	22 nm	22 nm	14 nm
Base Frequency	2.6 GHz	2.1 GHz	2.6 GHz
Max Frequency	3.3 GHz	3.0 GHz	2.6 GHz
# Cores	10	16	16
SMT (Hyperthreading)	2	2	4
L1 I/D Capacity	32 kB/32 kB	32 kB/32 kB	32 kB/32 kB
L1 I/D Associativity	8/8-way	8/8-way	8/8-way
# Line Fill Buffers	10	10	8
L2 Capacity	256 kB	256 kB	512 kB
L2 Granularity	1 core	1 core	2 cores
L2 Associativity	8-way	8-way	8-way
L3 Capacity	2.5 MB	2.5 MB	10 MB
L3 Granularity	1 core	1 core	2 cores
L3 Associativity	20-way	20-way	20-way
DTLB Entries (4 kB/2 MB)	64/32	64/32	64 (shared)
DTLB Associativity	4-way	4-way	Full
STLB (4 kB/2 MB)	1024	1024	1024
STLB Associativity	8-way	8-way	4-way
OS	Windows 10 1809	SLES 12 SP3	Ubuntu 18.04
Compilers	MSCV 14.1/Clang 6	Clang 8	Clang 8

Table 4.1 – System parameters (only DRAM)

Code compilation. We compile our code with two compilers: Microsoft Visual C++ (MSVC) and Clang. The compilation flags are:

- MSVC: `/Ox /std=c++17 /arch:AVX2 /await`
- Clang: `-std=c++17 -fcoroutines-ts -O3 -march=haswell`

Note that on the *Windows* system, we use the MSVC standard library and ABI with both MSVC and Clang, and on the *Linux*, *IBM*, and *NVM* systems we use the widely available GNU C++ Library (`libstdc++`). Using coroutines with Clang requires the `experimental/coroutine` header from the C++ standard library of LLVM (*libcxx*); to avoid an additional dependency, we use a modified copy of the header that works with the two aforementioned standard libraries. Moreover, we use our implementation of the `resumable` type and a port of the `task` type from Lewis Baker's *cppcoro* library[1].

Profiling. We profile our microbenchmarks and analyze their microarchitectural behavior on Intel processors using Intel VTune Amplifier XE 2018 and 2019.

Software prefetching. The instruction-set architectures of the processors we use support temporal prefetch instructions that modify cache contents and non-temporal ones that circumvent the cache hierarchy. In our microbenchmarks, we use non-temporal prefetches to avoid evictions due to cache conflicts, since prefetched data has low temporal locality and the instruction distance between a prefetch and the corresponding load is small. The latter is not true for the applications we study, in which the Line Fill Buffer that holds the prefetched cacheline might be cleared before the corresponding load instruction. In these cases, we use temporal prefetch instructions; these potentially evict other data to slower cache levels, but, as observed also by Jonathan et al. [24], fetching data that has been evicted to L2 or L3 due to collisions is preferable to fetching again from main memory data after a non-temporal prefetch.

4.2 Microbenchmarks

In this section, we study *interleaving with coroutines* in comparison to (a) non-interleaved execution, (b) group prefetching and asynchronous memory access chaining. In particular, we use six binary search implementations: two for sequential execution and four for interleaved. The sequential ones are `std::lower_bound` (abbreviated as `std`) from the standard library of C++, and `Baseline` from Section 3.2.5. From the same Section, we get also our interleaved implementations: `GP` for group prefetching, `AMAC` for asynchronous memory access chaining, `Coro` for compiler-based coroutines, and `Cont` for library-based coroutines. The two coroutine implementations as the interleaved implementations of `Coro-S` and `Cont-S`—we are interested in the best possible performance of the coroutine implementations, so we do not evaluate the performance of the unified implementations, `Coro-U` and `Cont-U`, which are currently not

[1] https://github.com/lewissbaker/cppcoro

System Identifier	*NVM*
Processor	Xeon Platinum 8280L
Architecture	Cascade Lake
Technology	14 nm
Base Frequency	2.6 GHz
Max Frequency	3.9 GHz
# Cores	28
SMT (Hyperthreading)	2
L1 I/D Capacity	32 kB/32 kB
L1 I/D Associativity	8/8-way
# Line Fill Buffers	10
L2 Capacity	1 MB
L2 Granularity	1 core
L2 Associativity	16-way
L3 Capacity	1.375 MB
L3 Granularity	1 core
L3 Associativity	11-way
DTLB Entries (4 kB/2 MB)	64/32
DTLB Associativity	4-way
STLB (4 kB/2 MB)	1536
STLB Associativity	12-way
DRAM (per socket)	96 GB (6 × 16 GB)
Optane DC PMM (per socket)	768 GB (6 × 128 GB)
OS	SLES 15
Compilers	Clang 7

Table 4.2 – System parameters (DRAM and NVM)

well-optimized by the compiler. Furthermore, we ensure the `if` statement, which conditionally assigns the `probe` value to the `low` variable (lines 7–9 in Listing 2.1, and corresponding lines in Listings 3.9, 3.10, 3.1, and 3.5), is executed not as a branch with speculative execution but as a predicated mov instruction. For each interleaving implementation, we can configure the `group size`, i.e., how many lookups run interleaved at any given point in time.

We use these single-threaded implementations to (a) demonstrate the advantages of interleaving with compiler-based coroutines over sequential execution for binary searches over `int` and `string` arrays (Section 4.2.1); (b) explain the performance gains through a microarchitectural analysis of the `int` case, where we also show how to estimate the best group size G (Section 4.2.2); (c) compare the performance of compiler- and library-based coroutines, highlighting the impact of code generation by different compilers (Section 4.2.4). Since compiler-based coroutines (`Coro`) are demonstrated to perform better than their library-based counterparts (`Cont`), the performance of the latter is studied only in Section 4.2.4.

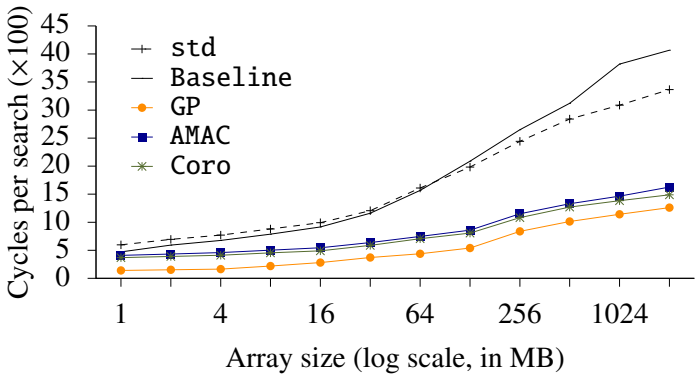
Apart from the single-threaded implementations, we implement also multi-threaded versions of `Baseline` and `Coro` that lookup a different chunk of values on each thread. We study these implementations in Section 4.2.5, where we compare interleaving coroutines to hyperthreading and multithreading, and in Section 4.2.6, where we explain why multithreaded interleaved execution scales better than simple multithreaded execution.

4.2.1 Comparison to GP and AMAC

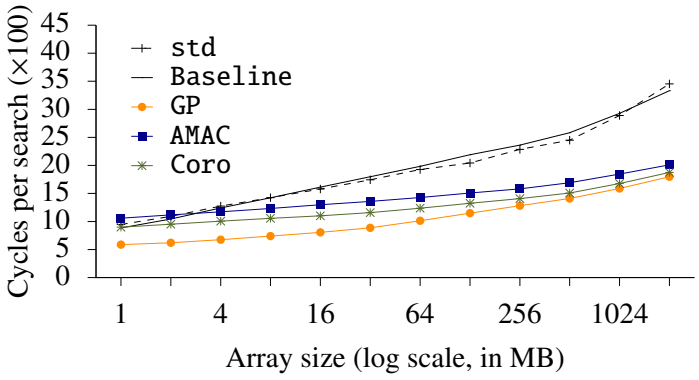
We evaluate the five aforementioned implementations on sorted arrays whose size ranges from 1 MB to 2 GB. We generate the array values using the array indices: for integer arrays, the values are the corresponding array indices, whereas for string arrays we convert the index to a string of 15 characters, suffixing characters as necessary. Furthermore, the list of lookup values is a subset of the array values, selected randomly using `std::mt19937` with seed `0` and `std::uniform_int_distribution`.

Figure 4.1 depicts the performance per binary search with lookup lists of 10K values. We report the average runtime of 100 executions, and, for interleaving implementations, the depicted measurements correspond to the best group size configuration (see Section 4.2.3). Since the instructions executed in a binary search are a logarithmic function of the array size, the horizontal axis has a logarithmic scale.

The difference between sequential and interleaved execution is clear for both integer and string arrays. Both `std` and `Baseline` incur a significant runtime increase for arrays larger than 16 MB. These arrays outsize the last level cache (25 MB), so binary search incurs main memory accesses that manifest as stall cycles, as we explained in Section 2.4. As a result, runtime diverges significantly from the logarithmic function we described above. Contrary to this behavior, runtime increases are less significant for `GP`, `AMAC` and `Coro`.



(a) Integer array.

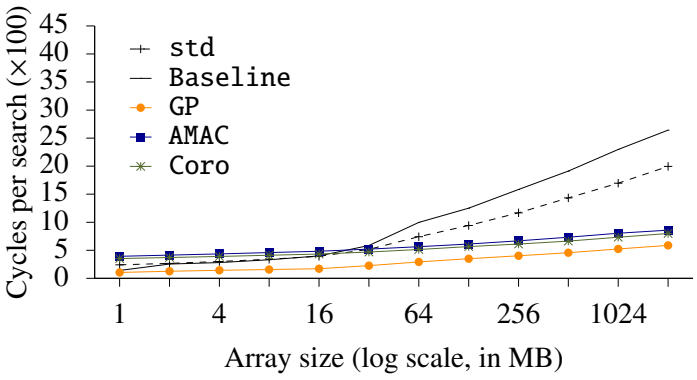


(b) String array.

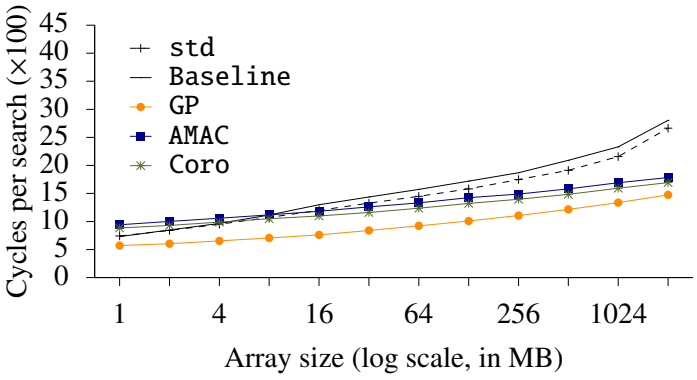
Figure 4.1 – Binary searches over sorted array. Interleaving increases runtime robustness. `Coro` performs similarly to `AMAC`, while the difference to `GP` is smaller for the string case.

Focusing on arrays larger than the last level cache, all interleaving implementations behave similarly. `GP` constantly has the maximum speedup, in the range 2.7–3.7× for integers and 1.8–2.2× for strings. As further explained in Section 4.2.2, this behavior arises thanks to the coupling of the binary searches in the group. `Coro` and `AMAC` follow with decent speedups, in the ranges 2.0–2.4× and 1.8–2.3× for integers, and 1.4–2.1× and 1.2–1.9× for strings. We should note that, thanks to compiler optimizations, `Coro` performs slightly better than `AMAC`, whose data alignment and layout we have carefully optimized.

Finally, as array size increases, we observe a smoother increase of the interleaved execution for strings than for integers. This observation reflects the computationally heavier string comparisons, which de-emphasize cache misses. In Section 4.2.2, we focus on the integer case, identifying how runtime behavior changes for different array sizes.



(a) Integer array.



(b) String array.

Figure 4.2 – Binary searches over sorted array with sorted lookup values. Sorting increases temporal locality, but does not eliminate compulsory cache misses.

Increasing locality with sorting

Sorting small lists is a cheap operation, and thus a valid preprocessing step. In this case, the lookup values are sorted before starting the binary searches. Figure 4.2 depicts the corresponding measurements for integers (strings). `std` and `Baseline` are up to 2.6× (1.8×) and 2.4× (1.8×) faster, owing to increased temporal locality: since subsequent lookups access monotonically increasing positions in the array, the values accessed in one lookup are likely to be cache hits in later lookups. This additional temporal locality benefits also `GP`, `AMAC` and `Coro` up to 2.2× (1.4×), 1.9× (1.3×) and 1.9× (1.3×) respectively. Still, sorting does not affect spatial locality: if the lookup values are not close to each other, which is likely for arrays much larger than the lookup lists, there will still be compulsory cache misses to hide.

Takeaway. Interleaved execution is robust to the increase of the array size, contrary to sequential execution. `Coro` has slightly better performance than the functionally equivalent `AMAC`, while `GP` performs best thanks to the minimal overhead of static interleaving. Furthermore, sorting the lookup values increases temporal locality between subsequent lookups, but does not eliminate compulsory cache misses.

4.2.2 Microarchitectural analysis

To understand the effect of interleaved execution, we perform a microarchitectural analysis of our binary search implementations. We study them for integer arrays and unsorted lookup values, analyzing them with TMAM (described in Section 2.1.3). Furthermore, we leverage the same analysis to determine the best group size for each implementation.

Where does the time go?

In Figure 4.3, we depict the execution time breakdown of a binary search as the array size increases, with the best group size for each technique, i.e., 10 for `GP`, and 6 for `AMAC` and `Coro` (Section 4.2.3 describes how to determine these values). We calculate the execution cycles spent on front-end, memory or resource stalls, wasted due to bad speculation, or retired normally (as specified by *TMAM*) by multiplying the respective percentages reported by VTune with the measured cycles per search.

Owing to the small instruction footprint of our implementations, the front-end and bad speculation components are negligible in all implementations except for `std`, which is penalized by bad speculation as explained in Section 2.4. Notably, however, `std` runs faster than `Baseline` for arrays larger than 16 MB; this means that speculation, even if it is bad half the time, is better that waiting for the data to be fetched from the main memory.

Compared to `std` and `Baseline`, memory stalls are reduced in `GP`, `AMAC` and `Coro`. Memory stalls are negligible until 4 MB, and they start to dominate `GP` execution from 32 MB. `AMAC` and `Coro` exhibit fewer memory stalls than with `GP`, as they execute more load and store instructions when switching. These additional instructions explain the more retiring instructions, as well as the more core stalls, which incur as switching saturates the load and store units of the core.

Takeaway. Interleaved execution eliminates most memory stalls caused by data cache misses.

How does interleaving reduce memory stalls?

Memory stalls occur when a load instruction fetches data from an address that is not in the *L1D* cache. In this case, the *Line Fill Buffers (LFB)* (*L1D MSHR*, see Section 2.1.2) are checked to see if there is a memory request for the same cacheline. If not, a new memory request is created, an empty *LFB* is allocated to track the status of the request, and the request itself gets forwarded to the *L2* cache. If the requested address is not in the *L2*, the request is next forwarded to the *L3* cache (also called last level cache, *LLC*), which is shared among the cores in a socket for the Intel processor used in our experiment. Finally, if the address is not in the *L3*, the request goes to the memory controller and subsequently to the main memory (*DRAM*). Depending on the level where the requested address is found, we categorize a load as a L1D hit, a LFB hit, a L2 hit, a L3 hit, or a DRAM access.

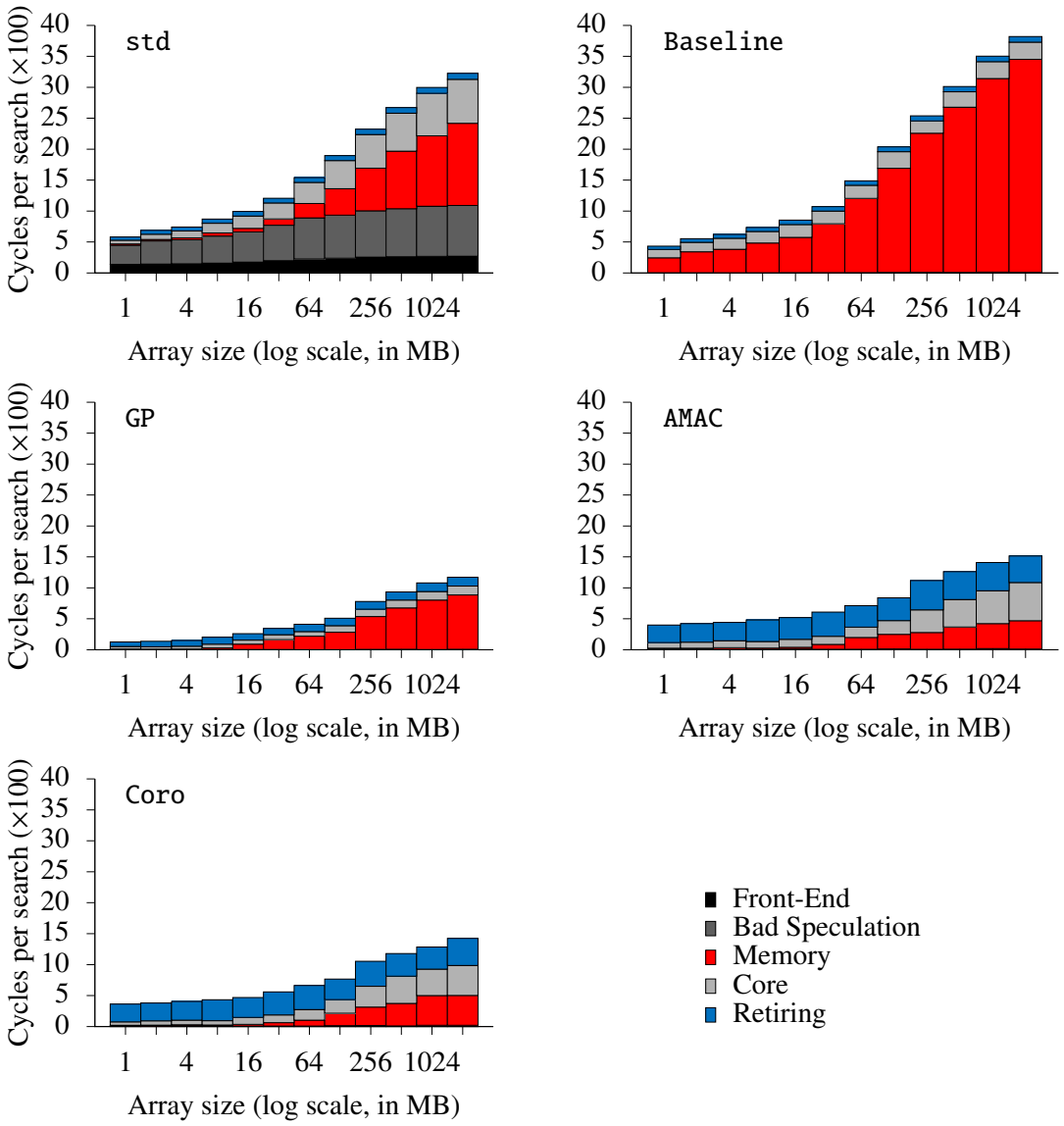


Figure 4.3 – Execution time breakdown of binary search. Interleaved execution reduces memory stalls significantly.

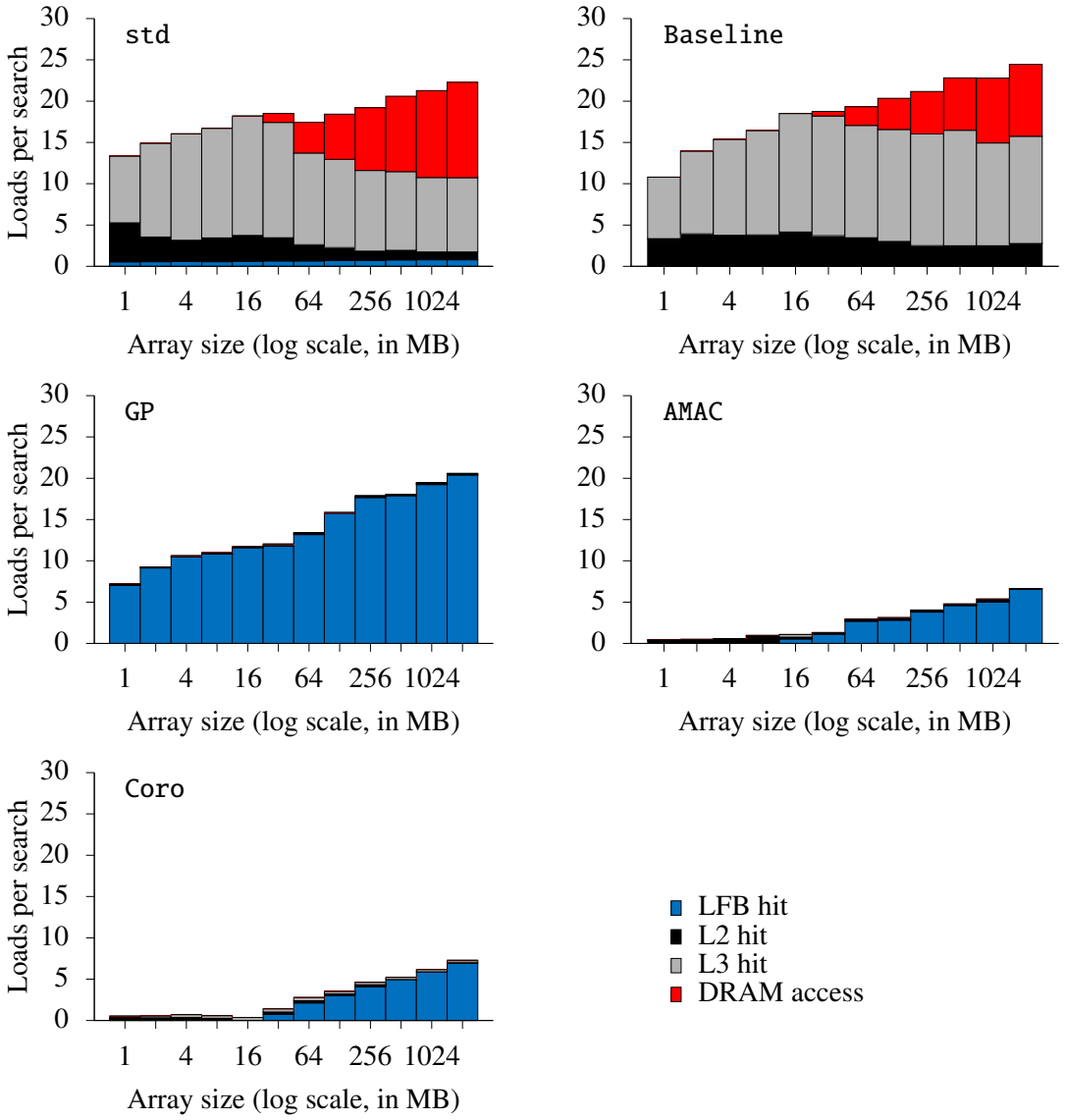


Figure 4.4 – Breakdown of L1D misses. Interleaved execution hides the latency of data cache misses.

In Figure 4.4, we depict a breakdown of the load instructions per implementation and array size, based on the memory hierarchy level in which they hit (we omit L1D hits as they do not cause lengthy memory stalls). We generally observe that, with interleaved execution, most L1D misses are LFB hits. The reason for this behavior is the use of prefetch instructions by the interleaving techniques: each prefetch that misses in L1D creates a memory request allocating an LFB; the corresponding load either finds the data in L1D in case enough instructions are executed between the prefetch and the load, or finds the allocated LFB otherwise. The instructions `GP` injects between a prefetch and the corresponding load are not enough to effectively hide L1D misses, despite using the best group size (see Section 4.2.3 for an explanation); still, they reduce the average miss latency, leading to the observed runtime improvement. Contrary to `GP`, `AMAC` and `Coro` eliminate most L1D misses for arrays up to 32 MB; for larger arrays, the effected L1D misses seem to be caused by address translation (see Section 4.2.2).

Takeaway. Interleaved execution introduces enough instructions between a prefetch and the corresponding load, decreasing the average memory latency of load instructions.

Why does `GP` perform best?

The performance difference between the three interleaving techniques can be explained by their respective instruction overhead: Compared to `Baseline`, from which they are derived, `GP`, `AMAC` and `Coro` execute 1.8×, 4.4× and 5.4× more instructions. These instruction overheads, also reflected as more retiring cycles in Figure 4.3, correspond to the overhead of switching among tasks, which mainly consists of managing state.

Many binary searches on the same array is a best case scenario for group prefetching: all tasks within a group execute the same code for the same number of iterations. The tasks share the binary search loop, reducing the number of instructions executed and the number of state variables that have to be tracked per task. As we describe in Listing 3.9, the tracked state variables include only the searched `value` and the current `low`, whereas `probe` is inexpensively recomputed. In addition to these variables, the non-coupling `AMAC` and `Coro` have to maintain the loop state separately per task, which means they execute more load and store instructions when switching between tasks.

Takeaway. Contrary to `AMAC` and `Coro`, `GP` shares computation among tasks and maintains less state per task. In other words, `GP` executes fewer instructions than `AMAC` and `Coro`, and thus performs best.

How does address translation affect execution?

In Section 4.2.1, we note that runtime increases smoothly for string arrays. However, in the measurements for integer arrays, we observe *runtime jumps* when increasing the array size from 4 MB to 8 MB, from 16 MB to 32 MB, and with every increase beyond 128 MB. Since the

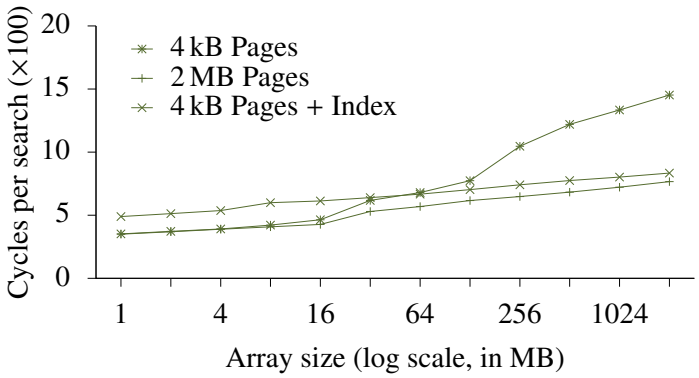


Figure 4.5 – `Coro` on sorted integer array using (a) 4 kB **pages, (b)** 2 MB **pages, and (c)** 4 kB **pages and an index on top of the sorted array. Using either** 2 MB **pages or the index leads to fewer TLB misses, smoothing out the related runtime jumps.**

memory load analysis of Section 4.2.2 cannot explain these runtime jumps, we monitor and analyze the address translation behavior.

Profiling shows most loads hit in the DTLB, the first-level translation look-aside buffer for data addresses. However, DTLB misses can hit in the STLB, the second-level TLB for both code and data addresses; or perform a page walk to find the address mapping in the page tables. In the latter case, the appropriate page tables can be located in any level of the memory hierarchy—we denote the page walks that hit L1D, L2, L3 and DRAM as PW-L1, PW-L2, PW-L3 and PW-DRAM respectively.

The aforementioned runtime jumps correspond to parameters related to address translation. The first runtime jump from 4 MB to 8 MB matches the STLB size, and our profiling results show PW-L1 hits for larger arrays, while the second one, from 16 MB to 32 MB, corresponds to PW-L2 hits. Since the latencies of L1D and L2 are partially hidden by out-of-order execution, the two first jumps are small. However, the PW-L3 hits that cause the third jump cannot be hidden, so increasing the array size beyond 128 MB incurs the most evident runtime increases.

We should note that interleaving works thanks to prefetch instructions. A prefetch does not block the pipeline in case of an L1D miss, and thereby allows subsequent instructions in the task to execute. Yet, the pipeline is blocked until the prefetched virtual address is translated to a physical one, possibly involving long page walks.

Reducing TLB misses

We can avoid TLB misses by using either large pages or an index. Beside the standard 4 kB pages, modern processors can address memory with 2 MB and 1 GB pages, and thereby reduce the stress on the TLB. Still, choosing a larger page size is a non-trivial decision, as their use requires special privileges, manual OS configuration, and special system calls so they cannot

be considered a general-purpose solution; transparent huge pages eliminate this configuration overhead, at the cost of a best-effort operation that does not guarantee the use of the appropriate page size. Alternatively, we can introduce a B^+-tree index with page-sized nodes on top of the sorted array. Lookups on this structure traverse the tree nodes, performing binary searches within each of them. By construction, each binary search in a node avoids TLB misses, since the involved memory accesses touch only one page.

In Figure 4.5, we compare these two alternatives to `Coro` with 4 kB pages. Both solutions improve performance for large arrays by eliminating TLB misses; 2 MB pages lead to strictly faster execution compared to 4 kB pages, up to 1.9×, whereas index traversal involves more instructions, which explains the slowdown for arrays smaller than 32 MB.

Takeaway. Larger array sizes cause TLB misses that interleaving cannot hide; yet we can avoid them with large pages or an index on top of the sorted array.

4.2.3 Choosing the group size

As already mentioned, the results we present correspond by default to the best group size configurations for each implementation. Given that all lookups execute the same instructions, we can estimate the best group sizes by applying the interleaving model of Section 3.1 to the profiling measurements. From `Baseline`, we map memory stalls to T_{stall} and all other cycles to $T_{compute}$. Further, we compute T_{switch} as the difference in retiring cycles between `Baseline` and each of the three interleaved implementations for group size 1. We apply these parameters to Formula 3.2 for a 256 MB integer array, yielding $G_{GP} \geq 12$ and $G_{AMAC} = G_{Coro} \geq 6$. Based on Formula 3.4, we expect the speedups to be at most 5.7×, 3.7× and 3.3× respectively.

To verify these results, we run our microbenchmarks for a 256 MB integer array and depict, in Figure 4.6, our runtime measurements as a function of the group size, which ranges between 1 and 12 concurrent binary searches (performance varies little for larger group sizes). We observe that the best group sizes are 10 for `GP`, and 5–6 for `AMAC` and `Coro`. For `GP`, $G_{estimated}$ differs from $G_{observed}$ owing to a hardware bottleneck: current Intel architectures have 10 LFBs (see Section 4.2.2), limiting the number of outstanding memory accesses and, thus, the benefit of interleaving (hence the smaller observed speedup). Nevertheless, 10 LFBs suffice for `AMAC` and `Coro`, matching our estimates[2].

We should note that interleaved execution with group size 1 makes no sense: `GP`, `AMAC` and `Coro` are slower than `Baseline` due to the overhead of the switching mechanism. The non-negligible overhead emphasizes the need for implementations that switch only in case of interleaved execution; otherwise, the switching mechanism should be bypassed. Finally, similar observations to the ones above can be made for the other array sizes and for string arrays. Varying the array size affects the number of cache misses and not the $T_{compute}$ nor the T_{stall} per cache miss, whereas

[2]The observed speedups differ from the expected ones mainly due to rounding; group sizes cannot be floats.

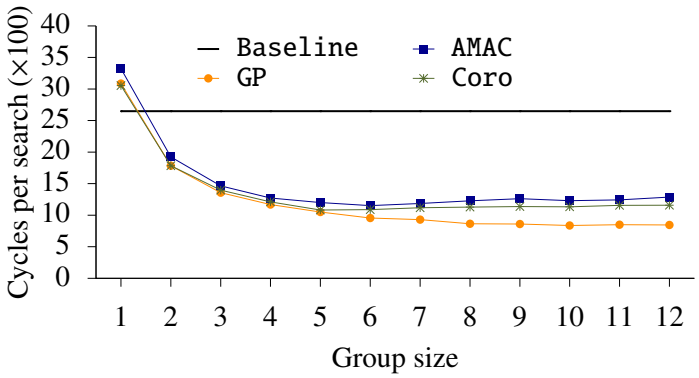


Figure 4.6 – The effect of group size on runtime (for 256 MB integer array). Best group sizes: 10 for `GP`, 5–6 for `AMAC`, `Coro`.

$T_{compute}$ for comparing strings with 15 characters seems to not differ significantly from integer comparison.

Takeaway. Knowing the available computation, the memory stalls, and the switch overhead per task, we can assess the effect of an interleaving technique on a group of lookups. Since the above parameters are similar for all lookups, Formula 3.2 provides a reasonable estimate of the best group size, as long as the hardware supports the necessary memory-level parallelism.

4.2.4 Library- vs compiler-based coroutines

To understand why it is important for coroutines to be supported and optimized by a compiler, we compare `Coro` to the lightweight `Boost.Context` implementation described in Section 3.2.3 (`Cont`). We compile these two implementations along with `Baseline` using both the MSVC 14.13 (`M` variants) and Clang 6.0 (`C` variants) compilers. In Figure 4.7, we report their performance as we increase the array size.

`Coro (C)` is slightly faster than `Coro (M)`, owing to the more comprehensive set of coroutine-related optimizations that Clang applies—the assembly instructions for binary search itself are identical. `Cont (M)` is 25% slower than `Coro (M)`, solely due the significantly more instructions `Cont (M)` executes to replace the execution context —a few registers vs the whole register file— despite the manual optimizations we apply to reduce the number of registers we replace (see Section 3.2.2). Moreover, the assembly generated by Clang causes more branch mispredictions leading to 70% longer execution time for `Cont (C)` compared to `Coro (C)`. Still, all `Coro` and `Cont` variants effectively hide cache misses for large array sizes.

Takeaway. As we also explain in Section 3.1, the cost of switching determines the performance of interleaved execution. A library-based context switch needs to be defensive and save all state when suspending execution. Contrary to that, a compiler-optimized suspension stores only the necessary state, which in binary search consists of two variables.

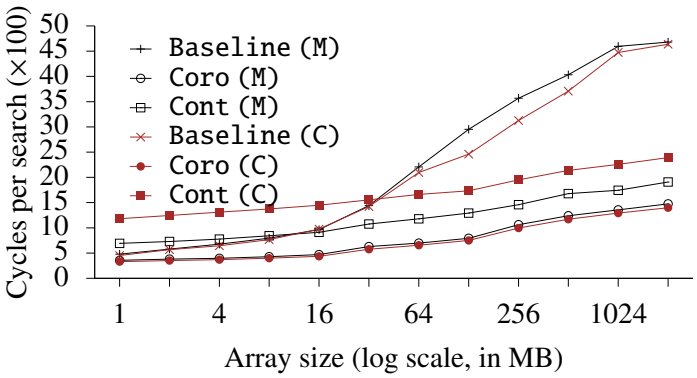


Figure 4.7 – Compiler-based (`Coro`) vs library-based (`Cont`) coroutines compiled with MSVC (`M`) and Clang (`C`). `Coro` performs better than `Cont` because it has more lightweight suspension/resumption.

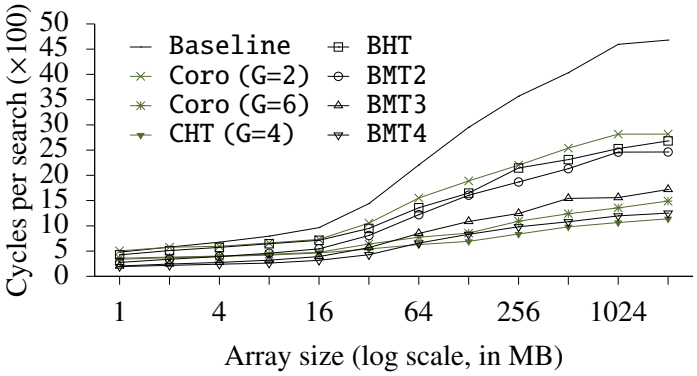


Figure 4.8 – Interleaving, hyperthreading, and multithreading. The combination of coroutines and hyperthreading (`CHT`) performs better than non-interleaved multithreaded execution on 4 cores (`BMT4`).

4.2.5 Hyperthreading and multithreading

Since interleaved execution assumes the existence of several independent tasks, one would first exploit this embarrassing parallelism with multithreaded execution. As explained in Section 2.1.1, modern Intel processors support hyperthreading, a form of interleaved execution implemented in hardware. Hyperthreading comes very close to the ideal form of interleaving that has instant context switches, and thus can hide arbitrarily short latencies, as long as the two hyperthreads have enough computation to execute; not the case for lookup code, which has only a few instructions between cache misses. On the other hand, coroutines support an arbitrary number of tasks, but have non-zero suspension/resumption cost.

To compare interleaving to hyperthreading and multithreading, we use multithreaded versions of `Baseline` and `Coro`; the former uses a simple parallel for loop, while the latter interleaves lookup execution per thread. Figure 4.8 depicts multithreaded performance (a) with hyperthreading

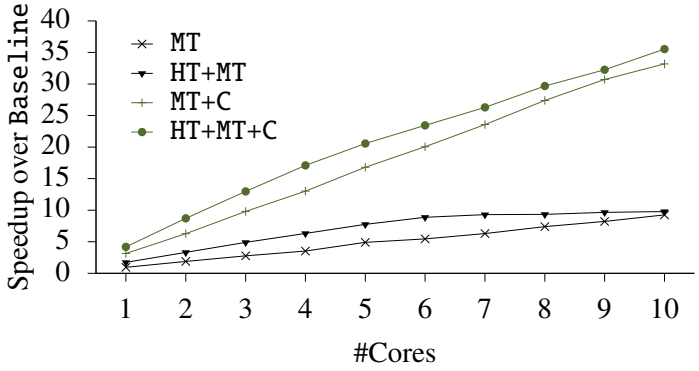


Figure 4.9 – Combining multithreading (MT), hyperthreading (HT) and interleaving with coroutines (C) performs best.

enabled (BHT for Baseline and CHT for Coro), on 2 logical cores sharing the same physical one; and (b) with hyperthreading disabled (BMT2-4 for Baseline), on 2 to 4 different physical cores. Note that we focus on the actual execution, not taking into account the non-trivial costs of creating and managing an OS thread. BHT is faster than Coro with two interleaved tasks (G=2), owing to the almost instant context switch between the two hyperthreads. Since the instructions of two binary search iterations can only partially hide memory latency, Coro with G=6, which is the optimal group size (see Section 4.2.3), outperforms not only hyperthreading but also real multithreaded execution with 2 and 3 cores, being somewhat slower than multithreaded execution with 4 physical cores. Nevertheless, combining coroutines with hyperthreading in CHT results in the best implementation of interleaved execution in a single core. Hyperthreading replaces a suspension/resumption pair with almost instant context switching, so more useful instructions can be executed. increasing the optimal group size per hyperthread from G=3, i.e., half the group size of the single-threaded execution, to G=4, This allows CHT to perform better than multithreaded execution on 4 physical cores.

Takeaway. For binary search, combining coroutines with hyperthreading is the most efficient way to interleave on a core, outperforming multithreaded execution with 4 cores.

4.2.6 The scalability of multithreaded interleaved execution

Interleaved execution improves single-thread performance by eliminating stalls. Here, we assess how this improvement scales with multiple threads. We use the multithreaded versions of Baseline and Coro described in Section 4.2.5, increasing the number of physical cores they run from 1 to 10 (the number of cores per socket) and we run with and without hyperthreading.

In Figure 4.9, we present the four aforementioned executions over Baseline for a 2 GB integer array. MT denotes multithreaded execution, HT means hyperthreading is enabled, and C implies interleaved execution with coroutines. As we observe also in Section 4.2.5, one physical core with hyperthreading achieves almost the same performance as two physical cores for non-interleaved

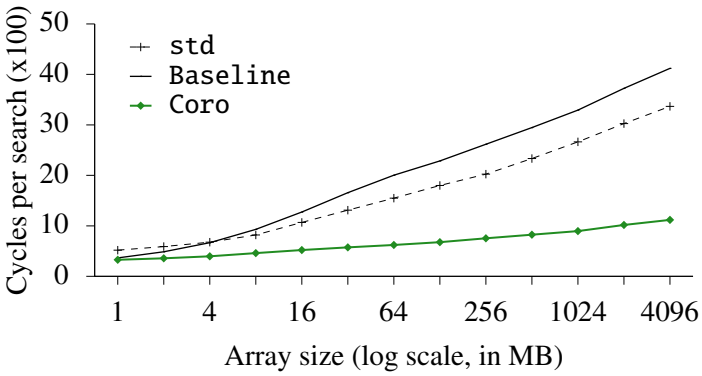


Figure 4.10 – 10K binary searches on IBM POWER9.

execution, while the benefit of hyperthreading, although visible, is much smaller for interleaved execution.

Moreover, `MT` scales linearly, whereas `MT+HT` achieves almost the same performance on 6 cores as `MT` on 10 cores; still, neither `MT` nor `MT+HT` gets 10× faster by fully utilizing the socket. The reason for this behavior is the increasing contention for LLC space: running more threads in parallel changes the overall memory access pattern and hurts temporal locality, increasing the data cache misses per thread. On the other hand, `MT+C` scales superlinearly and at 10 threads the speedup is 10.6× than `Coro`, because (a) interleaving decreases the dependency of lookup execution on array contents being in the cache or not, and (b) the multiple threads access the same page tables in parallel, increasing their temporal locality and thus reducing the latency of page walks. Finally, `MT+HT+C` with 20 hyperthreads is 11.4× faster than `Coro` (35.5× faster than `Baseline`) because hyperthreading hides stalls from the `Core` part of the breakdown in Figure 4.3.

Takeaway. Interleaving improves the scalability of a multithreaded implementation because it does not require the prefetched data to be in the cache and reduces the latency of address translation. Multithreaded interleaved execution on a 10-core socket with hyperthreading enabled shows a 35.5× speedup over single-threaded non-interleaved execution.

4.2.7 Interleaved execution on IBM POWER9

Our analysis so far has focused on Intel microarchitectures. However, interleaved execution applies to any microarchitecture that supports out-of-order execution, multiple outstanding memory requests, and software prefetching. In this subsection, we analyze the performance of interleaving with coroutines on our *Power* system (see Table 4.1), which features an IBM POWER9 processor that supports 2- and 4-way simultaneous multithreading (SMT2 and SMT4).

We run the `std`, `Baseline`, and `Coro` implementations of our binary search microbenchmark with **`int`** arrays. Figure 4.10 depicts the best runtimes, i.e., with the optimal group sizes, for

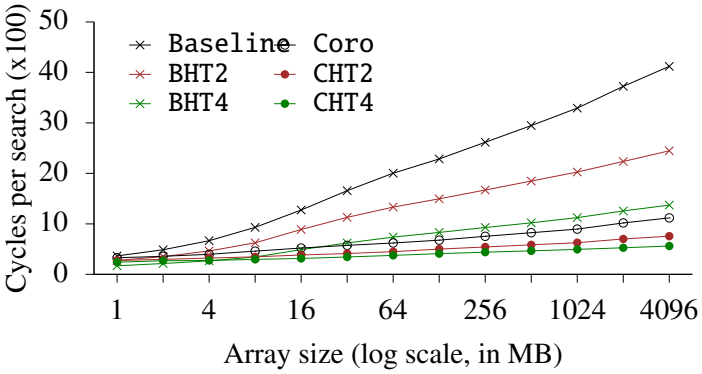


Figure 4.11 – 10K binary searches on IBM POWER9 single-threaded, with SMT2, and with SMT4.

each implementation for array sizes that range between 1 MB and 4 GB increased by 1 kB to reduce TBL conflicts and thereby TLB misses. The runtimes of `std` and `Baseline` increase significantly in the area of 10 MB, i.e., the size of the L3 slice that corresponds to the executing core. Accesses that miss in the corresponding slice are served with increasing latency either over the internal interconnect by other L3 slices which assume the role of victim caches, or by the memory controller which fetches the data from DRAM [21]. On the other hand, `Coro` exhibits runtimes that increase almost linearly, without visible L3 or TLB misses.

To compare with simultaneous multithreading (SMT), we use the multithreaded versions of `Baseline` and `Coro` described in Section 4.2.5. We run each implementation without SMT (`Baseline`, `Coro`), with 2-way SMT (`BHT2`, `CHT2`), and with 4-way SMT (`BHT4`, `CHT4`). The corresponding runtimes are depicted in Figure 4.11. Similarly to Figure 4.8, we observe that simultaneous multithreading almost doubles `Baseline` performance for array sizes where execution is dominated by cache misses, exploiting memory stalls for executing instructions from the other threads. Again, `Coro` also benefits from SMT, which not only replaces suspension/resumption pairs with instant context switch among SMT contexts, but also hides latencies that are too small for coroutine suspension/resumption to make sense.

Since SMT with enough hardware contexts outperforms coroutines when switching, one could argue that if we simply increase the number of SMT contexts, coroutines are an unnecessary complexity. However, we should keep in mind that, for general-purpose processors, adding hardware contexts might not be the best use of the available silicon, while no given number of contexts is guaranteed to suffice for future applications and the longer latencies of new memory technologies that have yet to come. A software-based interleaving technique, like coroutines, can adapt to new demands more flexibly than a hardware only technique.

Takeaway. Simultaneous multithreading (SMT) is the ideal implementation of interleaved execution in case the available hardware contexts provide enough work for the targeted latency. For long latencies, the combination of coroutines and SMT offers optimal performance.

4.3 Analytics and transactions

As already mentioned, a key differentiator between interleaving with coroutines and prior software-based interleaving techniques is the applicability of our technique to large codebases. In this section, we provide evidence that coroutines deliver on the promises of interleaved execution on two representative use cases from analytics and transactions. The index join we present first is a fundamental database operation that, in the form of hash join, consitutes the main hotspot in analytical workloads [30], whereas the `GET` and `PUT` we interleave next are key transactional operations for key-value stores.

4.3.1 Index join

When choosing the physical operator for an equi-join between two relations, A and B, a query optimizer checks if either of them has an index on the join attribute. Such an index, e.g., on A, can be used in an index join algorithm that scans B, looking up A's index to retrieve the matching records. In the absence of an index, the optimizer may decide to perform a hash join, i.e., build a hash index on one relation and probe it while scanning the other. Whether using an existing index or building a transient one, these join algorithms dominate analytical workloads.

Both the index join, as well as the probe phase of a hash join, are essentially a sequence of independent index lookups, like B^+-tree traversals, hash table lookups, or binary searches on sorted arrays. This sequence can be trivially expressed as a loop, which performs well so long as the index involved fits in the processor cache. Otherwise, the irregular memory access pattern of the lookup code leads to poor performance and execution times that are dominated by memory stalls due to data cache misses.

Without loss of generality, we focus on `IN`-predicate queries, a case of an index join found in main memory column stores that employ dictionary encoding. First, we elaborate on how column stores use dictionary encoding and explain why `IN`-predicate queries [41], i.e., dictionary lookups performed in bulk, are instances of index joins. We then quantify the negative effect of main memory accesses on dictionary lookups when the dictionary outsizes the last level cache, and present how to eliminate these effects with interleaved execution.

Dictionary encoding in main memory columm stores

Dictionary encoding is a common compression method for main memory column stores [16, 33, 35, 46, 50] that maps the value domain of one or more columns to a contiguous integer range [9, 11, 16, 22, 25, 34]. This mapping replaces column values with unique integer *codes* and is stored in a separate *dictionary*, a data structure that supports two access methods, `extract` and `locate`; `extract` returns the *value* for a *code*, whereas `locate` returns the *code* for a *value* that exists in the dictionary, or a special *code* that denotes absence. The resulting vector of codes and the dictionary constitute the encoded column representation. The code vector is usually smaller

than the original column, reflecting the smaller representation of codes, whereas the dictionary size is determined by the value domain, which can comprise from few to billions of distinct values as encountered by database vendors in customer workloads [44].

Here, we focus on the column store of SAP HANA, introduced in Chapter 2.2, As explained, the SAP HANA column store has two fragments per column: the read-optimized *Main* and the update-friendly *Delta*. A *Main* dictionary is a sorted array of the domain values, where the array positions correspond to codes, similarly to [16, 33, 50]. Hence, `extract` is a simple array lookup, whereas `locate` is a binary search on the array contents for the appropriate array position. On the other hand, *Delta* dictionaries are implemented as unsorted arrays indexed by a cache-conscious B^+-tree (CSB^+-tree) [51]; `extract` is again an array lookup, but `locate` is now an index lookup on the CSB^+-tree.

Sorted or not, a dictionary array can be considered a relation $D(code, value)$ that is indexed on both attributes: codes are encoded as array indices, whereas values can be retrieved through binary search or index lookup, respectively in the sorted and the unsorted case; here, we focus on the *value* index. Since a sequence of values is also a relation $S(value)$, every value lookup from a column involves a join $S \bowtie D$, which uses the dictionary index in case $|S| << |D|$. Such index joins dominate the `IN`-predicate queries that we discuss next.

`IN` predicates and their performance

`IN` predicates [41] are commonly used in ETL processes to extract interesting items from a table. An `IN` predicate is encountered in the WHERE clause of a query, introducing a subquery or an explicit list of values that the referenced column must match. Listing 4.1 illustrates an `IN` predicate from Q8 of TPC-DS [3], which extracts all zip codes from the *customer_address* table that belong in a specified list of 400 predicate values.

Listing 4.1 – `IN` predicate excerpt from TPC-DS Q8.

```
1 SELECT substr(ca_zip,1,5) ca_zip FROM customer_address
2 WHERE substr(ca_zip,1,5) IN ('24128', ..., '35576')
```

When `IN`-predicate queries are executed on dictionary-encoded data, the predicate values need to be encoded before the corresponding rows can be searched in the code vector. This encoding comprises a sequence of `locate` operations, which, as already described, can be viewed as an index join.

Like all index lookups, dictionary lookups become disproportionally expensive when the dictionary outsizes the last level cache of the processor. Figure 4.12 illustrates this disproportionality in the response time of the query `SELECT COUNT(*) FROM TBL WHERE COL IN LIST` running on a prototype based on SAP HANA, on our *Windows* system (see Table 4.1). We vary the size of `TBL` from 1 MB to 2 GB and we construct the `LIST` with 10K random predicate values. For both Main and Delta, we observe a runtime increase as the dictionaries outgrow the last level cache

(25 MB). To identify what causes the evident runtime increase, we profile the query execution for the smallest and largest dictionary sizes, i.e., 1 MB and 2 GB. The resulting list of hotspots in Table 4.3 identifies dictionary lookups (`locate`) as the main execution component.

Table 4.3 – Execution details of `locate`.

	Main		Delta	
	1 MB	2 GB	1 MB	2 GB
Runtime %	21.4	65.7	34.3	78.8
Cycles per Instruction	0.9	6.3	0.7	4.2

In the 1 MB case, `locate` contributes 21.4% (34.3%) of the total execution time for Main (Delta), a number which surges to 65.7% (78.8%) for the 2 GB case. These surges can be attributed to the 7× (6×) increase in the *cycles per instruction (CPI)* ratio between the 1 MB and the 2 GB cases.

Table 4.4 – Pipeline slot breakdown for `locate`.

	Main		Delta	
	1 MB	2 GB	1 MB	2 GB
Front-End	10.4%	3.5%	0.7%	0.2%
Bad speculation	43.3%	26.1%	0.0%	0.3%
Memory	2.8%	46.0%	30.8%	85.9%
Core	16.4%	20.5%	28.9%	7.3%
Retiring	27.0%	3.9%	40.0%	6.3%

In Table 4.4, we present the TMAM breakdown of `locate`'s execution retrieved from a profiling session of 60 seconds. In the 2 GB case, memory stalls account for 46.0% and 85.9% of the pipeline slots, respectively for Main and Delta, while they are relatively less prominent in the 1 MB case. The stalls occur from random accesses to the dictionary array for Main and to the index nodes for Delta. The 1 MB dictionary fits in the last level cache, so accesses to it incur small latencies [23] that do not dominate execution. For the 2 GB dictionary, however, only the first few binary search iterations (Main) or tree levels (Delta) are expected to be in a warmed-up cache, while the rest of the data requests lead to main memory accesses that translate into stalls.

Furthermore, Main execution exhibits a significant of bad speculation that arise from the binary search involved in Main dictionary lookups. As we have explained in Section 2.4, the decision on which subarray to search in the next iteration is inherently unpredictable, challenging branch predictors of the processor. The Delta uses such a predicated implementation for binary search in the tree nodes, so no pipeline slots get wasted due to bad speculation.

Moreover, we believe bad speculation is also the main reason for the front-end stalls we observe in Main, given that these stalls are negligible in Delta, and do not appear in the non-speculative microbenchmarks we study in Section 4.2. Finally, the core fraction in both Main and Delta comprises stalls due to unavailable execution units.

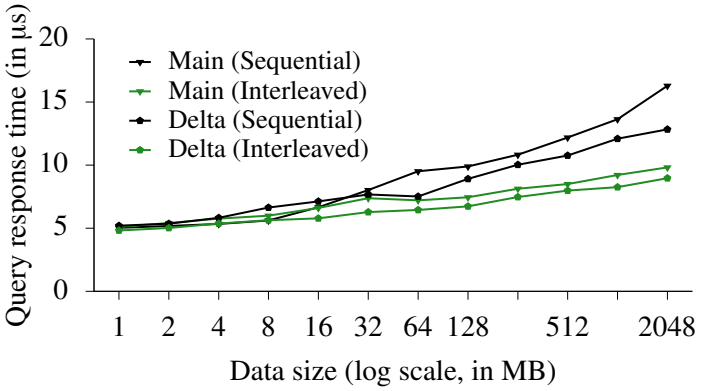


Figure 4.12 – `IN`-predicate queries executed on SAP HANA with increasing dictionary sizes. The original implementations incur evident runtime increases when the dictionaries do not fit in the cache (25 MB)**, due to expensive main memory accesses. Interleaved implementations exhibit robust performance despite accessing main memory.**

`IN` predicates with interleaving

To interleave the execution of the `IN`-predicate query we consider, we convert into `resumable` coroutines the involved dictionary lookups in both the Main (binary search) and Delta (CSB$^+$-tree lookup). Since `resumable` is not composable, we manually inline nested function calls into the main lookup functions, which we then convert into coroutines. The Main implementation is straightforward, but the Delta one differs from the CSB$^+$-tree described in Section 3.2.2: leaf nodes contain codes instead of values, so comparisons against the values of a leaf node incur accesses to the dictionary array to retrieve the actual values, adding an extra suspension point on the access to the dictionary array.

We evaluate the two implementations in the execution of `IN`-predicate queries with 1K, 5K, 10K, and 50K predicate values over `INTEGER` arrays with distinct values and dictionaries whose size ranges in 1 MB–2 GB. Figure 4.12 depicts the query response time for the 10K case. In the other cases, lookups account for a smaller part of total execution and the benefit of interleaving is less evident. For dictionaries larger than 16 MB, interleaving reduces the Main runtime for dictionary sizes larger than the cache, from 9% at 32 MB to 40% at 2 GB, corroborating the microbenchmark results. On the other hand, the Delta runtime is reduced for all dictionary sizes, from 10% at 1 MB to 30% at 2 GB; this can be explained by the memory stalls that Delta exhibits in the 1 MB case.

Takeaway. For both Main and Delta, interleaved execution hides the cache misses of dictionary lookups that appear as the dictionary grows larger than the last level cache. As a result, `IN`-predicate queries become robust to dictionary size.

4.3.2 Transactions

Transactional workloads also make heavy use of index structures and are thus susceptible to memory stalls. Contrary to analytical queries, transactions typically exhibit low intra-transaction parallellism, operating on one or few records each. At the same time, many transactions run concurrently, and this inter-transaction parallelism can be leveraged to hide memory access latency through interleaved execution and thereby increase transaction throughput.

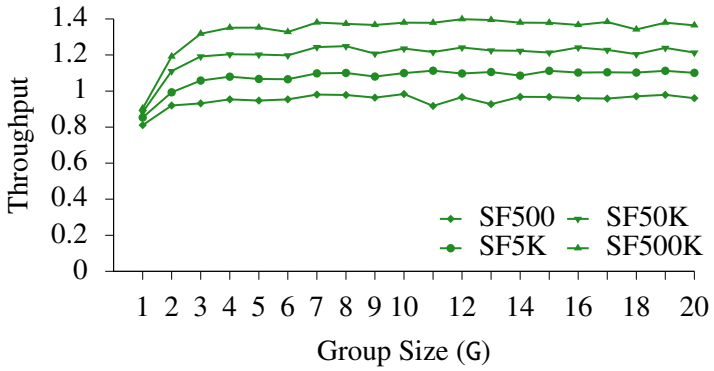
Here, we study the effects of transaction interleaving on Silo [58], an in-memory transactional engine that implements optimistic concurrency control (OCC) and employs the cache-efficient Masstree [38] for its indexes. Interleaving of Masstree operations has been studied by Jonathan et al. [24] in isolation, whereas we focus on the behavior of a complete transactional system that happens to use the Masstree. Moreover, we are interested not only in the throughput of the interleaved transactions but also their latency, which should satisfy given service quality guarantees.

We interleave the `GET` and `PUT` operations of the *Yahoo! Cloud Serving Benchmark* (YCSB) using `task` coroutines (Section 3.2.4). We introduce prefetches and suspensions to the traversals both operations perform on the Masstree, but we execute the tree modifications for the `PUT` operation "atomically", i.e., without any suspensions; the nodes to modify are already fetched in cache during the traversal, so there is no memory latency to hide. Since the YCSB driver in Silo has multiple worker threads, each of which generates and executes operations one after the other in a loop, we interleave by replacing the loop with our `for_each`. These changes do not alter the transactional guarantees of system, since multithreading subsumes multitasking in one thread.

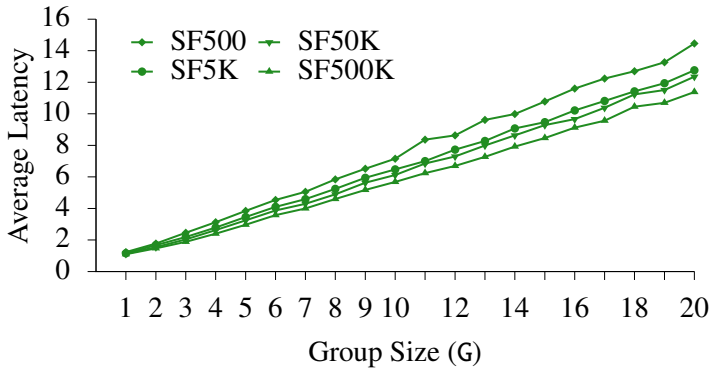
Table 4.5 – Performance metrics of non-interleaved YSCB execution on Silo.

Scale Factor	Throughput (in 10^6 ops/s)	Average Latency (in µs)	99th Percentile Latency (in µs)
500	1.8	0.5	1.0
5K	1.4	0.7	1.2
50K	1.0	0.9	1.6
500K	0.8	1.2	1.9

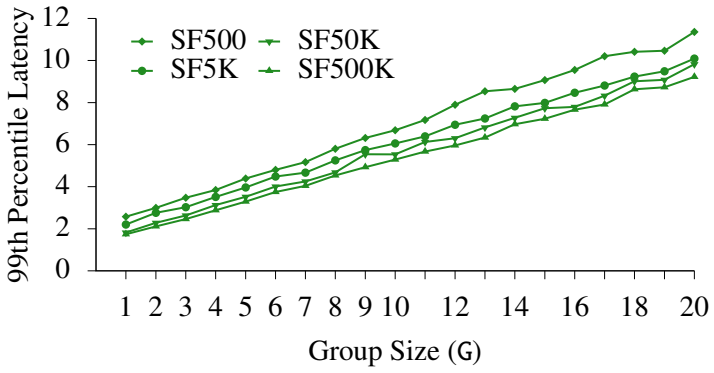
To evaluate Silo with interleaving, we use the Silo variation of the YCSB A workload [58], comprising 80% `GET` and 20% `PUT` operations. We execute this workload for 30 s with scale factors 500 (5 MB), 5K (50 MB), 50K (500 MB), and 500K (5 GB) running single-threaded on the *Linux* system (see Table 4.1). In Figure 4.13, we report the throughput, the average latency, and the 99th percentile latency of YCSB on Silo with interleaved execution; each of these three metrics is normalized to the corresponding measurements with non-interleaved execution, which are presented in Table 4.5). Note that operations in Silo are not queued, so latency does not include wait time—each new operation is created and submitted for execution only after the previous operation finishes.



(a) Throughput.



(b) Average Latency.



(c) 99th Percentile Latency.

Figure 4.13 – YCSB (80% `GET`, 20% `PUT`) **performance on Silo for scale factors 500, 5K, 50K, and 500K. The depicted throughput, average latency, 99th percentile latency measurements correspond to interleaved execution with different group sizes `G` and are normalized to non-interleaved execution. Larger scale factors imply more cache misses, hence more benefit from interleaving. In addition, relatively small group sizes offer near maximum throughput without increasing operation latency significantly.**

For scale factor 500, interleaving penalizes throughput, while leading to throughput increases up to 1.11×, 1.25×, and 1.40× respectively for scale factors 5K, 50K, and 500K. These maximum speedups correspond to group sizes 11, 8, and 12, for which the average and 99th percentile latency are 4×–7× longer than without interleaving. We observe that latency increases linearly with the group size, a side effect of the varying instruction distance between subsequent suspensions: a dynamic number of instructions need to be executed to modify the tree in a `PUT` operation, whereas going from one node to the appropriate child when traversing the tree involves a fixed number of instructions. To guarantee that memory latency is always hidden, we need to always account for the smallest possible instruction distance, which leads to high group sizes. Nevertheless, we also observe that throughput quickly becomes insensitive to the group size, with even small group sizes yielding near optimal speedup. 94% of the maximum throughput is already attained at group size 3, which means we can trade 6% throughput increase for roughly 2× better latency.

Takeaway. In transactional workloads, performance is determined not only from the aggregate throughput of the system but also from the latency of each individual transaction. This means, we can interleave to increase throughput, so long as transaction latency does not violate service levels guarantees.

4.4 DRAM vs NVM

Everything described so far about interleaving with coroutines applies to any use case with data- or task-level parallelism in which memory latency is exposed to runtime. What changes by placing data and/or working set to NVM, is the 4× latency increase and the 10× bandwidth reduction (see Section 2.1.2). To hide the higher latency, more instructions are needed. These instructions can be found by increasing the group size, i.e., the number of interleaved coroutines. However, the maximum memory-level parallelism (MLP) supported by current Intel processors is 10 in-flight memory requests per core [23]. This limit means higher group sizes offer little to no benefit in case all prefetches go to main memory. Furthermore, scaling interleaved execution to multiple cores is bound to reach the bandwidth limit of NVM—even in the absence of scans. Under these constrains, interleaving converts execution from latency- to MLP- or bandwidth-bound.

In this section, we show that interleaving with coroutines drastically narrows the performance gap between NVM and DRAM for latency-bound operations despite the significant latency difference. To that end, we interleave two types of operations: (a) lookups on a single index, and (b) lookups on different indexes. In Section 4.4.1, we compare the single-thread runtime and the scalability of interleaved and non-interleaved binary searches having sorted arrays on DRAM and NVM. Then, we assess the gap between DRAM and NVM and the effect of interleaving on the end-to-end execution of two queries running on a prototype based on SAP HANA: a query with an `IN`-predicate (Section 4.4.2), and a simple SELECT(*) query (Section 4.4.3). For our experiments, we compile our code with Clang 7.0 and use the *NVM* system, which is equipped with an Intel Xeon Platinum 8280L processor, 6 DRAM DIMMs, and 6 Optane DC PMMs per socket (see Table 4.2) and SUSE Linux Enterprise Server 15 (Linux kernel 4.12).

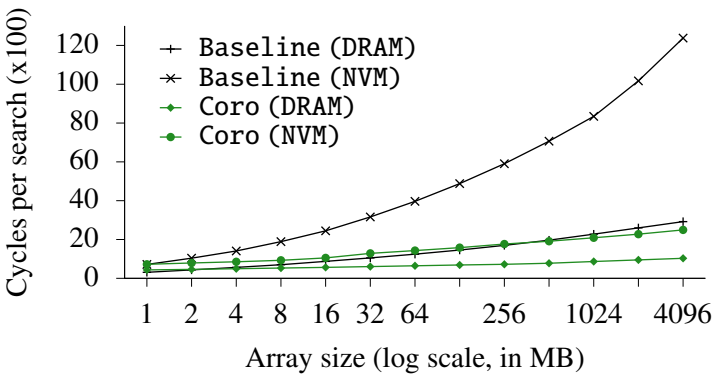


Figure 4.14 – Binary search performance on DRAM vs NVM.

4.4.1 Microbenchmarks

We use the binary search microbenchmark described in Section 4.2, as representative of a simple index join. We use two implementations that look for a list of values in a sorted array of 32-bit signed (`int32_t`) integers. We use the `Baseline` and `Coro` implementations, as described in Section 4.2, respectively for sequential and interleaved execution. The sorted array sizes range between 1 MB–4 GB, increased by 1 kB to circumvent alignment issues that hurt TLB performance (see Section 4.2.2). For the lookup list we select 10K values from the sorted array using `std::mt19937` with a fixed seed of value `0` and `std::uniform_int_distribution`. We run our experiments with the array placed first in DRAM and then in NVM.

Increasing the array size. Figure 4.14 depicts the cycles per binary search for all array sizes and for the optimal group sizes, i.e., the one for which `Coro` has the best performance. First, for `Baseline`, we notice the performance gap between `DRAM` and `NVM`, a gap that widens from 2.3× to 4.2× as the number of cache misses increases with the array size. For `Coro`, the performance difference ranges from 1.7× at 1 MB to 2.4× at 4 GB. The corresponding optimal group sizes are in the range 21–44 for DRAM and 22–49 for NVM. We observe values in the upper part of these ranges with small arrays and, conversely, values in the lower part with large arrays. For small arrays most accesses hit in the cache, allowing to interleave more than 10 coroutines at a time and thereby hide most of the latency of the few main memory accesses. As the number of main memory accesses increases, the hardware-imposed limit on in-flight memory requests becomes a bottleneck and decreases the optimal group size; still, the group size is above 20 because array values of the first binary search iterations fit in the cache, so the respective prefetches finish fast, allowing new prefetches to be issued[3]. The hardware-imposed limit explains also why interleaved execution does not eliminate the difference: the 7 assembly instructions that exist between subsequent array lookups in a binary search are inadequate for eliminating the latency gap given the group size limit. Still, interleaving improves NVM performance by up to 5× for

[3]Note that the difference to the optimal group size 10 reported in Section 4.2 is due to the higher suspension/resumption overhead of the code generated by the Microsoft Visual C++ compiler. Compiler support matters and has been improving over the years.

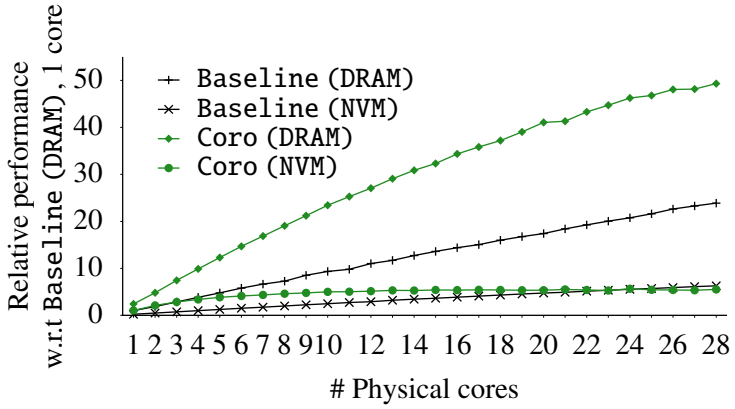


Figure 4.15 – Scalability of binary search for a 2 GB sorted array.

4 GB arrays, reaching runtimes similar to `Baseline` on DRAM.

Scaling up to 28 cores. We assess the scalability of inteleaved execution on NVM by running the multithreaded versions of `Baseline` and `Coro` on one socket using 1–28 physical cores. Figure 4.15 shows the speedups over `Baseline (DRAM)` with 1 core. Contrary to the latency-bound `Baseline`, that scales well on both DRAM and NVM, `Coro` on NVM becomes bandwidth-bound with 18 cores, reaching a maximum speedup of 8× and incurs a slight slowdown as the number of cores used increases further to 28 due to increasing resource contention on the socket; on DRAM performance scales linearly up to 8 cores and then sublinearly due to again resource contentions, reaching a maximum speedup of 50× at 28 cores.

4.4.2 Index join

In addition to the binary search microbenchmark, we evaluate the respective effects of NVM latency and interleaved execution on the semijoin of `IN`-predicate clauses involving dictionary-encoded columns. Similarly to Section 4.3.1, we run the following `IN`-predicate query on our prototype: `SELECT COUNT(*) FROM TBL WHERE COL IN` *`LIST`*. However, since our focus here is the difference between NVM and DRAM, we use only a `TBL` size with 100M rows that is placed first in DRAM and then in NVM, *`LIST`* contains 10K randomly generated values, and for `COL` we analyze two datatypes: either `INTEGER` or `VARCHAR(15)`. Note that we use the composable `task` coroutine type, converting the call stacks of dictionary lookups into coroutine chains.

In Figure 4.16, we report the average runtimes of 1000 executions per datatype. For the `Baseline` implementations and both datatypes, NVM runtime is 2× the DRAM runtime, while coroutines (`Coro`) reduce the difference to 30%—the difference is not eliminated because the index lookup is a variant of binary search with additional compression-related indirection and thus lacks the amount of work required to completely hide NVM latency. Still, `Coro` performs better than `Baseline`.

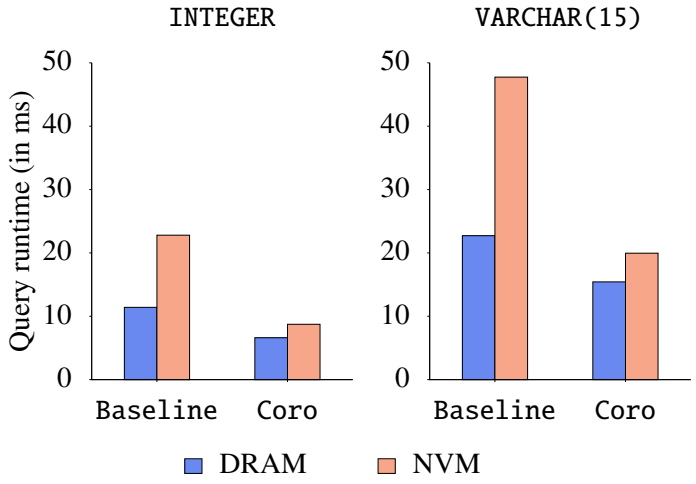


Figure 4.16 – `IN`-predicate query on tables with 100M rows.

4.4.3 Tuple reconstruction

To evaluate the effect of interleaved execution on tuple reconstruction, we use the query `SELECT(*) FROM TBL WHERE KEY=X`, where `KEY` has unique values.

Ranging column count. We assess the worst-case slowdown due to NVM by executing the query on a `TBL` with 1M rows and 10, 100, and 1000 `INTEGER` columns with unique values each. Retrieving the value from each `INTEGER` column incurs two cache misses—one when accessing the data vector and another one when accessing the dictionary—that dominate execution. In Figure 4.17, we depict the average runtime of 10K query executions, having placed `TBL` on DRAM and NVM, with (`Coro`) and without (`Baseline`) interleaving. We observe the runtime gap between `Baseline` and `Coro` widen from 6% to 150%, as the work in the reconstruction loop increases along with the column count; for `Coro`, the gap is 1%–8% and not eliminated due to lack of instructions. More interestingly, for 1000 columns, `Coro` on NVM is 30% faster than `Baseline` on DRAM.

Mixed datatypes. We show interleaving of three distinct codepaths by executing the query on a table `TBL` of 1000 columns, of which one third have datatype `INTEGER`, another third `DECIMAL(10,2)`, and the last third `VARCHAR(50)`. Moreover, we highlight the effect of processor frequency on execution by running the experiment with frequency scaling enabled (`Turbo On`) and disabled (`Turbo Off`). In Figure 4.18, we see interleaved execution reduces the performance gap from 123% to 37% for `Turbo On`. Lookups in `DECIMAL` and `VARCHAR` columns involve more instructions compared to `INTEGER` columns, explaining the higher runtime compared to the 1000 column case in Figure 4.17. However, the dictionaries of `VARCHAR` columns use prefix compression, which means dictionary lookups first retrieve the prefix and then the rest of the

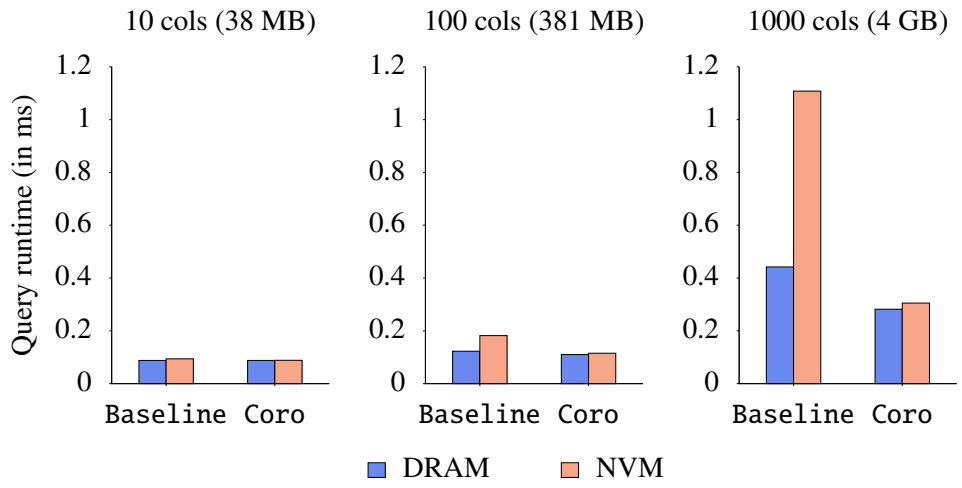


Figure 4.17 – 'SELECT (*)' query on INTEGER tables with 1M rows and varying column counts.

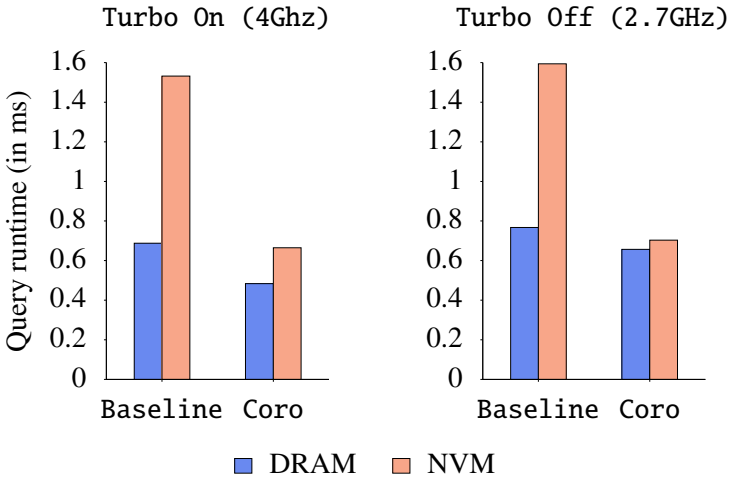


Figure 4.18 – 'SELECT (*)' query on a table with 1M rows and 1000 columns (of INTEGER, DECIMAL(10,2), and VARCHAR(50) type), with and without frequency scaling.

value with an additional indirection; contrary to the latter access that is a guaranteed cache miss for different values, the prefix is found in cache often enough to not justify a blind suspension for DRAM accesses, leading to the 37% gap for `Coro`—hardware support for cache content introspection would be beneficial for this case. Again, `Coro` runtime on `NVM` is faster than `Baseline` on `DRAM`. Finally, for `Turbo Off`, we see the gap reduces from 106% to 7% due to the lower frequency: the work needed to hide a given latency is less at 2.7 GHz than at 4 GHz.

Takeaway. Non-volatile memory exhibits a higher access latency that penalizes latency-bound operations. Interleaved execution eliminates this penalty, reducing the gap between NVM and DRAM by more than 65%, and exhibits runtimes better than even the original ones on DRAM.

4.5 Summary

Interleaved execution eliminates stalls due to main memory access, and transforms latency-bound code into compute- or bandwidth-bound, depending on which hardware resource is exhausted first. Given enough work, execution becomes compute-bound, so only more computation units in the processor core can increase performance. Otherwise, either the buffers that keep track of the outstanding memory request become exhausted at some level of the memory hierarchy, or the installed memory modules do not provide enough bandwidth. This is an impressive phenomenon, typical for sequential and not random access patterns.

Our coroutine-based technique constitutes a flexible software implementation of interleaved execution. Being a software technique without particural hardware support, interleaving with coroutines provides task switching that is slower than simultaneous multithreading, which is the hardware form of interleaving. However, our technique works, given the proper compiler support, on any platform that supports software prefetching or out-of-order execution. And, given hardware with simultaneous multithreading, latency-bound code performs best when combining software with hardware interleaving.

Finally, the benefits of interleaving can be reaped even in complex systems. The analytical and transactional use cases we interleave provide evidence that coroutines not only are indeed a practical way to implement code but also effectively hide the latency of both DRAM and NVM.

5 Conclusions and Future Directions

Main memory access is inherent to the von Neumann architecture and its latency poses a fundamental performance challenge for database systems and data-intensive applications in general. Locality of reference allows for drastic reduction of data fetch through caching, yet main memory accesses are inevitable. That said, memory access does not necessarily imply problems: given enough work to execute while fetching data, memory latency can be hidden. However, despite decades of work on latency hiding by computer architects, compiler writers, and systems builders, modern database systems still waste large numbers of CPU cycles in latency-bound operations, waiting for data to be fetched from memory.

This thesis demonstrates that cooperative multitasking is a general approach to hide the latency of unavoidable memory accesses in operations comprised of latency-bound tasks that do not depend on each other. By interleaving task execution upon memory operations that miss the cache, we can hide latency given enough independent work in the form of parallel tasks, sufficient memory parallelism, as well as a lightweight mechanism to switch among tasks. In this final chapter, we summarize our work and discuss the application scope, requirements, and limitations of interleaving with coroutines, ultimately presenting directions for future work.

5.1 What we did

The profound inefficiency of index join execution on modern database systems has motivated our study of latency-bound loops that consist of independent tasks. While such loops magnify the problem of exposed memory latency, they also hold the key to mitigating latency: task parallelism. The old idea of cooperative multitasking converts task parallelism into intruction-level parallelism, which out-of-order processors can leverage to avoid execution stalls, improving efficiency and thereby performance.

Despite the simplicity of the idea, cooperative multitasking has only been fully implemented in exotic processor architectures, or in limited form as simultaneous multithreading. On the software side, the compiler optimizations that essentially multitask has restricted application scope,

whereas manual code rewrites to put instructions in the optimal order results in incomprehensible code that nobody wants to maintain. As a result, memory latency penalizes the performance of computer systems to this day.

With this thesis, we have established interleaved execution as a general-purpose way to hide latency. First, we have modeled interleaved execution, augmenting the literature with the analogue to Amdahl's law for cooperative multitasking. Our model captures the intuition that, to avoid stalls due to main memory access, we need sufficient work to execute and this work can be found in one or more other tasks. Furthermore, the maximum performance improvement to expect from multitasking corresponds to an execution without stalls. In case of tasks with similar parameters, our model determines the optimal number of coroutines necessary to hide latency and estimates the performance benefits for any given interleaving technique. More importantly, we have introduced a practical way to implement this form of cooperative multitasking in software, finding that coroutines constitute the right abstraction to separate code logic from code execution and thereby avoid maintainability problems. Thanks to this separation, interleaving with coroutines applies to any group of parallel tasks with cache misses, can hide longer latencies than hardware-based interleaving techniques, and is practical to implement in production codebases, requiring only few focused changes that retain code structure.

Our evaluation provides evidence for these three characteristics, while independent follow-up works have corroborated our findings in a wider range of data structures, verifying the value of our technique. Based on our findings, we can provide a rule of thumb for when to interleave a parallel loop. For simplicity, we make the following assumptions: (a) no other processes run on the sockets we use, and (b) the conditional branches in the code are not mispredicted. Using the microarchitectural profile of any given parallel loop, we can decide whether to interleave or not based on the percentage of stalls:

$\%_{stall} << 50\%$: Eliminating memory stalls will have minimal to no positive performance impact, so the only ways to improve performance are vectorization (if applicable) and multithreading.

$\%_{stall} \approx 50\%$: From Figure 3.2, we see that only a very lightweight interleaving technique can meaningfully eliminate stalls and achieve the $2\times$ speedup estimated by our performance model, so use simultaneous multithreading to improve single-thread performance, along with vectorization and multithreading.

$\%_{stall} >> 50\%$: In this case, simultaneous multithreading is insufficient, calling for software-based interleaving. If available, use coroutine-based coroutines to achieve the maximum speedup; otherwise, more heavyweight techniques can be used to hide the longer latencies of disk and network I/O. With most stalls minimized, consider vectorization and multithreading as above.

5.2 Discussion and future directions

Interleaving with coroutines has a wide application scope, is easy to implement, and offers remarkable speedups. The programmers needs to first identify the independent tasks and in each task the loads that cause cache misses—in this thesis, we relied on offline profiling to identify the cache misses. Then, the programmer need to convert the functions of each task into coroutines, replace each memory load causing a cache miss with our `load` helper function, which prefetches the specified memory address, suspend the execution of the current coroutine, and loads the data upon resumption. Finally, the resulting coroutine chains need to be executed using an interleaved scheduler similar to the one we proposed.

With interleaving, random access no longer implies memory stalls, enabling us to rethink our data structures—with more indirection and less data preprocessing. Further, opportunities arise for hardware-software co-design: Can we ask the cache if a piece of data is there, so that we suspend only when necessary? Even better, can we eliminate the cost of coroutine suspension/resumption? Here, we discuss the application scope and requirements of interleaved execution, as well as desirable compiler and hardware support to lift current limitations.

Application scope and requirements

Interleaving with coroutines applies to any group of latency-bound parallel tasks, since it depends not on the code logic, but on the ability to switch between independent tasks. As we have demonstrated, this approach can effectively hide memory latency on the two distinct codepaths of binary search and CSB$^+$-tree traversal, while it also applies to all use cases covered by Kocberber et al. [29], i.e., the build and probe phases of a hash join, a group-by operator, bulk lookups in binary search trees and bulk insertions in lock-free skip-lists. Furthermore, Jonathan et al. [24] have shown that our technique is also effective in hiding not only the memory latency in bulk operations with even more complex data structures, like the Masstree and the Bw-tree, but also the remote access latency in NUMA environments.

Interleaved execution and multitasking in general are *universal latency-hiding techniques* [36] and, as such, they can be employed in any data- or task-parallel operation, like sorting, or operations related to the state management of the lock and the transaction manager. However, if the amount of computation and stalls varies among tasks, we have to consider statistical distributions of task parameters instead of concrete values, resulting in a statistical performance model. Moreover, since coroutines allow task execution to progress asynchronously, even different operations on multiple data-structures can, in principle, be interleaved, from simple lookups to whole transactional queries. The simple `GET/PUT` transactional scenario we studied indicates that individual task latency poses a restriction—unlike in a join—, while care should be taken to not increase instruction cache misses as well as to avoid deadlocks when combining cooperative multitasking with thread synchronization mechanisms. Finally, the identification of cache misses and the statical introduction of suspensions as described above becomes tricky

as the same load instruction might only occasionally cause a cache miss; for such cases, extra hardware support is required as described in Section 5.2.

On the other hand, the interleaved tasks need not be all latency-bound; compute-bound tasks can also be interleaved by introducing additional suspensions—not upon cache misses, but at such places to ensure the compute-bound tasks do not monopolize execution. Given compute-bound tasks, any latency can be hidden even with small group sizes, avoiding bandwidth problems such as the ones we observe in binary searches when combining interleaving with multithreading. Therefore, overlapping memory- with compute-bound operations is a promising direction to efficiently use the available computate resources.

Since the problem of memory latency transcendes databases, workloads like graph processing and machine learning could also benefit from interleaving with coroutines, so long as the following four requirements are satisfied: there is sufficient work to execute while fetching data, the latency to hide is that of a DRAM access or longer, the compiler used has a coroutine implementation similar to that of C++, and the processor supports software prefetching. Modern processors provide prefetch instructions, whereas at the time of writing already two major C++support coroutines, which means interleaving with coroutines can be applied todays to task-parallel operations that suffer from main memory accesses. All in all, the application potential of interleaved execution is vast and calls for further study.

Desirable compiler and hardware support

As we have mentioned, interleaving with coroutines is effective thanks to the low switch overhead of C++ coroutines. The compiler support for C++ coroutines is still maturing, and is not specific to interleaved execution, but covers all coroutine use cases. This means that interleaved coroutines are always converted into state machines, even in cases where the coupling of group prefetching enables better performance. Kiriansky et al. [28] show that a compiler can generate a state machine or *GP*-like assembly based on hints provided by the programmer; such hints allow to express interleaving with coroutines and still get assembly code that performs best in all cases.

Another negative aspect of C++ coroutines is the limited inlining performed by current compilers. When we want to suspend a deeply nested function, we have to convert the call stack into a coroutine chain, where each coroutine is allocated on the heap and has individually managed lifetime, both of which imply overhead. An efficient implementation relies not only on avoiding heap allocations per coroutine, but also on inlining coroutines to their parents, so that there are few coroutine objects to manage. At the time of writing, this coroutine elision is manual, while allocation avoidance is a fragile compiler optimization. Stackful coroutines such as the studied `Boost.Context` avoid these problems altogether by using a stack instead of translating functions into state machines with heap allocated state; however, instead of just a library, stackful coroutines should be a programming language feature with proper compiler support that minimizes switch overhead and enables stack allocation with the optimal stack capacity per case.

The aforementioned compiler optimizations improve the performance of interleaving with coroutines, yet they do not eliminate the overhead of task switching. Ideally, we want an almost instant switch mechanism that is similar to simultaneous multithreading, with the difference of assigning coroutines instead of OS threads to each hardware context. Such a switch mechanism would eliminate the need to swap register contents per coroutine switch and at the same time avoid the expensive creation of an OS thread per task. Still, this approach would require as many hardware contexts as the maximum number of coroutines we might interleave; such a large number of hardware contexts, if not infeasible, would not be the best transistor investement for a general-purpose processor that is not dedicated to interleaving. In addition, no number of contexts is guaranteed to suffice for all future applications that might arise.

The flexibility of a software-based technique is a desirable property, and a promising alternative to one hardware context per coroutine would be to have two hardware contexts in total, supporting interleaving among an arbitrary number of coroutines. In this scheme, one context is active and the other passive; while coroutine in the active context is running, the state of the coroutine in the passive context would be swapped in the background, overlapping short stalls in the active coroutine with the background context switch. We essentially introduce data movement between registers and cache and thereby limit the number of hardware contexts to two. Further, to be able to hide smaller latencies, the scheme described here could be augmented with a second pair of contexts that would enable simultaneous multithreading between the coroutines in the active contexts, while the coroutines in the passive contexts would be swapped in the background.

Nevertheless, even if we cannot make coroutine switching instant, we could still improve the performance of interleaving by suspending upon actual cache misses In this thesis, suspension are defensively introduced by the programmer upon loads that incur cache misses most of the time—as determined offline through profiling. This coarse-grained approach implies unnecessary suspensions for the few times that the load instruction does not miss the cache. In addition, in case of loads that occasionally miss the cache and for which suspension is deemed non-beneficial, there are memory stalls to avoid. To be able to suspend upon and only upon actual cache misses, suspension needs to be decided at runtime, based on whether the load is an L1 miss or not. A branch instruction that tests if the accessed addressed is cached in L1 would suffice to avoid introducing pure overhead when all data is in L1. The same capability for L2 and L3 would have a non-negligible cost, as it requires a roundtrip to these caches to retrieve the information; it is also unnecessary given that the cost of a coroutine switch is equivalent to two function calls. Such an instruction would be a fast way to extract cache contents, lending itself for use in security attacks that target speculative execution, e.g. Spectre; as a result, unless we embrace the fact that microarchitectural behavior is observable, such an instruction is unlikely to be provided by mainstream processors.

All the points discussed here indicate on the one hand that compiler support is still maturing and on the other hand that some hardware support is missing. Still, they should not be deemed as drawbacks that diminish the current value of interleaving with coroutines, but rather as opportunities towards an ideal technique that will arise through the synergy of compiler writers,

computer architects, and system builders.

5.3 Parting thoughts

Imagine we could fit big data in the processor cache and access data without spending hundreds of stall cycles on data cache misses. This is probably an infeasible fantasy, yet cache-like performance is within reach for workloads that consist of parallel tasks. Parallel tasks can be interleaved so that, when one waits for memory, others can use the otherwise idle processor core. Our thesis shows interleaved execution is practical to implement in real systems today and brings the old idea of cooperative multitasking back into the spotlight as the way to hide latency.

Bibliography

[1] 2013. Intel Memory Latency Checker. http://www.intel.com/software/mlc [Online; accessed 18-March-2019].

[2] 2015. C# Reference. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/yield [Online; accessed 16-May-2018].

[3] 2016. Transaction Processing Performance Council. TPC-DS Benchmark Version 2.3.0. http://www.tpc.org/tpcds/ [Online; accessed 14-August-2017].

[4] 2017. Programming Languages – C++ Extensions for Coroutines. Proposed Draft Technical Specification ISO/IEC DTS 22277 (E). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4680.pdf [Online; accessed 16-May-2018].

[5] 2018. Generators. Python Wiki. https://wiki.python.org/moin/Generators [Online; accessed 16-May-2018].

[6] 2019. Intel Optane DC Persistent Memory Module. www.intel.com/optanedcpersistentmemory [Online; accessed 20-March-2019].

[7] 2019. Working Draft, Standard for Programming Language C++. http://eel.is/c++draft/dcl.fct.def.coroutine [Online; accessed 25-March-2019].

[8] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1754–1765. https://doi.org/10.14778/3137765.3137780

[9] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 54–65. http://dl.acm.org/citation.cfm?id=645925.671364

[10] Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2004. Improving Hash Join Performance Through Prefetching. In *Proceedings of the 20th International*

Conference on Data Engineering (ICDE '04). IEEE Computer Society, Washington, DC, USA, 116–. http://dl.acm.org/citation.cfm?id=977401.978128

[11] Maria Colgan. 2015. *Oracle Database In-Memory*. Technical Report. Oracle Corporation. http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.pdf [Online; accessed 16-May-2018].

[12] Melvin E. Conway. 1963. Design of a Separable Transition-diagram Compiler. *Commun. ACM* 6, 7 (1963), 396–408. http://doi.acm.org/10.1145/366663.366704

[13] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Helen Li, and Yiran Chen. 2008. Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) As a Universal Memory Replacement. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08)*. ACM, New York, NY, USA, 554–559. https://doi.org/10.1145/1391469.1391610

[14] Ulrich Drepper. 2007. What Every Programmer Should Know About Memory. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[15] R. Kent Dybvig. 2009. *The Scheme Programming Language, 4th Edition* (4th ed.). The MIT Press.

[16] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[17] Brian Fields, Shai Rubin, and Rastislav Bodík. 2001. Focusing Processor Policies via Critical-path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*. ACM, New York, NY, USA, 74–85. https://doi.org/10.1145/379240.379253

[18] B Govoreanu, GS Kar, YY Chen, V Paraschiv, S Kubicek, A Fantini, IP Radu, L Goux, S Clima, R Degraeve, et al. 2011. $10\times 10nm^2$ Hf/HfO_x crossbar resistive RAM with excellent performance, reliability and low-energy operation. In *IEEE International Electron Devices Meeting (IEDM)*. IEEE, 31–6.

[19] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[20] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. 1996. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. *SIGARCH Comput. Archit. News* 24, 2 (May 1996), 260–270. https://doi.org/10.1145/232974.233000

[21] IBM Corporation. 2018. *POWER9 Processor User's Manual*.

[22] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.

[23] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

[24] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *PVLDB* 11, 11 (July 2018), 1702–1714. https://doi.org/10.14778/3236187.3236216

[25] Alfons Kemper and Thomas Neumann. 2011. HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*. 195–206.

[26] Michael Kerrisk. 2010. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook* (1st ed.). No Starch Press, San Francisco, CA, USA.

[27] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/1807167.1807206

[28] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. 2018. Cimple: Instruction and Memory Level Parallelism: A DSL for Uncovering ILP and MLP. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 30, 16 pages. https://doi.org/10.1145/3243176.3243185

[29] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *PVLDB* 9, 4 (2015), 252–263. https://doi.org/10.14778/2856318.2856321

[30] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proc. of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 468–479. http://doi.acm.org/10.1145/2540708.2540748

[31] Oliver Kowalke and Nat Goodspeed. [n. d.]. call/cc (call-with-current-continuation): A low-level API for stackful context switching. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0534r3.pdf [Online; accessed 16-May-2018].

[32] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. ASPLOS*. 12. http://doi.acm.org/10.1145/106972.106981

[33] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage

Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

[34] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. 2013. Enhancements to SQL Server Column Stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1159–1168. https://doi.org/10.1145/2463676.2463708

[35] Per-Ake Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL Server Column Store Indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 1177–1184. https://doi.org/10.1145/1989323.1989448

[36] James Laudon, Anoop Gupta, and Mark Horowitz. 1994. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, New York, NY, USA, 308–318. https://doi.org/10.1145/195473.195576

[37] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan. 2010), 143–143. https://doi.org/10.1109/MM.2010.24

[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[39] Jackson Marusarz. 2015. Understanding How General Exploration Works in Intel® VTune™ Amplifier XE. https://software.intel.com/en-us/articles/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe [Online; accessed 16-May-2018].

[40] T. J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Softw. Eng.* 2, 4 (July 1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[41] Jim Melton. 2002. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features*. Elsevier Science Inc.

[42] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (Feb. 2009), 31 pages. https://doi.org/10.1145/1462166.1462167

[43] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/143365.143488

[44] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proc. EDBT*. 283–294.

[45] Gor Nishanov. 2018. Memory Latency Troubles You? Nano-coroutines to the Rescue! (Using Coroutines TS, of Course). https://cppcon2018.sched.com/event/FnKT/memory-latency-troubles-you-nano-coroutines-to-the-rescue-using-coroutines-ts-of-course CppCon 2018.

[46] M. Poess and D. Potapov. 2003. Data Compression in Oracle. *Proc. VLDB* (2003), 937–947.

[47] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB* 11, 2 (Oct. 2017), 230–242. https://doi.org/10.14778/3149193.3149202

[48] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2018. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *The VLDB Journal* (14 Dec 2018). https://doi.org/10.1007/s00778-018-0533-6

[49] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap Between NVM and DRAM for Latency-bound Operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19)*. ACM, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3329785.3329917

[50] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: so Much More Than Just a Column Store. *PVLDB* 6, 11 (2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[51] Jun Rao and Kenneth A. Ross. 2000. Making B+-Trees Cache Conscious in Main Memory. In *Proc. ACM SIGMOD*. 475–486. http://doi.acm.org/10.1145/342009.335449

[52] RethinkDB Team. 2010. Improving a large C++ project with coroutines. https://www.rethinkdb.com/blog/improving-a-large-c-project-with-coroutines/ [Online; accessed 16-May-2018].

[53] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. 2012. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7* (6th ed.). Microsoft Press.

[54] Lars-Christian Schulz, David Broneske, and Gunter Saake. 2018. An Eight-dimensional Systematic Evaluation of Optimized Search Algorithms on Modern Processors. *Proc. VLDB Endow.* 11, 11 (July 2018), 1550–1562. https://doi.org/10.14778/3236187.3236205

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

[56] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-architectural Analysis of In-memory OLTP. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 387–402. https://doi.org/10.1145/2882903.2882916

[57] Richard Smith and Gor Nishanov. 2018. Halo: coroutine Heap Allocation eLision Optimization: the joint response. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0981r0.html [Online; accessed 15-March-2019].

[58] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[59] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 36–53. https://doi.org/10.1145/3299869.3300075

[60] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. 2005. Improving Database Performance on Simultaneous Multithreading Processors. In *Proc. VLDB*. 49–60. http://dl.acm.org/citation.cfm?id=1083592.1083602

[61] Jingren Zhou and Kenneth A. Ross. 2003. Buffering Accesses to Memory-resident Index Structures. In *Proc. VLDB*. 405–416.

Georgios Psaropoulos

Address: Zollhofgarten 6, 69115 Heidelberg, DE
Phone: +4915237967014 • +41791386185
Email/Skype: gpsar@outlook.com
LinkedIn: georgios-psaropoulos
Nationality: Hellenic (Greek)

Summary

Research-oriented software engineer with experience in large codebases and a solid background on computer systems, compilers, language design, and computer architecture. Currently, a PhD student researching ways to improve database performance without sacrificing code maintainability.

Experience

Doctoral Assistant – École Polytechnique Fédérale de Lausanne (EPFL), Switzerland **09.2013 – present**
- Main work on hardware-aware database systems (collaboration with the SAP HANA Database team).
- Presented work at 2 top-tier conferences and 6 other venues.
- Assisted in teaching 5 courses (material preparation, exercise sessions, exam grading).
- Participated in the ACM SIGMOD Programming Contest 2018 with a top-5 finalist team.

Developer Associate (Internship) – SAP, Germany **03.2016 – 08.2016**
- Optimized the evaluation of IN predicates in SAP HANA (C++, Git, Gerrit, CMake, Python).

Systems Developer (Trainee) – Consolidated Contractors Company (CCC), Greece **02.2013 – 04.2013**
- Automated a set of GUI tests for C3D Interactive (Scala, Java Swing, FEST framework).
- Developed an AutoCAD-to-C3D file converter (C#, F#, Microsoft Excel).

Education

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland **expected 2019**
Ph.D. in Computer Science
Thesis: *Improving Main-Memory Database System Performance through Cooperative Multitasking*
Advisor: Prof. Anastasia Ailamaki
- Courses: Principles of Computer Systems • Foundations of Software • Topics on Datacenter Design
- Participated in summer schools on *DSL Design and Implementation* and *Data Management Techniques*.
- Received a 1-year fellowship from the EPFL IC Doctoral School and a 3-year scholarship from SAP.

National Technical University of Athens (NTUA), Greece **July 2013**
Diploma (M.Eng. equivalent) in Electrical and Computer Engineering
GPA: 9.1/10
Thesis: *Concurrency and Parallelism in Erlang, F#, and Scala – A Comparative Study*
Advisor: Prof. Kostis Sagonas
- Core modules: Software Engineering • Algorithms and Complexity • Programming Languages • Databases • Operating Systems • Computer Architecture • Control Systems
- Received two awards for academic excellence.

Publications & Patents

- *Bridging the latency gap between NVM and DRAM for latency-bound operations*
  DAMON 2019 • G. Psaropoulos, I. Oukid, T. Legler, N. May, and A. Ailamaki • DOI: 10.1145/3329785.3329917
- *Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins*
  VLDBJ 2018 • G. Psaropoulos, T. Legler, N. May, and A. Ailamaki • DOI: 10.1007/s00778-018-0533-6
- *Interleaving with coroutines: a practical approach for robust index joins*
  PVLDB 2017 • G. Psaropoulos, T. Legler, N. May, and A. Ailamaki • DOI: 10.14778/3149193.3149202
- *Coroutines for optimizing memory access*
  Patent (pending) • Assignee: SAP SE • Inventors: G. Psaropoulos, T. Legler, N. May, A. Ailamaki
- *Access pattern based optimization of memory access*
  Patent (pending) • Assignee: SAP SE • Inventors: G. Psaropoulos, T. Legler, N. May, A. Ailamaki

Notable projects

EPFL studies

- A practical coroutine-based technique to hide memory latency in task-parallel operations (C++).
- A DSL to traverse pointer-based data structures using a research hardware accelerator (Scala).
- An interpreter, a type checker and a type inferencer for the simply typed lambda calculus (Scala).
- A port of the CloudSuite data caching benchmark to Docker (Memcached, Docker).

NTUA studies

- A compiler for a C-like procedural language (F#).
- An information system to regulate media companies (JSF, PL/SQL, Oracle Database Express Edition).
- Additional functionality modules for the SIP Communicator application (Java, Scala).

Hobby projects

- A port of an educational emulator of the 8085 microprocessor from C# to F# (F#, FParsec).
- A Windows Phone app for querying information about public transportation in Athens (C#, XAML).

Technical Skills

Programming languages: C++ • F# • Scala • C# • Java • C • Bash • PowerShell • Python • Erlang • Prolog
Programming paradigms: Procedural • Functional • Object-oriented • Message Passing
Database systems: SAP HANA • PostgreSQL • MySQL • SQL Server • Oracle Database Express Edition
Development tools: Git • Gerrit • CMake
Debugging & profiling: Visual Studio Debugger • GDB • Intel VTune Amplifier
Miscellaneous: Microsoft Office • Latex

Language Skills

Greek: Native proficiency
English: Full professional proficiency
German: Limited working proficiency
French: Elementary proficiency

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**