Performance Analysis and Acceleration of Learned Hash-Indexes on Supercomputing Architectures

Montserrat Palazón Balmaseda

Bachelor

Double degree in Computer Engineering and Biotechnology

Academic tutor

Carlos Molina Clemente (Department of Computer Engineering and Mathematics, URV)

Thesis supervisor

Noelia Oliete Escuín (Barcelona Supercomputing Center)

Santiago Marco Sola (Department of Computer Science, UPC)

Bachelor's thesis

Escola Tècnica Superior d'Enginyeria (ETSE)

Universitat Rovira i Virgili (URV)

Tarragona, 06/06/2025

Acknowledgments

I would like to thank Santiago Marco Sola for welcoming me into his group and offering me this project with such enthusiasm. Thank you for your guidance, advice, motivation, and time, and for making it so easy and enjoyable to work with you. I am also grateful to both Miquel Moretó and Santiago for giving me this opportunity.

A special thanks to Noelia Oliete, for being such an inspiring supervisor. Thank you for always being there to help, listen, and teach, because when you enjoy what you do, others can feel it too. And that is exactly what happens between you and hardware.

I would also like to thank the entire group for the warm welcome and for always being willing to lend a hand.

I am also thankful to my academic tutor, Carlos Molina, for his attention, valuable suggestions, and guidance during the writing.

Finally, I would like to thank my family and friends for their unconditional support. To my dad, thank you for unintentionally passing on your curiosity and interest in computer science. And to my parents and brother, thank you for your trust, encouragement, and for always being by my side. None of this would have been possible without you.

Abstract

Recent advances in sequencing technologies have made population-scale genome analysis a reality, enabling the discovery of valuable biological insights. However, the exponential growth of genomic data poses significant challenges for the performance scalability of genome analysis tools that require accessing large sequence databases. For instance, mapping long DNA sequences to a large reference genome is one of the most time-consuming steps in many genome sequencing analyses. In particular, seeding algorithms, which locate short DNA fragments in a reference genome, often become a major performance bottleneck in genome mapping tools.

For that, many performance-critical tools rely on optimized hash-tables to perform fast database lookups of DNA sequences. Despite their well-known efficiency and performance, hash-table-based tools suffer from irregular memory access patterns and limited spatial locality, limiting their performance in modern hardware architectures. The recently proposed learned index strategies have shown promise in accelerating traditional data structures, such as hash tables, by leveraging machine-learned models, such as RMI, to predict the location of keys and reduce the number of memory accesses. Notwithstanding, the performance of learned hash-tables remains constrained by low instruction-level parallelism, poor cache locality, and underutilized memory bandwidth.

Our evaluation shows that carefully optimized learned hash indexes can outperform traditional hash tables in memory-bound scenarios. The integration of an RMI into Minimap2, combined with software-level optimizations such as batching, prefetching, and vectorization, achieves a speedup of 2.90× over the baseline. These performance improvements are accompanied by a 3.87× reduction in MPKI and a significant decrease in memory-bound cycles, from 66.80% to 13.65%, highlighting the effectiveness of learned indexes when adapted to modern processor architectures.

Keywords: hash tables, learned indexes, genome analysis, read mapping, seeding, memory-bound, batching, prefetching, vectorization, minimap2, mm2-fast

Resum

Els recents avenços en les tecnologies de seqüenciació han fet realitat l'anàlisi genòmica a escala poblacional, permetent descobrir coneixements biològics valuosos. Tanmateix, el creixement exponencial de les dades genòmiques planteja grans reptes per a l'escalabilitat del rendiment de les eines d'anàlisi genòmica, que han d'accedir a grans bases de dades de seqüències. Per exemple, mapejar seqüències llargues d'ADN contra un genoma de referència és un dels passos més costosos en moltes anàlisis. En particular, els algorismes de *seeding*, que localitzen fragments curts d'ADN dins d'un genoma de referència, sovint esdevenen un coll d'ampolla important per al rendiment.

Per tal d'agilitzar aquest procés, moltes eines crítiques utilitzen taules hash optimitzades per fer cerques ràpides. Tot i la seva eficiència, pateixen patrons d'accés a memòria irregulars i baixa localitat espacial, limitant el seu rendiment en arquitectures modernes. Recentment, els *learned indexes* han sorgit com una alternativa prometedora, fent servir models, com l'RMI, per predir la posició de les claus i reduir el nombre d'accessos a memòria. Tot i això, el seu rendiment es veu restringit per un paral·lelisme limitat, baixa localitat de memòria cau i una utilització ineficient de l'amplada de banda de memòria.

Els nostres resultats mostren que els *learned hash indexes*, quan són optimitzats amb cura, poden superar les taules hash tradicionals en escenaris limitats per memòria. La integració d'un RMI a Minimap2 combinat amb optimitzacions a nivell software com *batching*, *prefetching* i vectorització, aconsegueix una acceleració de 2.90× respecte al *baseline*. A més, redueix el MPKI en 3.87× i els cicles dedicats a memòria passen del 66.80% al 13.65%, evidenciant l'eficàcia dels *learned indexes* quan s'adapten a les arquitectures de processador modernes.

Paraules clau: taules hash, learned indexes, anàlisi genòmic, mapatge de lectures, seeding, memory-bound, batching, prefetching, vectorization, minimap2, mm2-fast

Resumen

Los recientes avances en las tecnologías de secuenciación han hecho posible el análisis genómico a escala poblacional, permitiendo descubrir conocimientos biológicos valiosos. Sin embargo, el crecimiento exponencial de los datos genómicos plantea importantes retos para la escalabilidad del rendimiento de las herramientas de análisis genómico, que requieren acceder a grandes bases de datos de secuencias. Por ejemplo, mapear secuencias largas de ADN contra un genoma de referencia es uno de los pasos más costosos en muchos análisis. En particular, los algoritmos de *seeding*, que localizan fragmentos cortos de ADN en un genoma de referencia, suelen convertirse en un cuello de botella para el rendimiento de las herramientas de mapeo.

Para acelerar este proceso, muchas herramientas emplean tablas hash optimizadas para realizar búsquedas rápidas. A pesar de su eficiencia, presentan patrones de acceso a memoria irregulares y una baja localidad espacial, limitando su rendimiento en arquitecturas modernas. Recientemente, las estrategias de *learned indexes* han demostrado potencial para acelerar estructuras tradicionales como las tablas hash, mediante modelos, como los RMI, para predecir la ubicación de las claves y reducir así el número de accesos a memoria. No obstante, su rendimiento sigue limitado por un bajo paralelismo a nivel de instrucciones, una baja localidad en memoria y un bajo aprovechamiento del ancho de banda de memoria.

Nuestros resultados muestran que los *learned hash indexes*, cuando se optimizan cuidadosamente, pueden superar a las tablas hash tradicionales en escenarios limitados por memoria. La integración de un RMI en Minimap2 combinado con optimizaciones a nivel software como *batching*, *prefetching* y vectorización, logra una aceleración de 2.90× respecto al *baseline*, una reducción de 3.87× en el MPKI y una disminución de los ciclos dedicados a memoria del 66.80% al 13.65%, lo que demuestra la eficacia de los *learned indexes* cuando se adaptan a arquitecturas modernas de procesador.

Palabras clave: tablas hash, learned indexes, análisis genómico, mapeo de lecturas, seeding, memory-bound, batching, prefetching, vectorization, minimap2, mm2-fast

Contents

List of Figures

List of Algorithms

List of Tables

1 Introduction

1.1 Context and Motivation

In the past decade, the volume of genomic data has grown rapidly due to improvements in sequencing technologies. In particular, long-read technologies such as Oxford Nanopore and PacBio have enabled more complete genome assemblies and the detection of complex structural variations. These advances are transforming modern genome biology and health care research. However, current genome analysis pipelines are computationally demanding, requiring significant time and resources.

Notably, one of the most computationally demanding steps in genome analysis is read mapping. This task consists of finding where small fragments of DNA, called reads, come from in a known reference genome. The mapping process must allow inexact matches to account for sequencing errors and small mismatches. For that, most mapping tools divide the task into three stages: seeding, chaining, and alignment. In particular, the seeding phase is responsible for finding short exact matches between the read and the reference. These matches serve as starting points for the rest of the process. Notably, the seeding stage is particularly time-consuming because it requires frequent and repeated access to large volumes of data (e.g., large genomics), over a large memory footprint, which places significant stress on the memory hierarchy.

Many read-mapping tools, such as the popular Minimap2, use hash tables to index a minimum representative set of all reads from the reference genome, known as minimizers. Hash tables offer constant-time $O(1)$ lookups in theory, but in practice, they access memory in an irregular and unpredictable way. These random accesses occur throughout a large memory footprint, leading to poor cache usage and inefficient use of the memory hierarchy. As a result, hash-based seeding can limit performance, especially on modern processors with deep cache hierarchies.

To reduce these inefficiencies, learned hash indexes based on models like Recursive Model Indexes (RMIs) have been proposed as an alternative. These data structures use a lightweight machine-learning model to predict where a hash key is located in a sorted array, which helps minimize the number of memory accesses. Compared to traditional hash tables, learned indexes tend to reduce the number of cache misses and improve lookup efficiency in sorted arrays.

Despite their theoretical efficiency, traditional hash and learned indexes pose challenges for modern hardware. Their performance depends not only on algorithmic complexity but also on how they interact with the memory hierarchy and processor features. Large memory footprints, irregular access patterns, and limited spatial locality can lead to frequent cache misses and underutilization of available bandwidth. As a result, the actual efficiency of these data structures is strongly limited by how well they make use of the CPU computer architecture.

These limitations highlight the need for data structures that are not only algorithmically efficient but also adapted and optimized to the characteristics of the hardware. To fully exploit modern CPUs and their resources (e.g., cache hierarchies, wide vector units, and

high memory bandwidth), hash structures must be designed with memory access patterns, data locality, and compute efficiency in mind. Performance-aware indexing approaches are essential to bridge the gap between theoretical design and practical hardware efficiency.

1.2 Research Problem and Goal

Despite the good and promising theoretical properties of hash and learned hash indexes, their performance in real-world applications remains limited. While they can reduce the number of memory accesses in theory, their execution speed and efficiency vary depending on how they interact with system memory and the processor architecture. In practice, their performance is limited when used in demanding scenarios such as genomic read mapping.

A key question we want to address in this work is whether learned hash indexes can outperform traditional hash tables in realistic genome analysis scenarios.

In practice, many factors can contribute to the performance limitations of learned hash indexes. These include inefficient memory use, frequent cache misses, underutilization of available memory bandwidth, and poor exploitation of SIMD instructions. Such inefficiencies can occur from irregular memory access patterns, lack of data locality, and control flow that depends heavily on the data. When combined with the large memory footprint typical of genomic workloads, like large genomic references, these factors can prevent learned indexes from efficiently exploiting the capabilities of modern multicore processors.

This work aims to characterize and quantify these bottlenecks through methodological performance profiling. Using a range of tools (including hardware counters, instruction-level analysis, and cache profiling), this study aims to identify the causes of performance inefficiencies across different application configurations and use-case scenarios.

To answer this, we analyze the effects of several optimization techniques at the software level (e.g., batching and prefetching) and the hardware level (e.g., efficient cache usage and SIMD-aware design). The goal is to identify which parameter configurations and optimizations are most effective in closing the performance gap and making learned indexes more suitable for genomic applications.

We hypothesize that learned hash tables, if adapted and optimized for modern CPU architectures, can outperform traditional hash tables in seeding tasks. However, this requires addressing their interaction with the memory hierarchy and making explicit use of hardware features such as cache prefetching and vector units.

1.3 Project Objectives

The main goal of this thesis is to investigate whether learned hash indexes can outperform traditional hash tables used in HPC genome read mapping applications executed on modern CPU architectures. To this end, the following General Objectives **(GO)** and Specific Objectives **(SO)** have been identified.

GO1: Investigate the algorithmic design, data structure properties, and main operations of traditional and learned hash indexes used in genomic read mapping.

- SO1.1: Review the algorithmic foundations of traditional hash tables and their usage in minimizer query tasks for the seeding stage of read mapping.
- SO1.2: Examine the algorithmic structure and functioning of learned hash indexes, including Recursive Model Indexes (RMI) and associated last-mile search technique.
- SO1.3: Compare the algorithmic complexity, memory layout, and data access patterns of traditional versus learned index structures.

GO2: Analyse and characterize the performance of traditional and learned hash indexes for read mapping on an HPC platform and identify critical software-level and microarchitectural performance bottlenecks.

- SO2.1: Quantify instruction throughput, IPC, memory miss ratio, cache behavior, and instruction misspeculation using hardware performance counters.
- SO2.2: Analyse memory access patterns and cache spatial locality to evaluate their impact on cache efficiency and pipeline stall cycles.
- SO2.3: Characterize traditional and learned hash index performance and determine whether it is limited by memory bandwidth (memory-bound) or compute resources (compute-bound).

GO3: Explore and evaluate optimization techniques to improve the efficiency of learned hash indexes.

- SO3.1: Implement and evaluate software-level strategies such as batched query processing and software prefetching.
- SO3.2: Incorporate SIMD vectorization in the last-mile search and evaluate its effect on performance throughput.
- SO3.3: Perform a design space exploration on RMI configurations to optimize lookup performance on learned hash indexes.

GO4: Evaluate the scalability of learned hash indexes on an HPC platform.

- SO4.1: Measure multicore scalability on a HPC supercomputing node, such as MareNostrum 5.

GO5: Produce recommendations for deploying and accelerating learned indexes in production-ready genomic applications.

Moreover, from an educational perspective, this thesis offers the opportunity to gain experience with a wide range of profiling techniques, while covering diverse areas such as algorithms, computer architecture, high-performance computing (HPC), and machine learning, making it a highly comprehensive project. It provides the chance to work with profiling tools for the first time, understand and adapt complex codebases to extract performance metrics, analyze an RMI implementation, explore the HPC domain, and investigate microarchitectural bottlenecks in depth.

1.4 Project Planning

This project is structured around the theoretical study, software experimentation, and performance evaluation of learned hash indexes in the context of high-performance genome read mapping. The planning has been divided into different work tasks, accompanied by a corresponding timeline and a risk mitigation plan, to ensure the successful completion of the project.

1.4.1 Tasks Definition and Milestones

The project is divided into seven tasks that follow the natural steps of the work: starting with background research, then studying the algorithms, setting up the software, running experiments, applying optimizations, and finally writing the thesis and defense. The timeline for these tasks is shown in Figure 1.

- **T1. Literature Review and Background Study.** Review main algorithmic concepts regarding traditional hash tables, learned indexes, seeding algorithms in genome mapping, and HPC profiling techniques.
- **T2. Algorithmic Analysis of Hash Index Structures.** Formal comparison of traditional and learned hash table designs, focusing on algorithmic complexity, memory layout, and their practical implementation.
- **T3. Software Setup and Experimental Framework.** Set up the benchmarking environment using genome mapping applications (Minimap2 and Mm2-fast), including configuring dependencies, preparing datasets, and adapting the codebase to support profiling and performance testing.
- **T4. Profiling and Bottleneck Analysis.** Utilize performance analysis tools (e.g., perf, VTune, Valgrind) to identify microarchitectural limits and execution hot spots.
- **T5. Optimization and Evaluation.** Apply optimization strategies such as batching, prefetching, and SIMD to improve performance. Measure impact on performance and scalability.
- **T6. Thesis Writing and Final Report.** Draft, revise, and finalize the thesis document.
- **T7. Thesis Defense Preparation.** Prepare material and rehearsals for TFG defense.

Additionally, the project is structured around three milestones that help guide progress and ensure timely delivery. These milestones mark critical points in the project and serve as moments for internal review and evaluation. At each milestone, review meetings are held with the supervisors to assess the work completed so far, validate results, and adjust plans when needed.

Milestone 1: Theoretical Study and Experimental Setup Completed
Associated with: Task T1, T2, and T3
This milestone marks the end of the preparation phase. By this point, the review of theoretical concepts behind traditional and learned hash indexes will be completed, and

the experimental infrastructure will be ready. This includes the setup of benchmarking tools, integration of learned indexes, and preparation of datasets. This milestone confirms that the requirements necessary for proceeding to experimentation have been met.

Milestone 2: Benchmarking and Evaluation Completed
Associated with: Task T4 and T5
This milestone confirms that all experiments and performance evaluations have been carried out. All relevant data will have been collected, analyzed, and structured for interpretation.

Milestone 3: Final Report Delivered
Associated with: Task T6
The final milestone covers the completion of the thesis document and preparation for the oral defense. A final internal review will verify that the report accurately reflects the work and meets the expected results before it is submitted.

1.4.2 Project Timeline

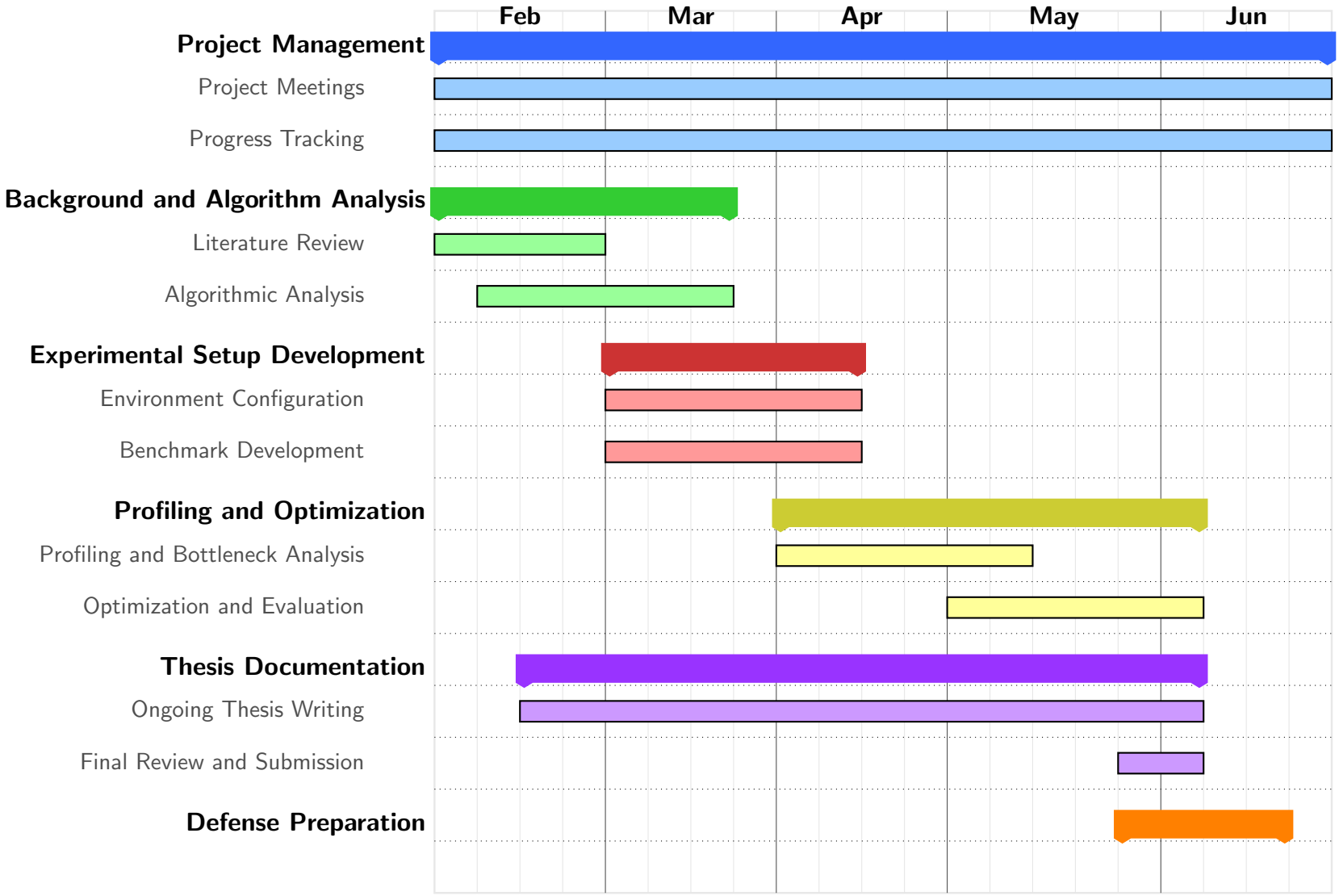

Figure 1: Gantt chart of the project (February–June 2025)

1.4.3 Risk Assessment and Mitigation

R1 Complexity of performance profiling tools *Mitigation:* Start early with simplified examples and seek guidance from supervisors or available documentation.

R2 Software integration challenges (e.g., modifying Mm2-fast) *Mitigation:* Maintain version-controlled branches and modularize testing to isolate issues quickly.

R3 Unclear performance benefits from learned indexes *Mitigation:* Clearly document all observations. If results are neutral or negative, shift focus to architectural insights and theoretical implications.

R4 Overlapping deadlines with other academic commitments *Mitigation:* Use the early months to advance TFG-specific tasks. Lock writing and testing slots into personal calendar.

1.5 Project Contributions

Overall, this work investigates the performance of learned hash indexes, compared to classical hash data structures, in the context of long-read genome alignment. The main contributions of this work are the following.

- This work provides a benchmarking and profiling software environment developed on the MareNostrum 5 (MN5) supercomputing platform to support fine-grained performance analysis. This included adapting and instrumenting the read-mapping applications Minimap2 and Mm2-fast for detailed profiling, and integrating a diverse set of tools such as perf, PAPI, callgrind, Intel SDE, and roofline analysis. This infrastructure enables low-level insight into execution behavior, providing accurate performance measurements and reproducible comparisons.
- This work presents an experimental comparison between the classical hash-based indexing used in Minimap2 and the learned RMI-based indexing in Mm2-fast. In particular, this comparative study focuses on the seeding stage executed on high-end server-class CPUs. Moreover, this study evaluates microarchitectural metrics that reflect the efficiency of each indexing strategy using realistic genomic workloads. This analysis provides experimental results on the strengths and limitations of learned hash indexes in practical scenarios.
- The profiling results of this study allow identifying performance bottlenecks and architectural limitations that impact the performance of classical hash-based indexing and learned hash indexes. These include high memory stall rates, poor cache utilization, irregular access patterns, and limited SIMD usage. Identifying these bottlenecks is key for understanding why learned indexes underperform on certain hardware and where tuning efforts should be focused. These results seek to help bridge the gap between theoretical efficiency and practical performance on real server-grade computing nodes.
- This work provides a space exploration analysis of different configurations and optimization strategies to improve the hardware efficiency of learned indexes. This

included tuning internal parameters and evaluating techniques such as software prefetching and batched query processing. The experiments presented in this work allow for the identification of configurations that reduce memory latency and improve throughput. These results seek to identify practical ways to adapt learned index implementations for better performance on modern CPUs.

- Ultimately, this work provides recommendations on how learned indexes can be adapted to better exploit the capabilities of modern high-end CPU architectures. These include considerations on memory usage, cache locality, and SIMD usage. We hope that this discussion contributes to the broader effort of designing hardware-aware data structures that are not only theoretically sound but also practically efficient on real-world hardware, particularly in the context of genomics and other data-intensive applications.

2 Background

This section provides the base knowledge required to understand this work. First, we discuss sequencing technologies and their role in genome analysis. Then, we introduce the concept of read mappers, focusing on the seeding phase. Next, we explore the role of optimized hash-table-based tools in the seeding phase. Finally, we present the emerging approach of learned index and its adoption in hash tables.

2.1 Sequencing Technologies

Every organism possesses a genome that encodes all the information required for its growth, development, and biological function. The genome is essential for the survival of the organism. In the vast majority of cellular life forms, including humans, the genome is composed of DNA (deoxyribonucleic acid), whereas some viruses have RNA (ribonucleic acid) genomes. DNA and RNA are polymeric molecules made from monomers, called nucleotides, that form long chains [1]. Each DNA nucleotide includes one of four nitrogenous bases: adenine (A), guanine (G), cytosine (C), and thymine (T). In RNA, thymine is replaced by uracil (U), while the other bases remain the same. To ensure proper organization and transmission of genetic material during cell division, DNA is packed into structures known as chromosomes.

Interestingly, the size of the genome does not have a consistent relationship with the biological complexity of an organism. For example, while the human genome contains about 3 billion nucleotides organized into 23 pairs of chromosomes, the genome of the Japanese flower Paris japonica contains around 150 billion nucleotides.

Since many of our characteristics as humans are encoded in our DNA, genomic research plays a crucial role in identifying mutations and revealing genetic variations critical for precision medicine. These insights shed light on faster and more effective treatments, enabling the prediction, diagnosis, and personalized treatment of diseases more precisely than ever.

In genomic research, the first step is to obtain genomic data from existing databases or by sequencing DNA samples to decode the nucleic acid composition of DNA molecules. This enables researchers to gain a broader understanding of nucleic acid structure and function.

The machines that perform the sequencing process can be classified into three generations based on technological advancements. In 1977, Sanger sequencing was introduced as the first method for sequencing DNA [2]. This method, based on the random addition of labeled chain-terminating dideoxynucleotides, gained wide acceptance and became the gold standard for sequencing small and large genomes. It enabled the first complete sequencing of the human genome [3].

The completion of the first human genome established a reference genome. It revealed the limitations of first-generation technologies, particularly in terms of throughput and cost, in answering complex biological questions. This was just the start of the modern DNA sequencing era, in which next-generation sequencing technologies (NGS), also known as

second-generation technologies, replaced first-generation technologies.

The key change in novel NGS methods was the parallel sequencing of single DNA fragments, dramatically increasing throughput and significantly reducing costs by producing thousands of bases (b) per second. These short sequences of DNA, known as reads, are the fundamental output of sequencing platforms. Currently, the Illumina platform [4] dominates the market for second-generation sequencing machines. Although NGS produces shorter reads and has higher error rates compared to Sanger sequencing, its high-throughput methods are expanding our knowledge, particularly in transcriptome and proteome research, enabling faster and more cost-effective whole-transcriptome sequencing.

However, the limitations of NGS, particularly the short read lengths that require complex post-processing pipelines and extended analysis times, led to the development of third-generation sequencing (TGS) technologies shortly after the introduction of NGS. TGS is characterized by producing long reads, with an average length of more than 10 Kb. This innovation has drastically improved the quality of genome assemblies and the analysis of genome structures to better characterize large insertions, deletions, translocations, and other structural variations. Pacific Biosciences (PacBio) [5] and Oxford Nanopore Technologies (ONT) [6] are the most notable platforms in this third generation, having revolutionized sequencing through their long-read capabilities.

PacBio platform employs Single-Molecule Real-Time (SMRT) sequencing technology, a sequencing-by-synthesis approach introduced by Pacific Biosciences. SMRT sequencing uses four-colour fluorescently labeled dNTPs and zero-mode waveguides (ZMWs), structures that allow real-time observation, to sequence single DNA molecules. In contrast, ONT uses protein nanopores embedded in an electrically resistant membrane. A potential is applied across the membrane, and as DNA molecules pass through the nanopores, they cause characteristic disruptions in the current. These disruptions can be measured and translated into nucleotide sequences.

Both PacBio and ONT have been successfully used to sequence and analyze full-length DNA sequences and even direct RNA sequencing (a notable advantage over NGS, which requires reverse transcription). In addition, both technologies are capable of detecting base modifications, enabling the identification of epigenetic events. They also offer relatively low-cost DNA and RNA sequencing of very long fragments.

Nevertheless, each platform has its particular advantages and disadvantages. PacBio uses cyclic consensus to sequence a molecule multiple times, achieving higher post-correction accuracy rates than ONT [7]. However, ONT provides higher throughput and enables ultra-long read sequencing, capabilities that are not achievable for PacBio, which produces a lower number of reads. ONT is particularly well-suited for field applications and quick diagnostics because it is also more portable and reasonably priced. In contrast, PacBio instruments are more expensive and less portable but offer better accuracy in applications that require high-confidence variant detection. To obtain more complete and reliable genome assemblies, many recent studies have adopted hybrid sequencing strategies that combine the high accuracy of PacBio with the ultra-long reads and scalability of ONT.

2.2 Genome Analysis Pipelines and Read Mapping Tools

Genome analysis involves studying an organism's entire DNA sequence to examine its structure, function, and genetic variations to understand its role in health and disease. Although genome analysis comprises many approaches, most of them involve three main phases: DNA sequencing, basecalling, and either genome resequencing or de novo genome assembly (Figure 2).

Genome analysis pipelines begin with the DNA sequencing phase, where the DNA is transformed into fluorescence images, fluorescence signals or ionic current signals. Then, the basecalling phase translates the output of the previous phase into sequences composed by the A, C, G and T bases.

Once the nucleotide sequences are decoded, the sequenced reads can be processed to extract meaningful biological insights. Although many types of genome analysis exist, most of them begin with either genome resequencing or genome assembly. Genome assembly reconstructs a genome de novo by piecing together sequence reads in order to create a new reference genome. However, this work will focus primarily on genome resequencing.

Once the nucleotide sequences are decoded, the sequenced reads can be processed to extract meaningful biological insights. Genome resequencing reconstructs a sample's genome using an existing reference genome as a template. In this process, each sequenced DNA from the sample is mapped to its most likely position of origin in the reference genome. This enables the subsequent detection of mutations and structural variations throughout the genome. This process is called read mapping and is typically implemented in three steps: seeding, chaining, and alignment, based on the seed-chain-extend technique.

Substrings (known as seeds) are extracted from each read, and the reference genome is indexed to allow a fast and rapid lookup of these seeds. The seeding step searches for possible locations where each read matches in the reference genome. In this first step, the number of potential matches is reduced, decreasing the amount of work for subsequent steps.

Then, the chaining step computes a colinear chain identifying a good cluster of exact matches that can be combined into plausible alignments.

Finally, the location found is where the input sequence is aligned against the reference genome during the extension or alignment step.

Once the reads are mapped, genomic differences between the reads and the reference genome can be detected. Variant calling algorithms can determine mutations and structural variations related to differences and are critical for understanding disease mechanisms and biological functions.

Although we have focused so far on genome assembly and resequencing, the described steps can be found in other applications, such as metagenomics tools such as CLARK [8], Centrifuge [9], Kraken2 [10], UNCALLED [11], and RawMap [12]. In metagenomics, given a set of DNA sequences to be classified (objects) and a set of reference sequences (targets), the goal is to identify the most likely origin of each object based on sequence similarity.

This allows researchers to characterize the diversity and function of microbials within complex communities.

Numerous tools, known as read mappers, implement the read mapping process. Mappers have been designed and optimized in parallel with advances in sequencing technologies. Among the most relevant tools of the first-generation sequencing era were BLAST [13] and BLAT [14], which were among the first read mappers to be developed. Years later, with the boom of second-generation technologies, SOAP (2008) [15] emerged, followed by Bowtie [16], BWA [17], BFAST [18], and SOAP2 [19] in 2009. As third-generation technologies became available, new tools were needed to handle longer reads. Bowtie2 [20], SeqAlto [21], GEM [22], and BWA-MEM [23] were published around 2012, with further advances such as Mosaik [24] and MHAP [25] in 2014 and Minimap [26] in 2016.

As this work focuses specifically on the seeding step, it is important to highlight the indexing structures employed by these mappers. The most widely used indexing methods are FM-Indexes [27] and hash tables.

Read mappers can be classified according to the sequencing technology generation they target (short-reads to first and second generation, and long-reads to third generation) and the type of index they use. Employing the FM-index for short reads, we have Bowtie [16], BWA [17], SOAP2 [19] and GEM [22], while BWA-SW [28], BWA-MEM [23], BWA-MEM2 [29], and Bowtie2 [20] were adapted for long-reads like SeqAlto [21]. On the other hand, tools such as BLAST [13], BLAT [14], SOAP [15], PASS [30], MOM [31], BFAST [18] are hash table-based mappers for short reads, and Mosaik [24], MHAP [25], Minimap [26] and Minimap2 [32] have been developed adapted to long-reads.

For many years, hashing was the only dominant technique until the BWT-FM index was introduced by Bowtie, which marked a shift towards more memory-efficient alternatives. However, hashing remains the most popular indexing technique used by read alignment tools, being implemented in approximately 60.80% of aligners across various areas of biological research [33]. The popularity of the BWT-FM index can be explained by its compact, compressed genome index, ease of implementation, and the ability to support fixed and variable seed lengths. Hash-based aligners provide shorter indexing times and fast seed query performance, especially when dealing with long reads. However, they consume significantly more memory than FM-index-based methods, although this drawback is becoming less critical with the decreasing cost and increasing availability of high-memory computing systems.

2.3 Hash Tables in Genomics: Strengths and Limitations

The seed-and-extend strategy fundamentally relies on the ability to quickly search for exact matches of seed sequences within the reference genome. To achieve this, the reference genome is stored in an index structure that allows fast in-memory lookups, enabling quick searches.

Several data structures have been proposed for indexing, including suffix arrays [34], suffix trees [35], FM-Index [27], and hash tables. Among these, FM-indexes and hash tables are the most widely used.

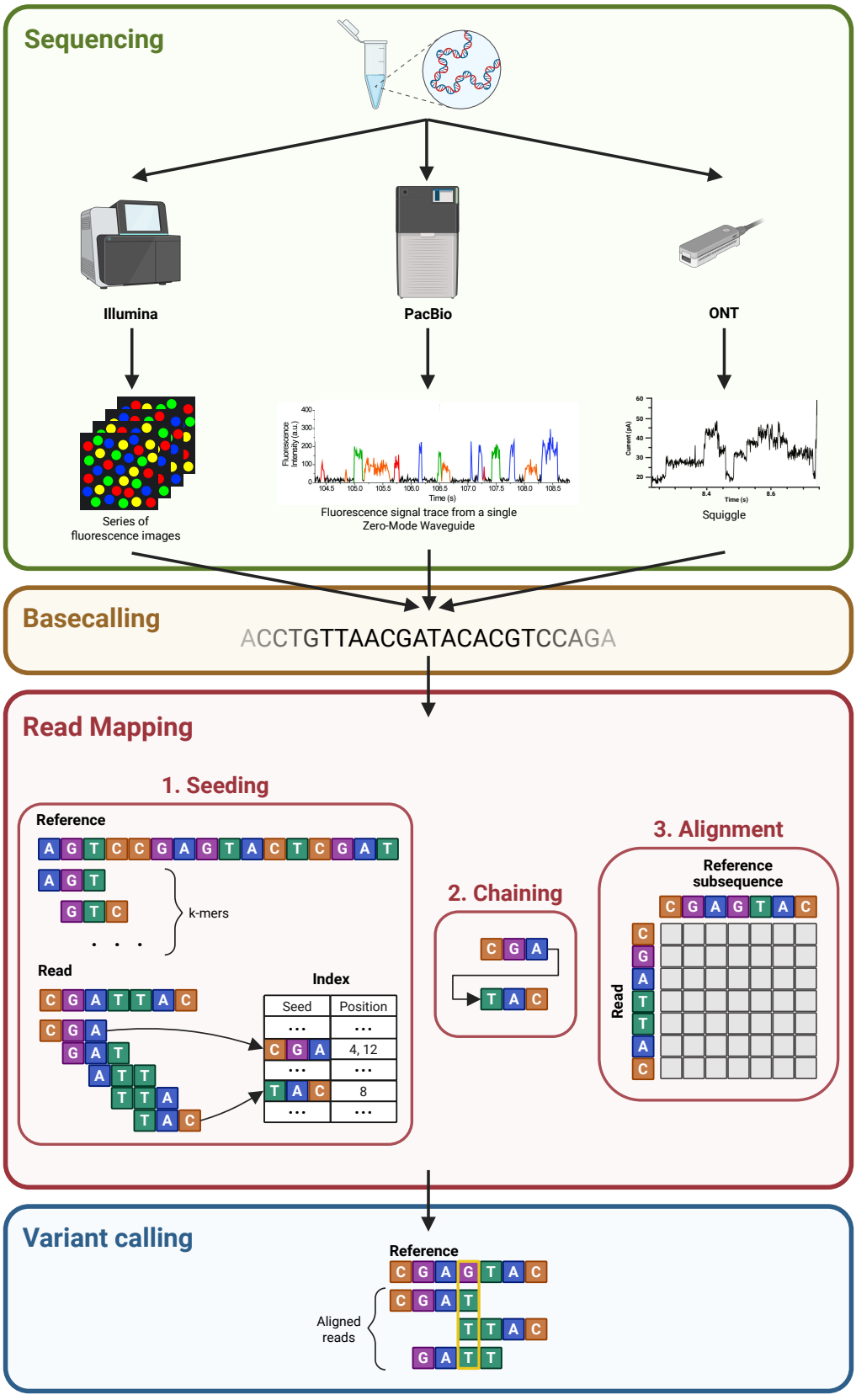

Figure 2: Genomic Analysis Workflow. Illustration of the main steps: sequencing, basecalling, read mapping, and variant calling.

Hashing was first developed in 1953 by IBM personnel, although their work was never formally published. The first descriptions of hashing methods were published independently by Dumey [36] and Peterson [37].

A hash table is a data structure that implements an associative array, an abstract data type that maps keys to values [38]. In hash tables, this mapping is performed by computing an index, also called a hash code. Hashing is the process of using a hash function to map a key in a reproducible way to a hash code that is a legal index in an array where the value will be stored.

However, when two elements with different keys are assigned the same hash code, a collision occurs. Various strategies have been developed to deal with collisions. One common approach is the chaining method, where each position in the hash table serves as a bucket to store multiple data items. Typically implemented as a linked list or an array. Another approach is open addressing, where all elements are stored directly in the hash table, and when the assigned position is occupied, the algorithm searches for the next position available through a process known as probing. There are several probing techniques: linear probing, which checks the next position sequentially until an empty one is found; quadratic probing, which uses a quadratic function to calculate the next index; and double hashing, which applies a second hash function for the increment for probing.

Hash tables are highly efficient for key-value lookups, since they offer constant time complexity $O(1)$ for both insertions and searches. However, in the worst case, due to excessive collisions, the performance is linear to the size of the hash table or the number of keys, depending on the collision resolution strategy and the size of the table. Therefore, a good hash function must minimize collisions by uniformly sharing the keys. With an appropriate hash function and table size, the performance of the hash is generally better than $O(\log n)$ and approaches $O(1)$.

Focusing on hash table indexes, optimizing storage, and looking up efficiency is critical. To minimize storage costs, many algorithms reduce the number of sequences stored in the index structure, especially the most recently developed long-read alignment algorithms. Instead of creating a hash table for the full set of sequences, recent long-read alignment algorithms select the minimum representative set of all reads from a group of adjacent sequences known as minimizers. Minimap2, a state-of-the-art read mapper for long reads, follows this minimizer-based strategy [39]. Its indexing process involves several steps:

1. **K-mers Obtaining**: The k-mers, defined as all possible contiguous subsequences of length k within a sequence, are extracted from the reads, and their hash values are calculated.
2. **Minimizers Extraction**: For each window of size w covering $w+k-1$ nucleotides, the k-mer with the smallest hash value is selected (called the minimizer) and added to form the sketch of the sequence s (that is, all minimizes). This process is shown in Figure 3.

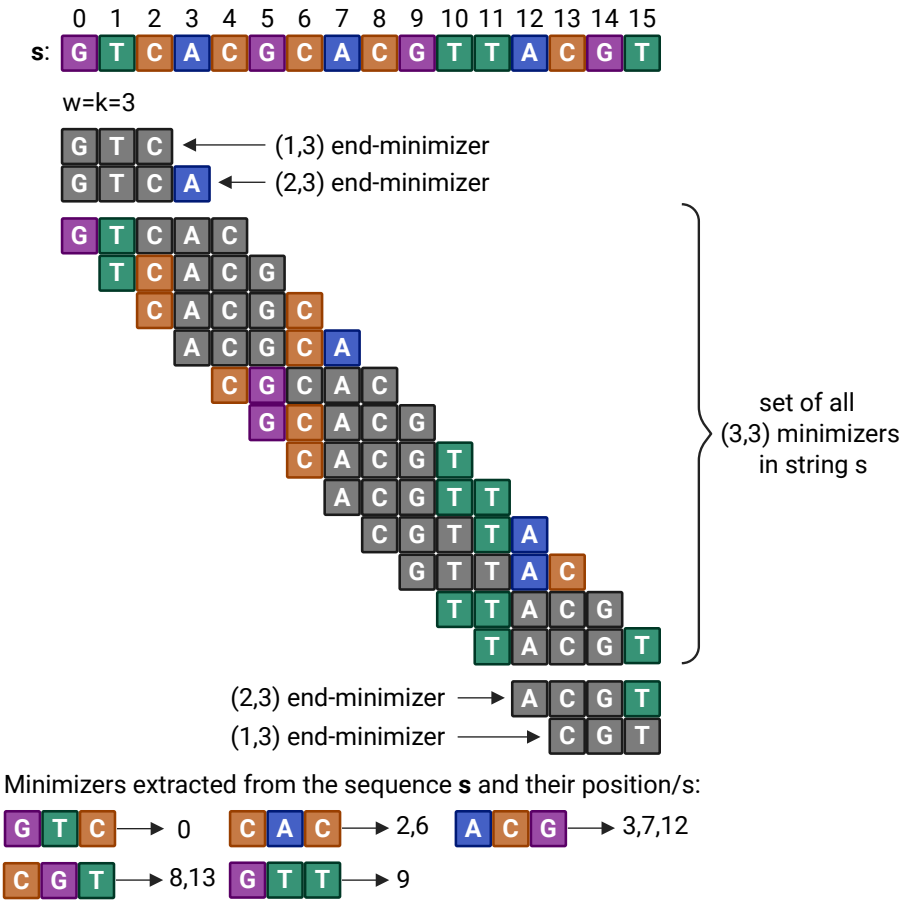

Figure 3: Sketch generation for sequence s. Minimizers are computed using k-mers ($k = 3$) within a sliding window of size $w = 3$.

3. **Index Construction**: The minimizers, along with their positions in the reference genome, are appended to an array.
4. **Sorting**: The minimizers in the array are sorted according to their hash values.
5. **Key-Value Table Building**: A table is constructed where keys are the unique minimizers and the values consist of their starting position in the position list and the number of occurrences (Figure 4).

Then, in the seeding phase, the minimizers from the query reads are collected and searched against this table to find exact matches in the reference.

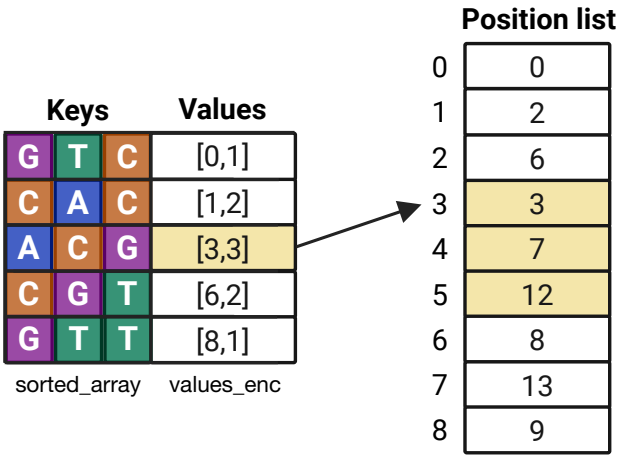

Figure 4: Hash table pointing to minimizer positions. A key-value data structure points to the sorted list of minimizer positions in the reference sequence.

2.4 Machine-Learned Indexes: Concepts and RMI Overview

Recent work proposes replacing conventional hash lookups with learned indexes, which are machine learning models trained to predict the position of a key in an index structure. Introduced by Kraska et al. (2017), learned indexes are a technique for accelerating queries on a variety of data structures by leveraging patterns present in the particular dataset being processed.

The idea is based on the key insight that an index is a model. More specifically, a hash index is a particular kind of model that, by a hash function, maps a key to a position in a data structure. Traditional implementations carry out these mapping functions using data structures, such as a B-tree for sorted data or a hash table for direct lookup. Kraska et al. (2017) observed that this function can instead be learned via regression, reframing the indexing problem as a machine learning task. This leads to an important observation: a model that predicts the position of a key in a sorted array effectively approximates the cumulative distribution function (CDF) of the key space. By learning this distribution, it becomes possible to optimize index structures.

In the context of hash tables, although traditional hash functions are extremely fast to compute, collisions have a significant impact on performance. Learned hash models, trained on the distribution of the input keys, can reduce the collision rate, thus achieving a better hashing quality.

Among the different learned indexes that can be used, the Recursive Model Index (RMI) has emerged as an effective solution for read-only in-memory dense arrays (REF). RMI is a multilayer tree structure with a model at each tree node. A typical two-layer RMI model consists of a root layer with a unique model that, in a lookup operation, predicts which among a set of leaf models should handle a given key. The selected leaf layer model then predicts the position of the key. In case the prediction is not exact, a last-mile search is

performed within the range determined by the error of the prediction ([pos-err,pos+err]). Since the key is expected to be close to the predicted position, this final search step is typically very short.

Mm2-fast [40], an optimized version of Minimap2, significantly enhances seeding performance by substituting the hash lookup with a two-layer RMI that predicts the position of the minimizer in the key-value table, as shown in Figure 5. This thesis will analyze and characterize the performance of learned hash-tables, particularly focusing on their application in genomic read mapping.

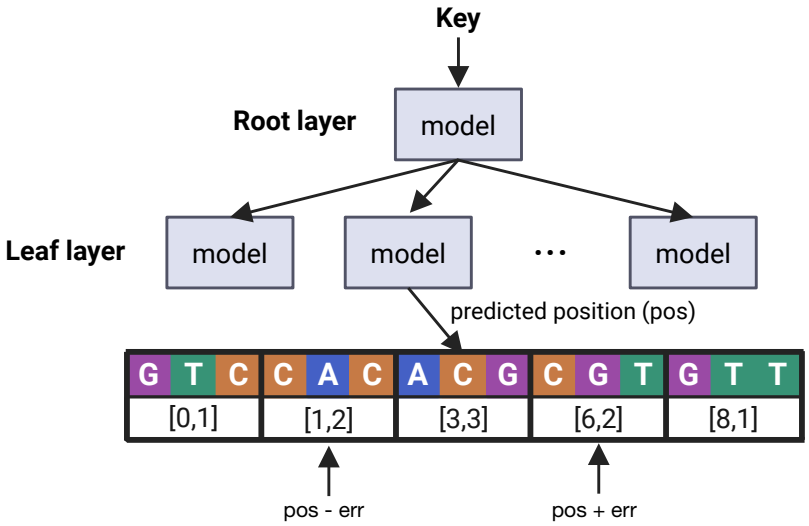

Figure 5: Data structure with RMI for minimizer lookup. The new data structure uses a Recursive Model Index (RMI) to predict the position of a minimizer in the key-value table.

2.5 Motivation for Hardware-Aware Indexing

Although traditional hash functions are extremely fast to compute, they often suffer from irregular memory access patterns and limited spatial locality, making them a poor fit for modern hardware. The random memory access pattern, along with a poor spatial and temporal locality, causes most of the accesses performed to the hash table to miss in all cache hierarchy levels. Consequently, the IPC is reduced, since the processor frequently stalls while waiting for data to be fetched from memory. In addition, branch mispredictions are more likely due to the unpredictable access behavior, further degrading the performance.

In contrast, learned data structures like the RMI predict key positions directly, avoiding collisions. Even in cases where the prediction is not exact, the subsequent last-mile search operates over a small, contiguous range of keys, which benefits from both spatial and temporal locality thanks to the sorted data layout. This access pattern enhances cache utilization and is more hardware-friendly.

In summary, learned indexes not only offer theoretical efficiency but are also more in line

with the memory hierarchies and execution models of modern hardware. This alignment results in noticeable performance improvements in real-world scenarios, such as genomic analysis applications, where a significant portion of the execution time depends on index data structures.

3 Experimental Methodology

This section describes the methodology used to evaluate and analyze the performance of learned hash indexes in the context of genomic read mapping. The goal is to assess how different indexing strategies behave under realistic computational conditions, quantify their efficiency, and identify performance bottlenecks and architectural limitations.

In the following, we define the experimental setup, including the hardware platform and runtime environment. Then, we describe the datasets used, selected to reflect realistic genome sequencing scenarios. The main body of the section focuses on the profiling strategy. It explains how performance was measured, which tools were used, and how different metrics were collected and interpreted. The analysis follows a top-down approach, combining high-level profiling with low-level hardware counter analysis. It includes function-level inspection, memory and cache behavior evaluation, microarchitectural bottleneck detection, and visualization through performance stacks. Altogether, this methodology aims to provide a comprehensive performance characterization of how hash indexes and learned index structures behave in practice.

3.1 Experimental Environment

Genome analysis tools such as Minimap2 and Mm2-fast are commonly executed on high-end computing infrastructures, as these applications are designed to process large-scale sequencing data efficiently. Their performance and scalability become especially relevant when applied to population-scale or real-time genomic studies, which demand substantial computational throughput and memory bandwidth. Benchmarking on a pre-exascale system, such as MareNostrum 5 (MN5), reflects realistic deployment scenarios and ensures that the results are representative of the conditions in which these tools are expected to execute.

This way, we evaluated Minimap2 and Mm2-fast on a General Purpose Partition (GPP) node of the MN5 supercomputer. MN5 is a pre-exascale EuroHPC system comprising two main partitions with distinct technical features (i.e., the GPP and ACC partitions). The system is composed of 125 racks, Infiniband NDR compute network interconnect, and a Red Hat Enterprise Linux operating system.

The GPP partition, where all the benchmarks of this work were executed, consists of 6408 nodes distributed across 90 racks, based on Intel Xeon Sapphire Rapids processors. An additional 72 nodes featuring Intel Sapphire Rapids High Bandwidth Memory (HBM). As of June 2025, this partition ranks 35th on the TOP500 list with a Linpack Rmax performance of 40.10 PFlop/s. Each GPP node is primarily equipped with two Intel Xeon Platinum 8480+ processors (each with 56 cores at 2.0 GHz, totaling 112 cores per node), 256 GB of DDR5 memory, and a 960 GB NVMe unit for local storage (Figure 6).

Furthermore, the MN5 is equipped with the Accelerated Partition (ACC), which includes 35 racks with 1,120 nodes, each with 80-core Intel Xeon Sapphire Rapids processors and NVIDIA GPUs. As of June 2025, this partition ranks 11th on the TOP500 list with a Linpack Rmax performance of 175.30 PFlop/s. This partition was not used for this work, as the execution on the GPUs falls out of the scope of this study.

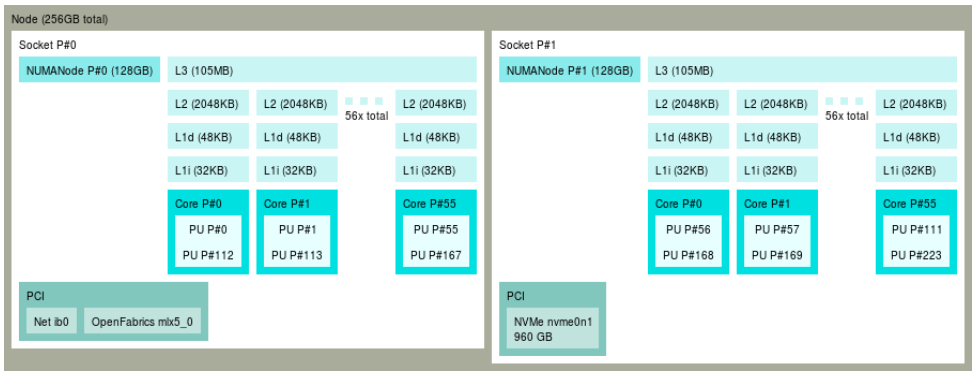

Figure 6: Topology of a GPP node. A GPP node has two sockets (56 cores each), 256GB of DDR5 RAM, and 960GB local NVMe storage. Each core has 32KB L1 instruction cache, 48KB L1 data cache, and 2MB L2 cache. Each socket includes a shared 105MB L3 cache. PCI connects the network (InfiniBand) and storage devices.

3.2 Experimental Datasets

Applications like Minimap2 and Mm2-fast require two main inputs to perform read mapping: a reference genome and a set of query sequences. The reference genome is indexed to build the internal data structures, such as hash tables and learned indexes, used for efficient seed lookups during alignment. Then, the query sequences, typically produced by sequencing machines, are aligned against this indexed genome reference to determine their genomic position of origin. For this reason, our experimental evaluation requires a representative reference genome (like the human genome) and a realistic set of input queries (a.k.a. reads), like those produced by modern sequencing machines, to perform a realistic benchmark.

For our experimental evaluation, we used GRCh38 reference genome (`GCF_000001405.40`), obtained from the National Center for Biotechnology Information (NCBI) . GRCh38 is the current human reference genome assembly, and it is widely used in many genomic analyses.

As input queries, we selected a subset of sequencing reads from the `SRR23704822_1.fastq` dataset , available through the European Nucleotide Archive (ENA). This dataset consists of Oxford Nanopore Technologies (ONT) long-read sequencing reads from Homo Sapiens. Specifically, we extracted 2 million reads, each 1000 base pairs in length, randomly sampled from the original FASTQ file. These sequences reflect the typical error profile and structure of ONT reads, including variable signal quality and higher indel rates, making them suitable for evaluating the performance and robustness of hash-based and learned indexing strategies.

Overall, these input datasets provide a realistic benchmarking scenario that captures the requirements of modern long-read genomic analysis tools. It also enables performance comparisons between different indexing strategies under realistic biological conditions.

3.3 Profiling Methodology

This section explains the profiling methodology used to evaluate and characterize the performance of the applications studied in this work (i.e., Minimap2 and Mm2-fast). It describes how the experiments were executed, what metrics were collected, and which analysis techniques were applied to understand high-level behavior and low-level bottlenecks. The methodology follows a top-down approach, starting with coarse-grain profiling and gradually incorporating results from fine-grained microarchitectural analyses.

The following subsections describe the specific profiling tools and techniques used in each stage. These include function-level profilers (such as gprof and callgrind), static binary inspection (objdump), hardware performance counter analysis (perf, PAPI, IntelSDE), memory and cache efficiency evaluation (using cache metrics and roofline modeling), and pipeline efficiency visualization through performance stacks. Together, these approaches provide a detailed characterization of the application's performance.

3.3.1 Evaluation Methodology and Performance Metrics

To ensure consistency and reproducibility of the results, each experiment was executed five times under identical conditions. For each measurement, we verified that the variance between runs remained low and discarded any outliers when deviations were significant. This averaging process helps reduce the impact of system noise, ensuring the reliability of our measurements.

Moreover, all benchmarking and profiling experiments were executed using cluster nodes in exclusive mode to ensure consistent and undisputed access to hardware resources. This setup removes interference from other users or jobs, minimizing variability in performance measurements. Hence, it guarantees that the results reflect only the behavior of the evaluated applications.

To evaluate performance, we adopted a top-down profiling approach. We began by measuring the execution time of the complete alignment application (i.e., Minimap2 baseline and Mm2-fast aligning DNA sequences against the GRCh38 reference genome). This initial timing provided a baseline for end-to-end performance comparisons between the original Minimap2 implementation and the optimized Mm2-fast version.

In this work, we employed callgrind, a Valgrind-based tool that records the number of instructions executed and visualizes function call relationships, to understand the execution flow of the code, making it easier to identify candidate regions for more in-depth profiling. Furthermore, to identify the most time-consuming functions, we execute the binaries using gprof to obtain how often each function is called and the total time spent, offering a coarse-grain view of program behavior.

In many cases, rather than profile the whole application, we isolate regions of interest (ROI) from the code that dominate execution time. In the context of this work, these regions primarily correspond to the seeding phase and its associated hash index lookup operations. By restricting measurements to these hot spots, we avoid unnecessary profiling overheads and focus our analysis on the most relevant and time-consuming kernels. Once the most computationally intensive routines were identified, we limited the scope

of the performance analysis to these ROIs. This focused approach allowed more detailed profiling to investigate microarchitectural behavior, memory usage patterns, and low-level bottlenecks.

Throughout the experimental evaluation, we collect standard performance metrics to capture a detailed view of the application's performance. These include total execution time, memory usage (particularly peak resident set size), and microarchitectural-level metrics such as retired instructions, instructions per cycle (IPC), front-end stalls, back-end stalls, and branch prediction accuracy. Using these metrics, we characterize top-level performance and its dependency on the underlying hardware-level architecture.

Subsequent sections describe how this deeper analysis was carried out using tools such as gprof, callgrind, perf, and PAPI.

3.3.2 Function-Level Profiling and Static Analysis

This section describes the tools used to perform function-level profiling and static inspection of the application code. These tools help identify performance-critical functions, understand how the application is structured at the call level, and inspect the compiled binary. The tools used in this work include gprof, callgrind, and objdump. Each offers different information from the studied applications and their performance (e.g., execution profiling, call graph exploration, and binary-level disassembly).

First, **gprof** was used as a starting point to obtain high-level performance metrics at the function level. After compiling the program with the `-pg` option, we executed the binary to collect runtime statistics, which gprof stores in a `gmon.out` file. This file records the number of calls to each function and the time spent within them. The output helps identify where execution time is concentrated, serving as an initial guide for more in-depth profiling. Although limited in granularity and sampling precision, gprof is lightweight and easy to use, making it suitable for a first function-level performance analysis.

Then, **callgrind**, part of the Valgrind suite, complements gprof by providing a detailed account of instruction counts, call relationships, and the dynamic structure of program execution. Unlike gprof, which uses instrumentation-based sampling, callgrind simulates CPU execution to record exact counts of operations and calls. This allows for precise visualization of call graphs and instruction flow. We used it to explore how the application's most time-consuming functions interact with each other. The output was later visualized using KCachegrind for a graphical and easier interpretation.

Lastly, **objdump** was used for static binary inspection, particularly when analyzing performance-critical functions. By disassembling the compiled code, we could examine the assembly instructions emitted by the compiler. This helped verify whether optimization flags were correctly applied and inspect how certain loops, branches, or memory operations were translated (including the usage of SIMD instructions). Although it does not provide runtime data, objdump is useful for confirming low-level implementation details that affect performance, such as unrolled loops, inlined functions, or branch instruction patterns.

Together, these tools allowed us to understand how execution time is distributed across

the application, how functions interact, and how they are encoded at the binary level. This analysis is key for guiding the more detailed microarchitectural profiling discussed in the next section.

3.3.3 Microarchitectural Profiling with Hardware Counters

This section focuses on the tools and techniques for collecting low-level performance data from hardware performance monitoring units (PMUs). These tools provide fine-grained metrics on how programs execute on the underlying microarchitecture, allowing the identification of memory bottlenecks, pipeline stalls, cache inefficiencies, and general throughput issues. The tools employed in this work include Perf, PAPI, and Intel SDE.

First, **perf** is the Linux-native profiling tool that interfaces directly with the CPU's hardware counters. It provides access to many metrics, including instructions per cycle (IPC), cache misses, and branch mispredictions. In this work, perf stat was used to gather summary statistics, while perf record and perf report were used to attribute events to specific functions and instructions. This allowed us to link high-level execution performance with low-level architectural metrics, such as cache efficiency and memory-bound execution kernels.

Also, **PAPI** (Performance Application Programming Interface) was used to complement perf with portable and programmatic access to hardware counters. Although not used as the primary profiling tool, PAPI helped double-check specific performance metrics, especially in regions of interest, such as the seeding phase. In some cases, it was also used to validate results when specific perf counters failed to report accurate data.

Moreover, **IntelSDE** (Software Development Emulator) was used to simulate program execution at the instruction level. Unlike Perf or PAPI, which rely on actual hardware support, SDE emulates the microarchitectural behavior of Intel CPUs, allowing fine-grained analysis of the instruction mix. Although significantly slower than native execution, it offers access to low-level execution details that are otherwise difficult to measure. In this study, we used Intel SDE to obtain a precise breakdown of the number and types of instructions executed across different application versions, which helped us evaluate the impact of various optimizations.

By combining data from these tools, we were able to characterize performance at the microarchitectural level. This included detecting memory bottlenecks, measuring how effectively the CPU pipeline was utilized, and assessing the performance impact of specific optimization techniques, such as software prefetching or batched processing.

3.3.4 Memory and Cache Behavior Analysis

Understanding memory behavior is key when working with data-intensive applications like genomic alignment tools, where large index structures and irregular access patterns can quickly become performance bottlenecks. To assess memory efficiency and identify possible sources of latency or underutilization, we combined runtime memory usage tracking, hardware event monitoring, and memory-boundness modeling.

We used the **time -v** command to collect runtime memory statistics, particularly the peak resident set size (RSS). This measurement reflects the highest amount of physical memory used during program execution and allows us to compare the memory footprint across different application configurations and optimizations. While not detailed at the microarchitectural level, this metric helped us assess the increased memory cost of learned indexing techniques relative to the baseline.

Moreover, we analyzed cache behavior using performance counters, such as **Last Level Cache (LLC) misses** (i.e., number of LLC requests that require DRAM access) and **MPKI** (misses per thousand instructions), to analyze the memory system efficiency. These counters were collected through Perf and PAPI, focusing on L1, L2, and L3 cache levels. The goal was to quantify how each application version used the cache hierarchy and to determine whether optimizations like batching or prefetching led to measurable improvements. For instance, high MPKI values often indicate poor spatial locality or insufficient reuse of data, which can degrade performance on modern CPUs. As a general rule of thumb, the MPKI can be divided into different ranges that indicate good cache usage (0-5), average cache usage (5-10), and poor cache usage (¿10).

We also computed a **roofline model**, using Intel VTune Amplifier, to understand the balance between computational throughput and memory bandwidth usage. The roofline model situates performance in relation to two limits: the peak computational performance of the processor and the maximum available memory bandwidth. The intersection of these defines the "roofline", and any program lies below this bound. This helped us determine whether specific regions of the code were memory-bound or compute-bound, guiding optimization efforts accordingly. For example, certain learned-index operations showed potential for improvement by reducing memory stalls or enhancing prefetch efficiency.

To construct a roofline model, one must first measure the performance, the arithmetic intensity, and the sustained memory bandwidth of the application or code region of interest. Performance is calculated as the total number of useful operations executed (in our case, integer operations) divided by the execution time, yielding a value in GINTOPS/s. Arithmetic intensity is defined as the number of executed operations divided by the volume of memory transferred, expressed in INTOPs per byte. The sustained memory bandwidth can be obtained using profiling tools such as Intel VTune, which can report actual bandwidth usage during execution.

In the roofline, the arithmetic intensity is plotted along the x-axis and performance along the y-axis. A diagonal line representing the DRAM bandwidth is included to indicate the maximum achievable performance for memory-bound workloads. A horizontal line represents the hardware's peak integer compute performance. The position of the application's point relative to these limits reveals the dominant bottleneck and guides decisions about where to focus optimization (e.g., whether on reducing memory stalls, improving vectorization, or increasing concurrency).

In summary, this memory and cache analysis provided a detailed view of how data movement and access patterns affect execution. By combining high-level memory tracking with low-level counter analysis and bandwidth modeling, we were able to identify criti-

cal inefficiencies and better understand the architectural impact of the learned indexing techniques.

3.3.5 Performance Stack Analysis

Lastly, we computed and plotted performance stacks to elaborate on the microarchitecture performance when executing our applications. We constructed performance stacks to visualize the contribution of different hardware bottlenecks to overall execution behavior. Performance stacks provide a high-level breakdown of stalled cycles, allowing us to reason about the relative impact of front-end, back-end, memory, and speculation-related inefficiencies. This type of analysis is key to identifying where time is lost in the CPU pipeline and prioritizing optimization strategies.

To generate the performance stacks, we used perf stat to collect low-level hardware counter metrics for each execution. These included event groups related to instruction dispatch, pipeline stalls, memory accesses, and cache performance. We then post-processed these counters to estimate the fraction of cycles spent in each stall category. The collected data was organized into structured reports and visualized using the Matplotlib library. These visual representations helped us compare different application versions and track how specific optimizations shifted the balance between compute-bound and memory-bound behavior.

The main goal of this analysis was to quantify the architectural efficiency of the applications beyond raw performance numbers. While execution time or IPC provides a global view, performance stacks expose why certain implementations are faster or slower, whether due to improved locality, reduced stalls, or better branch behavior. In our case, they were very helpful to confirm that learned hash index optimizations reduced memory-bound pressure and improved pipeline utilization in time-consuming regions of the code.

4 Learned Hash Performance Characterization

This section presents the results of an in-depth performance characterization conducted to assess how hash-based and learned index structures behave in practice. The analysis follows a top-down methodology, combining high-level profiling with low-level hardware counter examination. In the analysis, we include execution time comparison, function-level inspection, evaluation of microarchitectural metrics (e.g., instruction count, IPC, MPKI, and misprediction rate), bottleneck identification using performance stacks, and a roofline analysis to assess compute- vs memory-bound behavior. Finally, we present the scalability results for a multi-core environment.

First, we describe the indexing strategies and configurations evaluated in this study. The **Baseline** of this study is Minimap2, a state-of-the-art long-read mapper that uses a minimizer-based indexing strategy. Minimizers are stored in a hash table, and lookups are performed using a quadratic probing to handle collisions, as presented in **Algorithm 1**.

Algorithm 1: Minimizers Lookup in Classic Hash (Baseline)

Input: Array of n minimizers $M[1 \dots n]$, hash table H of size m
Output: Array of match positions $Match[1 \dots n]$

for $i \leftarrow 1$ **to** n **do**
  $key \leftarrow M[i].key$;
  $idx \leftarrow \text{hash}(key) \bmod m$;
  $step \leftarrow 1$;
  while *H[idx] is occupied and H[idx].key* $\neq key$ *and* $step \leq m$ **do**
    $idx \leftarrow (idx + step^2) \bmod m$;
    $step \leftarrow step + 1$;
  end
  $Match[i] \leftarrow idx$ if $step \leq m$ and $H[idx].key = key$; else -1;
end

To accelerate Minimap2 hash lookup operations, Mm2-fast, an optimized version of Minimap2, replaces traditional hash tables with learned indexes. Specifically, it uses a machine learning model implemented by a two-layer Recursive Model Index (RMI), which predicts the position of a minimizer (key) in a key-value table.

Additionally, we evaluate different configurations of Mm2-fast that isolate the contributions of diverse optimizations termed as **RMI Sequential**, **RMI Vectorized**, **RMI Batched**, **RMI Batch+Vect**, **RMI Batch+Pref**, and **RMI Fully Optimized**.

The first configuration, **RMI Sequential**, is a newly implemented version in this work that performs lookups sequentially using the base Mm2-fast. This version performs a sequential lookup of each queried minimizer (key). For each lookup, the RMI model defines a guarantee range where the queried key is located in the key-value table. Specifically, the root model first predicts which leaf model to use, and then that leaf model predicts the position of the key in the key-value table. After that, a last-mile search is performed using a binary search within the obtained range until the correct position is found. This process is presented in **Algorithm 2**, where `sorted_array` denotes the sorted array of

keys from the key-value table where the RMI searches, and the last-mile search is presented in **Algorithm 3**. As previously explained, since the key is expected to be close to the predicted position, the last-mile process is typically very short.

Algorithm 2: Minimizers Lookup in Learned Hash Structures (RMI Sequential)

Input: Array of n minimizers $M[1 \dots n]$, hash table H of size m
Output: Array of match positions $Match[1 \dots n]$

for $i \leftarrow 1$ **to** n **do**
  $key \leftarrow M[i].key$;
  $err \leftarrow 0$;
  $leaf \leftarrow get\_guess\_root\_step(key, err)$;
  $guess \leftarrow get\_guess\_leaf\_step(key, leaf, err)$;
  $pos \leftarrow last\_mile\_search(key, guess, err)$;
  $Match[i] \leftarrow pos$ if $key = sorted\_array[pos]$; else -1;
end

Algorithm 3: Last-Mile Binary Search

Input: Key k, predicted position $guess$, error bound err, sorted array of size n
Output: Predicted position p of the element

$first \leftarrow guess - err$;
if $first < 0$ **then**
  $first \leftarrow 0$
end
$last \leftarrow guess + err + 1$;
if $last > n$ **then**
  $last \leftarrow n$
end
$m \leftarrow last - first$;
while $m > 1$ **do**
  $half \leftarrow \lfloor m/2 \rfloor$;
  $middle \leftarrow first + half$;
  if $k \geq sorted\_array[middle]$ **then**
    $first \leftarrow middle$;
    $m \leftarrow m - half$;
  end
  else
    $m \leftarrow half$;
  end
end
return $first$

The second configuration, **RMI Vectorized**, extends the RMI Sequential version by vectorizing the last step of the last-mile search, leveraging AVX-512 SIMD instructions. This version is similar to the RMI Sequential configuration, but differs in the final step of the lookup. In summary, it iterates through all the minimizers, and for each one, its

position is predicted using the RMI. Then, the last-mile search is performed, and when the searched range contains 8 or fewer elements, a SIMD instruction is used to perform a parallel comparison to accelerate the last-mile process.

Next, in the **RMI Batched** configuration, lookups are processed in batches (default batch size: 32). The lookup process is divided into three steps: prediction by the root model (first layer of the RMI), prediction by the corresponding leaf model (second layer of the RMI), and range adjustment using the last-mile search. Thus, this batched version, instead of performing all lookup steps sequentially per key, executes each lookup step across the entire batch before proceeding to the next step. In this way, the step of predicting the leaf model cannot start until all minimizers in the batch have finished the previous step (root model prediction). This software optimization aims to exploit temporal locality in memory accesses, and while it can yield significant performance improvements, it requires careful tuning of the batch size to achieve these performance improvements.

Next, **RMI Batch+Vect** version combines batching (RMI Batched) and last-mile vectorization (RMI Vectorized), so it applies SIMD-based refinement to the last-mile final step after batch-based predictions.

Building upon batching (RMI Batched), **RMI Batch+Pref** configuration adds software prefetching. After the evaluation of each lookup step, the necessary data for the next step is prefetched into the cache. This information is the leaf model predicted by the root, the predicted position by the leaf, and the key-value entries accessed by the last-mile search.

The final configuration, referred to as **RMI Fully Optimized**, integrates all three optimizations: it performs the lookup in batches of 32 minimizers (batching), applies software prefetching to load the data required at each step in advance (software prefetching), and vectorizes the final step of the last-mile search to accelerate the final comparisons (vectorization). This represents the most optimized lookup strategy evaluated in this work.

As the primary goal is to assess the performance of the indexing data structures under realistic conditions, we limited the scope of the performance analysis to the region of interest (ROI) specifically corresponding to the lookup phase of the application. By focusing exclusively on this part, we can isolate and accurately evaluate the impact of each optimization and better understand the behavior of the different indexing strategies.

With these configurations defined, we proceed to present and analyze the results obtained.

4.1 Minimap2 and Mm2-fast Hash Performance Evaluation

As a first experiment, we evaluate which configuration is faster by comparing their execution times during the lookup phase. The results shown in Figure 7 indicate that the configuration achieving the best speedup is the RMI Fully Optimized version, with a 2.90× improvement compared to the baseline. Notably, we observe that simply replacing the hash-based lookup with an RMI-based prediction (RMI Sequential) already achieves a 1.32× speedup, demonstrating that learned indexes offer benefits even without further optimization.

However, vectorizing the last-mile search does not result in any noticeable improvement,

which aligns with expectations, as the RMI Vectorized optimization only applies to the final step of the last-mile search, which is typically expected to be short.

What does provide a significant improvement is the RMI Batched version. Moreover, this batching strategy also enhances the effect of vectorization, as the combination achieves better performance than either optimization alone, suggesting a positive interaction between the two.

In addition, the combination of batching and prefetching in RMI Batch+Pref achieves an even greater improvement. This result shows that prefetching is especially effective when combined with batching, as it helps mitigate memory latency by requesting future data accesses in advance.

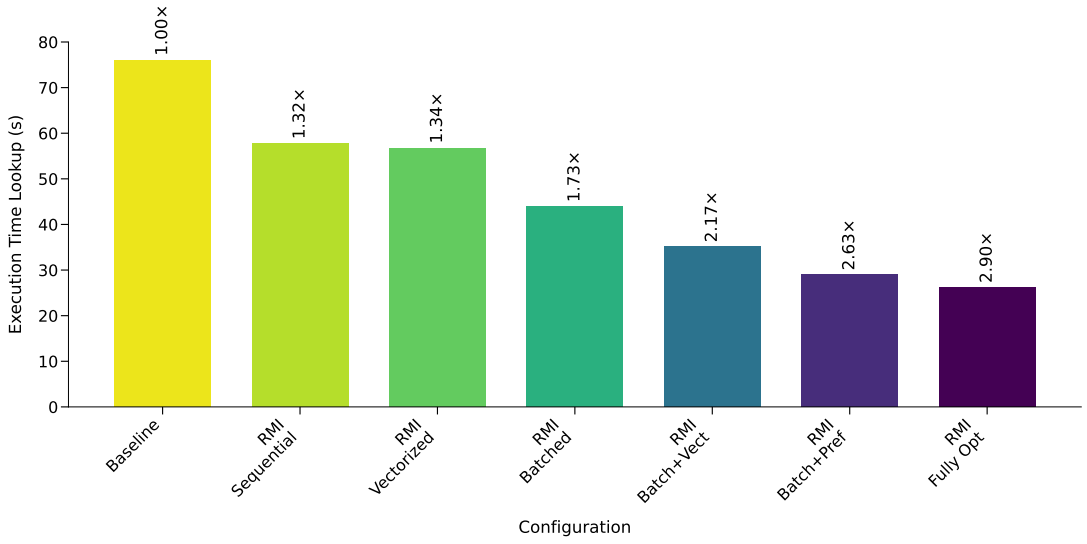

Figure 7: Execution time comparison. Comparison of the lookup execution time across all configurations, including the baseline and RMI-based configurations. Optimizations include vectorization (Vect), batching (Batch), and software prefetching (Pref).

To better understand the reasons behind the speedup observed in the RMI Sequential configuration compared to the baseline, we analyzed the behavior of the lookup phase in terms of collisions. First, in Figure 8, we show the collision distribution in hash lookups. We observe an interesting pattern: when the lookup corresponds to a minimizer not present in the index, the number of collisions is generally higher than when the minimizer is found. This is due to the nature of the hash table's collision resolution. When the minimizer is not found, the hash table applies quadratic probing, which leads to more random accesses across the index until the lookup concludes unsuccessfully. As the data shows, 96% of the found queries are solved with 5 or fewer accesses, whereas only 75% of the not-found queries are solved within the same number of accesses. To reach 96% coverage for not-found queries, up to 14 collisions are needed, resulting in a much larger number of accesses to the index.

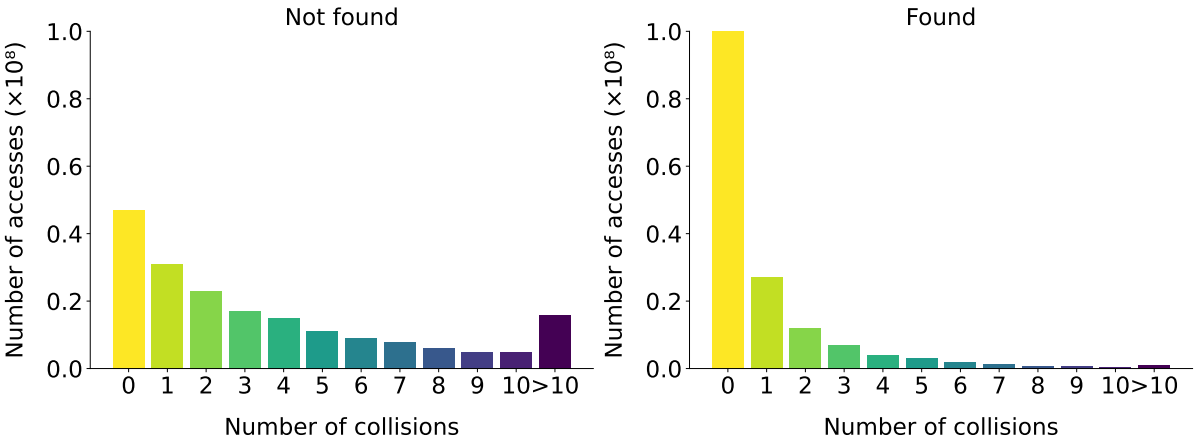

Figure 8: Collision distribution in hash lookups. Number of accesses (collisions) required to solve lookups in the baseline hash table. Not-found queries experience more collisions than found ones, increasing random memory accesses.

In contrast, when analyzing the behavior of learned indexes, we observe that the performance remains consistent between found and not-found minimizers. The RMI predicts a position with an associated error, and the last-mile search is performed within a small bounded range. This range remains nearly the same whether the key is present or not, and if the key is not found within it, the search terminates without further probing. As a result, in Figure 9 is shown that the number of memory accesses remains stable and limited, regardless of whether the key is found. This behavior contrasts with that of the hash table, where not-found queries incur a greater memory access cost.

The difference in access patterns explains the performance improvement of RMI-based approaches. When reducing the number of memory accesses, especially in cases of unsuccessful lookups, memory latency is reduced, and so is the overall execution time.

4.2 Microarchitecture Profiling and Bottleneck Analysis

4.2.1 Microarchitecture profiling

To continue, we analyze microarchitectural performance counters to better understand the hardware-level behavior of each configuration and identify where the different optimizations have their impact.

Overall, in terms of instruction count, vectorization reduces the number of executed instructions because SIMD instructions can process multiple data elements with a single instruction (Figure 10). In contrast, the batch-optimized versions execute a greater number of instructions compared to both the Baseline and RMI Sequential configurations due to the additional overhead required to split the lookup into distinct phases and coordinate execution across them. Similarly, software prefetching adds more instructions to the pipeline. The Fully Optimized version maintains a lower instruction count than the batching and batching+prefetching configurations, thanks to the instruction savings

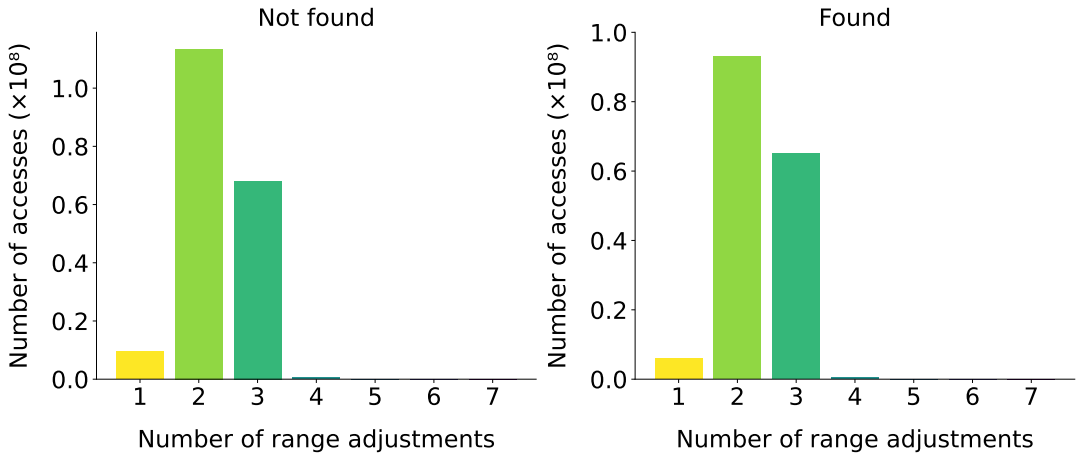

Figure 9: Range adjustment distribution in learned index lookups. Number of refinements during the last-mile search in learned index lookups. The number of adjustments is similar for both found and not-found queries, indicating stable performance.

introduced by vectorization.

The results of the IPC (Instructions Per Cycle) analysis, shown in Figure 10, are consistent with these findings. The configuration that achieves the highest IPC is RMI Batch + Prefetch. However, when vectorization is added on top of this version, the IPC slightly decreases, but not due to inefficiency, but because the same work is completed with fewer instructions. In potentially memory-bound scenarios, the total number of cycles may remain similar, while the instruction count decreases, thus lowering the IPC. In contrast, even though the Baseline version executes more instructions than RMI Sequential, it achieves a lower IPC due to its slower performance. Additionally, the reported IPC for both the baseline and the learned index configurations is very low. The huge dependencies between instructions, along with the random access pattern, limit the overall performance of the application.

The MPKI (Misses Per Kilo Instructions) metric provides further information, as shown in Figure 10. All RMI-based configurations show a significant reduction in cache misses compared to the baseline, which helps explain the performance improvements observed earlier. Notably, batching also leads to a decrease in MPKI, indicating that it successfully exploits temporal locality by grouping and reusing memory accesses.

However, batching also increases the number of misspeculated instructions. This is due to the control flow involved in processing different steps of the lookup pipeline. The branch predictor may mispredict which phase of the lookup is currently active or the last-mile next step, leading to the execution of misspeculative instructions. Moreover, prefetching further increases the number of mispredicted instructions, as it adds control complexity to each step.

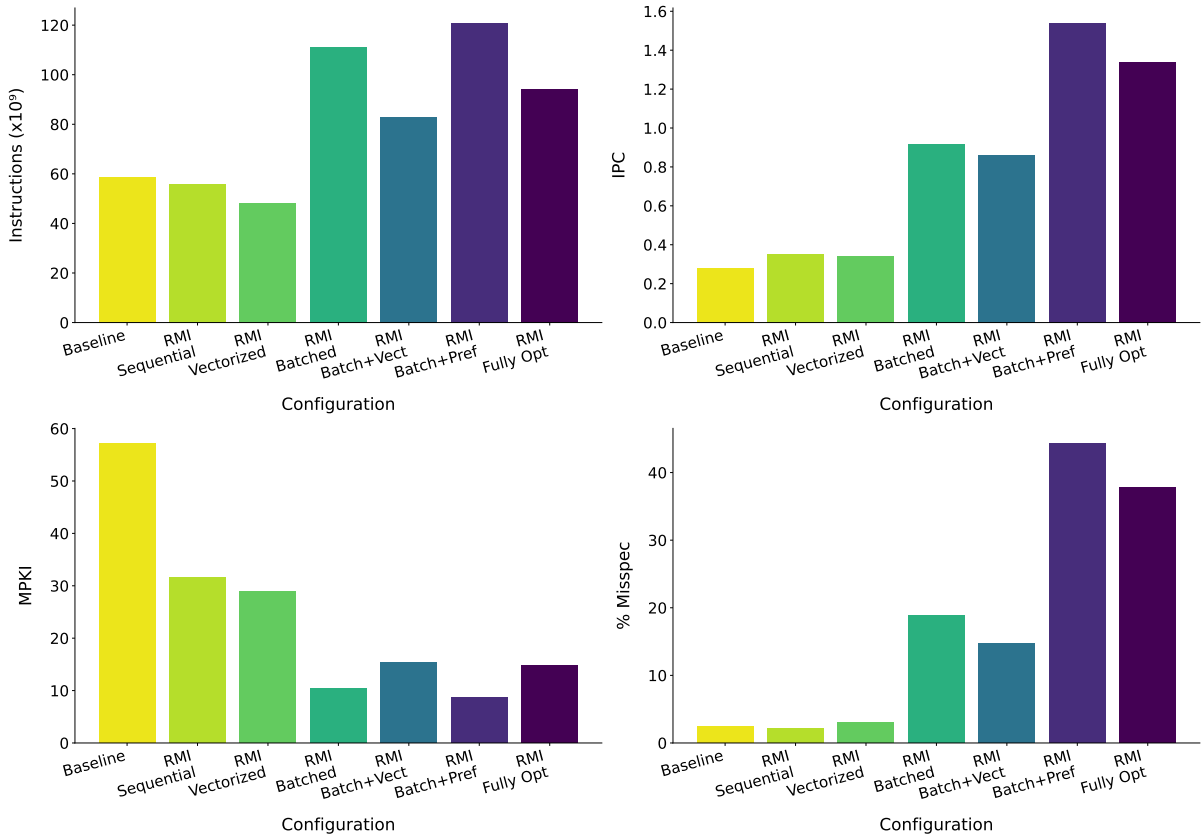

Figure 10: Microarchitecture metrics comparison. Instruction count, IPC, MPKI, and speculation behavior across Baseline and RMI-based optimized versions. Optimizations include batching (Batch), vectorization (Vect), and prefetching (Pref).

4.2.2 Bottleneck analysis

The key microarchitectural results are complemented by the performance stack analysis shown in Figure 11. This breakdown provides a more visual and detailed view of where execution time is spent across different pipeline components. The plot represents the number of cycles spent in each of the four main pipeline components:

- Retiring: representing cycles spent doing useful work.
- Misspeculation: showing cycles wasted on executing mispredicted instructions.
- Front-end: representing cycles stalled fetching and decoding instructions.
- Back-end: cycles when instructions are ready to execute but are stalled in the back-end of the pipeline. This is further divided into:
  - Core-bound: cycles limited by the computational resources of the core.
  - Memory-bound: cycles stalled waiting for data from memory.

The results highlight the inefficiency of the baseline, which spends the vast majority of its cycles stalled waiting for memory, leaving only a small fraction available for useful

work. One of the most revealing results from this experiment is that, although the RMI Sequential version outperforms the baseline in terms of execution time, it still suffers from the same memory bottleneck. The learned index prediction does not solve the memory latency issue: both RMI Sequential and RMI Vectorized remain heavily stalled waiting for memory.

The breakthrough comes with batching, which substantially reduces memory stalls. As shown in the performance stacks, this optimization helps the processor make better use of time that would otherwise be lost waiting for a memory response. Additionally, batching increases memory locality, reducing memory access latency.

Importantly, the combination of batching and prefetching further reduces memory stalls, and this directly correlates with another significant reduction in execution time. Finally, in the RMI Fully Optimized version, the time spent waiting for memory becomes smaller than the time used for useful work, back-end core execution, front-end activity, and even mispredicted speculation. This balance demonstrates a successful alleviation of the memory bottleneck, which makes the processor more efficient.

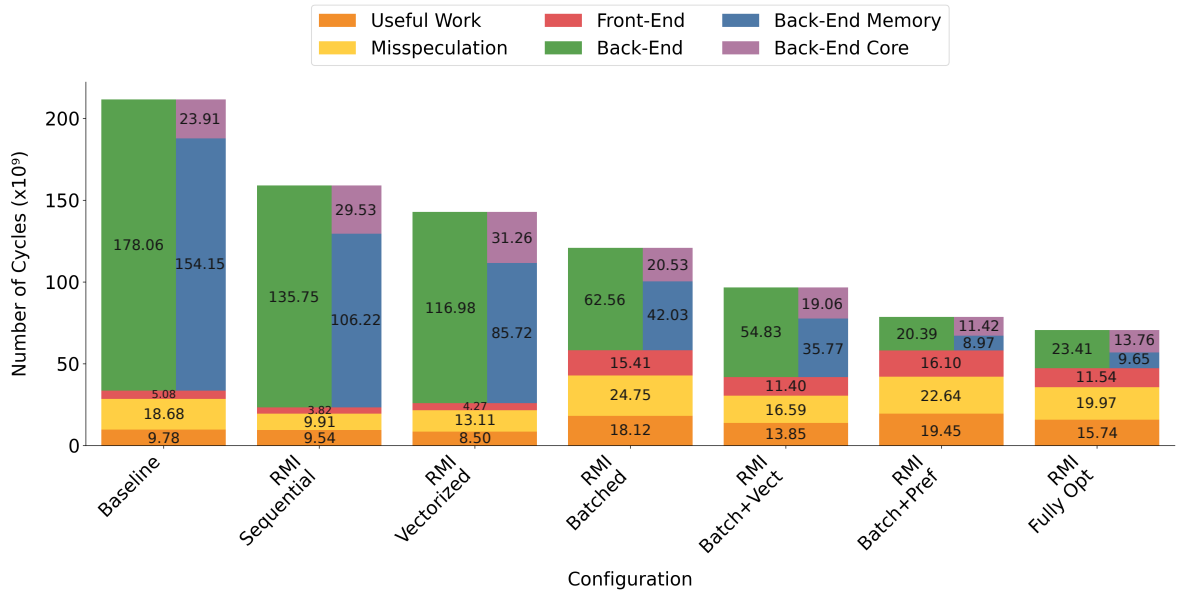

Figure 11: Performance stacks comparison. Distribution of CPU cycles across pipeline categories, including memory stalls, back-end/core execution, useful work, front-end, and mispredictions. Significant reductions in memory stalls are observed when batching and prefetching are combined.

4.3 Roofline Model Analysis

With the previous evaluations done, we now present the roofline model analysis to further identify and reaffirm the compute and memory behavior of each configuration. As explained in the Methodology section, in the roofline, the arithmetic intensity is plotted

along the x-axis and performance along the y-axis. Performance is calculated as the total number of useful operations executed (in our case, integer operations) divided by the execution time, yielding a value in GINTOPS/s. Arithmetic intensity is defined as the number of executed operations divided by the volume of memory transferred, expressed in INTOPs per byte.

As shown in Figure 12, none of the configurations reach the compute-bound region, as all points lie well below the scalar peak performance. Instead, all versions operate primarily in the memory-bound or memory-and-compute-bound regions due to their low instruction-level parallelism and poor memory locality.

Analyzing the specific points:

- Baseline: is clearly memory-bound, failing to exploit the theoretical memory bandwidth. This aligns with previous findings indicating frequent memory stalls that limit its throughput.
- RMI Sequential: performs the lookup using a different algorithm compared to the hash table used in the baseline. It requires fewer memory accesses and involves fewer overall operations, which makes it slightly faster in terms of total execution time. However, this version exhibits lower performance in the roofline model because the nature of its algorithm inherently performs less work. This lower performance value does not imply inefficiency or worse behavior.
- RMI Vectorized: achieves the highest performance among all versions. This is attributed to the use of SIMD instructions, which increase arithmetic intensity by processing multiple data elements per instruction.
- Batched versions: we have seen that show a notable reduction in memory stalls. However, their arithmetic intensity remains relatively low, as the batching strategy primarily adds control and coordination instructions rather than arithmetic operations. In this context, where the task consists exclusively of key lookups without further computation, higher operation throughput is not necessarily meaningful. What matters more is that batching improves data locality, effectively reducing memory stalls.
- RMI Fully Optimized: demonstrates reduced memory stall impact, although its arithmetic intensity remains modest due to the nature of the workload.

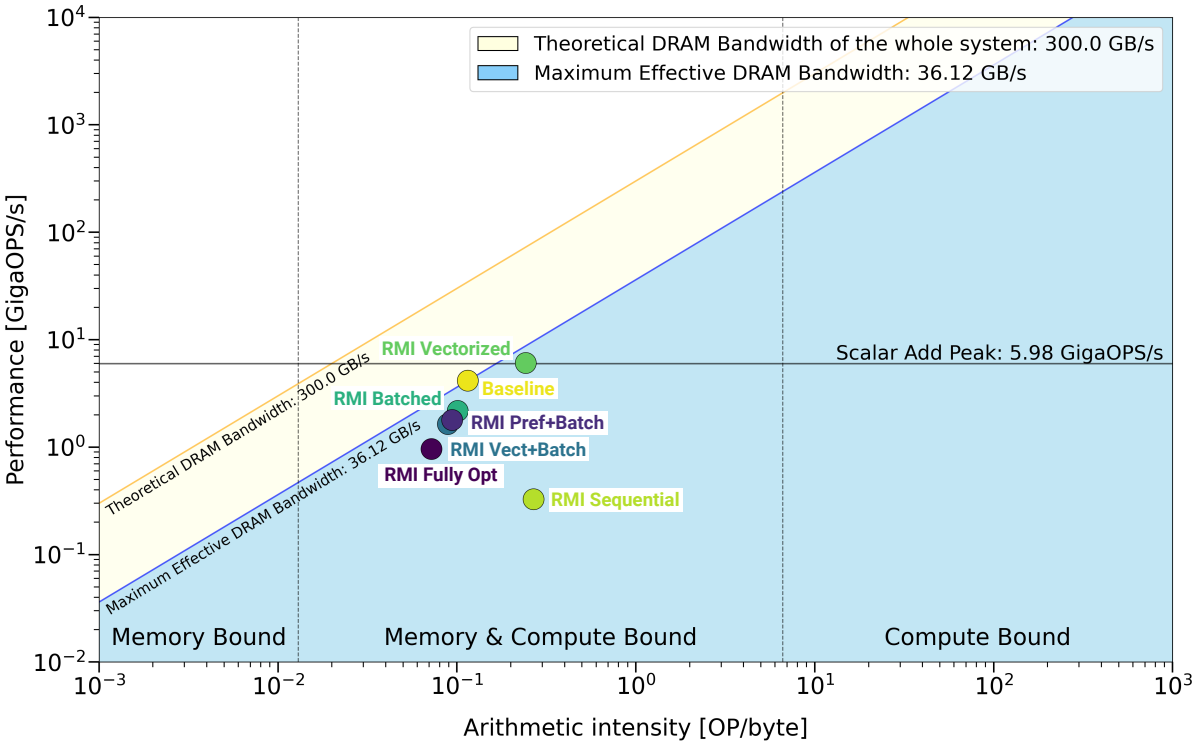

Figure 12: Roofline model. Roofline analysis of the baseline and RMI configurations, showing performance vs arithmetic intensity.

4.4 Multicore Scalability Analysis

After analyzing the behavior of each configuration, we evaluate the strong scalability of the Baseline and the RMI Fully Optimized version, which represents the most effective version overall.

To explore the effects of oversubscription, we extend the thread count beyond the number of physical cores by including 56, 112, and 224 threads. As we explained, the node where the experiments are performed contains two processors with 56 cores each, supporting up to 224 concurrent threads through simultaneous multithreading (SMT).

As shown in Figure 13, the RMI Fully Optimized version scales efficiently, closely following the perfect scalability line up to 112 threads. While performance continues to improve at 224 threads, it no longer matches perfect scalability. This is expected, as multithreading introduces overhead and requires more resources, as running multiple threads per core is not equivalent to having more physical cores.

In contrast, the baseline scales reasonably well up to 56 threads. Although there is some improvement at 112 threads, the speedup stagnates, and at 224 threads, performance degrades significantly. This behavior can be attributed to the memory stalls experienced by the application. Increasing the number of threads does not help once the memory subsystem becomes saturated. Moreover, the random access patterns lead to increased latency, eventually resulting in a performance collapse under the pressure of oversubscription.

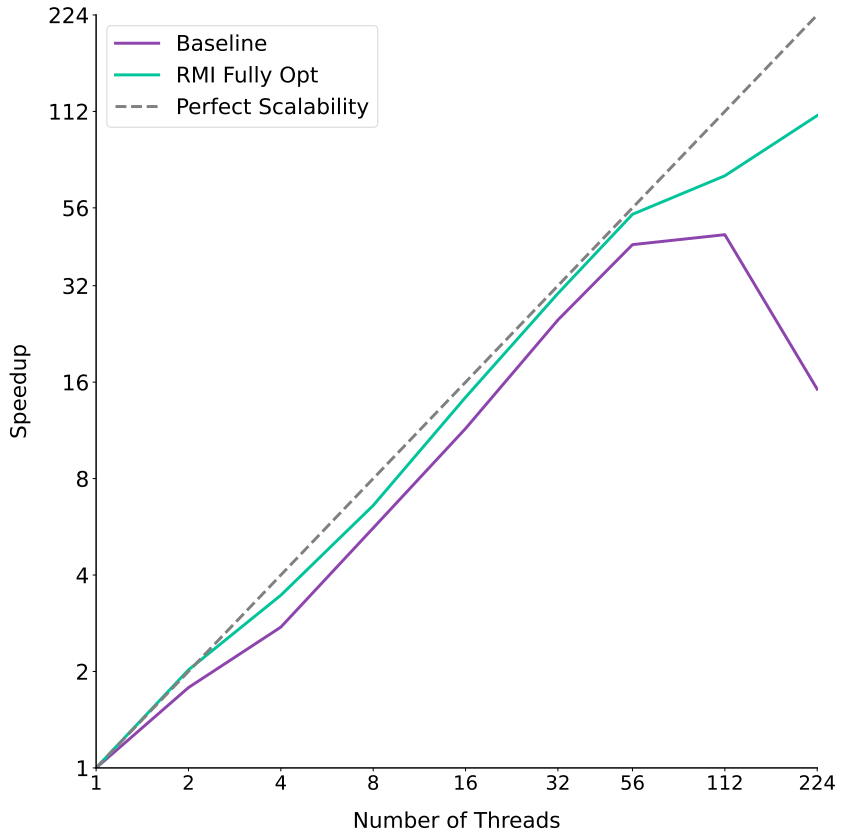

Figure 13: Scalability plot of the multithreading version. Speedup comparison for the Baseline and RMI Fully Optimized versions across different numbers of threads used.

5 RMI Learned Hash Performance Optimization

After completing the microarchitectural evaluation and identifying the best version among the different configurations, we proceed to study the RMI (Recursive Model Index) configuration and design space of the Mm2-fast application.

5.1 RMI Design Space Exploration

By default, the two-layer RMI consists of a single root model in the first layer and multiple leaf models, where each leaf is trained with 32 keys of the key-value table. We explore the design space by reducing the number of keys per leaf model to 16, 8, 4, and 2, and evaluate their impact in terms of execution time and microarchitectural performance using hardware counters. The results are shown in Figure 14.

In terms of the execution time of the lookup, the most notable reduction occurs when transitioning from 32 to 16 keys per leaf model. Further reductions in the number of keys per model continue to improve execution time, although the differences are not as significant. For example, using 2 keys per model achieves a speedup of 1.20×, while 16 keys already reach 1.10×.

The number of index accesses exhibits a more pronounced trend; each halving of the number of keys per model yields approximately 100 million fewer accesses. This linear reduction highlights the improved prediction accuracy of smaller models.

As expected, reducing the number of keys per leaf increases the number of models, thereby increasing the RMI's memory footprint from 12.01 GB (32 keys/model) to 12.30, 12.92, 14.02, and 16.32 GB for 16, 8, 4, and 2 keys per model, respectively, compared to the 11 GB memory usage of the baseline index.

These improvements are due to the fact that models responsible for fewer keys tend to make more accurate predictions. More accurate predictions reduce the error range, resulting in fewer range adjustments and fewer index accesses.

In terms of low-level metrics, the number of instructions executed decreases as the number of keys per model is reduced (due to fewer accesses). The IPC (Instructions Per Cycle) also slightly decreases from 32 to 16 keys but remains stable beyond that. Although MPKI may appear to increase, this is due to the decrease in total instruction count rather than an actual increase in cache misses. Notably, the slight increase in instructions at 4 keys per node is linked to an improvement in misspeculated instructions, as confirmed by the speculation-related counters.

Overall, we consider the configuration with 16 keys per model to offer a good trade-off between performance and memory overhead. It provides measurable gains with only a marginal increase in RMI size.

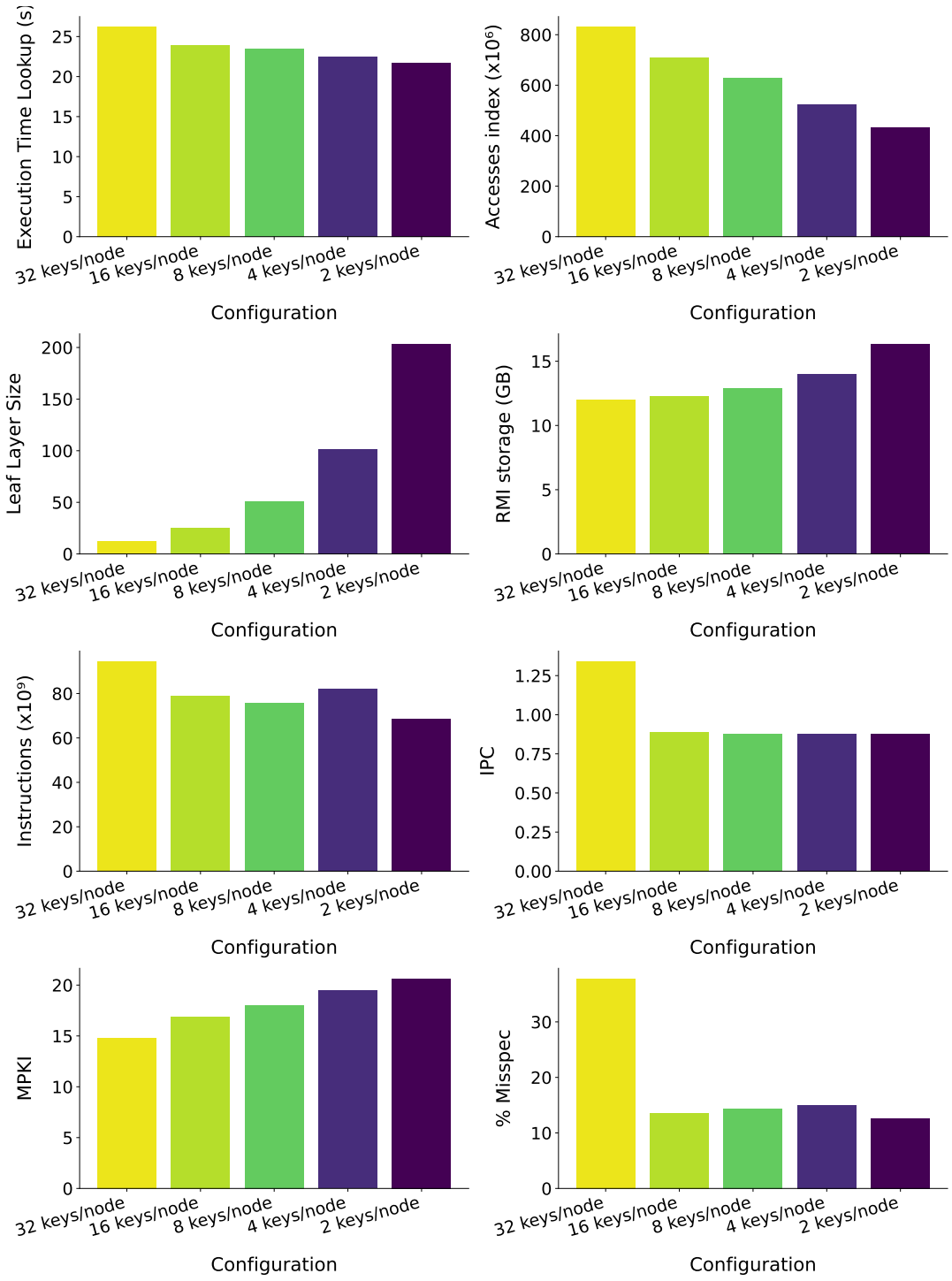

Figure 14: RMI Design Space Exploration Results. Microarchitectural analysis of different RMIs with varying number of keys per model.

To better understand these results, we examine the error distribution of the models in the leaf layer. Figure 15 shows that, as the number of keys per model decreases, the diversity of prediction errors decreases. This aligns with our expectations: fewer distinct

errors suggest better accuracy, which directly explains the reduction in index accesses. Interestingly, although the error distribution becomes smaller, the relative frequencies of errors remain consistent, with error 1 being dominant, followed by error 3.

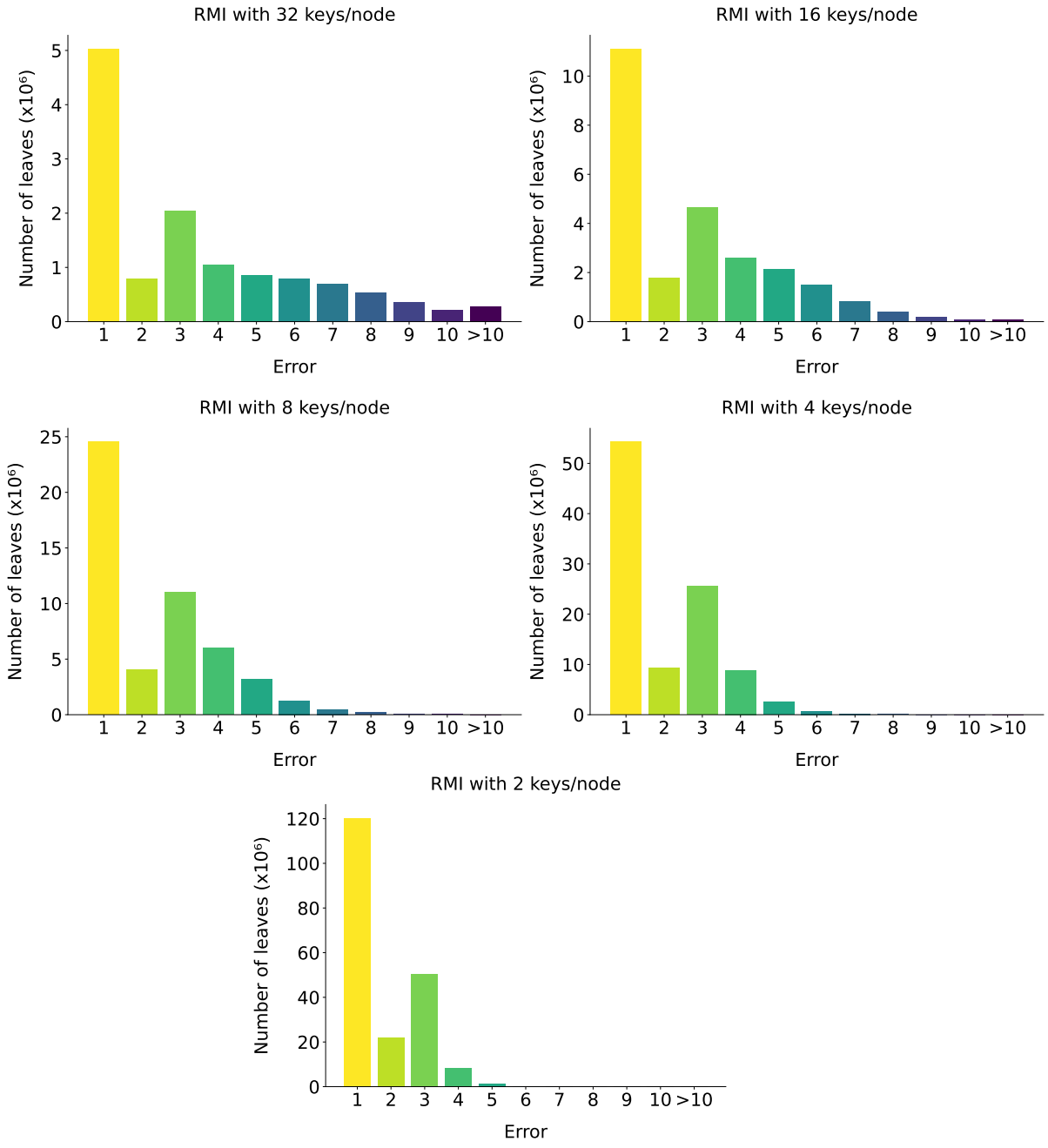

Figure 15: Leaf Layer Error Distribution. Error frequency among models in the leaf layer for different RMI configurations.

5.2 Batched Query Processing and Software Prefetching

We also explore the impact on application performance of varying batch sizes when software prefetching is activated. The results in Figure 16 show that smaller batch sizes lead to longer execution times, indicating a low prefetch timeliness. A low timeliness indicates that the prefetch has been made too late or too early, so that the data is not yet available in the cache when needed, resulting in memory stalls. In this case, the prefetch is made too late, so the prefetched data arrives late, and the memory instructions stall waiting for the response. As the batch size increases, the timeliness of the prefetcher rises, reducing the execution time.

For larger batch sizes, we initially expected a negative impact on performance due to early data prefetches. With many queries in a single batch, prefetching data for later queries could evict data needed by earlier ones, causing new petitions for this data and further stalls. However, this degradation was not observed in our case. This can be explained by the nature of Mm2-fast, which restricts batch sizes to the number of minimizers per query. Since typical queries (e.g., 1000 bp) have fewer than 200 minimizers, batch sizes do not exceed this limit. In conclusion, the default batch size of 32 provides the best balance and achieves optimal performance in our setting.

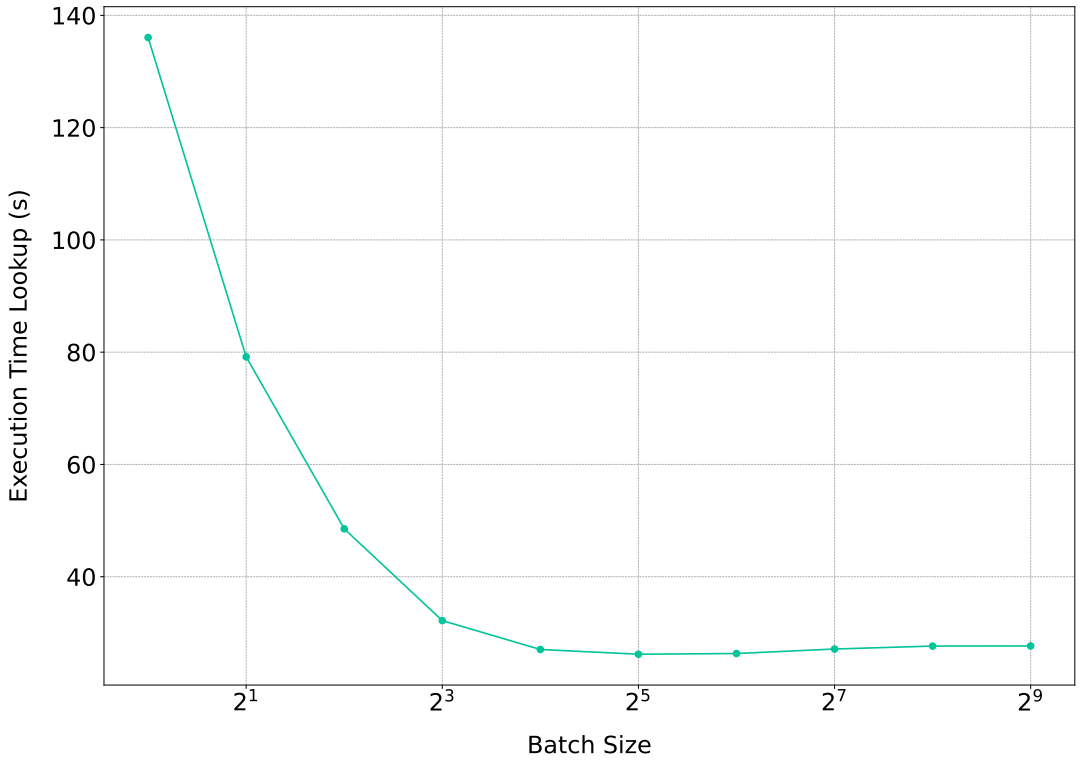

Figure 16: Performance comparison for different Batch Size. Performance of the RMI Fully Optimized configuration using varying batch sizes.

6 Related Work

Hash tables are widely used across various applications, and over the years, numerous solutions have been proposed to enhance their performance [43, 44, 45], particularly when handling large-scale datasets.

In the context of genomics, hashing plays a fundamental role, especially through the indexing of k-mers, which forms the basis of many bioinformatics tools. However, in other tasks, such as sequence comparison, techniques like spaced seeds are increasingly used to improve sensitivity. For example, spaced seeds are employed in read mapping tools, such as SHRiMP [46], and metagenomic classifiers, including CLARK-S [47]. Although these alternatives offer better sensitivity, they also tend to be slower than conventional k-mer hashing. To address this, several efforts have been made to optimize the hashing of spaced seeds, as seen in the work [48].

Beyond hashing, various alternative indexing structures have been proposed to improve performance in read mapping. These include suffix trees [35], suffix arrays [34], Burrows-Wheeler transform (BWT) with FM-index [27], which are commonly used in modern read mappings due to their space-efficiency and effectiveness in compressing genomic data, and q-grams [49], to name a few.

To reduce memory consumption and runtime for index construction associated with traditional hash tables, a memory-efficient structure called a succinct hash index has been proposed [50]. This proposal retains only a small portion of the suffix array entries and reconstructs the remaining ones using neighboring known values. The proposed design significantly reduces memory usage and accelerates index construction through parallelization.

Recently, as the use of learned indexes has become a trend, the rise of learned index structures has extended well beyond genomic applications, offering promising performance improvements in various domains where traditional hash tables are commonly used. For instance, TELEX [51] introduces a two-level learned indexing framework designed to significantly accelerate query processing across entire blockchain datasets. Similarly, other recent works explore ways to enhance learned indexes for diverse tasks such as fast approximate nearest neighbor search in massive databases [52], or improving the speed, accuracy, and scalability of data retrieval systems [53]. These studies demonstrate the growing interest in generalizing and optimizing learned indexes for real-world, large-scale data-intensive applications.

7 Discussion and Conclusions

7.1 Discussion

At the conception of this project, we wondered whether learned hash tables, when adapted and optimized for modern CPU architectures, could outperform traditional hash tables in seeding tasks. While learned hash indexes indeed show better performance than traditional hash tables, they retain some of the same memory-related limitations, requiring additional optimizations to address bottlenecks and leverage the full potential of current architectures.

Our work, in line with other recent studies, highlights the limits of traditional hash tables. Despite the extremely fast computation of hash functions, their performance degrades under demanding scenarios due to pointer chasing dependencies, irregular memory access patterns, poor data locality, and frequent collisions. These limitations lead to underutilized memory bandwidth, low instruction-level parallelism, and frequent cache misses. The margin of improvement is very small because any learned model will be slower to compute than most hash functions. However, learned index structures can leverage features of modern CPUs, such as vectorization, batching, and prefetching, to overcome these architectural challenges and achieve improved overall performance.

Our results support this hypothesis; the RMI Sequential version, which replaces the hash function with a learned model, slightly outperforms the baseline. This performance improvement is likely due to fewer index accesses because of fewer range adjustments than collisions. Nevertheless, a deeper performance analysis shows that this version suffers from the same bottlenecks as the baseline, remaining heavily memory-bound and spending around 66.80% of cycles stalled on memory operations.

Substantial performance gains appear in the RMI-optimized versions. The final RMI Fully Optimized version applies three optimizations: vectorization, batching, and software prefetching. First, vectorizing the last step of the last-mile search offers limited benefit on its own but becomes effective in combination with the other techniques. Second, batching the lookups significantly improves performance by exploiting temporal locality. Third, prefetching, used in conjunction with batching, further reduces memory stalls, thereby improving performance.

Altogether, these three optimizations result in a 2.90× speedup, which is a substantial achievement. However, it is important to note that batching and prefetching are complex to tune. Their effectiveness is highly dependent on system-specific parameters, and misconfiguration can lead to worse performance than not using them at all. This presents a challenge for general deployment, particularly for users without prior knowledge of low-level hardware.

Additionally, our design space exploration suggests that RMI may not be the ideal model for this specific use case. For instance, using only two keys per leaf model should, in theory, yield highly accurate predictions; yet, the performance improvement is marginal. We suspect that this is due to the root model's poor prediction accuracy when directing queries to a large number of leaf models. This observation highlights the need to explore

alternative model architectures or training strategies in future work.

7.2 Conclusions

The thesis focuses on the analysis and characterization of the performance of learned hash tables, identifies their limitations, and explores software and hardware acceleration strategies to enhance their efficiency on modern processors. In this section, we first analyze the limitations of state-of-the-art approaches. Then, we detail the characteristics and bottlenecks of learned hash indexes. Finally, we present the results obtained from evaluating the applied optimizations.

While hash tables are widely used due to their speed and simplicity, their performance suffers significantly when applied to large datasets, such as those encountered in genomic analysis. Tools like Minimap2, a state-of-the-art long-read mapper, rely on traditional hash tables to store the reference genome. However, their performance is limited by memory stalls arising from random access patterns, poor cache locality, and low instruction-level parallelism.

To address these limitations, learned index structures such as learned hash indexes have been proposed. These models replace the hash function with a machine-learned model, as the RMI, that predicts the key's position in a sorted array. We evaluated Mm2-fast, an optimized variant of Minimap2 that integrates a learned hash index through an RMI to substitute the traditional Minimap2's hash table.

Our results show that, despite its promising results, hash learned indexes still present performance limitations due to their memory-bound behavior. For instance, the RMI Sequential version, despite providing a 1.32× speedup over the baseline, spends 66.80% of its CPU cycles stalled on memory.

The most significant improvements come from applying software-level optimizations:

- **Batching** improves execution time by 1.73×.
- **Batching + Prefetching** achieves an speedup of 2.63×.
- **Vectorizing the last-mile search**, on top of batching and prefetching, further improves performance, achieving a total speedup of 2.90× in the RMI Fully Optimized version.

The RMI Fully Optimized version reduces the number of index accesses by 1.50× and also the MPKI by 3.87× compared to the baseline, and memory-bound cycles drop from 66.80% to 13.65%, demonstrating a significant reduction in memory stalls. Notably, we found no compute-bound limitations.

From the design space exploration of the RMI, reducing the number of keys per leaf model from 32 to 2 leads to a 1.20× speedup. This suggests the machine learning model could be better tailored, possibly due to limitations in the root model's ability to route queries to the correct leaf models accurately. Additionally, we observed that the success of batching is highly dependent on careful tuning, as incorrect configurations can degrade performance instead of improving it.

In conclusion, while well-optimized learned hash indexes can outperform traditional hash tables, the performance benefits rely heavily on tuning system-specific parameters. Optimizations such as batching, prefetching, and model configuration must be carefully tailored to the target architecture to fully unlock the potential of these data structures.

These findings, while focused on genome analysis, can be generalized to other hash-table-based applications (which, as previously discussed, are widely used) and extended to alternative processor architectures, highlighting the broad applicability of learned indexes and their optimization strategies.

7.3 Future Work

In future work, we aim to further enhance both traditional and learned hash index performance.

First of all, given the widespread usage of traditional hash tables, it is worth improving. Our profiling showed a clear distinction in collision patterns between found and not-found queries. To mitigate unnecessary lookups, we propose adding a Bloom filter to quickly rule out queries that are not present in the index, significantly reducing unnecessary memory accesses.

In terms of learned indexes, since the RMI model already provides highly accurate predictions, the binary search that follows only needs to compute a few steps to achieve the correct location, so it is often unnecessary. We suggest replacing it with a linear search, which may reduce branch mispredictions and instruction overhead in small search windows.

Lastly, as we mentioned, our findings suggest that the current RMI model may not be ideal, particularly when dealing with large numbers of leaf models. Future work could explore other machine learning models or improved RMI configurations. A more accurate model could potentially further reduce memory accesses and improve performance.

8 Cost Assessment

This section provides a breakdown of the resources used during the project, including infrastructure, tools, and human effort. As this is an academic project, there are no direct monetary costs. However, understanding the underlying effort and infrastructure gives perspective on the scale and investment involved.

8.1 Infrastructure and Equipment

The project made use of the following hardware platforms:

- **Personal laptop:** Dell Inc. Latitude 7490 provided by BSC used for development, testing, and documentation.
- **Peripherals:** two secondary screens, a cordless mouse, and a cordless keyboard provided by BSC.
- **HPC cluster (MareNostrum 5 – GPP partition):** used for benchmarking learned hash indexes at scale.

No hardware acquisition or usage costs were incurred, as all resources were made available by BSC, including access to MareNostrum.

8.2 Software and Tools

All software used was open-source or free for academic use:

- **Mapping tools:** `Minimap2`, `Mm2-fast`
- **Profiling tools:** `perf`, `Valgrind`, `Intel VTune`
- **Programming environments:** `GCC`, `Clang`, `Python`
- **Other tools:** `Git`, `GitHub`, `Overleaf`, `LaTeX`, `Visual Studio Code`, `Slack`

These tools enabled the full development and analysis pipeline without requiring commercial licenses. As these applications do not incur any costs, the total cost for software in this project is 0 €.

8.3 Human Resources

The project was conducted over a period of five months, with the following estimated human effort:

- **Student:** approx. 735–750 hours dedicated to literature review, software setup, performance profiling, optimizations, and thesis writing.
- **Supervisor:** regular guidance through scheduled meetings, feedback, and access management to compute resources.

8.4 Estimated Effort by Project Phase

Project Task	Estimated Hours
T0: Project Meetings	25
T1: Literature Review	60
T2: Algorithmic Study	115
T3: Software Setup	130
T4: Profiling and Benchmarking	155
T5: Optimization	135
T6: Documentation and Review	95
T7: Defense Preparation	20
Total Estimated Effort	**735 hours**

Table 1: Estimated effort per project phase

8.5 Summary

No direct monetary budget was necessary due to the use of institutional infrastructure and free software. However, the project demanded a sustained time investment and access to advanced computing resources. Table 2 summarizes the estimated resource usage.

Category	Cost / Value
Hardware	Provided by BSC (0 €)
HPC Usage	Granted through BSC (0 €)
Software Licenses	Open-source / academic use (0 €)
Student Effort	735 hours
Supervisor Effort	Academic duty
Total Direct Cost	**0 €**

Table 2: Cost summary

9 Legislation and Data Protection

This project complies with the applicable legal and institutional regulations concerning intellectual property, software usage, and data protection.

9.1 Software Licensing

All software tools and libraries used during this project are open-source or available under academic-friendly licenses. This includes genome mapping applications (`Minimap2`, `Mm2-fast`) and profiling tools (`perf`, `Valgrind`, `Intel VTune`). Their use respects the terms of their respective licenses, primarily MIT, BSD, or GPL, and no proprietary software was incorporated.

9.2 Data Usage and Protection

The project does not involve any personal, sensitive, or clinical data. All genomic datasets used for benchmarking purposes are synthetic, simulated, or publicly available from open-access repositories. Therefore, the work does not fall under the scope of the General Data Protection Regulation (GDPR) or Spain's Organic Law 3/2018 on the Protection of Personal Data and Guarantee of Digital Rights (LOPDGDD).

Nonetheless, good data handling practices were followed, including the following.

- Citing the sources of all datasets used.
- Ensuring data reproducibility and integrity through proper version control.
- Avoiding the download or storage of restricted or identifiable genomic data.

10 Ethical, Equity, and Environmental Considerations

This section analyzes the Final Bachelor's Thesis from the perspective of the transversal competence CT7: "Apply ethical principles and social responsibility as a citizen and professional". The project is examined concerning four key indicators: gender equality, environmental sustainability, social responsibility, and ethics in professional practice.

10.1 Gender Equality

The project does not directly involve user-facing components or decision-making algorithms that could manifest gender bias. However, care has been taken to ensure that the literature and tools used do not propagate biased assumptions or exclude contributions based on gender. In line with inclusive academic practice, gender-neutral language has been adopted throughout the written work.

10.2 Environmental Sustainability

Although the project does not focus on environmental issues, it involves intensive use of computational resources. Performance profiling and benchmarking tasks executed on high-performance computing systems (e.g., MareNostrum 5) consume a significant amount of energy. To mitigate environmental impact, experiments were planned to minimize unnecessary reruns and redundant tests. Efficiency was prioritized not only for performance gains but also as a means to reduce computational cost and energy consumption.

10.3 Social Responsibility

The project's objective is aligned with improving the efficiency of computational tools used in genomic research, which ultimately contributes to better scalability in biomedical applications. Efficient genome mapping tools are essential in areas such as population genomics, epidemiology, and precision medicine. By optimizing learned indexes, this work indirectly supports research infrastructures that benefit public health and scientific advancement.

10.4 Ethical Considerations

All software and data used in this project are publicly available and fall under open-source or academic licenses. No personal or sensitive data was processed. The project has been conducted with academic integrity, including proper citation of all external resources and tools. The principles of reproducibility, transparency, and adherence to community standards in software research guided decisions made throughout the development process.

10.5 Academic Integrity

The development of this project adheres to the URV's code of academic conduct (CODE OF ETHICS OF THE UNIVERSITAT ROVIRA I VIRGILI). All external contributions, including code, data, and literature, are properly credited. No third-party content has been plagiarized, and the research has been carried out independently under the guidance of the assigned supervisor.

11 Personal Evaluation of the Work

This project was conducted during a 5-month extracurricular internship in the second semester of the 2024-2025 academic year. During the initial weeks, I reviewed relevant literature to understand the context and algorithms. Afterwards, I began familiarizing myself with the two applications to compare (Minimap2 and Mm2-fast) and analyzed their algorithms and indexing structure. This stage was particularly interesting because I was excited to dive into the code and explore it in depth. However, it was also slow and challenging to understand a complex codebase not written by oneself. Tools like Callgrind proved especially useful in this phase, helping me trace the program's execution path. At this point, we accomplished the first objective, and I also began learning to use the MareNostrum 5 system. Submitting jobs on a supercomputer for the first time was an exciting milestone for me.

Next, we set up the benchmarking environment and adapted the codebase to extract the desired metrics to support profiling. At the same time, performance analysis tools were used to obtain various profiling data. This phase allowed me to learn to use several tools such as perf, VTune, IntelSDE, and gprof. These tools generate large amounts of data, and the most challenging aspect for me was learning to interpret this information to understand what was happening in the system. As I started to better understand how to identify bottlenecks, I became increasingly motivated to analyze the system's behavior in depth and explore potential improvements. With that, the second objective was achieved.

Having identified performance problems in the applications, we experimented with different configurations to determine which changes had the most significant impact. We also added missing optimization configurations and compared all possible combinations to identify the most effective one. This process helped me not only to learn the optimization techniques and how to apply them, but more importantly, to understand why they improve performance. After identifying the best version, we modified RMI parameters and optimization configurations to observe their impact. For me, this was the most interesting part of the project: understanding the internal workings of the techniques, identifying the issues they address, and exploring the RMI behavior using a wide range of metrics. This allowed us to reach both the third and fifth objectives by identifying useful optimizations and explaining their effectiveness. The final experiment focused on scalability to assess the solution's ability to parallelize, fulfilling the fourth objective.

Overall, I believe that completing this project, and the internship simultaneously, has greatly contributed to my personal and academic development. I learned a lot about algorithmics, computer architecture, and HPC concepts. At the same time, I begun developing essential research skills: learning how to deal with blocking situations, interpret complex results, and reason about performance and system behavior. I have always thought that doing research requires a wide range of capabilities that are gradually acquired through experience. Thanks to this internship and the guidance of my supervisors, I feel I have taken my first steps and acquired a few of those skills in computer science research. Although there is still a long way to go, I believe these skills will be valuable for my future. I am deeply grateful for the opportunity. This has been a very enriching and complete project, and I want to sincerely thank my supervisors and academic tutor for their guidance.

12 Resources

References

[1] T.A. Brown. *Genomes*. 2nd. Oxford: Wiley-Liss, 2002. URL: https://www.ncbi.nlm.nih.gov/books/NBK21128/.

[2] F. Sanger, S. Nicklen, and A. R. Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences* 74.12 (1977), pp. 5463–5467. URL: https://www.pnas.org/doi/abs/10.1073/pnas.74.12.5463.

[3] J. Craig Venter et al. "The Sequence of the Human Genome". In: *Science* 291.5507 (2001), pp. 1304–1351. URL: https://www.science.org/doi/abs/10.1126/science.1058040.

[4] Illumina. *Illumina: Sequencing and Array-based Solutions for Genetic Research*. Accessed 2025-05-02. 1998. URL: https://www.illumina.com/.

[5] Pacific Biosciences of California, Inc. *PacBio: Highly Accurate Long-Read Sequencing*. Accessed 2025-05-12. 2004. URL: https://www.pacb.com/.

[6] Oxford Nanopore Technologies. *Oxford Nanopore Technologies: Real-Time DNA/RNA Sequencing*. Accessed 2025-05-12. 2005. URL: https://nanoporetech.com/.

[7] Jason L Weirather et al. "Comprehensive comparison of Pacific Biosciences and Oxford Nanopore Technologies and their applications to transcriptome analysis". In: *F1000Research* 6 (2017), p. 100. URL: https://doi.org/10.12688/f1000research.10571.2.

[8] Rachid Ounit et al. "CLARK: Fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers". In: *BMC genomics* 16.1 (2015), p. 236. URL: https://doi.org/10.1186/s12864-015-1419-2.

[9] Daehwan Kim et al. "Centrifuge: rapid and sensitive classification of metagenomic sequences". In: *Genome research* 26.12 (2016), pp. 1721–1729. URL: https://doi.org/10.1101/gr.210641.116.

[10] Derrick E Wood, Jennifer Lu, and Ben Langmead. "Improved metagenomic analysis with Kraken 2". In: *Genome biology* 20.1 (2019), p. 257. URL: https://doi.org/10.1186/s13059-019-1891-0.

[11] Sam Kovaka et al. "Targeted nanopore sequencing by real-time mapping of raw electrical signal with UNCALLED". In: *Nature biotechnology* 39.4 (2021), pp. 431–441. URL: https://doi.org/10.1038/s41587-020-0731-9.

[12] Harisankar Sadasivan et al. "Rapid Real-time Squiggle Classification for Read until using RawMap". In: *Archives of clinical and biomedical research* 7.1 (2023), pp. 45–47. URL: https://doi.org/10.26502/acbr.50170318.

[13] Stephen F. Altschul et al. "Gapped BLAST and PSI-BLAST: A new generation of protein database search programs". In: *Nucleic Acids Research* 25.17 (Sept. 1997), pp. 3389–3402. ISSN: 0305-1048. URL: https://doi.org/10.1093/nar/25.17.3389.

[14] W James Kent. "BLAT—the BLAST-like alignment tool". In: *Genome research* 12.4 (2002), pp. 656–664. DOI: 10.1101/gr.229202.

[15] Ruiqiang Li et al. "SOAP: short oligonucleotide alignment program". In: *Bioinformatics* 24.5 (Jan. 2008), pp. 713–714. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btn025.

[16] Ben Langmead et al. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". In: *Genome biology* 10.3 (2009), R25. URL: https://doi.org/10.1186/gb-2009-10-3-r25.

[17] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows–Wheeler transform". In: *Bioinformatics* 25.14 (May 2009), pp. 1754–1760. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btp324.

[18] Nils Homer, Barry Merriman, and Stanley F Nelson. "BFAST: An alignment tool for large scale genome resequencing". In: *PloS one* 4.11 (2009), e7767. URL: https://doi.org/10.1371/journal.pone.0007767.

[19] Ruiqiang Li et al. "SOAP2: An improved ultrafast tool for short read alignment". In: *Bioinformatics* 25.15 (June 2009), pp. 1966–1967. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btp336.

[20] Ben Langmead and Steven L Salzberg. "Fast gapped-read alignment with Bowtie 2". In: *Nature methods* 9.4 (2012), pp. 357–359. URL: https://doi.org/10.1038/nmeth.1923.

[21] John C. Mu et al. "Fast and accurate read alignment for resequencing". In: *Bioinformatics* 28.18 (July 2012), pp. 2366–2373. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/bts450.

[22] Santiago Marco-Sola et al. "The GEM mapper: Fast, accurate and versatile alignment by filtration". In: *Nature methods* 9.12 (2012), pp. 1185–1188. URL: https://doi.org/10.1038/nmeth.2221.

[23] Heng Li. "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM". In: *arXiv preprint arXiv:1303.3997* (2013). URL: https://doi.org/10.48550/arXiv.1303.3997.

[24] Wan-Ping Lee et al. "MOSAIK: A hash-based algorithm for accurate next-generation sequencing short-read mapping". In: *PloS one* 9.3 (2014), e90581. URL: https://doi.org/10.1371/journal.pone.0090581.

[25] Konstantin Berlin et al. "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing". In: *Nature biotechnology* 33.6 (2015), pp. 623–630. URL: https://doi.org/10.1038/nbt.3238.

[26] Heng Li. "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences". In: *Bioinformatics* 32.14 (Mar. 2016), pp. 2103–2110. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btw152.

[27] P. Ferragina and G. Manzini. "Opportunistic data structures with applications". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science* (2000), pp. 390–398. DOI: 10.1109/SFCS.2000.892127.

[28] Heng Li and Richard Durbin. "Fast and accurate long-read alignment with Burrows–Wheeler transform". In: *Bioinformatics* 26.5 (Jan. 2010), pp. 589–595. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btp698.

[29] Md. Vasimuddin et al. "Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems". In: (2019), pp. 314–324. DOI: 10.1109/IPDPS.2019.00041.

[30] Davide Campagna et al. “PASS: A program to align short sequences”. In: *Bioinformatics* 25.7 (Feb. 2009), pp. 967–968. URL: https://doi.org/10.1093/bioinformatics/btp087.

[31] Hugh L. Eaves and Yuan Gao. “MOM: Maximum oligonucleotide mapping”. In: *Bioinformatics* 25.7 (Feb. 2009), pp. 969–970. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btp092.

[32] Heng Li. “Minimap2: pairwise alignment for nucleotide sequences”. In: *Bioinformatics* 34.18 (May 2018), pp. 3094–3100. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/bty191.

[33] Mohammed Alser et al. “Technology dictates algorithms: recent developments in read alignment”. In: *Genome biology* 22.1 (2021), p. 249. URL: https://doi.org/10.1186/s13059-021-02443-7.

[34] Udi Manber and Gene Myers. “Suffix arrays: a new method for on-line string searches”. In: *siam Journal on Computing* 22.5 (1993), pp. 935–948. URL: https://doi.org/10.1137/0222058.

[35] Peter Weiner. “Linear pattern matching algorithms”. In: *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)* (1973), pp. 1–11. DOI: 10.1109/SWAT.1973.13.

[36] Arnold I Dumey et al. “Indexing for rapid random access memory systems”. In: *Computers and Automation* 5.12 (1956), pp. 6–9.

[37] W. W. Peterson. “Addressing for Random-Access Storage”. In: *IBM Journal of Research and Development* 1.2 (1957), pp. 130–146. DOI: 10.1147/rd.12.0130.

[38] W. D. Maurer and T. G. Lewis. “Hash Table Methods”. In: *ACM Comput. Surv.* 7.1 (Mar. 1975), pp. 5–19. ISSN: 0360-0300. DOI: 10.1145/356643.356645. URL: https://doi.org/10.1145/356643.356645.

[39] Michael Roberts et al. “Reducing storage requirements for biological sequence comparison”. In: *Bioinformatics* 20.18 (July 2004), pp. 3363–3369. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/bth408.

[40] Saurabh Kalikar et al. “Accelerating minimap2 for long-read sequencing applications on modern CPUs”. In: *Nature Computational Science* 2.2 (2022), pp. 78–83. URL: https://doi.org/10.1038/s43588-022-00201-8.

[41] National Center for Biotechnology Information (NCBI). *GCF_000001405.40 - Genome Reference Consortium Human Build 38 (GRCh38.p14)*. Accessed 2025-02-20. 2025. URL: https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.40/.

[42] European Nucleotide Archive (ENA). *SRX19566475 - ENA Browser*. Accessed: 2025-02-20. 2025. URL: https://www.ebi.ac.uk/ena/browser/view/SRX19566475.

[43] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo hashing”. In: (2001), pp. 121–133. URL: https://doi.org/10.1007/3-540-44676-1_10.

[44] H. Donovan et al. *Sparsehash Library*. https://code.google.com/p/sparsehash/. Accessed: 5 May 2025. 2013.

[45] Nikolas Askitis and Justin Zobel. “Cache-conscious collision resolution in string hash tables”. In: (2005), pp. 91–102. URL: https://doi.org/10.1007/11575832_11.

[46] Stephen M. Rumble et al. “SHRiMP: Accurate Mapping of Short Color-space Reads”. In: *PLOS Computational Biology* 5.5 (May 2009), pp. 1–11. URL: https://doi.org/10.1371/journal.pcbi.1000386.

[47] Rachid Ounit and Stefano Lonardi. "Higher classification sensitivity of short metagenomic reads with CLARK-S". In: *Bioinformatics* 32.24 (Aug. 2016), pp. 3823–3825. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btw542.

[48] Samuele Girotto, Matteo Comin, and Cinzia Pizzi. "Efficient computation of spaced seed hashing with block indexing". In: *BMC bioinformatics* 19 (2018), pp. 29–38. URL: https://doi.org/10.1186/s12859-018-2415-8.

[49] Gonzalo Navarro and Ricardo Baeza-Yates. "A practical q-gram index for text retrieval allowing errors". In: *CLEI Electronic Journal* 1.2 (1998), pp. 3–1. URL: https://doi.org/10.19153/cleiej.1.2.3.

[50] Haowen Zhang et al. "Fast and efficient short read mapping based on a succinct hash index". In: *BMC bioinformatics* 19 (2018), pp. 1–14. URL: https://doi.org/10.1186/s12859-018-2094-5.

[51] Haotian Wu et al. "TELEX: Two-Level Learned Index for Rich Queries on Enclave-based Blockchain Systems". In: *IEEE Transactions on Knowledge and Data Engineering* (2025), pp. 1–16. DOI: 10.1109/TKDE.2025.3564905.

[52] Wei Liu et al. "Discrete graph hashing". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 3419–3427.

[53] Ankita Sappa. "Neural Network Powered Indexing Techniques for High Performance Data Retrieval". In: *Research Briefs on Information and Communication Technology Evolution* 11 (Mar. 2025), pp. 22–41. URL: https://doi.org/10.69978/rebicte.v11i.210.