Accelerators for Data Processing

THÈSE N^{O} 6710 (2015)

PRÉSENTÉE LE 25 SEPTEMBRE 2015

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DE SYSTÈMES PARALLÈLES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Yusuf Onur KOÇBERBER

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

Acknowledgements

First and foremost, I would like to thank my PhD advisor, Babak Falsafi, for giving me the opportunity to do a PhD in his lab. I not only learned how to become a better person from him but I also greatly benefited from the decent research and social environment he created at EPFL. Babak has been a constant source of inspiration with his extremely high standards, which brought out the best in me. At the same time, I always admired how he made hard goals simple and achievable. Undoubtedly, doing a PhD with Babak was the best thing ever happened to me.

Besides my advisor, I would like to thank the rest of my thesis committee, Edouard Bugnion, Partha Ranganathan, and Karu Sankaralingam, for their encouragement, insightful comments, and also the constructive criticism during my thesis proposal, which directly influenced the ideas presented in my thesis. I would also like to thank Christoph Koch who accepted to be my thesis jury president and provided valuable feedback during my defense.

A very special thanks goes to Boris Grot who truly made an impact on this thesis. Boris spent countless hours on my work and he always managed to take it to the next level. Meanwhile he was always positive, passionate and encouraging. I am not sure if I will ever find a mentor/advisor/collaborator like Boris in the rest of my life!

I am also grateful to Partha Ranganathan and Kevin Lim for being excellent mentors and teachers during the joint project we pursued and my internships at HP Labs. Partha had a big influence on me with his vision, leadership and scientific skills. Kevin was a vast source of technical and practical information. Both Partha and Kevin spent tremendous amount of energy and time while helping me to reach my deadlines.

Stavros Volos, Javier Picorel and Djordje Jevdjic have been marvelous collaborators, great friends, thirsty drinking buddies, and tireless travel companions. They were my go-to people whenever I needed help with my research and they always provided top-quality feedback. Outside of the lab, I enjoyed their company. Stavros made winters fun with his never-ending passion for skiing. Javier made me laugh all the time with his good humor and impersonations. Djordje made my coffee breaks interesting with his music and political knowledge.

During early years of my PhD, Mike Ferdman was my *PhD survival guide* as he thought me countless things about speaking, writing and doing research. Alisa Yurovsky and Mike Ferdman also created a social lab culture by throwing parties and organizing outings, which allowed me to make new friends in Lausanne.

If it was not for my undergrad and MS advisor, Oguz Ergin, I wouldn't have gone to graduate school. His guidance throughout my time at TOBB ETU has changed my life. When Cansu and I wanted to start a PhD, Oguz made a big sacrifice by letting and helping us leave the *Kasırga* lab at the same time. Thanks Oguz, the Kasırga spirit never dies!

I would like to thank all members of the PARSA team for their friendship, collaboration and support. Specifically, Sotiria Fytraki provided delicious Greek food during deadlines and truly helped me before the candidacy exam in my first year. Pejman Lotfi-Kamran was a great office-mate who made my life easier with his deep knowledge of the simulation infrastructure. Effi Georgala, the honorary member of PARSA, has been a great friend, always keeping me updated with the most recent exercise trends and food recipes. I would also like to thank Effi for being my personal pronunciation coach throughout the years. A warm thanks to Almutaz Adileh, Evangelos Vlachos, Jennifer Sartor, Alexandros Daglis, Yorgos Psaropoulos, Nooshin Mirzadeh, Mario Drumond, Dmitrii Ustiugov and Nancy Chong for their friendship and support throughout my PhD journey. I highly appreciated the assistance of Stéphanie Baillargues, Valérie Locca and Rodolphe Buret in making all the administrative and technical tasks as painless as possible throughout the years. I also wish to acknowledge Damien Hilloulin for translating my thesis abstract into French.

Several people and institutions outside of PARSA and/or EPFL have been instrumental in this thesis. I would like to thank Pınar Tözun and Danica Porobic from the DIAS lab at EPFL for tirelessly answering my questions about database systems and proofreading my papers. Tim Harris of Oracle Labs kindly helped me to run experiments on their servers. Google Doctoral Fellowship and HP Labs Innovation Research Program graciously funded my research.

The *Turkish Gang* (Duygu Ceylan, Kerem Kapucu, Barıs Can Kasıkçı, and Pınar Tözun) was my laughing therapy and kept me sane in graduate school. I really enjoyed the nights we spent together laughing, cooking and drinking. You are my life-long friends and I will miss you.

These acknowledgments cannot be possibly complete without expressing my gratitude to my parents, Emine and Abdulkadir Koçberber, and my sister, Nur Kılıç Koçberber, for their unconditional love and support. I will forever remain indebted to them.

Finally, words cannot describe my feelings, nor my gratitude to my better half, Cansu Kaynak, for always being there for me, her patience, and undying love.

Abstract

The explosive growth in digital data and its growing role in real-time analytics motivate the design of high-performance database management systems (DBMSs). Meanwhile, slowdown in supply voltage scaling has stymied improvements in core performance and ushered an era of power-limited chips. These developments motivate the design of software and hardware DBMS accelerators that (1) maximize utility by accelerating the dominant operations, and (2) provide flexibility in the choice of DBMS, data layout, and data types.

In this thesis, we identify pointer-intensive data structure operations as a key performance and efficiency bottleneck in data analytics workloads. We observe that data analytics tasks include a large number of independent data structure lookups, each of which is characterized by dependent long-latency memory accesses due to pointer chasing. Unfortunately, exploiting such inter-lookup parallelism to overlap memory accesses from different lookups is not possible within the limited instruction window of modern out-of-order cores. Similarly, software prefetching techniques attempt to exploit inter-lookup parallelism by statically staging independent lookups, and hence break down in the face of irregularity across lookup stages. Based on these observations, we provide a dynamic software acceleration scheme for exploiting inter-lookup parallelism to hide the memory access latency despite the irregularities across lookups. Furthermore, we propose a programmable hardware accelerator to maximize the efficiency of the data structure lookups. As a result, through flexible hardware and software techniques we eliminate a key efficiency and performance bottleneck in data analytics operations.

Key words: Performance, energy efficiency, database systems, analytics, hash tables, trees, indexes, accelerators, prefetching

Résumé

L'augmentation massive du volume de données numériques et son importance croissante dans l'analyse de données rend crucial la conception de systèmes de gestion de bases de données (SGBD) haute-performance. En parallèle, le ralentissement des améliorations apportées aux tensions d'alimentation a contrecarré les plans d'améliorations des performances des cœurs de processeurs, et ouvert l'ère des puces limitées en puissance. Ces développements motivent la conception d'accélérateurs logiciels et matériels de SGBD qui (1) maximisent l'utilité en accélérant les opérations dominantes, et (2) fournissent une plus grande flexibilité dans le choix du SGBD ainsi que le format et le type des données.

Dans cette thèse, nous identifions les structures de données utilisant de manière intensive les pointeurs comme déterminant et limitant les performances dans les tâches d'analyse de données. Nous observons que les tâches d'analyse des données comprennent un nombre important d'accès indépendants à des structures de données, chacun étant caractérisé par des accès mémoires dépendants et à fortes latences induits par le suivi des différents pointeurs. Malheureusement, exploiter le parallélisme inhérent à ces différents accès indépendants pour faire se chevaucher les accès mémoires n'est pas possible de par le set d'instructions limité des coeurs non ordonnées actuels. De même, les techniques logicielles de préchargement des données ("prefetching") essaient d'exploiter le parallélisme des accès indépendants en les ordonnant par étape de manière statique, et peinent face à l'irrégularité des accès entre les différentes étapes. En se basant sur ces observations, nous fournissons une méthode d'accélération logicielle dynamique pour exploiter le parallélisme entre les accès indépendants pour cacher la latence d'accès à la mémoire malgré les irrégularités entre les différentes étapes d'accès. Nous proposons de plus un accélérateur matériel qui maximise l'efficacité de ces accès à travers les structures de données. En résumé, nous éliminons un élément clé limitant l'efficacité et les performances dans les opérations d'analyse des données.

Mots clefs : Performance, efficacité énergétique, systèmes de base de données, analytique, tables de hachage, arbres, indices, accélérateurs, préchargement des données (prefetching)

Contents

List of Figures

List of Tables

1 Introduction

The information revolution of the last decades is being fueled by the explosive growth in digital data. Enterprise server systems reportedly operated on over 9 zettabytes (1 zettabyte = 10^{21} bytes) of data in 2008 [69], with data volumes doubling every 12 to 18 months. As businesses such as Amazon and Wal-Mart use the data to drive business intelligence and decision support logic via databases with several petabytes of data, IDC estimates that almost 40% of global server revenue ($22 billion out of $57 billion) goes to supporting database workloads [27].

Businesses have been leveraging analytics to improve decision making processes for more than fifty years. However, since the information revolution data-driven business processes has become a key differentiator among business competitors, a phenomena sometimes referred to as *competing on analytics* [19]. Serious competition entails real-time and interactive analytic query processing over copious data. As a result, to keep up with the business requirements, database vendors and researchers started considering custom-tailored analytics systems [72, 73] by departing from the traditional all-purpose database system architectures.

In the era of rethinking the database system architectures, column-oriented databases gained significant importance as they have been shown to deliver at least an order of magnitude better performance than the traditional row-oriented databases for emerging analytics workloads. The key enabler behind the significant performance improvements goes beyond the columnar data layout; these systems introduced novel column-oriented execution engines for analytics, which implement numerous techniques to minimize data accesses during query execution such as late materialization, compression [2].

Furthermore, the advent of large main-memory capacity combined with the benefits of column-oriented processing made vasts datasets accessible in the blink of an eye [54]. To sustain the trend towards main-memory processing, database servers integrate as much memory as possible within the limits of current DRAM technology. Because the cost of DRAM constitues a significant fraction of the server operation and capitalization costs, it is imperative that software and hardware is optimized to fully utilize the available memory bandwidth and capacity. Given the abundant parallelism offered by analytics tasks, software optimizations are centered around algorithms that can take advantage of the parallelism offered by general-purpose multi-core processors to hide the memory access latency. Driven by the Moore's law, processor designers were able to keep increasing the number of cores on a chip to match the needs of software.

Unfortunately, with the end of Dennard scaling, which has historically been the primary mechanism for lowering the energy per transistor switching event, the ability to double the processor performance at a constant power budget has greatly diminished [24, 34]. Increasing the number of transistors on the semicondutor area directly leads to a higher energy consumption at the chip level and higher server electricity and cooling costs at the server level. As a result, electricity cost of operating servers constitute a significant fraction of the total cost of ownership (TCO) for datacenters.

With rising server operation costs, energy efficiency in database management systems has become a first-class optimization criteria. After two decades of its foundation, Transaction Processing Council (TPC), an organization that sets the industry standard for database benchmarks, introduced energy-related metrics to all database benchmark specifications besides performance and price metrics [79]. Researchers from industry and academia have attempted to improve energy efficiency of the database systems by proposing novel benchmarks and optimization techniques on commodity hardware [30, 48, 49, 62, 80].

Similarly, constrained by power at the chip level in the post-Dennard era, computer architects struggled to improve the performance and efficiency of general-purpose processors. Consequently, system researchers advocate for improving general-purpose processor throughput through *hardware-conscious* software and algorithms, which are tuned to the underlying hardware via architecture-specific parameters and instructions [5, 10, 17, 46, 53, 92]. At the same time, an increasing number of hardware proposals are calling for specialized server hardware to increase performance and energy efficiency by tailoring the hardware to the specific needs of the database operations [18, 37, 55, 85, 86, 90].

While these efforts represent a step towards the right direction in the post-Dennard era, meeting the computation demand fueled by ever-growing data necessitates a holistic approach by tailoring the hardware to the needs of the software and vice versa. Although software and hardware specialization are not mutually exclusive, the two critical challenges are (1) identifying the service that would benefit the most from specialization by delivering significant value for a large number of users (i.e., maximize utility), and (2) specializing just the right functionality of the targeted service to provide significant performance and/or energy efficiency gain without limiting applicability (i.e., avoiding over-specialization).

We observe that emerging data analytics applications are tasked with finding critical pieces of information in vast data sets – a "needle-in-the-haystack" problem, often with an additional constraint of being real-time. Doing so rapidly mandates the use of in-memory pointer-intensive data structures, which convert linear-time search operations into near-constant-time lookups. As a result, performance and efficiency bottleneck for many database operations is accessing main memory, as vast datasets overwhelm on-chip caches and dependent access patterns frequently leave the contemporary CPUs waiting on a long-latency memory access [4].

This thesis identifies pointer-intensive data structure lookups as a critical performance and efficiency bottleneck in data analytics tasks and proposes software and hardware mechanisms to maximize the throughput and efficiency for a wide range of data structure operations without limiting their applicability.

1.1 High-Throughput and Efficient Data Lookups

1.1.1 Hiding Memory Access Latency

A modern CPU employs multiple power-hungry out-of-order (OoO) cores, which are designed to hide the memory access latency by identifying and issuing multiple independent memory accesses. The number of in-flight memory accesses at a given point in time is called *memory-level parallelism*. The amount of memory-level parallelism is dictated by the number of independent memory operations within the instruction window of the processor core. A lookup in a pointer-intensive data structure (e.g., a hash table) may require chasing pointers, resulting in low memory-level parallelsim as the next pointer cannot be discovered until the current access completes.

Fortunately, many analytics tasks that leverage pointer-intensive data structures (e.g., hash join, indexed-join, and group-by) involve a large number of independent key lookups. For example, a group-by operation involves grouping a large number of key-value tuples according to tuple keys by inserting each tuple into a hash table one at a time. Similarly, an indexed-join, involves probing an index tree or a hash index for every join attribute in the input database table or column. Therefore, there is abundant inter-lookup parallelism that can be exploited to increase the memory-level parallelsim of each core. Exploiting such inter-lookup parallelism within an OoO core is difficult due to the limits on instruction window size imposed by technology [3]. Nevertheless, hardware multi-threading and prefetching are two important techniques that can possibly exploit such parallelism and hide the memory access latency within a core.

Simultaneous multi-threading (SMT) enables multiple hardware contexts to share single core resources to exploit memory-level parallelism. Major processor vendors, provide general-purpose OoO cores with various degrees of SMT [1, 43]. Although these processors can overlap multiple long-latency cache misses that belong to different threads, they fall short of an ideal multi-core processor due to imperfect interleaving of memory and computation phases of different threads leading to pipeline resource contention. More specifically, recent research showed that although hash table lookups in database operations are memory-bound, SMT provides marginal throughput benefits for hash join [10, 13]. Furthermore, SMT deteriorates the overall throughput for hardware-conscious hash join algorithms, which are relatively more compute intensive [11].

Prefetching is another effective technique to hide the memory access latency by demanding data blocks ahead of their consumption. Hardware data prefetching techniques for pointer-intensive data structures [23, 64, 88] can identify dependent access patterns and predict future prefetch requests ahead of the core. Although, these techniques can eliminate memory stalls that will be encountered in the future, they suffer from several fundamental limitations, which hinder their applicability. First, the pointer address calculation mechanisms are limited to regular and simple data structure layouts to generate prefetch requests, while database systems can have arbitrarily complex data structures (i.e., unclustered indexes, variable-length keys). Second, these prefetching techniques require distributive changes in the cache hierarchy, otherwise require address translation mechanisms as they are placed far from the core and rely on virtual addresses to generate prefetch requests.

Software prefetching techniques for pointer-intensive data structures have several advantages over hardware prefetching techniques. First, in modern architectures software prefetch instructions complete as long as a TLB miss does not cause a fault [43, 71], which is rare with in-memory database algorithms. Second, generating prefetch requests for complex data structure layouts is trivial given the layout is exposed to the programmer or compiler.

Unfortunately, the main drawback of the software prefetching is that it is not possible to issue prefetch requests within the traversal of single pointer chain due to dependent address calculations. To solve this problem, prior work has proposed data-linearization prefetching [52] to calculate the addresses without needing pointers so that the prefetches can be issued ahead of time. Similarly, history-based prefetching techniques [15] maintain an array of jump pointers containing the pointers from recent traversals. However, these techniques either assume that the data structure is traversed in a similar order more than once or incur both space and time overhead to increase the prefetch accuracy.

To exploit inter-lookup parallelism, researchers proposed loop transformations augmented with software prefetching instructions. State-of-the-art software prefetching approaches for database systems work by arranging a set of independent lookups into a *group* (Group Prefetching [14]) or *pipeline* (Software Pipelining [14, 46]), in effect synchronizing their memory accesses into a highly structured sequences.

These approaches work well whenever the number of pointer dereferences is known ahead of time and is constant across lookups, in which case the group size or the number of pipeline stages can be provisioned to perfectly accommodate the memory access pattern. Whenever such perfect knowledge or regularity are not present, some lookups may exhibit *irregularity* with respect to the expected or average case. Examples of irregularity include variable number of nodes per bucket in a hash table, early exit (e.g., on a match of a unique key), and read/write dependencies that require serialization of subsequent accesses via a latch. When such irregularities occur, existing software prefetch techniques must execute expensive and complex cleanup or bailout code sequences that greatly diminish the techniques' effectiveness.

1.1.2 Specialized Hardware

The idea of specialized database hardware (i.e., database machines) was explored in 1980s but long design turnarounds and high cost of specialized hardware made these systems unattractive in the face of cheap commodity hardware [22]. Today, the data deluge and efficiency constraints are reviving similar approaches via field-programmable gate arrays (FPGAs) in commercial analytic appliances (e.g., Netezza [37], Teradata [75]). Ideally, the reconfigurable and parallel hardware offered by the FPGAs can be leveraged as a high-utility and -performance acceleration substrate by synthesizing specialized circuits for every incoming analytics query. In practice, due to the impractical synthesis time of complex operations (e.g., joins) on FPGAs [57, 76], they can only accelerate the basic data processing operations (e.g., select) and delegate the rest of the query to the general-purpose CPUs. Regardless of the specialization substrate (FPGAs or on-chip accelerators), practical specialized hardware requires fast programmability to achieve high utility given that reconfigurability is not practical for complex data processing operations, database queries, and data-centric workloads [50, 60].

General-purpose processors also include specialized on-chip units to improve the performance and efficiency of the database operations. SPARC64 X+ [90] architecture employs a customized execution unit to accelerate bit vector and byte compare operations. Similarly, recent x86 architectures employ wider SIMD units [40, 68], which are beneficial for increasing data sort and aggregation throughput [17, 92]. Overall the specialized data processing hardware comes in the form of an execution unit in the processor pipeline, offering extremely low utility for memory-intensive database operations, while carrying the inefficiencies of the general-purpose hardware.

1.2 Thesis and Dissertation Goals

Pointer-intensive data structure lookups are a key performance and efficiency bottleneck in analytics. Therefore, the goal of this dissertation is to first maximize the memory-level parallelism via specialized software techniques for higher throughput and achieve higher efficiency via hardware specialization. The statement of this thesis is as follows:

High-throughput and energy-efficient data processing requires exploiting existing inter-lookup parallelism to maximize memory-level parallelism through software and hardware specialization.

1.3 Asynchronous Memory Access Chaining

We observe that many real-world execution scenarios involving pointer chasing entail irregularity across lookups. Achieving high memory-level parallelsim in these circumstances requires a degree of dynamism that is beyond the capability of today's software prefetching techniques.

To overcome the existing capability gap, this thesis introduces a new software prefetching scheme that avoids the need for rigidly arranging independent lookups into a group or a pipeline. By preserving and exploiting the lack of inter-dependencies across lookups, we are able to attain high memory-level parallelism even for highly irregular access patterns.

Our proposed software prefetching technique achieves its dynamism by maintaining the state of each in-flight lookup separately from that of other lookups. State maintenance operations are explicit, meaning that once a prefetch is launched, the state associated with that lookup is saved into a dedicated slot in a software-managed buffer, at which point a different lookup can be handled by loading its respective state. By decoupling the state of all in-flight lookups from each other, we are able to achieve unprecedented flexibility in initiating, completing, and waiting on lookups. Such flexibility directly translates into high memory-level parallelism, as potential memory access opportunities are not wasted due to common issues such as variable-length pointer chains.

1.4 On-chip Accelerator for Index Traversals

The explosive growth in digital data and its growing role in real-time analytics support motivate the design of high-performance database management systems. Meanwhile, slowdown in supply voltage scaling has stymied improvements in core performance and ushered an era of power-limited chips. These developments motivate the design of database system accelerators that (1) maximize utility by accelerating the dominant operations, and (2) provide flexibility in the choice of database system, data layout, and data types.

In this thesis, we study data analytics workloads on contemporary in-memory databases and find hash index lookups to be the largest single contributor to the overall execution time. The critical path in hash index lookups consists of ALU-intensive key hashing followed by pointer chasing through hash table nodes. Based on these observations, we propose a specialized on-chip accelerator for database hash index lookups, which achieves both high performance and flexibility by (1) decoupling key hashing from the list traversal, and (2) processing multiple keys in parallel on a set of programmable *walker* units. By tightly integrating our mechanism with a conventional core, we are able to reduce design cost and complexity thus eliminate the need for a dedicated TLB and cache.

1.5 Memory Subsystem Bottlenecks

Software and hardware acceleration techniques work in tandem with the memory hierarchy and virtual memory subsystem. Although, these techniques are effective at improving overall throughput, a sub-optimal memory hierarchy design can become a significant performance bottleneck because it is on the critical path of each data structure lookup. For effective implementations of software acceleration, the bottlenecks in the memory subsystem should be minimized or completely eliminated. This is especially important for specialized cores and accelerators that rely on the existing memory hierarchy, because a mismatch between the memory subsystem and the accelerator design can offset the efficiency benefits of specialization and waste on-chip resources.

In this thesis, we analyze the impact of memory subsystem on software acceleration with the goal of eliminating possible memory hierarchy and address translation overheads. We observe that three important features of the memory hierarchy play a crucial role in achieving high throughput data lookups, (1) virtual memory page size, (2) number of miss status handling registers that keep track of outstanding cache misses, and (3) data structure padding. Once we eliminate these bottlenecks for our software acceleration scheme, we take a next logical step and explore the options to leverage a specialized hardware accelerator for improving a broad class of data structure lookup throughput and efficiency.

1.6 Contributions

In this thesis, we explore specialized software and hardware mechanisms as a means to improve memory-level parallelsim and hide memory access latency in analytics workloads. We begin by analyzing the irregularity in pointer-intensive data structures in analytics and describe a robust software prefetching method to hide the memory access latency during regular and irregular data structure operations. We then study data analytics workloads on a contemporary database system to quantify the impact of data structure operations to overall analytics query execution. Based on our findings we propose a specialized hardware mechanism, which achieves high-throughput and efficiency hash index lookups. We conclude by exploring the limits of software and hardware specialization designs specifically focusing on the memory subsystem design.

Through a combination of real-hardware measurements, cycle-accurate modeling of CMP systems running data analytics workloads, and RTL modeling, we demonstrate:

- **Robust software prefetching technique for pointer-intensive data structures**. We demonstrate and evaluate a prefetching technique implementation on x86 and SPARC processors for various data structures used in data analytics systems. Our technique achieves 4.3x speed-up over the no-prefetching hash join baseline for uniform lookups while maintaining its performance advantage in the presence of irregular accesses. In the skewed hash join workloads with irregular accesses, our technique improves the performance by 3x over the no-prefetch baseline and by 1.8x over the state-of-the-art techniques. In doing so, our prefetching technique relies on a simple circular buffer occupying less than one KB in software regardless of the actual data structure size.
- **Characterization of modern data analytics workloads on contemporary database systems.** We study modern in-memory databases running modern data analytics workloads (TPC-H and TPC-DS) on real server hardware and show that hash index (i.e., hash table) accesses are the most significant single source of runtime overhead, constituting 14-94% of total query execution time.
- **Programmable hardware accelerator for hash index accesses.** We demonstrate a hardware accelerator for performing hash index traversals on contemporary database systems. The limited programmability afforded by the simple hardware accelerator allows our accelerator to support a virtually limitless variety of schemas and hashing functions. An evaluation

of the proposed accelerator on a set of modern data analytics workloads using full-system simulation shows an average speedup of 3.1x over an aggressive OoO core on bulk hash table operations, while reducing the OoO core energy by 83%.

- **Effectiveness of memory subsystem optimizations.** We quantify the mismatch between the memory subsystem design and the requirements of prefetch-based software optimization techniques. We show that properly optimized memory subsystem paves the way for a purely compute-bound execution for pointer-intensive data structure traversals despite the large datasets. In light of an effective memory subsystem, we holistically study the benefits of hardware and software acceleration. Using a combination of real hardware measurements and analytical modeling, we demonstrate the possibility of reducing the OoO core energy by up to 50x and fully utilizing the available off-chip memory bandwidth while performing pointer-chasing operations.

2 Background

2.1 DBMS Basics

Database management systems (DBMS) play a critical role in providing structured semantics to access large amounts of stored data. Based on the relational model, data is organized in *tables*, where each table contains some number of *records*. Queries, written in a specialized query language (e.g., SQL) are submitted against the data and are converted to *physical operators* by the DBMS. The most fundamental physical operators are *scan, join, group-by* and *sort*. The scan operator reads through a table to select records that satisfy the selection condition. The join operator iterates over a pair of tables to produce a single table with the matching records that satisfy the join condition. The group-by operator classifies records according to their key attributes and applies an aggregation function to summarize their values. The sort operator outputs a table sorted based on a set of attributes of the input table. As the tables grow, the lookup time for a single record increases linearly as the operator scans the entire table to find the required record.

In order to accelerate accesses to the data, database management systems commonly employ auxiliary data structures with sub-linear access times. These data structures can either be an *index* generated by the database administrator or they can be built on the fly as a query plan optimization. In the latter case, the data structure built during the execution of the query can

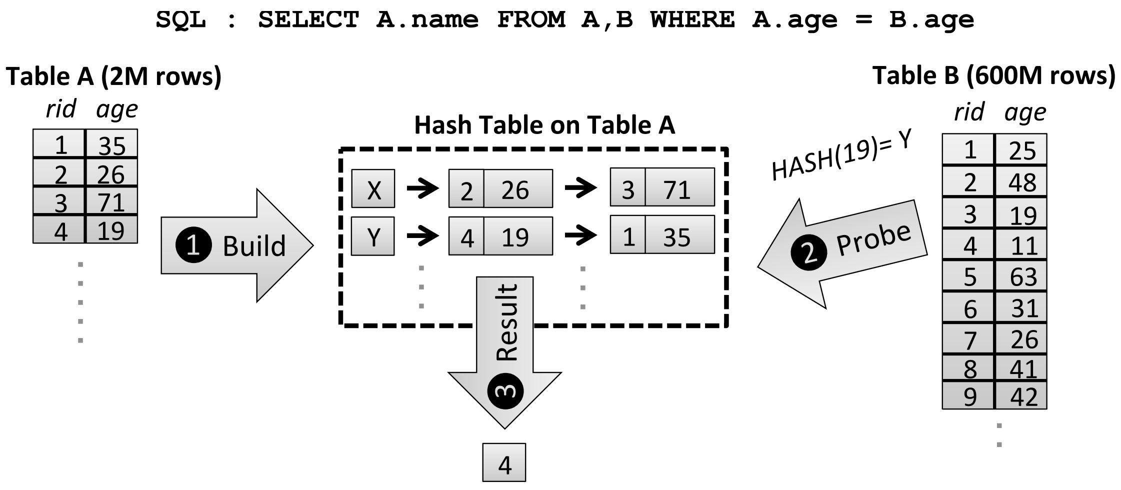


Figure 2.1 – Join via hash index.

be either be destroyed to save memory space after the query completes or it can materialized so that future queries with the same input can use the same data structure to avoid the build overhead (i.e., automatic indexing [38]). The most commonly used data structures are hash tables and trees, which have complementary properties. Hash tables are commonly preferred for their constant lookup time but they do not preserve any order across the keys. In contrast, trees are beneficial when the order of the lookup matters (i.e., range queries) but require traversing several pointers before locating an entry.

2.2 Pointer-Intensive Data Structures in Database Systems

2.2.1 Hash Tables

Hash tables are prevalent in modern databases for accelerating data-finding and grouping operations. We consider the use of hash tables for two frequent database operators: hash join and group-by.

Figure 2.1 shows a query resulting in a join of two tables, A and B, each containing several million rows in a column-store database. The tables must be joined to determine the tuples that match $A.age = B.age$. To find the matches by avoiding a sequential scan of all the tuples in Table A for each tuple in Table B, a hash index is created on the smaller table (i.e., Table A).

This index places all the tuples of Table A into a hash table, hashed on $A.age$ (Step 1). The index executor is initialized with the location of the hash table, the key field being used for the probes, and the type of comparison (i.e., is_equal) being performed. The index executor then performs the query by using the tuples from Table B to *probe* the hash table to find the matching tuples in Table A (Step 2). The necessary fields from the matching tuples are then written to a separate output table (Step 3).

Listing 2.1 shows the pseudo-code for the core index probe functionality, corresponding to Step 2 in Figure 2.1. The do_index function takes as input table t, and for each key in the table, probes the hash table ht. The canonical $probe\_hashtable$ function hashes the input key and walks through the node list looking for a match.

Hash table lookup throughput is the main bottleneck of the join operation, and its performance strictly depends on the number of dependent memory accesses (i.e., number of pointers chased) required to locate an item. Balkesen et al. [10] argue that in certain hash join scenarios (i.e., uniformly distributed unique keys), a hash table can be searched with only one memory access. Meanwhile, Barber et al. claim that due to space efficiency and imperfect hash functions, at least two independent memory accesses are required to probe a space-efficient hash table [12]. Yet Blanas et al. leverage a hash table requiring three dependent accesses [13]. Moreover, when there is value skew in the build relation, the hash collisions are unavoidable as the probe keys are identical but carry different payloads. Probing such buckets require as many memory accesses as the number of hash table nodes present in that bucket.

Moreover, in real database systems, the hash join and the index code tends to differ from the abstraction in Listing 2.1 in a few important ways. First, the hashing function is typically more robust than what is shown above, employing a sequence of arithmetic operations with multiple constants to ensure a balanced key distribution. Second, instead of storing the actual key, nodes can instead contain pointers to the original table entries, thus trading space (in case of large keys) for an extra memory access. The bottom line is that hash tables offer a tradeoff between performance (i.e., number of chained memory accesses) and space efficiency, and it

```
/* Constants used by the hashing function */
#define HPRIME 0xABC
#define MASK 0xFFFF
/* Hashing function */
#define HASH(X) (((X) & MASK) ^ HPRIME)

/* Key iterator loop  */
do_index(table_t *t, hashtable_t *ht) {
    for (uint i = 0; i < t->keys.size; i++)
        probe_hashtable(t->keys[i], ht);
}

/* Probe hash table with given key */
probe_hashtable(uint key, hashtable_t *ht) {
    uint idx = HASH(key);
    node_t *b = ht->buckets+idx;
    while(b) {
        if (key == b->key)
            { /* Emit b->id */ }
        b = b->next; /* next node */
    }
}
```

Listing 2.1 – Pseudo-code for join via hash index

is neither possible to generalize a single type of hash table layout nor guarantee a constant number of memory accesses for each probe.

Another use of hash tables in database systems is the group-by operator, which collects payloads of an input relation and groups them according to relation keys. The group-by operator is used in conjunction with an aggregation function (e.g., sum, min, max). Similar to the hash-join build phase, each payload in the input relation is added into a hash table. However, the difference is that in the case of non-unique keys, which is the common case for group-by, first the matching hash table node is located and then the necessary aggregation function is applied on the payload of the matching node. It is also possible that the aggregation operation is postponed to the end of the hash table build and just the payloads are collected in a separate list pointed to by the hash table node for each group (i.e., late aggregation). As a result, depending on the group-by scenario, hash table layout, and relation cardinality, the number of memory accesses per each tuple can differ significantly.

2.2.2 Tree Search

Tree index search is a fundamental operation in database systems to handle large datasets with low-latency and high-throughput. Given a list of key-value tuples (as a database table or column), a tree is built with nodes that are arranged in a specific order and connected with pointers. A tree index search involves continuous lookup to the tree nodes starting from root until the desired key is found. At each tree node, the node key is compared against the search key to determine the next child node to access.

The performance of a lookup in a search tree is directly related to the number of nodes traversed before finding a match. A single tree lookup is an inherently serial operation as the next tree node (i.e., child) to be traversed cannot be determined before the comparison in the current (parent) node resolves. The combination of branching control flow and large datasets leads to frequent memory stalls due to low cache and TLB locality. There are numerous proposals to optimize the layout of the index trees to improve their locality characteristics [16, 46, 61], but crossing the cache and TLB boundary, which is unavoidable, still incurs significant memory access penalties.

3 AMAC: Asynchronous Memory Access Chaining

In-memory databases rely on pointer-intensive data structures to quickly locate data in memory, often resulting in long-latency stalls due to pointer dereferences, which result in random memory accesses. Hiding the memory latency by executing useful instructions for other lookups – particularly, by launching other memory accesses – is an effective way of improving performance of pointer-chasing code, such as hash table probes and tree traversals. The ability to exploit such inter-lookup parallelism is beyond the reach of modern out-of-order cores due to the limited size of their instruction window. Instead, recent work has proposed software prefetching techniques that exploit inter-lookup parallelism by arranging a set of independent lookups into a group or a pipeline, and navigate their respective pointer chains in a synchronized fashion. While these techniques work well for highly regular access patterns, they break down in the face of irregularity across lookups. Such irregularity includes variable-length pointer chains, early exit, and read/write dependencies.

In this chapter, we describe *Asynchronous Memory Access Chaining* (AMAC), a new approach for exploiting inter-lookup parallelism to hide the memory access latency. AMAC achieves high dynamism in dealing with irregularity across lookups by maintaining the state of each lookup separately from that of other lookups. This feature enables AMAC to accommodate events such as early exit or variable-length pointer chains by generating new lookups as soon as any of the in-flight lookups completes. In contrast, the static arrangement of lookups into

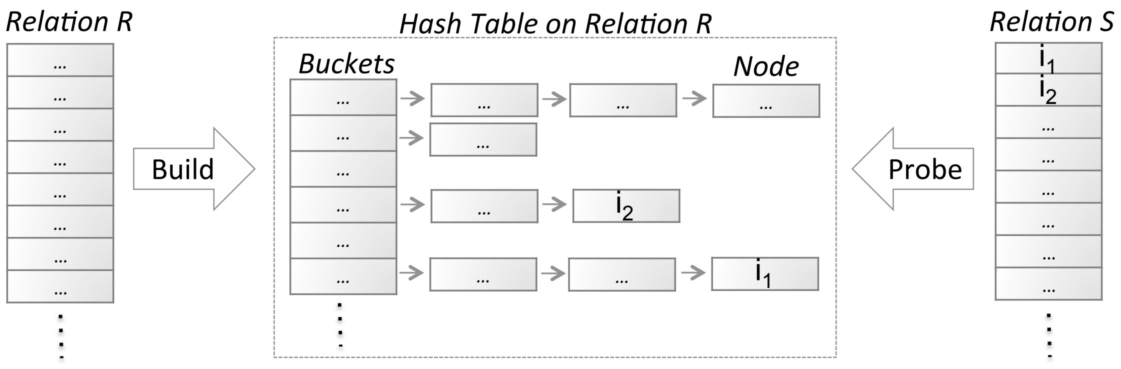


Figure 3.1 – Hash Join

a group or pipeline in existing techniques precludes such adaptivitity. Our results show that AMAC outperforms state-of-the-art prefetching techniques on regular access patterns, while delivering up to 2.3x higher performance over the existing techniques while performing hash joins of relations with skewed keys. AMAC fully utilizes the available micro-architectural resources, generating the maximum number of memory accesses allowed by hardware in both single- and multi-threaded execution modes.

3.1 Hiding Memory Access Latency

3.1.1 Software Prefetching Techniques for Pointer-Chasing Database Operations

Many database operations have abundant inter-lookup parallelism that can be be leveraged to increase MLP [14, 46, 91]. However, one complete lookup sequence involves many dozens of instructions to, for instance, hash a key, walk a chain of pointers, and produce a result. As such, even a high-end processor core with an instruction window of around 100 is not able to exploit inter-lookup parallelism. Instead, software prefetching can be used to initiate a memory access in a "fire-and-forget" fashion, allowing the core to execute instructions for other lookups and later return to complete the processing of the first one, with its memory latency completely hidden by useful work.

The state-of-the-art pointer-chasing prefetching techniques – namely, Group Prefetching (GP) and Software-Pipelined Prefetching (SPP) [14] – exploit inter-lookup parallelism in exactly this

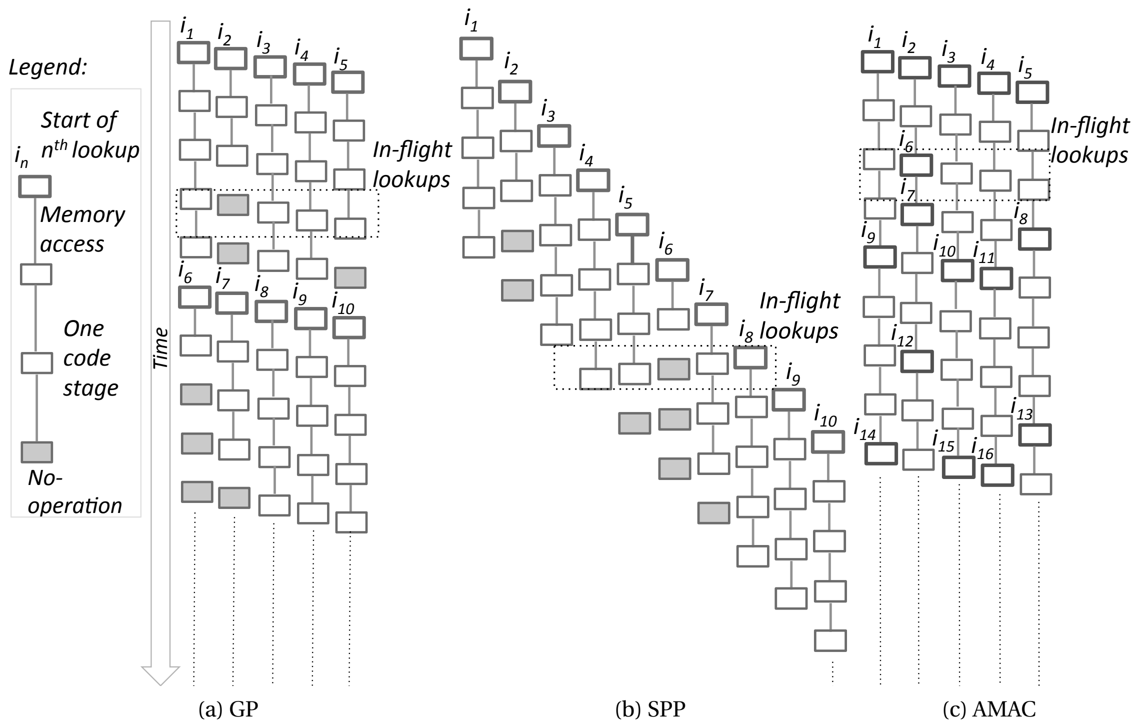


Figure 3.2 – Execution patterns of Group Prefetching (GP), Software-Pipelined Prefetching (SPP) and Asynchronous Memory Access Chaining (AMAC) in the case of traversal divergence.

fashion to improve the performance of hash table operations. SPP has also been applied to balanced search trees [46]. Both GP and SPP are loop transformations that break down a loop with N dependent memory accesses into a loop that contains $N+1$ code stages where each stage consumes the data from the previous stage and prefetches the data for the next stage. To hide the memory access latency by doing useful work, Group Prefetching executes each code stage for a group of M lookups,[1] therefore performing a maximum of M independent memory accesses at a time. Similarly, Software-Pipelined Prefetching forms a pipeline of $N+1$ stages, each code stage in the pipeline belongs to a different lookup and a single lookup completes after going through $N+1$ pipeline stages. In the cases where N is too small to hide the memory access latency, the pipeline is initiated with a prefetch distance so that M independent memory acesses ($M \geq N$) are performed in-flight.[2]

[1]This parameter is referred to as G in the original work [14]

[2]This parameter is referred to as D in the original work [14]. Hence, $M = N * D$.

Obviously, neither of the techniques is parameter-free. Both techniques require the number of stages, N, and the number of in-flight lookups M to be determined ahead of time. Setting M is relatively easy, as it is a function of the underlying hardware's memory-level parallelism capabilities; specifically, the maximum number of outstanding L1 data misses that can be in flight at once.[3] In contrast, N is a data structure- and algorithm-specific parameter, which makes both GP and SPP vulnerable to irregularities in the execution because N explicitly structures the execution pattern of all the M lookups, which leads to three issues:

1. Lookups might require less than N stages, due to the irregular data structure layout (e.g., as shown in Figure 3.1 for probes i_1 vs i_2). A similar situation could also occur on a regular structure in cases when certain lookups terminate earlier than others (i.e., early exit after finding a match). Regardless of the reason, when the actual number of stages is less than N, the remaining code stages must be skipped (i.e., no-operation) for that lookup.

2. Lookups might require more than N stages as the data structure is irregular (e.g., unbalanced trees or due to bucket collisions in a hash table). These cases require a bailout mechanism to complete the lookup sequentially.

3. Lookups might have a read/write dependency on each other (e.g., hash table build or update). When this occurs, the actual code stage and the subsequent ones should be executed later when the dependency is resolved.

Whenever any one of the above cases occurs, the maximum M will not be reached, necessarily lowering MLP. Over-provisioning M does not help, as it increases the number of no-operations (as explained in #1 above) or the likelihood of an inter-dependency (as in #2 or #3).

Figure 3.2a illustrates the Group Prefetching execution of ten independent lookups (i_{1-10}), where the number of code stages required and the maximum number of in-flight lookups

[3] In microarchitectural terms, it is the number of L1 Miss Status Handling Registers (MSHRs) that bounds a core's peak MLP.

are five ($N = 4, M = 5$). Each white box indicates a code stage and lines connecting the boxes indicate a memory access (prefetch) produced in the earlier code stage and consumed in the later code stage. The boxes with thick lines show the code stage that initiates a new lookup. The presence of gray boxes indicate a no-operation due to a lookup terminating earlier than expected (e.g, because of an empty bucket or an early exist). The dashed box depicts all the lookups within a group ($i_1 - i_5$) going through the same code stage. Once all the stages complete for all the lookups in the group, a new group of lookups are initiated ($i_6 - i_{10}$).

Figure 3.2b depicts the same example for Software-Pipelined Prefetching , the main difference being that one iteration (shown in a dashed box) contains different code stages for different lookups. One difference compared to GP is that the pipelined approach avoids the need to stop at a group boundary.

When we analyze the execution patterns of $i_{1,3,4,8,9,10}$ for both GP and SPP in Figure 3.2, we see that there are four memory accesses (five code stages), which perfectly matches parameter N in the example. In contrast, lookups $i_{2,5,6,7}$ turn out to be irregular as they terminate at different stages of the execution. To handle such complexities, GP and SPP maintain status information per lookup so that code stages can be skipped (necessary for correctness), resulting in a loss of memory-level parallelism and a waste of CPU cycles checking and propagating the status of completed lookups. The other irregularities, which include the case where the lookups turn out to be requiring more than N accesses or have a read/write dependency (not shown in the example), are more difficult to handle and require special "clean-up" passes and/or bailout mechanisms.

3.1.2 Performance Analysis of Software Prefetching

In order to understand the performance impact of the irregularities, we implemented GP and SPP for hash table probes. Our baseline implementation uses a chained hashed table with linked lists, resembling the hash table layout used in recent hash join studies [10]. The first hash table node is clustered with the bucket header to save one memory accesses as shown

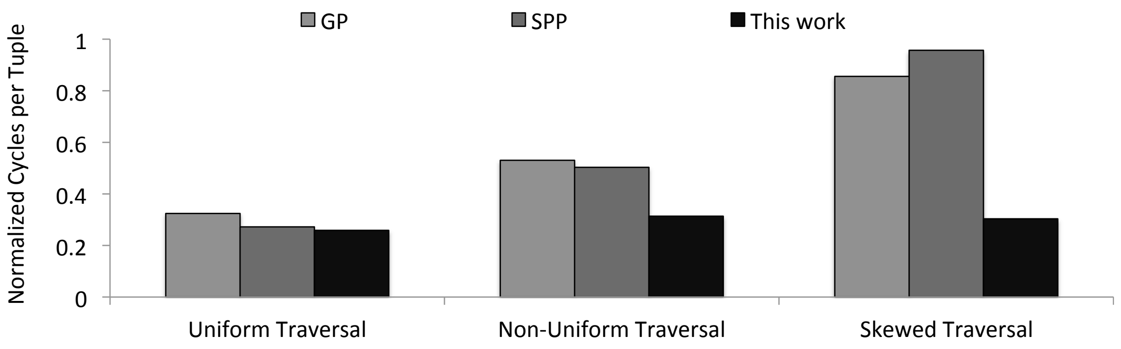


Figure 3.3 – Cycles per lookup tuple normalized to uniform lookups on baseline. Measurement on Xeon x5670.

in Figure 3.1. We run the experiment on Xeon x5670 with uniformly distributed random 2^{27} lookup tuples with 8B key and 8B payloads (2GB in total) and we report cycles spent per each tuple key lookup normalized to baseline code with uniform lookups. In all experiments, we pick the best configuration, which is $M = 15$ for GP and $M = 12$ SPP.

Figure 3.3 shows the results of the performance experiment. In the first part of the experiment (the first set of bars), we populate the hash table with uniformly distributed 2^{27} keys and we size the number of hash table buckets to contain exactly four nodes. This highly idealized setup may occur, for example, in hash joins when the join key is the primary key in the build relation and forms a strictly contiguous sequence of integers, and the hash function used for building is a simple modulo operation. We further assume *uniform traversals*, by allowing each lookup to reach the end of the link list in each bucket. Under these assumptions, each lookup visits all the nodes in its bucket, thus always requiring exactly four memory accesses ($N = 4$, similar to i_1 in Figure 3.1). We see that on uniform traversals, GP and SPP achieve an impressive speedup of 3.1x and 3.7x, respectively, over the baseline thanks to their ability to fully reach their memory-level parallelism potential. SPP performs 17% better than GP because it injects a new lookup continuously into the pipeline, as opposed to GP, which has to stall at the group boundary.

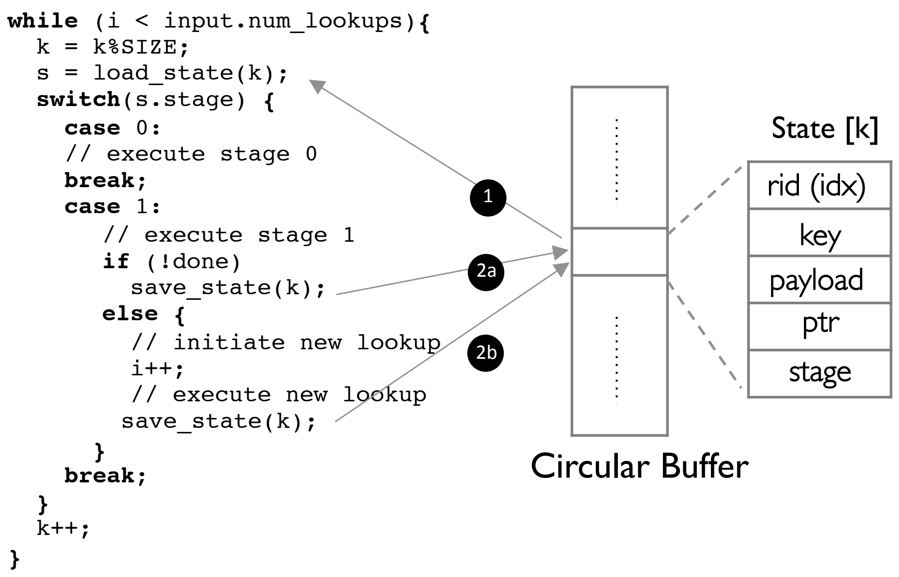
The second part of the experiment (the set of bars in the middle) is more realistic. We load the hash table of the same size with the same input data, however this time we relax the assumption that the build key has to be a contiguous sequence of integers in the build relation.

We also assume a non-trivial hash function that imperfectly distributes the sparse set of build keys accross the hash table, creating some variance in the occupancy across hash buckets. We also enable *non-uniform traversals* by allowing the lookups to terminate upon a key match. The average number of node traversals per each lookup is still four. Compared to the uniform lookups, the GP and SPP with irregular traversals are 1.6x and 1.8x worse than before in terms of nodes per cycle due to the wasted code stages on lookups that terminated early. The performance gap between SPP and GP decreases given that both techniques partially lost their ability to extract MLP.

In our final experiment, we build a hash table with 2^{27} build keys following a Zipf-skewed distribution (Zipf factor = .75). Therefore, some hash table buckets contain more nodes than the others, and the three most populous hash table buckets contains almost 3% of the total input keys. This situation may arise, for example, in hash joins when the join key is a non-unique attribute in the build relation. As a result, each lookup performs a *skewed traversal* meaning that each bucket has to be traversed until the end node, but the number of nodes per bucket varies. Similar to the two previous experiments, the average number of node traversals per each lookup is almost four. We observe that with skewed traversals, GP and SPP get 2.6x and 3.5x worse compared to the uniform lookup case, delivering virtually no improvement over the baseline. We also observe that GP outperforms SPP by 20%. Meanwhile, in the uniform traversal case, SPP was a better performer than GP. The inconsistent performance of existing techniques thus underscores the need for a robust software prefetching solution.

3.2 Asynchronous Memory Access Chaining

The main drawback of Group Prefetching and Software-Pipelined Prefetching is the static staging of all in-flight memory accesses. In effect, the set of lookups comprising these accesses are coupled within a group or pipeline, resulting in artificial inter-dependencies across the otherwise independent look-ups. This coupling is the reason for the lack of robustness in existing prefetching techniques in the face of irregularity in the data structure (e.g., unbalanced tree) or in the traversal path (e.g., early exit).



```
while (i < input.num_lookups){
  k = k%SIZE;
  s = load_state(k);
  switch(s.stage) {
    case 0:
    // execute stage 0
    break;
    case 1:
       // execute stage 1
       if (!done)
         save_state(k);
       else {
         // initiate new lookup
         i++;
         // execute new lookup
        save_state(k);
       }
    break;
  }
  k++;
}
```

Figure 3.4 – AMAC execution.

This work introduces Asynchronous Memory Access Chaining , a new prefetching scheme whose distinguishing feature is the ability to deal with irregular and divergent memory access patterns. AMAC accomplishes this by preserving the independence across look-ups, thus avoiding the coupling behavior that plagues existing techniques. Figure 3.2 shows a cartoon comparison of AMAC to the existing prefetching techniques.

3.2.1 Design Overview

The core idea of AMAC is to keep the full state of each in-flight memory access separate from that of other in-flight accesses. Whenever an access completes, a new one can be initiated in its place without any knowledge of the state of other accesses. If the completed memory access was the last one for a given lookup (e.g., in the case of a key match), a new look-up sequence can be started with similar ease and, again, without any regard for the state of other look-ups.

Figure 3.4 shows the key components of the proposed scheme in the context of a hash table probe in a hash-join operation. All in-flight requests are kept in a software-managed circular buffer, whose total number of entries is sufficient to cover the memory access latency. Once a lookup has been initiated, its state is saved in one entry of the circular buffer. This state,

Stage	Hash Join Probe	Next Stage
0	Get new tuple	
	Compute bucket addr.	1
1	Compare keys?	
	T: Output matches	0
	F: Next node?	
	T: Move to next node	1
	F: No match	0

Table 3.1 – Hash join (probe) code stages for AMAC.

comprised of the five fields shown in Figure 3.4 contains all the information necessary to continue and terminate the lookup. The key field is used for node comparisons. Upon a key match, the $rid(idx)$ and $payload$ fields are used for output materialization. The $stage$ field indicates the appropriate code stage to execute. Finally, ptr points to the node being prefetched but not yet visited. Using the combination of $stage$ and ptr fields, the exact status of each in-flight lookup is preserved.

To execute a code stage of a lookup, the first step is to dequeue a single in-flight request from the circular buffer. As a single in-flight request is dequeued from the circular buffer, the state of the lookup is loaded into the local variables of the software thread (step 1 in Figure 3.4). Once the state is loaded, the execution starts by jumping to the necessary code stage directed by the $stage$ information in the state entry. The execution stage retrieves the lookup key and the key is compared against the $ptr->key$ to determine the outcome of the stage. If the lookup is not completed (step 2a in Figure 3.4), a new memory prefetch is issued to the next data structure node and the state is updated with the address of the node that will be visited in the next stage. If the lookup is completed (step 2b in Figure 3.4), a new lookup is initiated by incrementing the index of the input array and the state is saved into the circular buffer entry after executing the necessary code to initiate a lookup. Then, the next entry is popped unless all the entries in the input array are consumed.

The pseudo-code of AMAC , shown in Listing 3.1, depicts a more realistic implementation of the scheme described above. The differences between our example in Figure 3.4 and Listing 3.1 are minor. One difference is, the modulo operation to access the circular buffer is

```
struct state_t {
int64_t idx;
int64_t key;
int64_t pload;
node_t  * ptr;
int32_t stage;
};
/* Hash table probe loop  */
void probe
  (table_t *input, hashtable_t *ht, table_t *out) {
state_t s[SIZE];
node_t  * n;
int32_t   k;
int32_t   i;
/* Prologue */
//...
/* Main loop */
while (i < input->num_keys){
  k = (k == (SIZE-1)) ? 0 : k;
  if (s[k].stage == 1){
    n   = s[k].ptr;
    /* Code 1: Output matches
    /* or visit next node  */
    if (n->key == s[k].key){
      out[s[k].idx] = n->pload;
      s[k].stage = 0;
    } else if (n->next){
      prefetch(n->next);
      s[k].ptr = n->next;
    } else {
      /* initiate new lookup (Code 0) */
    }
  } else if (s[k].stage == 0){
    /* Code 0: Hash input key, calculate bucket addr.  */
    int64_t hashed = HASH(input->tuple.key[i]);
    bucket_t * ptr = ht->buckets + hashed;
    /* Prefetch for next stage */
    prefetch(ptr);
    /* Update the state */
    s[k].idx   = ++i;
    s[k].key   = input->tuple.key[i];
    s[k].ptr   = ptr;
    s[k].stage = 1;
    /* Optionally fetch payload for emitting results  */
    s[k].pload = input->tuple.pload[i];
  }
  k++;
}
/* Epilogue */
//...
}
```

Listing 3.1 – AMAC hash table probe pseudo-code.

costly when the number of in-flight accesses is not a power of two, as a division instruction is required. Therefore, we implement a rolling counter that is reset to zero when it reaches the size of the buffer, which allows us to increment AMAC 's in-flight operations one by one. We also use the $if/else$ statements instead of a $switch/case$ statement and put the most commonly executed state upfront, which is *stage 1* in our example. This optimization possibly

Stage	Hash Join Build	Next Stage
0	Get new tuple	
	Compute bucket addr.	1
1	Latch?	
	T: Retry	1
	F: Node empty?	
	T: Insert tuple	0
	F: Next node?	
	T: Move to next node	2
	F: Get new node	3
2	Node empty?	
	T: Insert tuple	0
	F: Get new node	3
3	Insert tuple	0

Table 3.2 – Hash join (build) code stages for AMAC.

Stage	Group-by	Next Stage
0	Get new tuple	
	Compute bucket addr.	1
1	Latch?	
	T: Retry	1
	F: Compare keys?	
	T: Update tuple	3
	F: Next node?	
	T: Move to next node	1
	F: Get new node	2
2	Insert new tuple	3
3	Update the aggr. field	0

Table 3.3 – Group-by code stages for AMAC.

allows for executing fewer instructions in the cases where there are many stages. We also use additional state entries wherever is needed, which simplifies the code and in certain cases avoids re-execution of the same functionality.

Code Stages: The simplified code stages for hash join probe, hash join build, group-by, and binary search tree are depicted in Table 3.1, Table 3.2, Table 3.3, and Table 3.4 respectively. To identify the code stages, we analyze the baseline implementations and create the stages based on the pointer accesses. The entries in the table define the state transitions throughout the execution of the algorithm. The *Next Stage* field indicates which of the stages should be executed next based on the outcome(s) of the present stage. For example, the hash table probe stages depicted in Table 3.1 show that *stage 0* initiates a new lookup and access to data structure and key comparison happens in the next stage, which is *stage 1*. For simplicity, we

Stage	Binary Search Tree	Next Stage
0	Get new tuple Access root node	 1
1	Compare keys? T: Output result F: Move to next node	 0 1

Table 3.4 – Binary tree code stages for AMAC.

depict the hash join probe stages for unique keys, therefore upon a match in *stage 1*, a new lookup is initiated by transiting to *stage 0*. Other algorithms have similar stages and actions, one difference is the *latch?* action, which returns a *true* value when the latch is acquired by another lookup.

While accurate, the states we show in the table are simplified; for instance, an optimization not captured in the table is the following: in order to not lose an opportunity to initiate a new memory accesses, we merge the terminating stages of each lookup with the initial stage (for the next lookup) wherever it is applicable. Thus, when one lookup completes, a new lookup starts immediately (similar to our example in Figure 3.4), hence guaranteed a constant number of memory accesses in flight at all times. Similarly, in the data structures with latches, we might employ extra intermediate stages to avoid deadlocks during the lookups. An example of such implementation is the *stage 1* of group-by (not shown in Table 3.3), which is implemented as two different stages depending on whether the latch was already acquired for a given lookup or not.

Output order: Even though the lookup sequence does not follow the sequential order of row ids, the original order is preserved through the $rid(idx)$ field of the state. This ensures that results are materialized in the input order.

3.2.2 Handling Read/Write Dependencies

The baseline implementation of the hash join build and group-by contains a latch per bucket for updating the contents of the bucket. When the latch is not acquired, the thread spins on the latch until it gets it. Obviously, a spinlock will diminish the benefits of AMAC given that if one lookup cannot acquire the latch, there are still in-flight lookups pending.

When an AMAC thread executes a code stage, which requires acquiring a lock (e.g., *stage 1* of group-by and hash join build), we try to acquire the latch. If the attempt fails, we move on to the next lookup and retry when the same lookup is performed. As a result, we are still spinning on the latch but at a coarser-granularity. In the cases where, there is a probability of acquiring a lock but failing to complete the state (i.e., group-by insert), we employ an extra intermediate state to avoid any deadlocks (not shown in Table 3.3) as described in the previous subsection. Therefore, if one wants to run AMAC in a multi-threaded fashion, the spinlock should be replaced with an atomic swap instruction only. Moreover, in the highly contented cases, we spin on the dirty cache line as opposed to atomic instruction analogous to a Test and Test-and-set lock implementation. In the single-threaded runs, the same ideas apply but there is no need for an atomic instruction to acquire the latch.

3.3 Methodology

Workloads: For the all workloads evaluated in this work, we use 16-byte tuples containing an 8-byte key and an 8-byte payload, representative of an in-memory columnar database storage representation. In all the cases, the data structure nodes are aligned to 64-byte cache block boundary with the *aligned* attribute.

For the hash join workload, we adopt the highly optimized chained hash table implementation of Balkesen et al. [10, 11]. Each hash table bucket contains a 1-byte latch for synchronization, two 16-byte tuples and an 8-byte pointer to the next hash table node to be used in the case of collisions. For the uniform hash join scenario, we again use the no-partitioning hash join workload of Balkesen et al. [10, 11] with uniformly distributed random *R* and *S* relation keys following a foreign key relationship. In the uniform workload, the key value ranges are dense and when the sizes of *R* and *S* are equal both relations contain unique values given the foreign key relationship. In the case where the relation sizes are not equal, the *S* relation key range is restricted to the keys in R relation. As our study stresses the robustness of algorithms, we also relax the foreign key relationship and evaluate the case where R and S keys follow various Zipfian distributions, similar to prior studies [47, 5].

For the group-by workload, we extend the hash table used in hash join with an additional aggregation field in the hash table. The input relation contains uniformly distributed random keys, where each key appears three times. We also evaluate the Zipf-skewed key distributions of 0.5 and 1 [89]. The values (payloads) in the input relation always contains uniformly distributed random unique values. The values are aggregated with MIN and $COUNT$ functions, which are applied immediately upon a match in the hash table.

We use a canonical implementation of a binary search tree. We build the tree by using a uniformly distributed input relation with 8-byte integer keys. Each binary tree node contains an 8-byte key, an 8-byte payload and two 8-byte child pointers (i.e., left and right) and one 8-byte parent pointer. The probe relation contains uniformly distributed random unique keys. The probe relation size is always equal to the number of tree nodes, each of which finds a single match in the tree, resembling an indexed join scenario in data analytics workloads.

Experimental Setup: The server machines used in our experiments are shown in Table 3.5. The Intel Xeon x5670 server features a two-socket CPU with 6 cores per socket. We use just one socket in our experiments. The server runs Red Hat Linux (kernel version 2.6.32). On x86, we compile our code with `gcc 4.7.2` using `-O3` flag. On Oracle T4, we use a single 8-core CMP. The server runs Sun OS 5.11 and we compile our code with the `C compiler 5.13` found in `Oracle Solaris Studio 12.4`. In addition to the Oracle Solaris Studio compiler flags recommended by prior work [11], we use `-xprefetch=no%auto` option to disable the automatic generation of prefetch instructions by the compiler, which leads to overall performance improvements for all the evaluated techniques including the baseline code.

In all measurements, we use large VM pages, 2 MB on x86 and and 4 MB on SPARC. For all the prefetch-based techniques (GP, SPP, and AMAC), we parametrize the code and perform a sensitivity analysis to pick the best performing parameters for each experiment. For the

Table 3.5 – Architectural parameters.

Processor	**Intel Xeon 5670**	**Oracle SPARC T4**
Technology	32nm @ 2.93GHz	40nm @ 3GHz
ISA	x86	SPARC v9
CMP Features	6 cores / 12 Threads	8 cores / 64 Threads
Core Types	4-wide OoO	2-wide OoO
L1-I/D Caches (per core)	32KB	16KB
L2 Caches (per core)	256KB	128KB
L3 Cache	12MB	4MB
TLB entries (L1/L2)	64/512	128/-
Main Memory	24GB, DDR3	1TB, DDR3

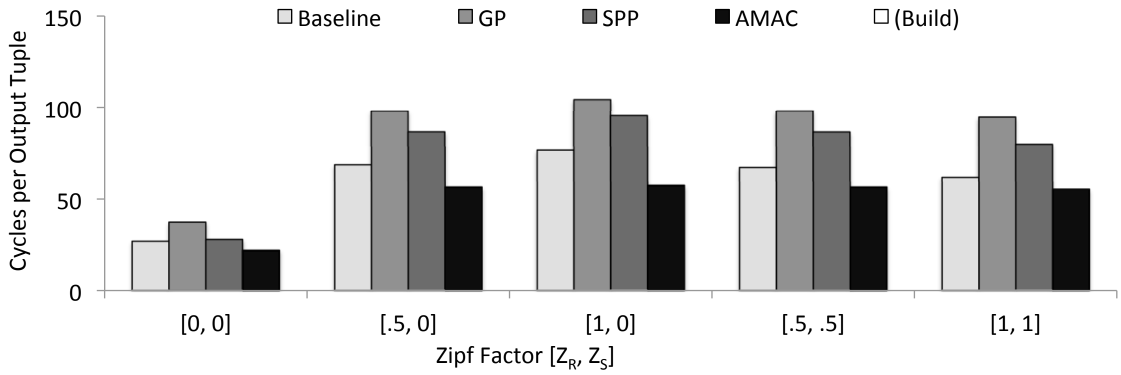
prefetch operations on x86, we use the `PREFETCHNTA` instruction via the built-in gcc functions. On SPARC, we use the strong prefetch variant [71]. In both platforms, prefetch instructions complete as long as a TLB miss does not cause a fault, which is rare with in-memory execution.

3.4 Evaluation

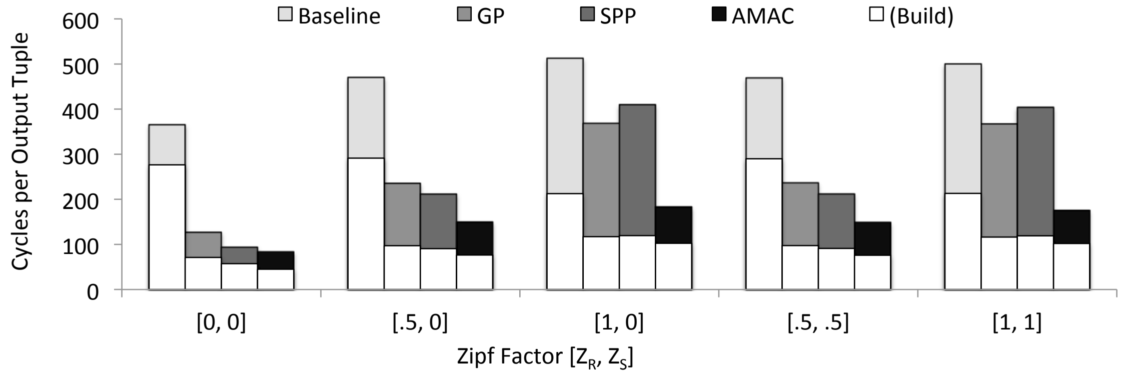
3.4.1 Hash Join

In order to analyze the performance implications of hash join with various dataset sizes, we keep the size of the probe relation constant ($|S| = 2^{27}$) and evaluate two different build relation sizes, *s*mall ($|R| = 2^{17}$) and *l*arge ($|R| = 2^{27}$), to study the behavior of the relative relation size difference and effect of on-chip locality. In addition, we evaluate skewed datasets, where the keys of R and S follow a Zipfian data distribution. The Zipf factor of each relation is denoted by $[Z_R, Z_S]$. We run the experiments on a single core and pick the best tuning parameter for all the techniques for each experiment.

Figure 3.5 depicts the cycles per output tuple for build and probe in hash join. We observe that for the join of the differently sized columns ($2MB \bowtie 2GB$), shown in Figure 3.5a, the build time is negligible and all the cycles are spent on probing the hash table, which fits in the



(a) Small build relation $2MB \bowtie 2GB$

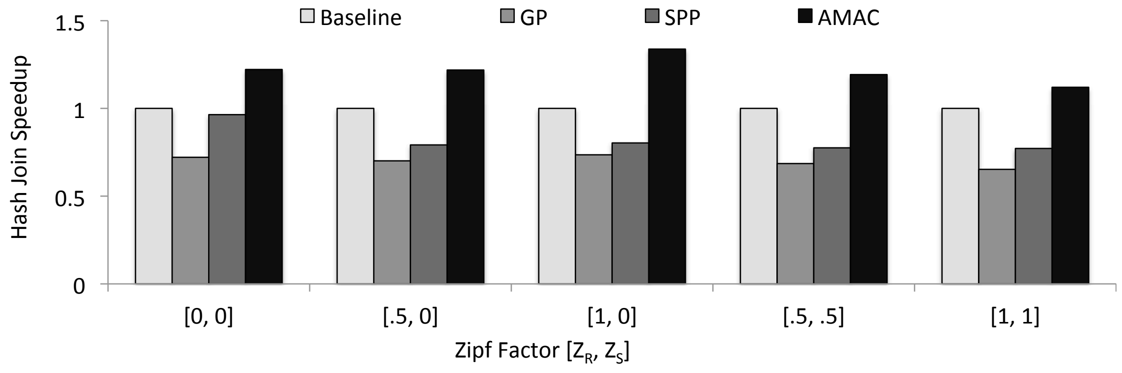


(b) Large build relation $2GB \bowtie 2GB$

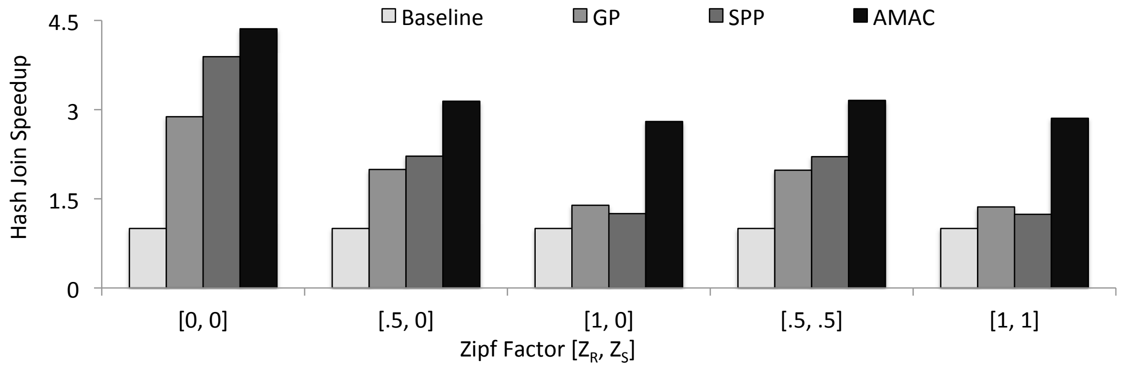
Figure 3.5 – Hash Join cycles breakdown over baseline hash join under different data distributions. Measurement on Xeon x5670.

last-level cache (LLC) of the Xeon processor. For the same reason, the skew in the build and probe relation keys does not have a significant impact on the execution cycles.

In contrast, in the hash join with equally sized relations ($2GB \bowtie 2GB$), shown in Figure 3.5b, the build cycles constitute half of the join cycles as the size of the build relation is beyond the LLC capacity. Similarly, analyzing the experiments with different Zipf-skewed R relation keys shows that the average number of probe cycles for GP and SPP increase by 1.8x and 2.4x respectively. In contrast, the probe cycles for the Zipf-skewed experiments of AMAC only shows an increase of 10% on average underscoring the robustness of AMAC under irregular data structure accesses. However, this is not the case for the build phase as the build operation inserts the build keys into the head of the bucket link list, which is a uniform operation regardless of the data distribution.



(a) Small build relation $2MB \bowtie 2GB$



(b) Large build relation $2GB \bowtie 2GB$

Figure 3.6 – Hash Join speedup over baseline hash join under different data distributions. Measurement on Xeon x5670.

Figure 3.6 shows the performance of the evaluated techniques normalized to the baseline. Figure 3.6a shows the performance of the small relation join ($2MB \bowtie 2GB$). We observe that the baseline code is faster than both GP and SPP by 32% on average, while AMAC outperforms the baseline by 22% .The root cause of this behavior is that the hash table fits in the last-level cache (LLC) of Xeon, therefore the core partially hides the LLC latency in the baseline case. Unfortunately, GP and SPP hurt the performance compared to the baseline. To investigate the cause of the GP and SPP slowdown, we perform a profiling analysis. Table 3.6 shows the number of instructions executed per output tuple and the performance obtained. We find that GP and SPP have a 2.5x and 1.9x overhead in the number of instructions per output tuple over the baseline code, therefore offset the benefits of prefetching. In contrast, AMAC has only a 1.6x overhead in the number of instructions, which explains the relatively better performance.

Table 3.6 – Execution profile of uniform ([0,0]) join with unequal table sizes ($2MB \bowtie 2GB$). Measurement on Xeon x5670 with x86 binary.

	Baseline	**GP**	**SPP**	**AMAC**
Instructions per Tuple	35	90	66	55
Cycles per Tuple	26	37	28	22

Figure 3.6b shows that, with equally sized relations ($2GB \bowtie 2GB$), all three techniques, GP, SPP and AMAC achieve significant speedups (2.8x, 3.9x, and 4.3x respectively) under uniform relations ([0,0]) as they all effectively hide the memory latency. It is important to note that the 25% performance gap between AMAC and SPP in the previous unequally sized join is bridged as both techniques simply hit the limit of the memory-level parallelism provided by the hardware. However, for the skewed R ([.5,0], [1,0]), GP and SPP lose their effectiveness at generating memory-level parallelism due to the irregular traversal paths in the hash table and deliver average speedups ranging from 1.3x to 2.0x and 1.2x to 2.2x respectively. As expected, AMAC gracefully handles the divergence in the hash table walk and achieves a robust performance of 3.2x on average for all the cases. Moreover, adding skew to the relation S ([.5,.5], [1,1]), which can help with the locality of the hash table buckets and nodes, has a minor impact on the performance as the hash table working set size is too big to be captured in the LLC even when both relations are skewed.

Figure 3.7 depicts the execution cycle sensitivity to the tuning parameters of GP, SPP, and AMAC . For all the techniques, we vary the parameters that increase the number of parallel lookups performed within a thread. For the uniform probes ([0,0]), we observe that increasing the number of parallel lookups lowers the cycles-per-tuple due to the increase in memory-level parallelism and, in general, ten in-flight lookups deliver the best performance except for GP. Further increasing the number of in-flight requests does not improve the performance of SPP and AMAC as the limit of L1-D outstanding misses (i.e., 10 L1-D MSHRs) is reached on the Xeon core [43]. For GP, (Figure 3.7a), the best performance is achieved with a group size of 15 due to the fact that GP is limited by the instruction-count overhead, instead of the number of MSHRs, and larger group sizes yield fewer outer-loop iterations helping slightly with reducing the instruction-count overhead.

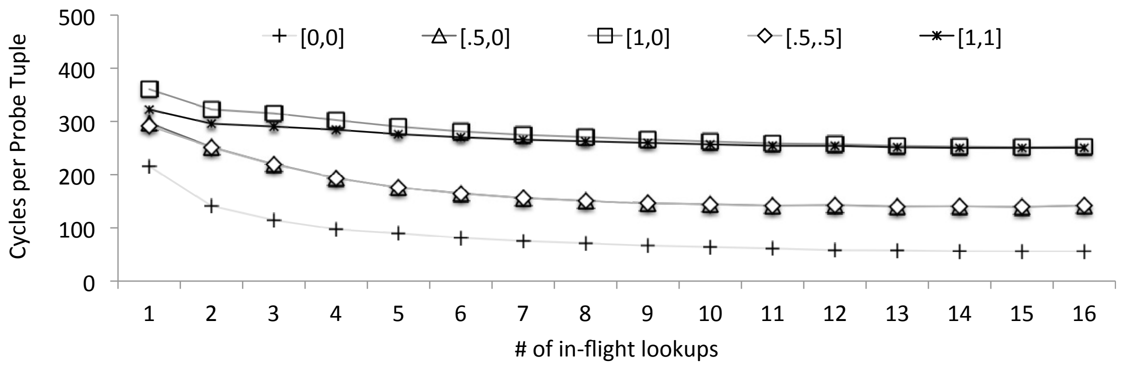
For the skewed data distributions, we observe that GP (Figure 3.7a) and SPP (Figure 3.7b) have limited benefits from multiple parallel lookups as the skewed key distribution leads to buckets with long pointer chains, which cannot be handled by GP and SPP. Especially in the cases where $Z_R = 1$, the performance difference of the single in-flight vs. the best case is only 22%. In contrast, AMAC (Figure 3.7c) is robust to various data distributions and handles the non-uniform cases without incurring any additional performance overhead.

3.4.2 Scalability Analysis

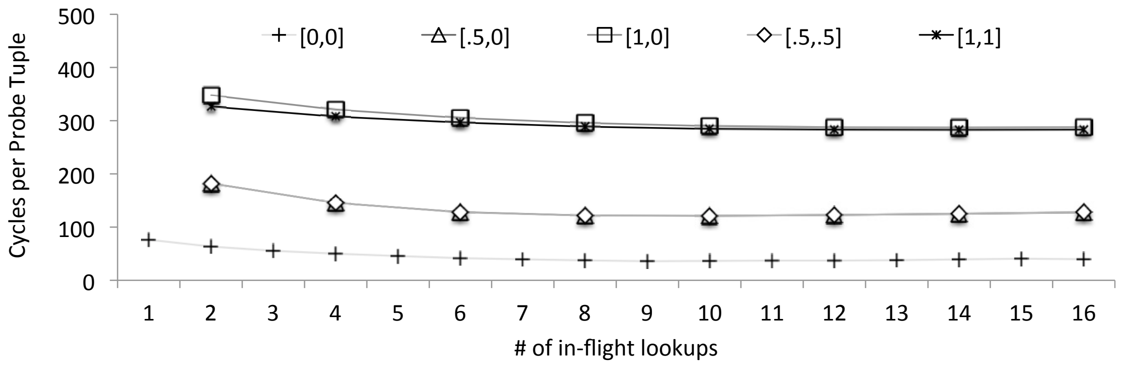
In this section, we study the scalability of AMAC , which delivers impressive speedups, but also stresses the memory subsystem of the machines. To mitigate any algorithm-related scalability issues, we only focus on the read-only probe phase of hash join. We report the probe throughput, which is calculated as $|S|/probeExecutionTime$ on the Xeon and T4 machines. On both platforms, we perform the experiment by assigning software threads first to physical cores (six on Xeon and eight on T4) and once we run out of physical cores, we start using the SMT threads.

The results on Xeon with non-skewed data, shown in Figure 3.8a, indicate that GP, SPP, and AMAC throughputs scale well up to four threads. However, the throughputs start leveling off after four cores and increasing the number of contexts does not result in any significant improvement. Meanwhile, the baseline algorithm achieves better scalability and brings the initial 2.5x throughput gap down to 80% by taking advantage of all the hardware contexts on Xeon. In contrast, the same experiment on T4, shown in Figure 3.9a, shows that GP, SPP, and AMAC scale well with the available physical cores (eight) and even benefit from SMT threads moderately.

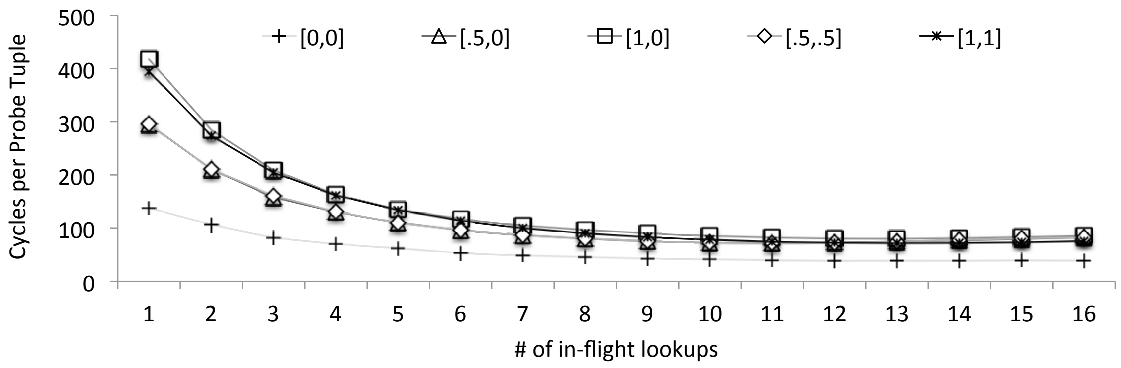
For the skewed experiments on Xeon (Figure 3.8b and Figure 3.8c), we see a similar trend for AMAC but for the others the performance drops. As a result, the scalability gets better as the pressure on the hardware resources decreases. On T4 (Figure 3.9b and Figure 3.9c), we see that increasing the skew severely hurts the performance of GP and SPP. Up to eight



(a) GP



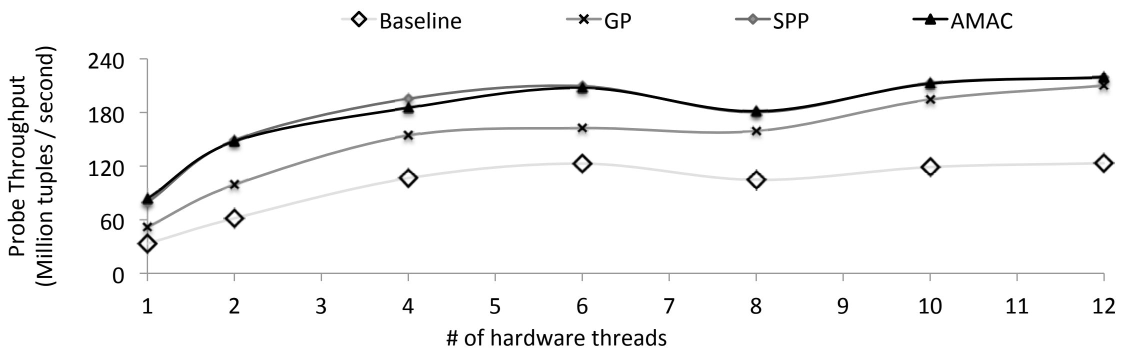
(b) SPP



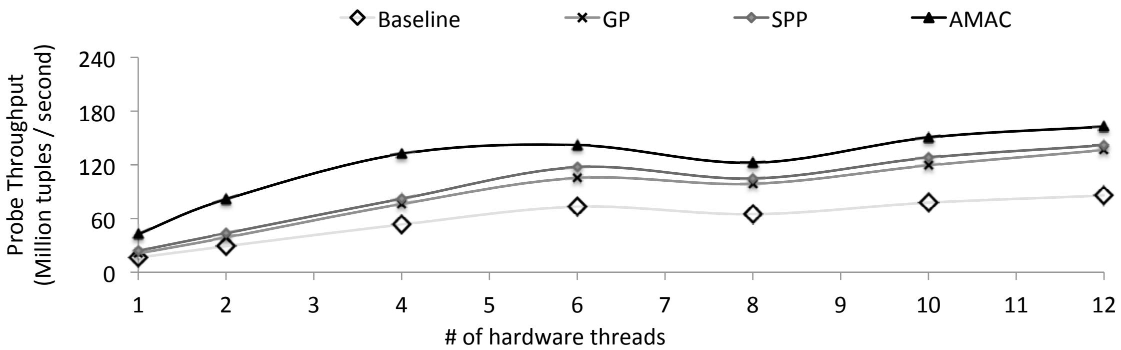
(c) AMAC

Figure 3.7 – Probe performance sensitivity to the tuning parameters of GP, SPP and AMAC ($2GB \bowtie 2GB$).

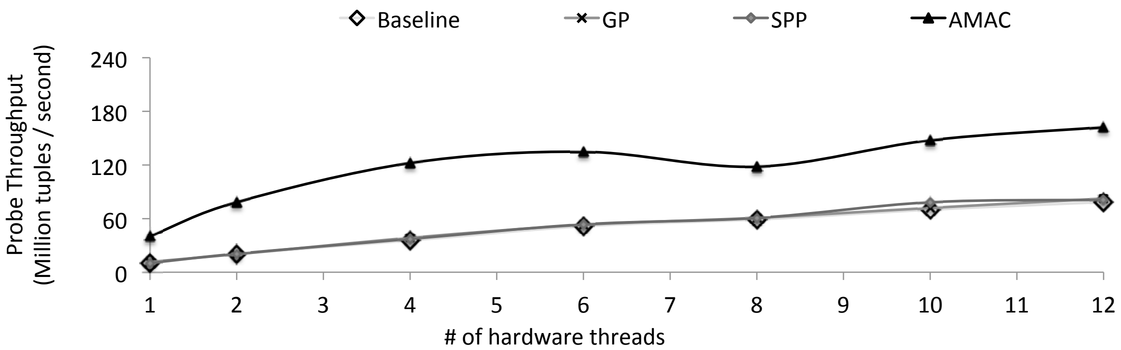
cores, the performance of SP and GPP are almost comparable to the baseline. However, beyond eight threads (i.e., SMT threads) GP and SPP yield an inferior performance to the baseline. An important trend in all of these T4 experiments is the fact that AMAC achieves better performance than the other techniques while the performance scales almost linearly with physical cores.



(a) $[0,0]$



(b) $[.5,.5]$

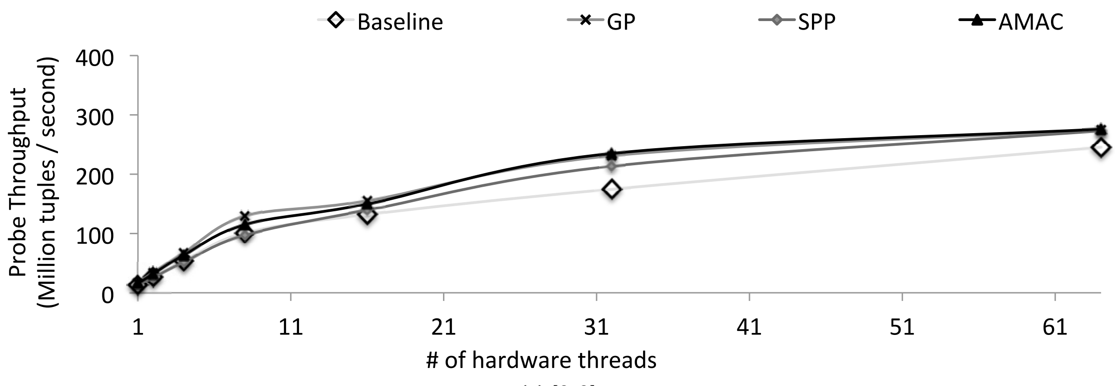


(c) $[1,1]$

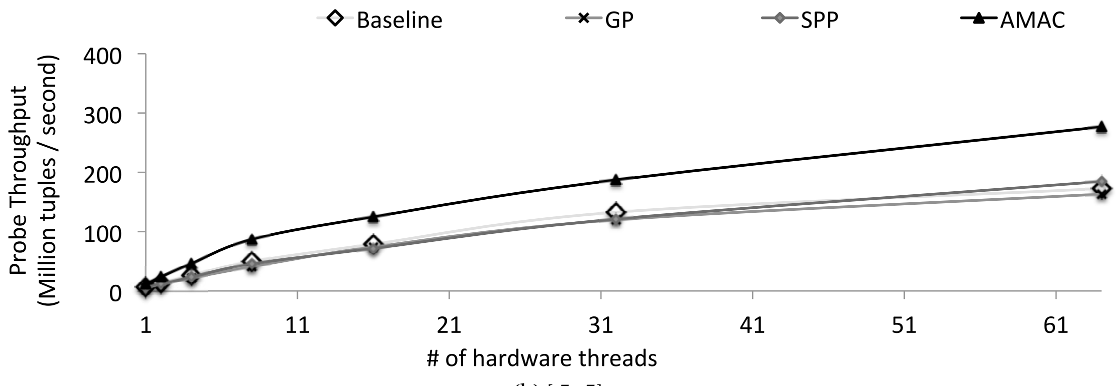
Figure 3.8 – Scalability of the hash table probes with uniform and Zipf-skewed keys ($2GB \bowtie 2GB$). Measurement on Xeon x5670.

3.4.3 Group-By

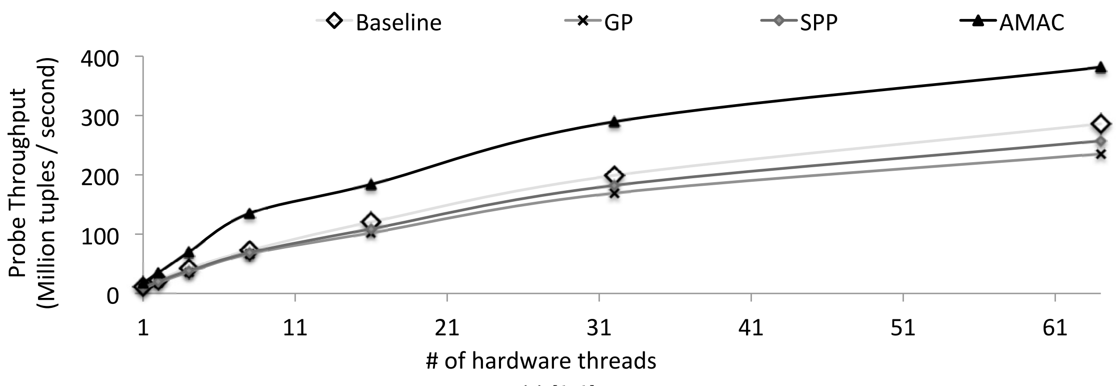
Figure 3.10 shows the group-by speedup for different input relation sizes. We observe that, for the small relation (2^{17}), the GP performance and SPP performance can be equal to or worse than the baseline but deliver 14% and 15% better performance on average respectively. In contrast, AMAC improves provides speedups ranging from 1.3x to 2x. In the big relation (2^{27}),



(a) [0, 0]



(b) [.5, .5]



(c) [1, 1]

Figure 3.9 – Scalability of the hash table probes with uniform and Zipf-skewed keys ($2GB \bowtie 2GB$). Measurement on Oracle T4.

GP and SPP deliver 2.5x and 2.75x speedup respectively and AMAC yields 3.2x speedup on average. It is important to note that, in the uniform case, SPP's performance is 7% lower than AMAC and higher than GP by 8%. Under the skewed input, SPP maintains an average of 10% higher performance than GP but achieves 20% lower performance compared to AMAC as

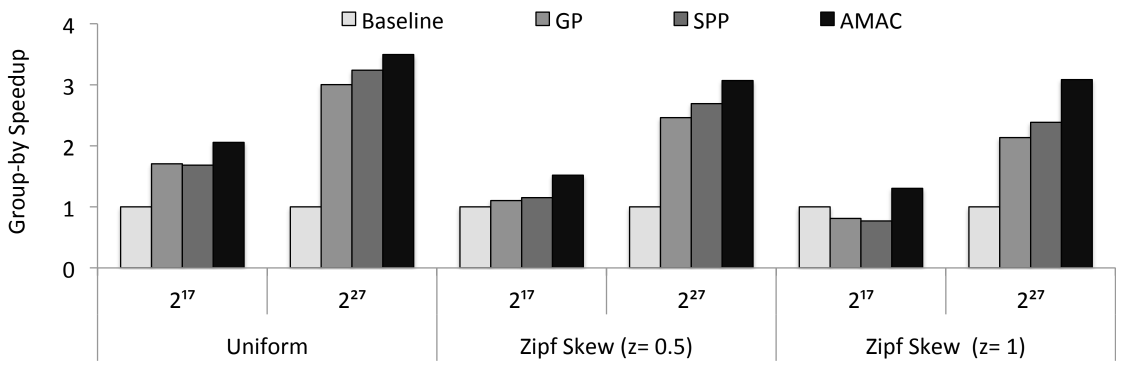


Figure 3.10 – Performance comparison of group-by operator with an input relation keys following a uniform and Zipf-skewed keys. Measurement on Xeon x5670.

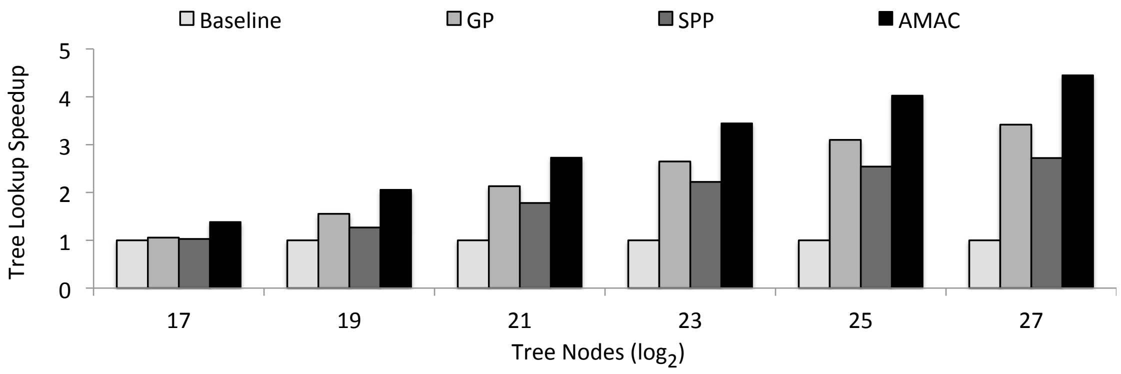


Figure 3.11 – Unbalanced binary search tree lookup performance. Measurement on Xeon x5670.

the skew causes read/write dependencies within the pipeline, therefore extra code should be executed to serialize the conflicting accesses.

3.4.4 Tree Search

Figure 3.11 depicts our results for unbalanced binary search tree. In general, the benefit of all prefetching techniques compared to the baseline increases with the height of the tree, as the baseline fails to generate MLP on long pointer chains. We observe that AMAC achieves an average speedup of 2.8x (geomean) over the baseline implementation, compared to 2.1x and 1.8x for GP and SPP, respectively.

We also note that in contrast to the group-by case, where SPP outperforms GP by up to 11%, in the unbalanced tree search GP performs 18% better than SPP on average. The reason for the poor performance of SPP in tree search is the loss of MLP in case of bailouts, which occur for the longest traversals. While SPP's pipeline can be stretched to match the height of the tree, in certain cases we found that the average performance attained in this configuration is inferior to that with a slightly shorter pipeline that favors the common-case (i.e., average) traversal length but incurs an occasional bailout. GP, however, never bails out because a group is considered done when all the lookups within that group terminates. As a result, although GP also loses the opportunity to achieve maximum MLP in the case of an irregularity, it can still overlap the remaining lookups in a group to extract the limited available parallelism.

3.5 Discussion

Other workloads and data structures: We evaluated AMAC in the context of analytical database systems with abundant inter-lookup parallelism and optimized data structures [38]. However, bulk accesses to data structures is also a common operation in other parts of the computing stack such as network layer. Similarly, in certain cases data structure lookups can contain complex and/or recursive function calls. Even in the cases where the bulk lookups are executed in small batches (i.e., 32, 64 lookups at a time) and contain function calls, AMAC can be beneficial as long as the available lookups suffer from long-latency memory misses.

AMAC automation: In this work, we manually created AMAC stages and implemented state save and restore functionality. Ideally this process should be automated and hidden from the software developer. We believe that event-driven programming language concepts such as *coroutines* that allow for cooperative multitasking (e.g., escape-and-reenter loops) within a single-thread can help creating a generalized software model and framework for hiding long-latency memory accesses. The benefit of such framework includes minimal modifications to baseline code, easier programmability, and portability across platforms. The disadvantages can be the user-land threads' state maintenance and space overhead, which is an overkill as the thread state carries a lot of redundancy across the threads of the same data structure lookup.

3.6 AMAC Summary

This chapter introduced Asynchronous Memory Access Chaining, a new approach for achieving high MLP on pointer-intensive operations with irregular behavior across lookups. AMAC is effective in managing irregularity by maintaining the state of each lookup separately from other in-flight requests. This separation allows AMAC to react to the needs of each individual lookup, such as executing more or fewer memory accesses than the common case, without affecting the execution of other lookups.

AMAC achieves a competitive 4.3x speed-up over the no-prefetching hash join baseline for uniform lookups. Moreover, AMAC is robust and maintains its performance advantage in the presence of irregular accesses. In the hash join workloads with irregular accesses and AMAC improves the performance by 3x over the no-prefetch baseline and by 1.8x over the existing techniques.

4 Widx: On-chip Accelerator for Index Traversals

The rapid growth in data volumes necessitates a corresponding increase in compute resources to extract and serve the information from the raw data. Meanwhile, technology trends show a major slowdown in supply voltage scaling, which has historically been the primary mechanism for lowering the energy per transistor switching event. Constrained by energy at the chip level, architects have found it difficult to leverage the growing on-chip transistor budgets to improve the performance of conventional processor architectures. As a result, an increasing number of proposals are calling for specialized on-chip hardware to increase performance and energy efficiency in the face of dark silicon [24, 34]. Two critical challenges in the design of such dark silicon accelerators are: (1) identifying the codes that would benefit the most from acceleration by delivering significant value for a large number of users (i.e., maximizing utility), and (2) moving just the right functionality into hardware to provide significant performance and/or energy efficiency gain without limiting applicability (i.e., avoiding over-specialization).

This chapter proposes *Widx*, an on-chip accelerator for database hash index lookups. Hash indexes are fundamental to modern database management systems (DBMSs) and are widely used to convert linear-time search operations into near-constant-time ones. In practice, however, the sequential nature of an individual hash index lookup, composed of key hashing followed by pointer chasing through a list of nodes, results in significant time constants even in highly tuned in-memory DBMSs. Consequently, a recent study of data analytics on a state-of-the-art commercial DBMS found that 41% of the total execution time for a set of TPC-H queries goes to hash index lookups used in hash-join operations [35].

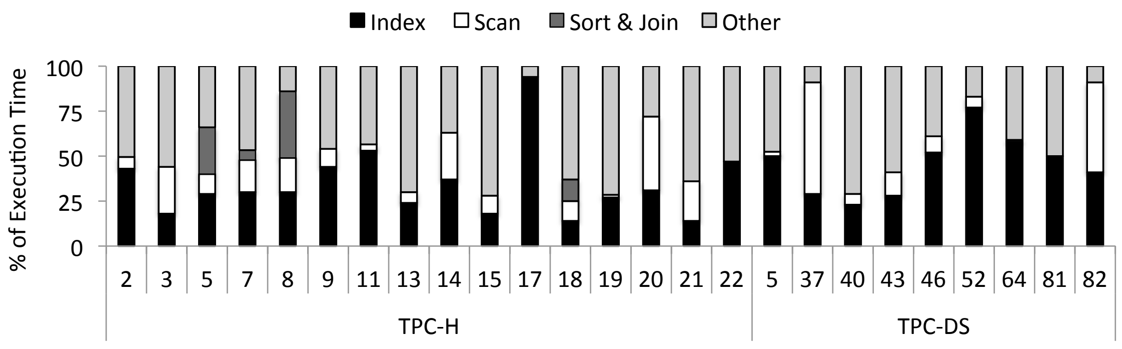
By accelerating hash index lookups, a functionality that is essential in modern DBMSs, Widx ensures high utility. Widx maximizes applicability by supporting a variety of schemas (i.e., data layouts) through limited programmability. Finally, Widx improves performance and offers high energy efficiency through simple parallel hardware.

The rest of this chapter is organized as follows. Section 4.1 motivates our focus on database index traversals as a candidate for acceleration. Section 4.2 presents an analytical model for finding practical limits to acceleration in index traversals. Section 4.3 describes the Widx architecture. Sections 4.4 and 4.5 present the evaluation methodology and results, respectively. Sections 4.6 and 4.7 discuss additional issues and concludes the chapter.

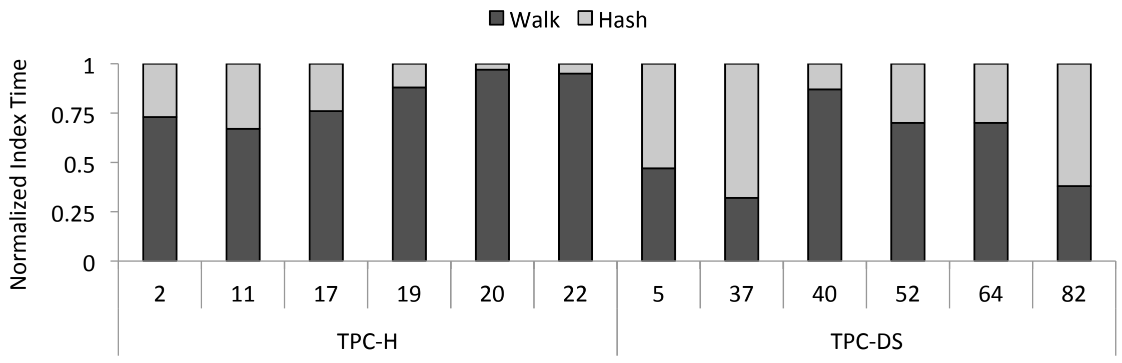
4.1 Profiling Analysis of a Modern DBMS

In order to understand the chief contributors to the execution time in database workloads, we study MonetDB [38], a popular in-memory DBMS designed to take advantage of modern processor and server architectures through the use of column-oriented storage and cache-optimized operators. We evaluate Decision Support System (DSS) workloads on a server-grade Xeon processor with TPC-H [78] and TPC-DS [59] benchmarks. Both DSS workloads were set up with a 100GB dataset. Experimental details are described in Section 4.4.

Figure 4.1a shows the total execution time for a set of TPC-H and TPC-DS queries. The TPC-H queries spend up to 94% (35% on average) and TPC-DS queries spend up to 77% (45% on average) of their execution time on indexing. Indexing is the single dominant functionality in these workloads, followed by scan and coupled sort&join operations. The rest of the query execution time is fragmented among a variety of tasks, including aggregation operators (e.g., sum, max), library code, and system calls.



(a) Total execution time breakdown



(b) Index execution time breakdown

Figure 4.1 – TPC-H & TPC-DS query execution time breakdown on MonetDB.

To gain insight into where the time goes in the indexing phase, we profile the index-dominant queries on a full-system cycle-accurate simulator (details in Section 4.4). We find that hash table lookups account for nearly all of the indexing time, corroborating earlier research [35]. Figure 4.1b shows the normalized hash table lookup time, broken down into its primitive operations: key hashing (Hash) and node list traversal (Walk). In general, node list traversals dominate the lookup time (70% on average, 97% max) due to their long-latency memory accesses. Key hashing contributes an average of 30% to the lookup latency; however, in cases when the index table exhibits high L1 locality (e.g., queries 5, 37, and 82), over 50% (68% max) of the lookup time is spent on key hashing.

Summary: Indexes are an essential database management system functionality that speeds up accesses to data through hash table lookups and is responsible for up to 94% of the query execution time. While the bulk of the index time is spent on memory-intensive node list traversals, key hashing contributes 30% on average, and up to 68%, to each index operation.

Due to its significant contribution to the query execution time, index traversals presents an attractive target for acceleration; however, maximizing the benefit of an index traversal accelerator requires accommodating both key hashing and node list traversal functionalities.

4.2 Database Index Traversal Acceleration

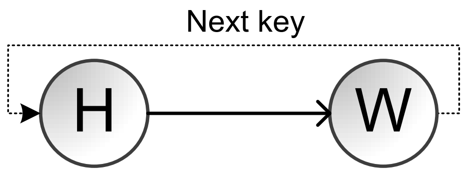
4.2.1 Overview

Figure 4.2 highlights the key aspects of our approach to index traversal acceleration. These can be summarized as (1) walk multiple hash buckets concurrently with dedicated *walker* units, (2) speed up bucket accesses by decoupling key hashing from the walk, and (3) share the hashing hardware among multiple walkers to reduce hardware cost. We next detail each of these optimizations by evolving the baseline design (Figure 4.2a) featuring a single hardware context that sequentially executes the code in Listing 2.1 with no special-purpose hardware.

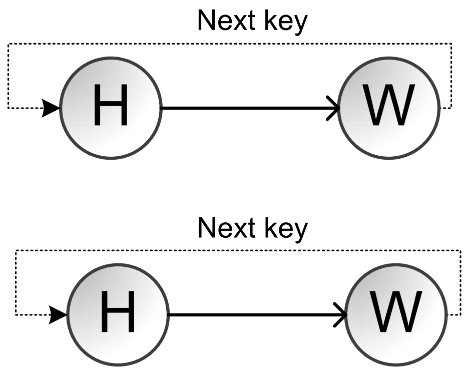
The first step, shown in Figure 4.2b, is to accelerate the node list traversals that tend to dominate the index traversal time. While each traversal is fundamentally a set of serial node accesses, we observe that there is an abundance of inter-key parallelism, as each individual key lookup can proceed independently of other keys. Consequently, multiple hash buckets can be walked concurrently. Assuming a set of parallel walker units, the expected reduction in index traversal time is proportional to the number of concurrent traversals.

The next acceleration target is key hashing, which stands on the critical path of accessing the node list. We make a critical observation that because index operations involve multiple independent input keys, key hashing can be decoupled from bucket accesses. By overlapping the node walk for one input key with hashing of another key, the hashing operation can be removed from the critical path, as depicted in Figure 4.2c.

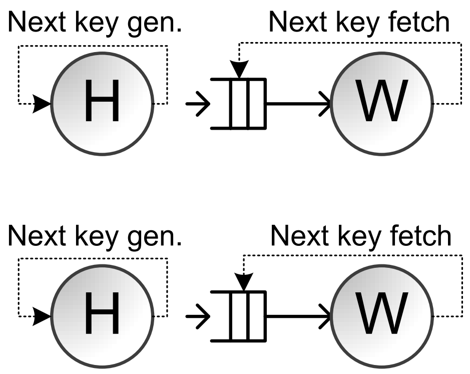
Finally, we observe that because the hashing operation has a lower latency than the list traversal, the hashing functionality can be shared across multiple walker units as a way of reducing cost. We refer to a decoupled hashing unit shared by multiple cores as a *dispatcher* and show this design point in Figure 4.2d.



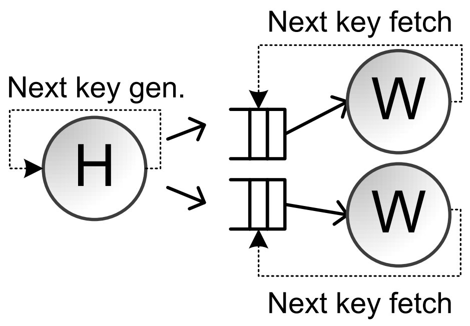
(a) Baseline design



(b) Parallel walkers



(c) Parallel walkers each with a decoupled hashing unit



(d) Parallel walkers with a shared decoupled hashing unit

Figure 4.2 – Baseline and accelerated index traversal hardware.

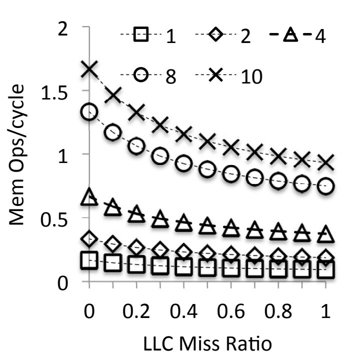
4.2.2 First-Order Performance Model

An index operation may touch millions of keys, offering enormous inter-key parallelism. In practice; however, parallelism is constrained by hardware and physical limitations. We thus need to understand the practical bottlenecks that may limit the performance of the index traversal accelerator outlined in Section 4.2.1. We consider an accelerator design that is tightly coupled to the core and offers full offload capability of the index traversal functionality, meaning that the accelerator uses the core's TLB and L1-D, but the core is otherwise idle whenever the accelerator is running.

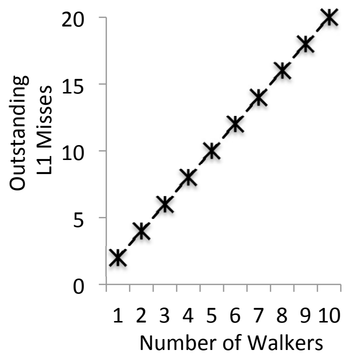
We study three potential obstacles to performance scalability of a multi-walker design: (1) L1-D bandwidth, (2) L1-D MSHRs, and (3) off-chip memory bandwidth. The performance-limiting factor of the three elements is determined by the rate at which memory operations are generated at the individual walkers. This rate is a function of the average memory access time (AMAT), memory-level parallelism (MLP, i.e., the number of outstanding L1-D misses), and the computation operations standing on the critical path of each memory access. While the MLP and the number of computation operations are a function of the code, AMAT is affected by the miss ratios in the cache hierarchy. For a given cache organization, the miss ratio strongly depends on the size of the hash table being probed.

Our bottleneck analysis uses a simple analytical model following the observations above. We base our model on the accelerator design with parallel walkers and decoupled hashing units (Figure 4.2c) connected via an infinite queue. The index lookup code, MLP analysis, and required computation cycles are based on Listing 2.1. We assume 64-bit keys, with eight keys per cache block. The first key to a given cache block always misses in the L1-D and LLC and goes to main memory. We focus on hash tables that significantly exceed the L1 capacity; thus, node pointer accesses always miss in the L1-D, but they might hit in the LLC. The LLC miss ratio is a parameter in our analysis.

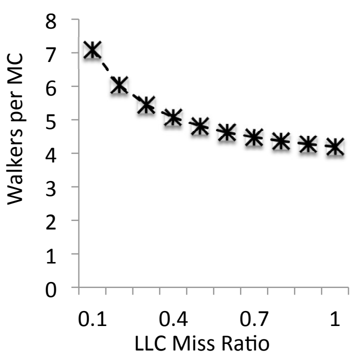
L1-D bandwidth: The L1-D pressure is determined by the rate at which key and node accesses are generated. First, we calculate the total number of cycles required to perform a fully



(a) Constraint: L1-D bandwidth

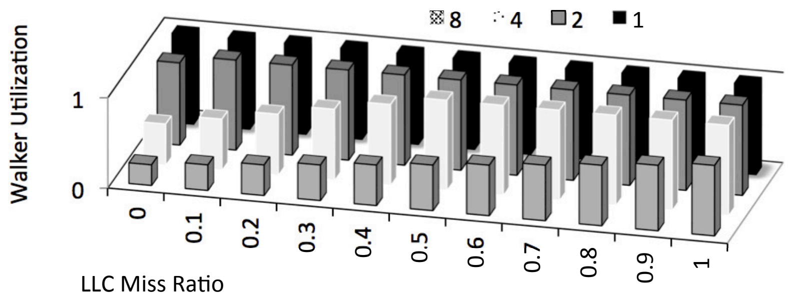


(b) Constraint: L1-D MSHRs

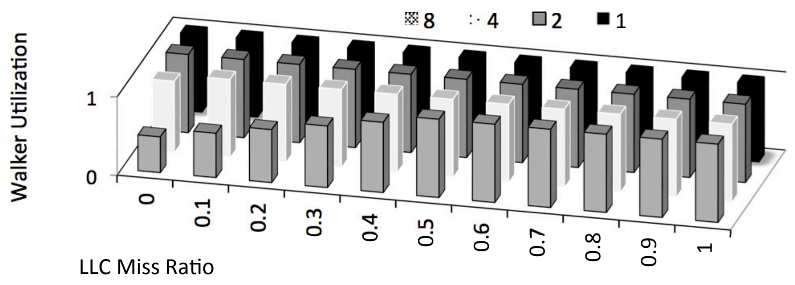


(c) Constraint: memory bandwidth

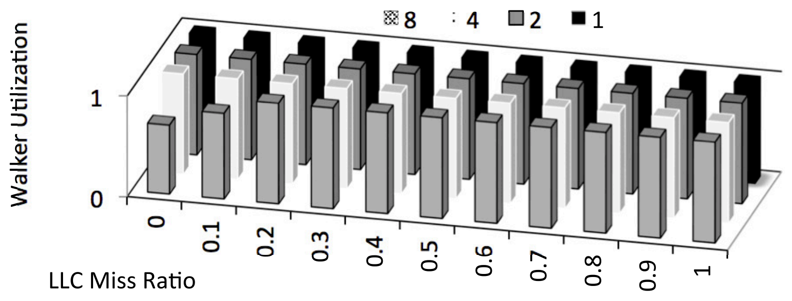
Figure 4.3 – Accelerator bottleneck analysis.



(a) 1 node per bucket



(b) 2 nodes per bucket



(c) 3 nodes per bucket

Figure 4.4 – Number of walkers that can be fed by a dispatcher as a function of bucket size and LLC miss ratio.

pipelined probe operation for each step (i.e., hashing one key or walking one node in a bucket). Equation 4.1 shows the cycles required to perform each step as the sum of memory and computation cycles. As hashing and walking are different operations, we calculate the same metric for each of them (subscripted as H and W).

Equation 4.2 shows how the L1-D pressure is calculated in our model. In the equation, N defines the number of parallel walkers each with a decoupled hashing unit. $MemOps$ defines the L1-D accesses for each component (i.e., hashing one key and walking one node) per operation. As hashing and walking are performed concurrently, the total L1-D pressure is calculated by the addition of each component. We use a subscripted notation to represent the addition; for example: $(X)_{H,W} = (X)_H + (X)_W$.

$$Cycles = AMAT * MemOps + CompCycles \tag{4.1}$$

$$MemOps/cycle = (\frac{MemOps}{Cycles})_{H,W} * N \quad \leq L1ports \tag{4.2}$$

Figure 4.3a shows the L1-D bandwidth requirement as a function of the LLC miss ratio for a varying number of walkers. The number of L1-D ports (typically 1 or 2) limits the L1 accesses per cycle. When the LLC miss ratio is low, a single-ported L1-D becomes the bottleneck for more than six walkers. However, a two-ported L1-D can comfortably support 10 walkers even at low LLC miss ratios.

MSHRs: Memory accesses that miss in the L1-D reserve an MSHR for the duration of the miss. Multiple misses to the same cache block (a common occurrence for key fetches) are combined and share an MSHR. A typical number of MSHRs in the L1-D is 8-10; once these are exhausted, the cache stops accepting new memory requests. Equation 4.3 shows the relationship between the number of outstanding L1-D misses and the maximum MLP the hashing unit and walker can together achieve during a decoupled hash and walk.

$$L1_{Misses} = max(MLP_H + MLP_W) * N \quad \leq MSHRs \tag{4.3}$$

Based on the equation, Figure 4.3b plots the pressure on the L1-D MSHRs as a function of the number of walkers. As the graph shows, the number of outstanding misses (and correspondingly, the MSHR requirements) grows linearly with the walker count. Assuming 8 to 10 MSHRs in the L1-D, corresponding to today's cache designs, the number of concurrent walkers is limited to four or five, respectively.

Off-chip bandwidth: Today's server processors tend to feature a memory controller (MC) for every 2-4 cores. The memory controllers serve LLC misses and are constrained by the available off-chip bandwidth, which is around 12.8GB/s with today's DDR3 interfaces. A unit of transfer is a 64B cache block, resulting in nearly 200 million cache block transfers per second. We express the maximum off-chip bandwidth per memory controller in terms of the maximum number of 64-byte blocks that could be transferred per cycle. Equation 4.4 calculates the number of blocks demanded from the off-chip per operation (i.e., hashing one key or walking one node in a bucket) as a function of L1-D and LLC miss ratio ($L1_{MR}$, LLC_{MR}) and memory operations. Equation 4.5 shows the model for computing memory bandwidth pressure, which is expressed as the ratio of the expected MC bandwidth in terms of blocks per cycle (BW_{MC}) and the number of demanded cache blocks from the off-chip memory per cycle. The latter is calculated for each component (i.e., hashing unit and walker).

$$OffChipDemands = L1_{MR} * LLC_{MR} * MemOps \tag{4.4}$$

$$WalkersPerMC \leq \frac{BW_{MC}}{(\frac{OffChipDemands}{Cycles})_{H,W}} \tag{4.5}$$

Figure 4.3c shows the number of walkers that can be served by a single DDR3 memory controller providing 9GB/s of effective bandwidth (70% of the 12.8GB/s peak bandwidth [20]). When LLC misses are rare, one memory controller can serve almost eight walkers, whereas at high LLC miss ratios, the number of walkers per MC drops to four. However, our model assumes an infinite buffer between the hashing unit and the walker, which allows the hashing unit to increase the bandwidth pressure. In practical designs, the bandwidth demands from the hashing unit will be throttled by the finite-capacity buffer, potentially affording more walkers within a given memory bandwidth budget.

Dispatcher: In addition to studying the bottlenecks to scalability in the number of walkers, we also consider the potential of sharing the key hashing logic in a dispatcher-based configuration shown in Figure 4.2d. The main observation behind this design point is that the hashing functionality is dominated by ALU operations and enjoys a regular memory access pattern with high spatial locality, as multiple keys fit in each cache line in column-oriented databases. Meanwhile, node accesses launched by the walkers have poor spatial locality but also have minimal ALU demands. As a result, the ability of a single dispatcher to keep up with multiple walkers is largely a function of (1) the hash table size, and (2) hash table bucket depth (i.e., the number of nodes per bucket). The larger the table, the more frequent the misses at lower levels of the cache hierarchy, and the longer the stall times at each walker. Similarly, the deeper the bucket, the more nodes are traversed and the longer the walk time. As walkers stall, the dispatcher can run ahead with key hashing, allowing it to keep up with multiple walkers. This intuition is captured in Equation 4.6. Total cycles for dispatcher and walkers is a function of AMAT (Equation 4.1). We multiply the number of cycles needed to walk a node by the number of nodes per bucket to compute the total walking cycles required to locate one hashed key.

$$WalkerUtilization = \frac{Cycles_{node} * Nodes/bucket}{Cycles_{hash} * N} \tag{4.6}$$

Based on Equation 4.6, Figure 4.4 plots the effective walker utilization given one dispatcher and a varying number of walkers (N). Whenever a dispatcher cannot keep up with the walkers, the walkers stall, lowering their effective utilization. The number of nodes per bucket affects the walkers' rate of consumption of the keys generated by the dispatcher; buckets with more nodes take longer to traverse, lowering the pressure on the dispatcher. The three subfigures show the walker utilization given 1, 2, and 3 nodes per bucket for varying LLC miss ratios. As the figure shows, one dispatcher is able to feed up to four walkers, except for very shallow buckets (1 node/bucket) with low LLC miss ratios.

4.3 Widx

4.3.1 Architecture Overview

Figure 4.5 shows the high-level organization of our proposed index traversal acceleration widget, Widx, which extends the decoupled accelerator in Figure 4.2d. The Widx design is based on three types of units that logically form a pipeline:
(1) a dispatcher unit that hashes the input keys,
(2) a set of walker units for traversing the node lists, and
(3) an output producer unit that writes out the matching keys and other data as specified by the index function.

To maximize concurrency, the units operate in a decoupled fashion and communicate via queues. Data flows from the dispatcher toward the output producer. All units share an interface to the host core's MMU and operate within the active application's virtual address space. A single output producer generally suffices as store latency can be hidden and is not on the critical path of hash table probes.

A key requirement for Widx is the ability to support a variety of hashing functions, database schemas, and data types. As a result, Widx takes the programmable (instead of fixed-function) accelerator route. In designing the individual Widx units (dispatcher, walker, and output producer), we observe significant commonality in the set of operations that each must support. These include the ability to do simple arithmetic (e.g., address calculation), control flow, and to access memory via the MMU.

Based on these observations, each Widx unit employs a custom RISC core featuring a minimalistic ISA shown in Table 4.1. In addition to the essential RISC instructions, the ISA also includes a few unit-specific operations to accelerate hash functions with fused instructions (e.g., xor-shift) and an instruction to reduce memory time (i.e., touch) by demanding data blocks in advance of their use. This core serves as a common building block for each Widx

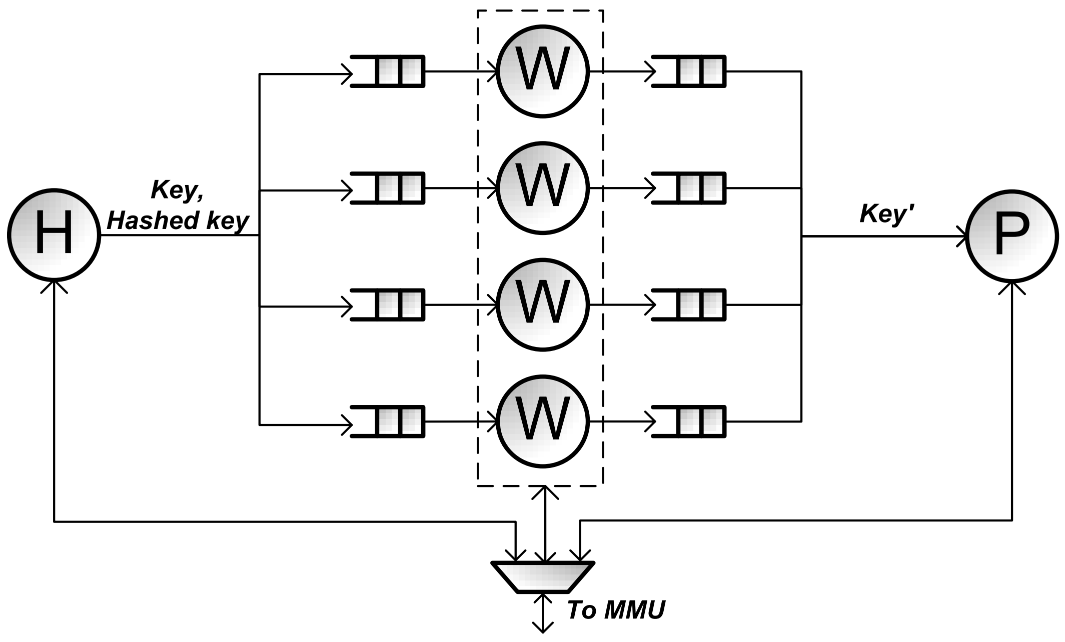


Figure 4.5 – Widx overview. H: dispatcher, W: walker, P: output producer.

unit shown in Figure 4.5. The compact ISA minimizes the implementation complexity and area cost while affording design reuse across the different units.

Figure 4.6 shows the internals of a Widx unit. We adopted a design featuring a 64-bit datapath, 2-stage pipeline, and 32 software-exposed registers. The relatively large number of registers is necessary for storing the constants used in key hashing. The three-operand ALU allows for efficient shift operations that are commonly used in hashing. The critical path of our design is the branch address calculation with relative addressing.

4.3.2 Programming API

To leverage Widx, a database system developer must specify three functions: one for key hashing, another for the node walk, and the last one for emitting the results. Either implicitly or explicitly, these functions specify the data layout (e.g., relative offset of the node pointer within a node data structure) and data types (e.g., key size) used for the index operations. Inputs to the functions include the hash table pointer and size, input keys' table pointer and size, and the destination pointer for emitting results.

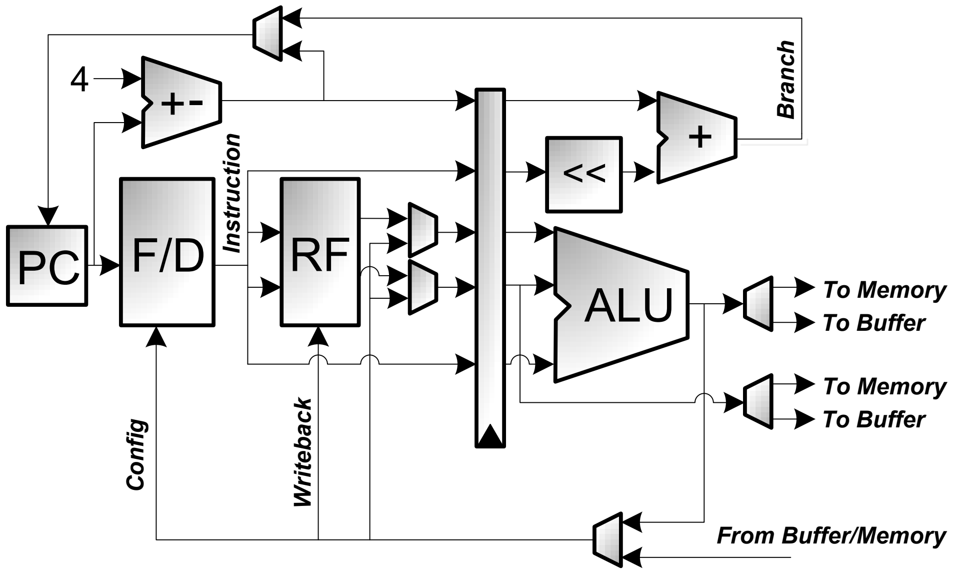


Figure 4.6 – Schematic design of a single Widx unit.

The functions are written in a limited subset of C, although other programming languages (with restrictions) could also be used. Chief limitations imposed by the Widx programming model include the following: no dynamic memory allocation, no stack, and no writes except by the output producer. One implication of these restrictions is that functions that exceed a Widx unit's register budget cannot be mapped, as the current architecture does not support push/pop operations. However, our analysis with several contemporary DBMSs shows that, in practice, this restriction is not a concern.

4.3.3 Additional Details

Configuration interface: In order to benefit from the Widx acceleration, the application binary must contain a Widx control block, composed of constants and instructions for each of the Widx dispatcher, walker, and output producer units. To configure Widx, the processor initializes memory-mapped registers inside Widx with the starting address (in the application's virtual address space) and length of the Widx control block. Widx then issues a series of loads to consecutive virtual addresses from the specified starting address to load the instructions and internal registers for each of its units.

Table 4.1 – Widx ISA. The columns show which Widx units use a given instruction type.

Instruction	H	W	P
ADD	✓	✓	✓
AND	✓	✓	✓
BA	✓	✓	✓
BLE	✓	✓	✓
CMP	✓	✓	✓
CMP-LE	✓	✓	✓
LD	✓	✓	✓
SHL	✓	✓	✓
SHR	✓	✓	✓
ST			✓
TOUCH	✓	✓	✓
XOR	✓	✓	✓
ADD-SHF	✓	✓	
AND-SHF	✓		
XOR-SHF	✓		

To offload an index operation, the core (directed by the application) writes the following entries to Widx's configuration registers: base address and length of the input table, base address of the hash table, starting address of the results region, and a NULL value identifier. Once these are initialized, the core signals Widx to begin execution and enters an idle loop. The latency cost of configuring Widx is amortized over the millions of hash table probes that Widx executes.

Handling faults and exceptions: TLB misses are the most common faults encountered by Widx and are handled by the host core's MMU in its usual fashion. In architectures with software-walked page tables, the walk will happen on the core and not on Widx. Once the missing translation is available, the MMU will signal Widx to retry the memory access. In the case of the retry signal, Widx redirects PC to the previous PC and flushes the pipeline. The retry mechanism does not require any architectural checkpoint as nothing is modified in the first stage of the pipeline until an instruction completes in the second stage.

Other types of faults and exceptions trigger handler execution on the host core. Because Widx provides an atomic all-or-nothing execution model, the index operation is completely re-executed on the host core in case the accelerator execution is aborted.

Table 4.2 – Evaluation parameters.

Parameter	Value
Technology	40nm, 2GHz
CMP Features	4 cores
Core Types	In-order (Cortex A8-like): 2-wide OoO (Xeon-like): 4-wide, 128-entry ROB
L1-I/D Caches	32KB, split, 2 ports, 64B blocks, 10 MSHRs, 2-cycle load-to-use latency
LLC	4MB, 6-cycle hit latency
TLB	2 in-flight translations
Interconnect	Crossbar, 4-cycle latency
Main Memory	32GB, 2 MCs, BW: 12.8GB/s 45ns access latency

4.4 Methodology

Workloads: We evaluate three different benchmarks, namely the hash join kernel, TPC-H, and TPC-DS.

We use a highly optimized and publicly available hash join kernel code [10], which optimizes the "no partitioning" hash join algorithm [13]. We configure the kernel to run with four threads that probe a hash table with up to two nodes per bucket. Each node contains a tuple with 4B key and 4B payload [46]. We evaluate three index sizes, *Small, Medium* and *Large.* The Large benchmark contains 128M tuples (corresponding to 1GB dataset) [46]. The Medium and Small benchmarks contain 512K (4MB raw data) and 4K (32KB raw data) tuples respectively. In all configurations the outer relation contains 128M uniformly distributed 4B keys.

We run DSS queries from the TPC-H [78] and TPC-DS [59] benchmarks on MonetDB 11.5.9 [38] with a 100GB dataset (a scale factor of 100) both for hardware profiling and evaluation in the simulator. Our hardware profiling experiments are carried out on a six-core Xeon 5670 with 96GB of RAM and we used Vtune [42] to analyze the performance counters. Vtune allows us to break down the execution time into functions. To make sure that we correctly account for the time spent executing each database operator (e.g., scan, index), we examine the source code of those functions and group them according to their functionality. We warm up the DBMS and memory by executing all the queries once and then we execute the queries in succession and report the average of three runs. For each run, we randomly generate new inputs for queries with the *dbgen* tool [78].

For the TPC-H benchmark, we run all the queries and report the ones with the index execution time greater than 5% of the total query runtime (16 queries out of 22). Since there are a total of 99 queries in the TPC-DS benchmark, we select a subset of queries based on a classification found in previous work [59], considering the two most important query classes in TPC-DS, *Reporting* and *Ad Hoc*. Reporting queries are well-known, pre-defined business questions (queries 37, 40 & 81). Ad Hoc captures the dynamic nature of a DSS system with the queries constructed on the fly to answer immediate business questions (queries 43, 46, 52 & 82). We also choose queries that fall into both categories (queries 5 & 64). In our runs on the cycle-accurate simulator, we pick a representative subset of the queries based on the average time spent in index lookups.

Processor parameters: The evaluated designs are summarized in Table 4.2. Our baseline processor features aggressive out-of-order cores with a dual-ported MMU. We evaluate the Widx designs featuring one, two, and four walkers. Based on the results of the model of Section 4.2.2, we do not consider designs with more than four walkers. All Widx designs feature one shared dispatcher and one result producer. As described in Section 4.3, Widx offers full offload capability, meaning that the core stays idle (except for the MMU) while Widx is in use. For comparison, we also evaluate an in-order core modeled after ARM Cortex A8.

Simulation: We evaluate various processor and Widx designs using the Flexus full-system simulator [84]. Flexus extends the Virtutech Simics functional simulator with timing models of cores, caches, on-chip protocol controllers, and interconnect. Flexus models the SPARC v9 ISA and is able to run unmodified operating systems and applications.

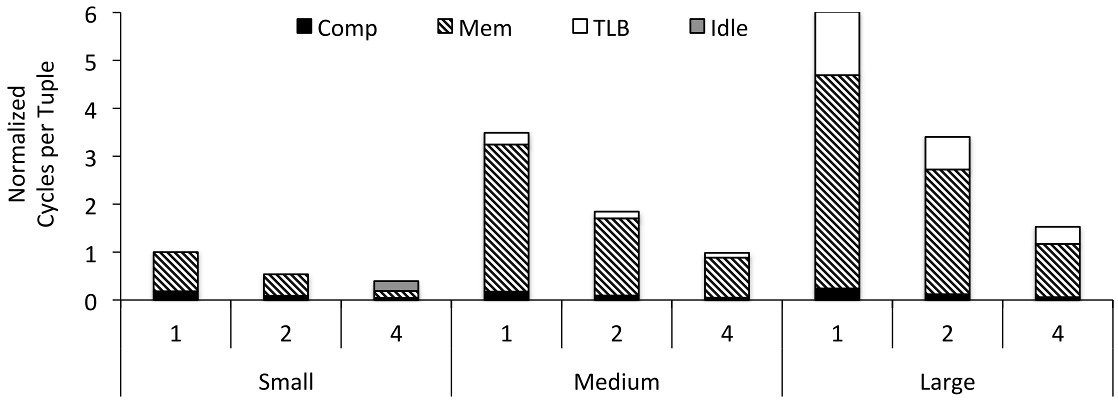
We use the SimFlex multiprocessor sampling methodology [84], which extends the SMARTS statistical sampling framework [87]. Our samples are drawn over the entire index execution until the completion. For each measurement, we launch simulations from checkpoints with warmed caches and branch predictors, and run 100K cycles to achieve a steady state of detailed cycle-accurate simulation before collecting measurements for the subsequent 50K cycles. We measure the index traversal throughput by aggregating the tuples processed per cycle both for the baseline and Widx. To measure the index traversal throughput of the baseline, we mark the beginning and end of the index code region and track the progress of each tuple until its completion. Performance measurements are computed at 95% confidence with an average error of less than 5%.

Power and Area: To estimate Widx's area and power, we synthesize our Verilog implementation with the Synopsys Design Compiler [74]. We use the TSMC 45nm technology (Core library: TCBN45GSBWP, V_{dd}: 0.9V), which is perfectly shrinkable to the 40nm half node. We target a 2GHz clock rate. We set the compiler to the *high* area optimization target. We report the area and power for six Widx units: four walkers, one dispatcher, and one result producer, with 2-entry queues at the input and output of each walker unit.

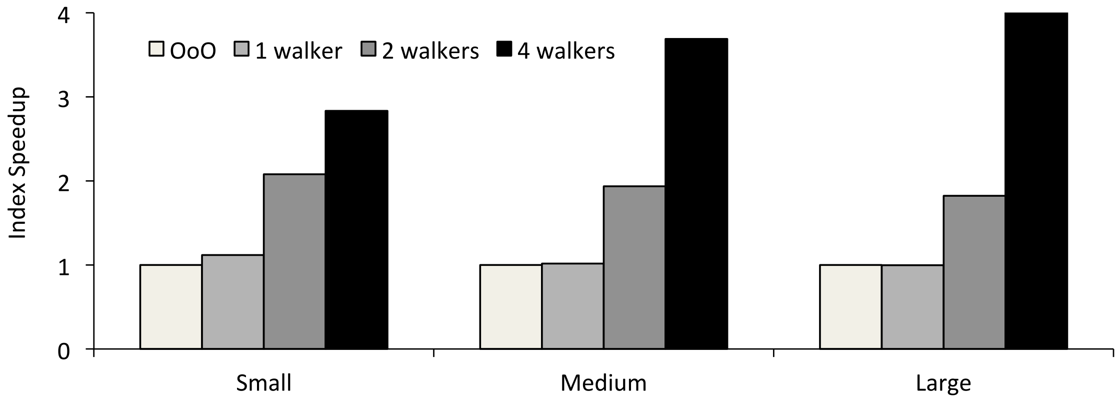
We use published power estimates for OoO Xeon-like core and in-order A8-like core at 2GHz [51]. We assume the power consumption of the baseline OoO core to be equal to Xeon's nominal operating power [66]. Idle power is estimated to be 30% of the nominal power [43]. As the Widx-enabled design relies on the core's data caches, we estimate the core's private cache power using CACTI 6.5 [56].

4.5 Evaluation

We first analyze the performance of Widx on an optimized hash join kernel code. We then present a case study on MonetDB with DSS workloads, followed by an area and energy analysis.



(a) Widx walkers cycle breakdown for the Hash Join kernel (normalized to Small running on Widx with one walker)



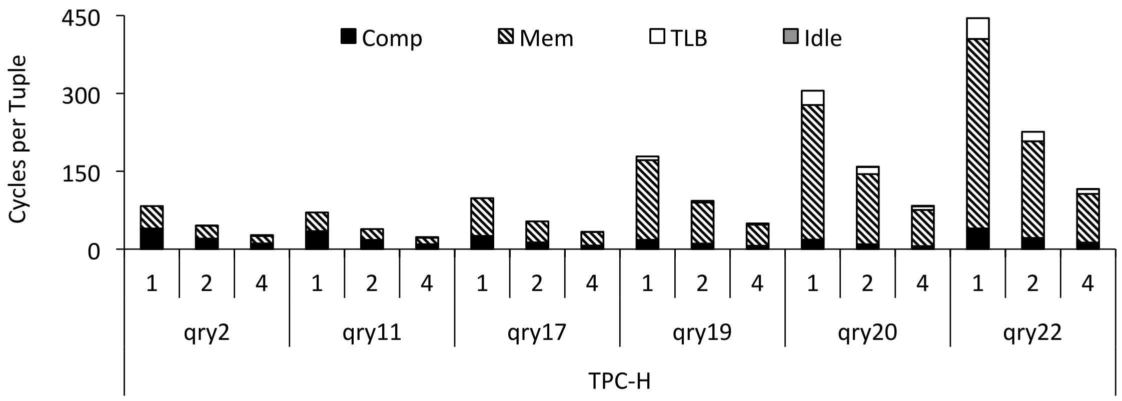
(b) Speedup for the Hash Join kernel

Figure 4.7 – Hash Join kernel analysis.

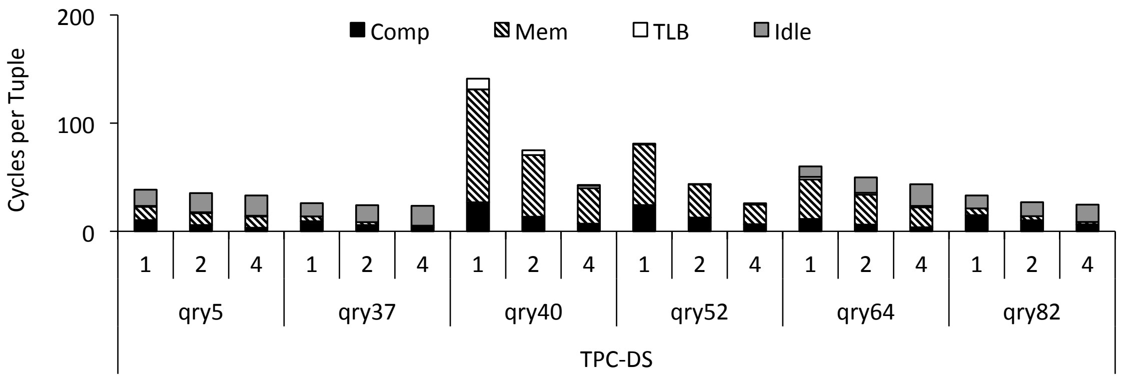
4.5.1 Performance on Hash Join Kernel

In order to analyze the performance implications of index walks with various dataset sizes, we evaluate three different index sizes; namely, Small, Medium, and Large, on a highly optimized hash join kernel as explained in Section 4.4.

To show where the Widx cycles are spent we divide the aggregate critical path cycles into four groups. *Comp* cycles go to computing effective addresses and comparing keys at each walker, *Mem* cycles count the time spent in the memory hierarchy, *TLB* quantifies the Widx stall cycles due to address translation misses, and *Idle* cycles account for the walker stall time waiting for a new key from the dispatcher. Presence of Idle cycles indicates that the dispatcher is unable to keep up with the walkers.



(a) Widx walkers cycle breakdown for TPC-H queries



(b) Widx walkers cycle breakdown for TPC-DS queries

Figure 4.8 – DSS on MonetDB. Note that Y-axis scales are different on the two subgraphs.

Figure 4.7a depicts the Widx walkers' execution cycles per tuple (normalized to Small running on Widx with one walker) as we increase the number of walkers from one to four. The dominant fraction of cycles is spent in memory and as the index size grows, the memory cycles increase commensurately. Not surprisingly, increasing the number of walkers reduces the memory time linearly due to the MLP exposed by multiple walkers. One exception is the Small index with four walkers; in this scenario, node accesses from the walkers tend to hit in the LLC, resulting in low AMAT. As a result, the dispatcher struggles to keep up with the walkers, causing the walkers to stall (shown as Idle in the graph). This behavior matches our model's results in Section 4.2.

The rest of the Widx cycles are spent on computation and TLB misses. Computation cycles constitute a small fraction of the total Widx cycles because the Hash Join kernel implements a simple memory layout, and hence requires trivial address calculation. We also observe that the fraction of TLB cycles per walker does not increase as we enable more walkers. Our baseline core's TLB supports two in-flight translations, and it is unlikely to encounter more than two TLB misses at the same time, given that the TLB miss ratio is 3% for our worst case (Large index).

Figure 4.7b illustrates the index traversal speedup of Widx normalized to the OoO baseline. The one-walker Widx design improves performance by 4% (geometric mean) over the baseline. The one-walker improvements are marginal because the hash kernel implements an oversimplified hash function, which does not benefit from Widx's decoupled hash and walk mechanisms, which overlap the hashing and walking time. However, the performance improvement increases with the number of Widx walkers, which traverse buckets in parallel. Widx achieves a speedup of 4x at best for the Large index table, which performs poorly on the baseline cores due to the high LLC miss ratio and limited MLP.

4.5.2 Case study on MonetDB

In order to quantify the benefits of Widx on a complex system, we run Widx with the well-known TPC-H benchmark and with the successor benchmark TPC-DS on a state-of-the-art database management system, MonetDB.

Figure 4.8a breaks down the Widx cycles while running TPC-H queries. We observe that the fraction of the computation cycles in the breakdown increases compared to the hash join kernel due to MonetDB's complex hash table layout. MonetDB stores keys indirectly (i.e., pointers) in the index resulting in more computation for address calculation. However, the rest of the cycle breakdown follows the trends explained in the Hash Join kernel evaluation (Section 4.5.1). The queries enjoy a linear reduction in cycles per tuple with the increasing number of walkers. The queries with relatively small index sizes (query 2, 11 & 17) do not experience any TLB misses, while the memory-intensive queries (query 19, 20 & 22) experience TLB miss cycles up to 8% of the walker execution time.

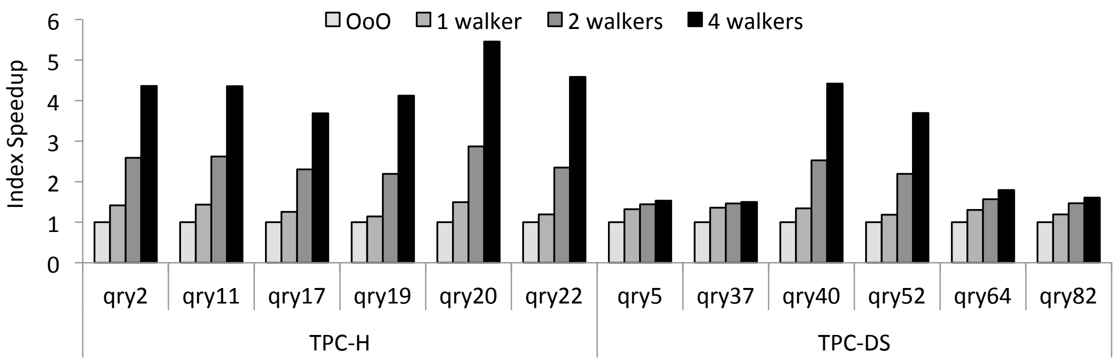


Figure 4.9 – Performance of Widx on DSS queries.

Figure 4.8b presents the cycles per tuple breakdown for TPC-DS. Compared to TPC-H, a distinguishing aspect of the TPC-DS benchmark is the small-sized index tables. [1] Our results verify this fact as we observe consistently lower memory time compared to that of TPC-H (mind the y-axis scale change). As a consequence, some queries (query 5, 37, 64 & 82) go over indexes that can easily be accommodated in the L1-D cache. Widx walkers are partially idle given that they can run at equal or higher speed compared to the dispatcher due to the tiny index, a behavior explained by our model in Section 4.2.

Figure 4.9 illustrates the performance of Widx on both TPC-H and TPC-DS queries. Compared to OoO, four walkers improve the performance by 1.5x-5.5x (geometric mean of 3.1x). The maximum speedup (5.5x) is registered on TPC-H query 20, which works on a large index with double integers that require computationally intensive hashing. As a result, this query greatly benefits from Widx's features, namely, the decoupled hash and multiple walker units with custom ISA. The minimum speedup (1.5x) is observed on TPC-DS query 37 due to L1-resident index (L1-D miss ratio <1%). We believe that this is the lower limit for our design because there are only a handful of unique index entries.

We estimate the benefits of index traversal acceleration at the level of the entire query by projecting the speedups attained in the Widx-accelerated design onto the index traversal portions of the TPC-H and TPC-DS queries presented in Figure 4.1a. By accelerating just the index portion of the query, Widx speeds up the query execution by a geometric mean of 1.5x

[1]There are 429 columns in TPC-DS, while there are only 61 in TPC-H. Therefore, for a given dataset size, the index sizes are smaller per column because the same size of dataset is divided over a large number of columns.

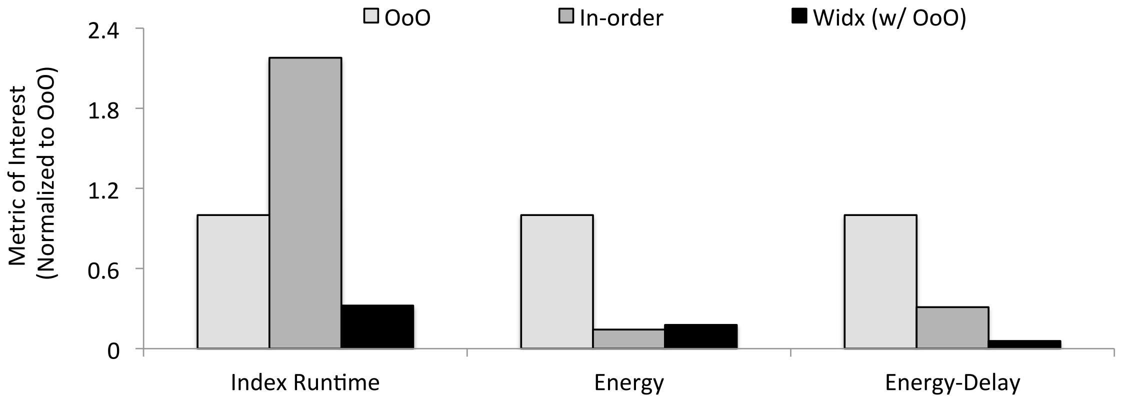


Figure 4.10 – Index Runtime, Energy and Energy-Delay metric of Widx (lower is better).

and up to 3.1x (query 17). Our query speedups are limited by the fraction of the time spent in index traversals throughout the overall execution. In query 17, the achieved overall speedup is close to the index-only speedup with the four-walker design as 94% of the execution time is spent in index lookups. The lowest overall speedup (10%) is obtained in query 37 because only 29% of the query execution is offloaded to Widx and as explained above, the query works on an L1-resident index.

4.5.3 Area and Energy Efficiency

To model the area overhead and power consumption of Widx, we synthesized our RTL design in the TSMC 40nm technology. Our analysis shows that a single Widx unit (including the two-entry input/output buffers) occupies $0.039mm^2$ with a peak power consumption of $53mW$ at 2GHz. Our power and area estimates are extremely conservative due to the lack of publicly available SRAM compilers in this technology. Therefore, the register file and instruction buffer constitute the main source of area and power consumption of Widx. The Widx design with six units (dispatcher, four walkers, and an output producer) occupies $0.24mm^2$ and draws $320mW$. To put these numbers into perspective, an in-order ARM Cortex A8 core in the same process technology occupies $1.3mm^2$, while drawing $480mW$ including the L1 caches [51]. Widx's area overhead is only 18% of Cortex A8 with comparable power consumption, despite our conservative estimates for Widx's area and power. As another point of comparison, an ARM M4 microcontroller [8] with full ARM Thumb ISA support and a floating-point unit occupies roughly the same area as the single Widx unit. We thus conclude that Widx hardware is extremely cost-effective even if paired with very simple cores.

Figure 4.10 summarizes the trade-offs of this study by comparing the average runtime, energy consumption, and energy-delay product of the index portion of DSS workloads. In addition to the out-of-order baseline, we also include an in-order core as an important point of comparison for understanding the performance/energy implications of the different design choices.

An important conclusion of the study is that the in-order core performs significantly worse (by 2.2x on average) than the baseline OoO design. Part of the performance difference can be attributed to the wide issue width and reordering capability of the OoO core, which benefits the hashing function. The reorder logic and large instruction window in the OoO core also help in exposing the inter-key parallelism between two consecutive hash table lookups. For queries that have cache-resident index data, the loads can be issued from the imminent key probe early enough to partially hide the cache access latency.

In terms of energy efficiency, we find that the in-order core reduces energy consumption by 86% over the OoO core. When coupled with Widx, the OoO core offers almost the same energy efficiency (83% reduction) as the in-order design. Despite the full offload capability offered by Widx and its high energy efficiency, the total energy savings are limited by the high idle power of the OoO core.

In addition to energy efficiency, QoS is a major concern for many database workloads. We thus study the efficiency of various designs on both performance and energy together via the energy-delay product metric. Due to its performance and energy-efficiency benefits, Widx improves the energy-delay product by 5.5x over the in-order core and by 17.5x over the OoO baseline.

4.6 Discussion

Other join algorithms and software optimality: In this study, we focused on hardware-oblivious hash join algorithms that run on the state-of-the-art software. In order to exploit on-chip cache locality, researchers have proposed hardware-conscious approaches that have a table-partitioning phase prior to the main join operation [53]. In this phase, a hash table is built on each small partition of the table, thus making the individual hash tables cache-resident. The optimal partition size changes across hardware platforms based on the cache size, TLB size, etc.

Widx's functionality does not require any form of data locality, and thus is independent of any form of data partitioning. Widx is, therefore, equally applicable to hash join algorithms that employ data partitioning prior to the main join operation [53]. Due to the significant computational overhead involved in table partitioning, specialized hardware accelerators that target partitioning [85] can go hand in hand with Widx.

Another approach to optimize join algorithms is the use of SIMD instructions. While the SIMD instructions aid hash-joins marginally [35, 46], another popular algorithm, sort-merge join, greatly benefits from SIMD optimizations during the sorting phase. However, prior work [9] has shown that hash join clearly outperforms the sort-merge join. In general, software optimizations target only performance, whereas Widx both improves performance and greatly reduces energy.

Broader applicability: Our study focused on MonetDB as a representative contemporary database management system; however, we believe that Widx is equally applicable to other DBMSs. Our profiling of HP Vertica and SAP HANA indicate that these systems rely on index strategies, akin to those discussed in this work, and consequently, can benefit from our design. Moreover, Widx can easily be extended to accelerate other index structures, such as balanced trees, which are also common in DBMSs.

LLC-side Widx: While our study focused on a Widx design tightly coupled with a host core, Widx could potentially be located close to the LLC instead. The advantages of LLC-side placement include lower LLC access latencies and reduced MSHR pressure. The disadvantages include the need for a dedicated address translation logic, a dedicated low-latency storage next to Widx to exploit data locality, and a mechanism for handling exception events. We believe the balance is in favor of a core-coupled design; however, the key insights of this work are equally applicable to an LLC-side Widx.

Widx unit candidates: In this work, we focused on designing our own hardware unit to fully demonstrate the energy efficiency benefits of specialized hardware. However, Widx-style execution is applicable to energy-efficient hardware units such as embedded cores and microcontrollers. A prime example of such simple core is ARC HS38 by Synopsis [21], which is a high-performance embedded core consuming almost the same power as a single Widx unit while operating at 1.6GHz. HS38 instructions are customizable and the design employs a dedicated MMU, which can avoid the need for a dedicated host core.

4.7 Widx Summary

We introduced Widx, a programmable widget for accelerating hash index accesses. Widx features multiple *walkers* for traversing the node lists and a single *dispatcher* that maintains a list of hashed keys for the walkers. Both the walkers and a dispatcher share a common building block consisting of a custom 2-stage RISC core with a simple ISA. The limited programmability afforded by the simple core allows Widx to support a virtually limitless variety of schemas and hashing functions. Widx minimizes cost and complexity through its tight coupling with a conventional core, which eliminates the need for dedicated address translation and caching hardware. Compared to an aggressive out-of-order core, the proposed Widx design improves index traversal performance by 3.1x on average, while saving 83% of the energy by allowing the host core to be idle while Widx runs.

5 Quantifying the Impact of Memory Subsystem on Acceleration

In this chapter, we study the impact of memory subsystem on software and hardware acceleration mechanisms. We identify and quantify the features of the memory subsystem that would improve the accelerator throughput and energy efficiency. We first look at the benefits of using large memory pages to minimize the overhead of TLB miss penalties. We then study the performance sensitivity of software acceleration to the features of L1-D cache, namely cache block alignment and the number miss status handling registers, which improve the effectiveness of the prefetched cache blocks and maximize memory-level parallelism respectively. Furthermore, to understand the significance of shared on-chip resources, we perform a throughput scalability analysis with multiple hardware threads to reveal possible bottlenecks on the shared on-chip resources. After quantifying the performance degradation imposed by the bottlenecks of the on-chip cache hierarchy and memory subsystem, we finally conclude by demonstrating a simple analysis of leveraging our software acceleration scheme on an optimized memory subsystem and also on our specialized hardware accelerator to maximize throughput by fully utilizing the off-chip bandwidth while achieving high energy efficiency.

5.1 Quantifying the Overhead

In Chapter 3 we demonstrated a software accelerator (AMAC) that greatly enhances data structure traversal throughput for a wide variety of pointer-intensive data structures. In this chapter, we study the limits of the AMAC execution, by providing an in-depth analysis of AMAC critical path and scalability. We identify and analyze three bottlenecks in the memory subsystem, which are (i) virtual memory page size, (ii) data structure alignment, and (iii) the number of miss status handling registers for on-chip caches.

Using large virtual memory pages is an important optimization for applications that deal with large amounts of data. Given that TLB capacity is limited because it is on the critical path of the L1-D cache accesses, large pages allow for reducing (i) the number of TLB misses by increasing the amount of memory space covered, and (ii) the cycles spent to translate a page in case of a TLB miss by reducing the page table size drastically. The latter is especially important as the total size of the page table entries can reach beyond the capacity of on-chip caches and result in severe TLB miss penalties.

In addition to perfect TLB coverage, completely hiding the memory latency necessitates demanding the required data block or blocks in advance. Given that prefetch instructions typically work at cache-line granularity (e.g., 64-byte on Intel Xeon), when the required data blocks are larger than the cache line size, the number of prefetch demands has to be adjusted accordingly. However, even if the required data blocks are smaller than a single cache line, there is a possibility that the target block of data can span two cache lines (i.e., unaligned) due to the memory layout of the data structure. Although, it is possible to prefetch two consecutive cache blocks with two prefetch instructions, unaligned accesses still result in sub-optimal performance because more than one cache line has to be accessed during the execution. Therefore, data structure padding is an effective solution by adding extra fields in data structure nodes at compile time to ensure that consecutive data structure nodes start at the boundary of a new cache line whenever it is possible.

Lastly, the number of outstanding memory accesses that can be supported by the cache hierarchy is another possible limitation for AMAC . Each level of the cache hierarchy contains miss status handling registers (MSHRs) to handle outstanding misses. When the MSHRs are fully occupied, the prefetch instructions are dropped [43] and lose the opportunity to hide memory access latency for that traversal. In AMAC execution, the number of MSHRs required is a function of the number of in-flight lookups (i.e., memory-level parallelism) that targets hiding the memory access latency completely. Moreover, the shared caches (e.g., last-level cache) should be able to accommodate the aggregate number of outstanding misses from all cores. When the number of MSHRs is not enough in private and/or in shared caches, the memory latency is exposed to the core as the desired memory-level parallelism cannot be achieved.

5.1.1 Experimental Setup

The experimental parameters in this chapter is almost identical to the setup used in Chapter 3 and except with a few modifications as described below:

To mitigate any algorithm-related scalability issues, we only focus on the read-only probe phase of hash join. We run our experiments on a Xeon x5670 machine except the scalability analysis where we include Oracle T4 experiments (more details in Table 3.5).

To stress different levels of the memory hierarchy with random pointer accesses, we leverage a uniform hash table with eight nodes per bucket while varying the size of the hash table from 0.5 MB to 2048 MB. In all configurations, the probe relation contains 512 MB of key/value tuples with uniformly distributed unique values. Each lookup uniformly traverses eight hash table nodes and we report the average number of cycles required to traverse a single hash table node as this is the critical path of the lookups. The AMAC code always assumes padded data structure nodes, therefore one prefetch instruction (i.e., one cache block) is issued only for a hash table node traversal. As a result, we keep the number of critical path operations the same throughout the experiments.

To estimate the energy consumption of Xeon x5670 and Widx, we leverage the methodology described in Section 4.4. As we report real hardware performance numbers for Xeon x5670 running at 2.93 GHz (as opposed to 2 GHz), we take the increase in OoO core energy consumption into account for our energy estimations.

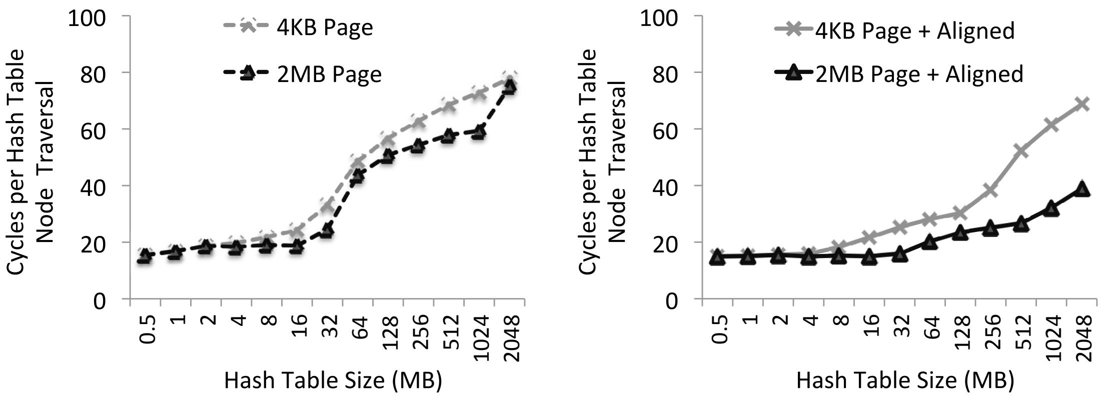
5.2 Data Structure Padding

Figure 5.1a shows our sensitivity analysis of the critical path of the hash table lookups (i.e., cycles per hash table node traversal) with unaligned hash table nodes for 4KB and 2MB virtual memory pages. We observe that for 0.5MB hash table, both 4KB and 2MB pages achieve 15 cycles per hash table node traversal. In other words, to traverse a single cache- and TLB-resident hash table node, the Xeon processor has to do 15 cycles of work.

As we increase the size of the hash table from 0.5MB to 16MB, the number of cycles per node traversal increases by 40% on average. The datasets beyond 16MB of dataset cannot be captured by the 12 MB last-level cache of Xeon x5670, and hence both configurations perform up to 3.4x worse compared to the LLC-resident configurations. Although AMAC should be able to hide the memory access latency when running out of the cache, the hash table nodes (without cache block alignment) are 48-bytes while demanded cache blocks are 64-bytes. As a result, the opportunity to hide the off-chip memory accesses is lost as the traversal of an unaligned hash table node may require two cache line accesses, while AMAC prefetches only one. As we mentioned in the previous section, although AMAC can be configured to prefetch two consecutive cache lines, we do not evaluate that configuration to keep the AMAC code the same across experiments.

5.3 Virtual Memory Page Size

Figure 5.1b shows our sensitivity results with the padded hash table nodes. In contrast to the experiments without padding, for 4KB page experiments, we observe a constant critical path between 0.5MB and 2MB, which is 15 cycles similar to the previous experiment that



(a) Without cache block alignment (b) With cache block alignment

Figure 5.1 – Virtual memory page size and cache block alignment sensitivity on AMAC hash table traversal performance. Measurement on Xeon x5670.

leverages the same code base. These results are not surprising given that the data is LLC-resident for 4KB pages, and the TLB can cover just a little more than 2MB virtual memory space (64-entry L1 TLB and 512-entry L2 TLB). As a result, when using 4KB pages we start observing address translation penalties increasing the critical path of the lookups beyond the 2MB dataset. When we turn on the support for large (2MB) pages, we observe that critical path latency (15 cycles) stays constant up to the 32 MB dataset, meaning that address translation overheads are mitigated completely as the TLB can cover up to 64MB of address space with large pages (32-entry TLB for large pages).

5.4 Miss Status Handling Registers

To understand the limitations imposed by the size of the L1-D miss status handling registers (MSHRs), we again analyze our results with large pages (2MB) with padded data structures (depicted in Figure 5.1b) by focusing on the dataset sizes beyond the capacity of the LLC (32 MB - 2048 MB). At the 2048 MB dataset point, we observe that the cycles per node traversal (i.e., critical path latency) increases by 2.5x compared to the 32 MB case. We find that the root cause of this increase is the combination of address translation misses and the lack of MSHR capacity in the L1-D cache. To avoid any TLB-related penalties, we also analyze the critical path latency with the 64MB dataset, which can be captured by the TLB but not by the LLC, and we still observe a 33% increase in the critical path cycles, which can be attributed to the limited MSHR capacity in the L1-D cache.

To further analyze the limitations of MSHR capacity at the L1-D cache, we measure the average memory access time (AMAT) of our Xeon machine as 202 cycles by using the Intel Memory Checker tool [41]. Given the 10 MSHRs at the L1-D cache, which dictates the maximum number of parallel accesses to the lower levels of the cache hierarchy, the workload can only generate a maximum MLP of 10. Therefore, in the best case, the cycles per one traversal is $202/10 \simeq 20$ despite the 15 cycles of work (i.e., computation) found in the previous analysis. As a result, the execution will be bounded by the memory access latency as opposed to the computation latency due to the lack of sufficient L1-D MSHR entries.

In summary, for data structure traversals that are properly aligned and require little computation with a near-perfect TLB, the number of L1-D MSHRs entries is not sufficient to cover the memory access latency. It is important to note that recent Intel CPU micro-architectures still employ 10 MSHRs at the L1-D [39, 44], therefore new architectures with aggressive OoO cores and a higher number of MSHRs will benefit AMAC-style execution. In addition, researchers proposed associative miss handling structures that are practical and scalable [81]. Nonetheless, AMAC does not require associative miss handling structures and can operate with a simple hardware FIFO to track outstanding misses given that in-flight operations are consumed sequentially.

5.5 Throughput Scalability and Bottleneck Analysis

In this section, we perform a throughput scalability experiment by running AMAC in a multi-threaded fashion to take advantage of the multiple cores on Xeon and Oracle T4. To do so, we equally divide the work among software threads and assign them only to physical cores (six on Xeon and eight on T4). In these experiments, we do not use SMT threads as we studied the effect of SMT in Chapter 3.

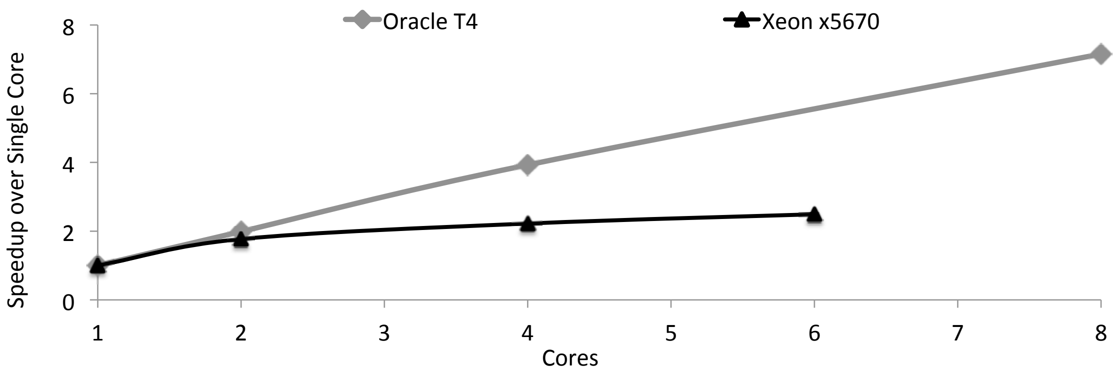


Figure 5.2 – Performance scalability of AMAC on Xeon x5670 and Oracle T4 with physical cores

Figure 5.2 depicts the AMAC throughput normalized to single-threaded hash table lookup throughput. The T4 machine offers a 7.1x speedup with eight physical cores, which is almost linear scalability with the number of physical cores. Therefore, the results on Oracle T4 clearly indicate that the AMAC approach does not affect the inherently scalable nature of the algorithm. In contrast, the Xeon processor delivers 2.7x speedup with six physical cores, which is almost half of the throughput we were expecting to observe. Such results signal a significant hardware bottleneck as the scalable algorithm does not scale with the number of physical cores (i.e., six cores). Moreover, recent work by Balkesen et al. [11] also reports relatively low benefits for prefetch-based hash joins on a fully loaded Xeon processor corroborating our results.

We further investigate the source of the bottleneck on Xeon by using performance counters. Table 5.1 depicts the instructions per cycle (IPC) of the CPU while increasing the number of threads for the probe phase of the large join. We observe that the average IPC of six threads is 2.5x worse than the single-threaded execution, which corroborates with the drop in the speedups explained above.

To identify the root cause of the drop in the IPC, we further investigate the behavior of memory accesses throughout the memory hierarchy. Figure 5.3 breaks down the L1-D cache misses into four categories depending on where they hit in the memory hierarchy. *L1-D MSHR* hits are the memory references that miss in the L1-D but hit in the L1-D MSHRs, meaning that the memory access was already issued by the core but the data has not arrived yet (i.e., outstanding miss). L2 and LLC hits count the references that hit in the on-chip cache hierarchy. Off-chip hits count the memory references that were not filtered by the on-chip caches.

We observe an almost 2.3x increase in the L1-D MSHR of a single-thread vs. six threads as the prefetches do not arrive in a timely manner in the six-thread experiment. While the cause of this problem can be off-chip accesses, the data shows that increasing the thread count has a marginal impact on the number of off-chip accesses. Therefore, our last hypothesis is that there is a resource contention in the LLC. To verify our hypothesis, we re-run the experiments with four threads, but this time we distribute the four threads to two sockets (i.e., two physical CPUs) two by two (Figure 5.3 and Table 5.1 rightmost bar and field) and see that the memory behavior and IPC with four threads on two sockets is identical to two threads on a single socket.

As a result, we find that the root cause of this problem is the number of LLC MSHRs provisioned for off-chip load requests (referred to as Global Queue). While this structure is limited to 32 entries (for loads) in a Nehalem processor [58], the aggregate number of outstanding load misses generated by six cores can reach up to sixty (i.e., 10 L1-D MSHRs per core). As a result, in the Nehalem architecture, the number of LLC MSHRs limits the scalability of read-only workloads with high MLP and low LLC locality.

We conclude that utilizing the upper levels of the hierarchy causes severe contention in the LLC when all the threads issue irregular accesses. Therefore, the throughput on Xeon saturates with four threads, explaining why prefetch-based techniques do not deliver the expected performance on fully loaded multi-core Xeon processors.

5.6 Putting It All Together

Having identified and quantified the impact of possible bottlenecks in the memory subsystem, we will describe necessary modifications to the memory subsystem to achieve a fully compute-bound execution via a simple analytical model. The parameters for our model are derived

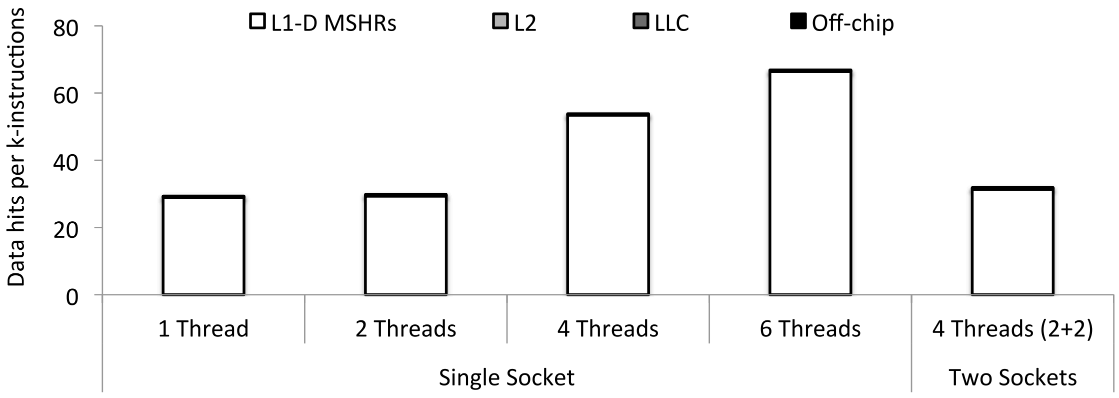


Figure 5.3 – Breakdown of memory reference hits per kilo-instruction on a two-socket Xeon x5670.

Table 5.1 – Instructions per cycle for the pointer-chasing workload with AMAC on a two-socket Xeon x5670.

Threads	**1**	**2**	**4**	**6**	**2+2**
IPC	1.0	1.0	0.6	0.4	0.9

from the hash table traversal workload with a 2048 MB dataset used in this chapter, which is sometimes referred to as the *pointer-chasing workload.*

Based on the analysis presented in Section 5.3 and Section 5.4, Equation 5.1 calculates the number of cycles per node traversal, which can be either computation-bound ($CompCycles$) or memory-bound ($\frac{AMAT}{MLP} + TLB_{MissPenalty}$) depending on which term is larger. $CompCycles$ defines the number of cycles required to process one hash table node including the address calculations, comparisons and AMAC state management operations. The average memory access latency per traversal is calculated by the $\frac{AMAT}{MLP}$ term, where MLP defines the number of in-flight lookups. $TLB_{MissPenalty}$ is the average number of non-overlapping cycles spent in TLB miss handling for each hash table node. It is important to note that, although TLB miss cycles can be overlapped with memory accesses, in the cases where there is a high number simultaneous TLB misses, the hardware page walker can act as a serialization point, therefore a separate term is used for indicating non-overlapping cycles (i.e., exposed) cycles.

$$Cycles_{NodeTraversal} = max(CompCycles, (\frac{AMAT}{MLP} + TLB_{MissPenalty})) \tag{5.1}$$

Based on the analysis in Section 5.3 and Section 5.4, we find that $CompCycles$ is equal to 15 cycles for the pointer-chasing workload. Ideally, AMAC execution should be compute-bound with a perfect TLB and a sufficiently high number of in-flight lookups (MLP) to reduce the $\frac{AMAT}{MLP}$ term. Given that $AMAT$ is equal to 202 cycles (from Section 5.4), the number of L1-D MSHRs required (i.e., MLP) is $202/15 \simeq 14$ for the pointer-chasing phase of the execution. Moreover, the MSHRs are also required by the sequentially accessed lookup keys, which are used for initiating the lookups and possibly by the hardware prefetchers and page walkers. In order not to create any MSHR contention between our workload and the rest of the hardware components, we assume that in total 16 MSHRs are required for effective execution of pointer-based lookups. Obviously, this number is specific to our pointer-chasing micro-benchmark and for the workloads that require fewer compute ops or the hardware platforms that have relatively higher memory access latencies, the required number of MSHRs can be calculated by using our methodology.

Table 5.2 summarizes the desired modifications to the existing memory hierarchy of the Xeon processor. We find that our pointer-chasing workload allocates a little less than 4GBs in total (including the probe table), therefore we require 4x1GB VM pages to achieve a 100% hit ratio in the TLB. To fully cover the memory access latency, we need to have 16 L1-D MSHRs, and also we need an LLC design, which does not suffer from the scalability problem described in Section 5.5. Finally, after all these optimizations, the execution is expected to be compute-bound. Therefore, we need to increase the number of memory channels to be able to fulfill the maximum bandwidth requirements of six OoO cores of Xeon x5670.

Figure 5.4 depicts the normalized execution time of our pointer-chasing workload with a 2048 MB dataset on OoO (Xeon x5670) including the AMAC execution with and without an optimized memory subsystem. The first two bars (OoO and OoO + AMAC) are real hardware measurements on Xeon x5670 and we observe that OoO + AMAC achieves almost 7x

Table 5.2 – Memory subsystem parameters.

Threads	Baseline	Optimized Memory Subsystem
VM Page Size	2MB	1GB
L1-D MSHRs	10-entry	16-entry
Max. Memory B/W	32GB/s (3 DDR3-1333Mhz channels)	76.8GB/s (6 DDR3-1600MHz channels)

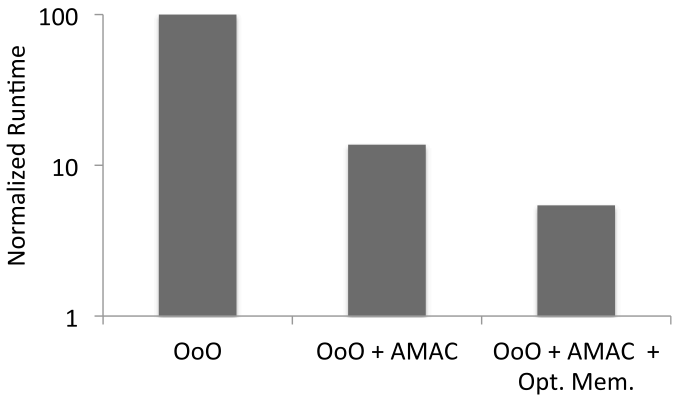


Figure 5.4 – Normalized execution time of AMAC with optimized memory subsystem compared to Xeon x5670.

improvement over the baseline OoO thanks to high-MLP execution enabled by AMAC . By using Equation 5.1 with parameters in Table 5.2, we estimate the performance benefits of AMAC with the optimized memory system parameters and find that the optimized memory subsystem delivers 2.5x improvement over OoO + AMAC and 18x over baseline OoO by achieving purely compute-bound execution, while maximizing the MLP extracted during the execution. Furthermore, we also estimate that the optimized design with a single core will sature one DDR3-1600 MHz memory channel, therefore maximizing the utilization of entire memory hierarchy.

5.7 Combining Hardware and Software Acceleration

The results presented so far indicate that optimizing memory hierarchy can enable dramatic improvements in pointer-chasing throughput while utilizing the available off-chip memory bandwidth. Therefore, a next logical step is to target improving the energy efficiency of the system by mitigating the inefficiencies of the general-purpose OoO cores. To do so, we will leverage specialized Widx units described in Chapter 4 and program them to run the AMAC code to improve the energy efficiency of the system, while achieving the same off-chip bandwidth utilization with the general-purpose OoO running the AMAC code.

The key aspect of AMAC is that the MLP generated during the pointer-chasing operations is a function of how fast the computation instructions are executed to issue non-blocking load (i.e., prefetch) operations as opposed to the baseline execution, which solely relies on the OoO instruction window capacity to generate MLP. Therefore, AMAC on Widx can also generate MLP as long as blocking load operations are converted to prefetch instructions. Luckily, the Widx ISA already supports prefetch instructions (i.e., *touch* instruction) and uses the L1-D cache of the OoO, which supports multiple outstanding misses through its MSHRs. Therefore a single Widx unit coupled with OoO can generate MLP without requiring the complex instruction window.

The main challenge, however, is that each Widx unit is inferior to an OoO core (e.g., Xeon x5670) in terms of computation capabilities such as the ability to extract instruction-level parallelism, the ability to hide L1-D cache access latency, and the operating frequency. To estimate the performance degradation of these aspects, we leverage our first-order performance model described in Section 4.2 and find that a single Widx unit is 5.8x slower than an OoO for every hash table node traversal. At the same time, each Widx unit is expected to generate an MLP of 2.5 on average. Although possible enhancements to the Widx unit design can improve the single Widx unit performance, such as managing the lookup state in hardware buffers to avoid L1-D cache accesses or shrinking the size of L1-D cache, we do not investigate such options as we can increase the number of Widx units to keep the design simple. As a result, we estimate that six Widx units running the AMAC code will reach the same performance as the OoO running AMAC code.

Figure 5.5 shows the energy consumption of all the designs presented normalized to the energy consumed by the baseline OoO. As expected, given the speedups achieved by OoO + AMAC

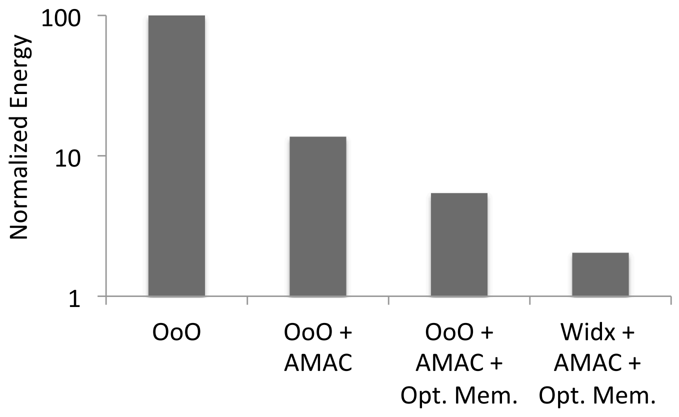


Figure 5.5 – Energy benefits of AMAC and Widx with optimized memory subsystem.

and OoO + AMAC + optimized memory subsystem , these designs deliver 7x and 18.5x energy reduction over the OoO respectively. Overall, offloading the execution to the Widx units with AMAC execution improves the energy efficiency over the best performing general-purpose design (OoO + AMAC + Opt. Mem.) by 2.5x, which translates to almost 50x energy efficiency improvement over the baseline OoO. It is important to note that, in Widx execution, most of the energy consumption comes from the idle energy of OoO execution.

To conclude, software and hardware acceleration schemes are individually effective in improving the performance and efficiency of the system, however, our results show that there is an interplay between the acceleration schemes and the memory subsystem. Therefore, we believe that our approach underscores the importance of holistic designs for future specialized data-centric systems.

5.8 Discussion

Heterogeneous Architectures: The ideas presented in this chapter can be applied to heterogeneous architectures for improving performance and efficiency. The prime examples of heterogeneous designs include ARM big.LITTLE [7] and AMD Fusion Architectures [6]. These heterogeneous architectures either offer efficient in-order cores similar to the Widx units or highly threaded GPU cores that target increasing memory-level parallelism akin to AMAC execution. The main advantage of these architectures is the unified coherent memory enabling fast communication across different computation units for seamless execution. The benefit of Widx over these architectures is the tightly coupled design, which avoids the need for dedicated L1-D caches and MMUs for every core present in heterogeneous CMPs. Similarly, the software-managed buffer in AMAC mitigates the need for per-thread hardware register file in GPUs, therefore achieving an area-efficient design, while extracting high MLP within a single compute unit. Overall, the presence of such heterogeneous hardware in the system offers a different area-performance-energy tradeoff, given the degree of specialization of our approach is not the same. Nonetheless, heterogeneous CMPs are promising and create an important opportunity for improving the throughput and efficiency of data structure traversals.

Compute-intensive operations: In this chapter, we leveraged a pointer-chasing workload with minimal computation requirements to reveal the bottlenecks in the memory subsystem. However, some data structure traversals such as shallow hash tables and group-by operations with several aggregation functions can be computationally intensive during hashing functions and mathematical calculations (e.g., min, max, avg). In the presence of compute-intensive data structure traversals, we expect that the importance of the memory optimizations presented in this chapter will gracefully degrade depending on the intensity of the required computation. In such cases, the AMAC code should take advantage of the SIMD instructions to reduce the ops executed per memory access wherever it is applicabile. For Widx acceleration, the compute-intensive operations can be staged by taking advantage of the Widx topology like we demonstrated for hash index traversals in Chapter 4.

6 Related Work

6.1 Software and Hardware Prefetching

Software prefetching instructions are effective in hiding the memory access latency when the required cache blocks can be demanded early enough. Unfortunately, prefetching within the traversal of single pointer chain is not possible due to dependent address calculations. To solve this problem, prior work has proposed data-linearization prefetching [52] to calculate the addresses without needing pointers so that the prefetches can be issued ahead of time. Similarly, history-based prefetching techniques in software [15] or in hardware [65] maintain an array of jump pointers containing the pointers from recent traversals. However, these techniques either assume that the data structure is traversed in a similar order more than once or incur both space and time overhead to increase the prefetch accuracy.

Hardware data prefetching techniques for pointer-intensive data structures [23, 64, 88] eliminate memory stalls by predicting the pointer accesses ahead of the core. These techniques analyze the data structure traversal instructions in a separate hardware context to produce future pointer accesses. However, they can be either applied to simple data structure layouts, which can only be accessed by address offsets or they require specialized computation hardware, which is similar in spirit to our approach.

6.2 Hardware-Conscious Algorithms

Hardware conscious algorithms and data structures have gained importance in the last decade [13, 45, 53, 54, 89, 93]. The data structure layout tuned to the underlying hardware is a promising approach for reducing the number of cache and TLB misses. In addition, SIMD instructions can be used to minimize the number of individual memory accesses [46]. Our work is orthogonal and can be used in conjunction with these techniques. Moreover, our approach is a step towards robust performance and tuning [31] for hardware conscious algorithms.

6.3 Decoupled Architectures

There have been several proposals that targeted to decouple and overlap pointer production latency with the pointer consumption latency through a hardware or software queue akin to Decoupled Access/Execute architectures [26, 63, 70]. To take advantage of the extra hardware contexts on the processor cores, Zhou et al. [91] divide a single software thread into producer and consumer threads, which communicate through shared memory via software queues. Ideally, the two contexts can work cooperatively to overlap long-latency memory misses with useful work. However, the synchronization overhead in the software queue negates the benefits of overlapping, as the producer and consumer threads have read/write dependencies on the software queue. To mitigate such synchronization overheads, speculative decoupled software pipelining [82] splits the recursive data structure traversal into two hardware contexts and performs synchronization via specialized hardware arrays with support for dealing with data dependencies. While such approach is beneficial for extracting parallelism out of a single-threaded program, we observe that in the case of inter-lookup parallelism (i.e., thread-level parallelism) almost all the complexity of the specialized hardware queue can be eliminated by managing the lookup state in software.

6.4 Specialized Hardware and Accelerators

Recent work has proposed on-chip accelerators for energy efficiency (dark silicon) in the context of general-purpose (i.e., desktop and mobile) workloads [25, 29, 32, 33, 36, 67, 83]. While some of these proposals target improving the efficiency of the memory access operations, the applicability of the proposed techniques to database workloads is limited due to the deep software stacks and vast datasets in today's server applications. Also, existing dark silicon accelerators are unable to extract memory-level parallelism, which is essential to boost the efficiency of data structure operations.

1980s witnessed proliferation of database machines, which sought to exploit the limited disk I/O bandwidth by coupling each disk directly with specialized processors [22]. However, high cost and long design turnaround time made custom designs unattractive in the face of cheap commodity hardware. Today, efficiency constraints are rekindling an interest in specialized hardware for DBMSs [18, 28, 37, 55, 85]. Some researchers proposed offloading hash-joins to network processors [28] or to FPGAs [18] for leveraging the highly parallel hardware. However, these solutions incur invocation overheads as they communicate through PCI or through high-latency buses, which affect the composition of multiple operators. Moreover, offloading the joins to network processors or FPGAs requires expensive dedicated hardware, while Widx utilizes the on-chip dark silicon.

Support for specialized vector instruction extensions for hash table probes [35] aims to exploit inter-lookup parallelism in similar spirit to our approach. Although promising, the work has several important limitations. One major limitation is the DBMS-specific solution, which is limited to the *Vectorwise* DBMS. Another drawback is the vector-based approach, which limits performance due to the lock-stepped execution in the vector unit. Finally, the vector-based approach keeps the core fully engaged, limiting the opportunity to save energy.

7 Concluding Remarks

Businesses, governments and societies are increasingly relying on collecting, analyzing and exchanging data to improve their products, services and ultimately to enhance our lives. The fundamental operation of data processing is to locate and extract the vital pieces of knowledge from a dataset. Given the copious data, a linear search through the entire dataset leads to a prohibitively large response time. Therefore, modern data processing systems rely on pointer-intensive data structures, which convert linear search time to sub-linear search time. While pointer-intensive data structures are essential for all data processing systems, vast datasets and dependent access behavior of the operations compromise performance and energy efficiency on contemporary processors.

In this thesis, we made the observation that pointer-intensive data structure lookups have abundant inter-lookup parallelism in data analytics workloads. Exploiting such inter-lookup parallelism requires dynamism in dealing with irregularity across lookups. We proposed *Asynchronous Memory Access Chaining* (AMAC) , a new software acceleration approach to hide memory access latency in the presence of regular and/or irregular lookups. We showed that dealing with irregular lookups require maintaining the state of each lookup separately from other lookups' state. As a result, AMAC achieves high flexibility in exploiting inter-lookup parallelsim even in the presence of irregularity in the data structure traversal path or data structure layout.

We further analyzed a modern in-memory database system (MonetDB), on a set of commercial data analytics workloads showed that hash index operations are the largest single contributor to the overall execution time. Nearly all of the index execution time is split between ALU-intensive key hashing operations and memory-intensive node list traversals. These observations, combined with a need for energy-efficient silicon mandated by the slowdown in supply voltage scaling, led to *Widx*, an on-chip accelerator for index operations. As a building block, we leveraged a set of programmable and simple hardware units to achieve high performance, efficiency, and flexibility.

While both software and hardware acceleration schemes are extremely effective at improving the performance efficiency, acceleration benefits go hand in hand with the memory subsystem. We presented the most important aspects of the memory subsystem design and studied their impact on the accelerated mechanisms. We quantified the impact of the virtual memory page size, number of miss status handling registers and data structure padding. We further performed a throughput scalability analysis and revealed bottlenecks on the shared on-chip resources. In light of these optimizations, we showed that holistically designed accelerators will enable dramatic improvements in throughput and efficiency of the future data-centric systems.

7.1 Future Directions

There are several important challenges in customizing the processor micro-architecture for higher efficiency and performance. First, reducing the number of computation instructions during data structure lookups is essential for accelerators. Therefore, a natural next step is to leverage customized instructions to reduce the number of computation operations in AMAC-style execution. Second, our results identified important bottlenecks in modern server processors, which limit the amount of MLP that is extracted by software. Given the growing importance of analytics in today's business and society, and the futility of caching the ever-growing datasets on-chip, it is imperative that processor designs evolve not to limit software's ability to exploit MLP. Therefore, we plan to study practical miss handling structures for general-purpose processors as well as the emerging near-memory processing techniques, which do not rely on traditional cache structures.

While our evaluation focused on pointer-intensive operations in the context of relational databases, we believe that applicability of AMAC and Widx goes beyond that. Our future work will examine the efficacy of our approaches on graph workloads and operations over unstructured data, which have high degrees of irregularity in data structure lookups. Moreover, recent in-memory database systems explore the use of additional index structures besides the hash tables and trees, such as skip lists [77]. Therefore, our accelerators will likely to maintain their effectiveness in improving the performance and efficiency of the emerging in-memory systems.

Bibliography

[1] Oracle's SPARC T4-1, SPARC T4-2, SPARC T4-4, and SPARC T4-1B server architecture. http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o11-090-sparc-t4-arch-496245.pdf.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[3] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[5] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB Endowment*, 5(10), 2012.

[6] AMD Compute Cores. http://www.amd.com/en-us/innovations/software-technologies/processors-for-business/compute-cores.

[7] ARM big.LITTLE Technology. http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[8] ARM M4 Embedded Microcontroller. http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php.

[9] C. Balkesen, G. Alonso, and M. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *VLDB Endowment*, 7(1), 2013.

[10] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 29th International Conference on Data Engineering*, 2013.

[11] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on modern processor architectures. In *IEEE Transactions on Knowledge and Data Engineering*, volume PP, 2014.

[12] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. 8(4), 2014.

[13] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011.

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems*, 32(3), 2007.

[15] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 2001.

[16] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[17] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *VLDB Endowment*, 1(2), 2008.

[18] E. S. Chung, J. D. Davis, and J. Lee. LINQits: Big data on little clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[19] H. T. Davenport. Competing on analytics. *Harvard Business Review*, Jan. 2006.

[20] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, 2011.

[21] DesignWare ARC HS38 Processor. http://www.synopsys.com/dw/ipdir.php?ds=arc-hs38-processor.

[22] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The GAMMA database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[23] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the 15th Annual Symposium on High Performance Computer Architecture*, 2009.

[24] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[25] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[26] A. Garg and M. Huang. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*, 2008.

[27] Global Server Hardware Market 2010-2014. http://www.technavio.com/content/global-server-hardware-market-2010-2014.

[28] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware*, 2005.

[29] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 17th Annual International Symposium on High Performance Computer Architecture*, 2011.

[30] G. Graefe. Database servers tailored to improve energy efficiency. In *Proceedings of the 2008 EDBT Workshop on Software Engineering for Tailor-made Data Management*, 2008.

[31] G. Graefe. Robust query processing. In *Proceedings of the 27th International Conference on Data Engineering*, 2011.

[32] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[33] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[34] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4), 2011.

[35] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero. Vector extensions for decision support DBMS acceleration. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[36] C.-H. Ho, S. J. Kim, and K. Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd International Symposium on Computer Architecture*, 2015.

[37] IBM Netezza Data Warehouse Appliances. http://www-01.ibm.com/software/data/netezza/.

[38] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1), 2012.

[39] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[40] Intel Advanced Vector Extensions Programming Reference Programming Reference. https://software.intel.com/sites/default/files/4f/5b/36945.

[41] Intel Memory Latency Checker v2.0. http://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[42] Intel Vtune. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[43] Intel Xeon Processor 5600 Series Datasheet, Vol 2. http://www.intel.com/content/www/us/en/processors/xeon/xeon-5600-vol-2-datasheet.html.

[44] Intel's Haswell CPU Microarchitecture. http://www.realworldtech.com/haswell-cpu/5/.

[45] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. 8(6), 2015.

[46] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[47] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. In *Proceedings of the 35th International Conference on Very Large Data Bases*, 2009.

[48] W. Lang, R. Kandhan, and J. M. Patel. Rethinking query processing for energy efficiency: Slowing down to win the race. *IEEE Data Eng. Bull.*, 34(1), 2011.

[49] W. Lang and J. M. Patel. Towards eco-friendly database management systems. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*, 2009.

[50] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[51] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[52] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[53] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4), 2002.

[54] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3), Dec. 2000.

[55] R. Mueller, J. Teubner, and G. Alonso. Glacier: A query-to-hardware compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[56] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[57] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered instructions:

A control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[58] Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[59] M. Poess, R. O. Nambiar, and D. Walrath. Why you should run TPC-DS: A workload analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.

[60] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st International Symposium on Computer Architecture*, 2014.

[61] J. Rao and K. A. Ross. Making B+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[62] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. Joulesort: A balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 2007.

[63] W. W. Ro, S. P. Crago, A. M. Despain, and J.-L. Gaudiot. Design and evaluation of a hierarchical decoupled architecture. *The Journal of Supercomputing*, 38(3), 2006.

[64] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[65] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[66] S. Rusu, S. Tam, H. Muljono, D. Ayers, and J. Chang. A dual-core multi-threaded Xeon processor with 16MB L3 cache. In *Solid-State Circuits Conference*, 2006.

[67] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. Taylor. Efficient complex operators for irregular codes. In *Proceedings of the 17th Annual International Symposium on High Performance Computer Architecture*, 2011.

[68] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *Micro, IEEE*, 29(1), 2009.

[69] J. E. Short, R. E. Bohn, and C. Baru. How Much Information? 2010 Report on Enterprise Server Information, 2011.

[70] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4), Nov. 1984.

[71] SPARC Strong Prefetch. https://blogs.oracle.com/d/entry/strong_prefetch.

[72] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[73] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.

[74] Synopsys Design Compiler. http://www.synopsys.com/.

[75] Teradata Data Warehouse Appliance. http://www.teradata.com/data-appliance/.

[76] J. Teubner, L. Woods, and C. Nie. Skeleton automata for FPGAs: Reconfiguring without reconstructing. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[77] The Story Behind MemSQL's Skiplist Indexes. http : / / blog . memsql . com / the-story-behind-memsqls-skiplist-indexes/.

[78] The TPC-H Benchmark. http://www.tpc.org/tpch/.

[79] Transaction Processing Performance Council (TPC) Launches Energy Performance Specification. http://www.tpc.org/information/press/tpcpress20100202.asp.

[80] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[81] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[82] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[83] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[84] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4), 2006.

[85] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[86] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[87] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[88] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 14th International Conference on Supercomputing*, 2000.

[89] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *Proceedings of the 7th International Workshop on Data Management on New Hardware*, 2011.

[90] T. Yoshida. SPARC64 X+: Fujitsu's Next Generation Processor for UNIX Servers . HotChips: A Symposium on High Performance Chips, 2013.

[91] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[92] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[93] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proceedings of the 29th International Conference on Very Large Data Bases*, 2003.

Yusuf Onur KOÇBERBER
EPFL IC ISIM PARSA INJ 215 (Bâtiment INJ)
Station 14, CH-1015, Lausanne, Switzerland
e-mail: onur.kocberber@epfl.ch
url: http://parsa.epfl.ch/~kocberbe

RESEARCH INTERESTS

- Design for Dark Silicon
- Efficient server architectures for large-scale datacenters
- Hardware acceleration for software debugging and security

EDUCATION

- Doctor of Philosophy (PhD), Parallel Systems Architecture Lab, 10/2009 – 9/2015
  École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
  Advisor: Prof. Babak Falsafi
  Thesis title: Accelerators for Data Processing

- Master of Science (MS), Computer Engineering, 09/2008 – 09/2009
  TOBB University of Economics and Technology, Ankara, Turkey
  Advisor: Asst. Prof. Oğuz Ergin
  Thesis title: Reducing Static Energy Dissipation of Data-Holding Components of Modern Microprocessors

- Bachelor of Science (BS), Electrical and Electronics Engineering, 09/2005 – 08/2008
  TOBB University of Economics and Technology, Ankara, Turkey
  Senior Design Project Title: FPGA Implementation of a 16-bit Microprocessor

HONORS and AWARDS

- Google Europe Doctoral Fellowship in Computer Systems, 2014
- IEEE Micro Top Picks from Computer Architecture Conferences for "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware", 01/2014

- Best Paper Runner-up Award at the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13) for "Meet the Walkers: Accelerating Index Traversals for In-Memory Databases", 12/2013

- ACM Turing Centenary Celebration Student Scholarship ($1K prize), 04/2012

- Best Paper Award at the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) for "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware", 03/2012

- TOBB University of Economics and Technology fellowship for graduate study covering full tuition and stipend, 09/2008 – 09/2009

- Awarded second prize at CPU-Turkey competition by designing a microprocessor architecture for FPGA (www.cpu-turkey.com), 09/2008

PUBLICATIONS

Refereed Conference Publications

1. FADE: A Programmable Filtering Accelerator for Instruction-Grain Monitoring
S. Fytraki, E. Vlachos, O. Kocberber, B. Falsafi and B. Grot. *In 20th International Symposium on High Performance Computer Architecture (HPCA'14), Orlando, FL, USA, 2015.*

2. Meet the Walkers: Accelerating Index Traversals for In-Memory Databases
O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim and P. Ranganathan. *In 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Davis, CA, USA, 2013.*
(recognized as Best Paper Runner-up by the program committee).

3. Scale-Out Processors
P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer and B. Falsafi. *In 39th International Symposium on Computer Architecture (ISCA), Portland, OR, USA, 2012.*

4. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware
M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A.D. Popescu, A. Ailamaki and B. Falsafi, *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), London, UK, 2012.*
(recognized as Best Paper by the program committee).

5. Dynamic Register File Partitioning in Superscalar Microprocessors for Energy Efficiency
M. Ozsoy, Y. O. Koçberber, M. Kayaalp, O. Ergin, *In 28th International Conference on Computer Design* (ICCD), *Amsterdam, Netherlands, 2010.*

6. Reducing Parity Generation Latency through Input Value Aware Circuits
Y. Osmanlioglu, Y. O. Koçberber, O. Ergin, *In 19th ACM Great Lakes Symposium on VLSI (GLSVLSI), Boston, MA, USA, 2009.*

Journal Publications

7. Quantifying the Mismatch between Emerging Scale-Out Applications and Modern Processors
M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A.D. Popescu, A. Ailamaki and B. Falsafi, *In ACM Trans. Comput. Syst., ACM, vol 30, 2012.*

8. Reducing the energy dissipation of the issue queue by exploiting narrow immediate operands
I. C. Kaynak, Y. O. Kocberber, O. Ergin, *In Journal of Circuits, Systems, and Computers, vol. 19, No.8, 2010.*

9. Exploiting Narrow Values for Faster Parity Generation
Y.O. Kocberber, Y. Osmanlioglu, O. Ergin, *In Microelectronics International, Vol 26, No.3, 2009.*

Refereed Workshop Publications

10. Sort vs. Hash Join Revisited for Near-Data Execution.
N. S. Mirzadeh, O.Kocberber, B. Falsafi and B. Grot. *5th Workshop on Architectures and Systems for Big Data (ASBD). Portland, OR, USA 2015 (co-located with ISCA).*

11. Dark Silicon Accelerators for Database Indexing
O. Kocberber, B. Falsafi, K. Lim and P. Ranganathan, *1st Dark Silicon Workshop (DaSi), Portland, OR, USA, 2012 (co-located with ISCA)*

TEACHING ASSINTANSHIPS

- Topics on Approximate Computing Systems, Spring 2015.
- Advanced Multiprocessor Architecture, Fall 2013.
- Topics on Datacenter Design, Spring 2013, 2014.
- Introduction to Multiprocessor Architecture (undergraduate), Spring 2010-2012.

EXPERIENCE

- HP Labs, Palo Alto, CA
  Research Intern at Intelligent Infrastructure Lab,
  07-08/2011 and 03-08/2012.
  Mentor: Parthasarathy Ranganathan

- TOBB University of Economics and Technology,
  Department of Computer Engineering, Ankara, Turkey
  Research assistant, 09/2008 – 09/2009
  "Detecting and exploiting narrow value phases for energy efficiency by analyzing the workloads of high performance superscalar microprocessors that employs out-of-order execution" project funded by The Scientific and Technological Research Council of Turkey

- Vivante Corporation CA, Ankara, Turkey
  Remote Engineering Intern, 04/2008 - 08/2008
  GPU implementation on FPGA for embedded systems

- Turkish Aerospace Industry (TAI), Ankara, Turkey
  Engineering intern at the Avionics department, 01/2006 - 04/2006 and 04/2007 - 08/2007

PROFESSIONAL ACTIVITIES

- Primary architect of CloudSuite a benchmark suite for scale-out applications.

- Co-developer of Flexus, an open-source, scalable, full-system, cycle-accurate multi-processor and multi-core simulation framework.

- Tutorial Organizer & Presenter: Rigorous and Practical Server Evaluation.
  ISPASS - 2014, Monterey CA, USA.
  EPFL - 2015, Lausanne, Switzerland.

PATENTS

- Method of Using a Buffer Within an Indexing Accelerator During Periods of Inactivity
  O. Kocberber, K. Lim and P. Ranganathan, USPTO #8,984,230. Granted Mar. 2015.

- Indexing Accelerator for Memory-Level Parallelism Support
  O. Kocberber, K. Lim and P. Ranganathan, PCT/US13/53040. Filed July 2013.

- Executing Requests from Processing Elements with Stacked Memory Devices
  O. Kocberber, K. Lim and P. Ranganathan, USPTO #13/755,661. Filed Jan. 2013.