Understanding, Modeling, and Improving Main-Memory Database Performance

Stefan Manegold

About the cover: The image depicts a relief on a wall of the *Temple of Horus* at Edfu in Upper Egypt between Aswan and Luxor. This major Ptolemaic temple is dedicated to the falcon god *Horus* and was built over a 180-year period from 237 BC to 57 BC. I took the picture during a memorable two-week sightseeing tour through Egypt in September 2000, after having attended the VLDB Conference in Cairo.

Understanding, Modeling, and Improving Main-Memory Database Performance

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. mr. P. F. van der Heijden
ten overstaan van een door het
college voor promoties ingestelde commissie,
in het openbaar te verdedigen
in de Aula der Universiteit
op dinsdag 17 december 2002, te 11.00 uur

door

Stefan Manegold

geboren te Eschwege, Duitsland

Promotiecommissie

Promotor:	M. L. Kersten
Overige leden:	D. J. DeWitt
	P. van Emde Boas
	L. O. Hertzberger
	A. P. J. M. Siebes
	G. Weikum

Faculteit

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam

The research reported in this thesis has been partially carried out at *CWI*, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme *Data Mining and Knowledge Discovery*, a subdivision of the research cluster *Information Systems*.

SIKS Dissertation Series No 2002-17.
The research reported in this thesis has been carried out under the auspices of *SIKS*, the Dutch Research School for Information and Knowledge Systems.

ISBN 90 6196 517 9

Contents

Acknowledgements

My first visit to Amsterdam during an InterRail-tour through Europe in summer 1992 lasted only a few hours and left an impression of anything else but hospitality: All left-luggage lockers occupied, endless queues in front of the left-luggage office, and sightseeing while carry a heavy backpack was not much fun. It took another 5 years, a move from C. where I had studied to B. where I got my first (real) job, two much more pleasant visits in summer 1997, during which A. presented itself in an almost Mediterranean atmosphere, and finally a job offer, to convince me that Amsterdam might be a place to life and work. However, with the contract signed and everything else arranged, there was no need anymore for a Mediterranean atmosphere, hence, when I arrived on October 1st 1997 it started raining and did not stop until March 1998... Nevertheless, I did stay, enjoyed life and work in Amsterdam, and eventually wrote this booklet.

Although only one name appears on the cover, this thesis would not exist without the direct or indirect support of various helpful people who accompanied me during the last five years, both locally and remotely. I take this opportunity to express my thanks to all of them.

First of all, I would like to thank my supervisor Martin Kersten. Together with Arno Siebes, he provided me with this great opportunity to come to Amsterdam and do research in an inspiring and well-funded group. Back then, both of us had no idea, which "problems" would pop-up a few years later. The more I am grateful to him for convincing me, whenever I lost faith, that all the struggling will eventually lead to a successful and (hopefully) happy end.

Further, I would like to thank David DeWitt, Peter van Emde Boas, Louis Hertzberger, Arno Siebes, and Gerhard Weikum for being on my committee. It is a great honor for me, that all of them found time to read my theses.

Special thanks go to all my Dutch and international colleagues within INS for the warm welcome they gave me right from the beginning, and their support after my most memorable encounter with Amsterdam traffic. Some of them shall be mentioned here, but I am thankful to all the others as well.

Two colleagues at CWI played major roles during my PhD track. Working together with Florian and Peter, and writing successful papers with both of them (though not at the same time) has been a privilege, an enlightning experience, and a great pleasure that I would not want to miss. Long deadline-nights with last-minute-submission adrenaline were always rewarded by "traditional dinners", at least as long nights with “contemporary music” or Pauwel Kwak and his friends, and finally joint conference trips. I hope there is more to come in the future.

Work at CWI would not have been that interesting, if there had not been “some” distractions from the pure research work. I could not imagine a better colleague for system administration and “Monet-hacking” than Niels. We do speak the same language, not only when talking about “common enemies”. Menzo has been the perfect room mate. Not only did he accept my irregular office hours, but he also showed great patience and helpfulness with all my nasty questions and problems, ranging from Dutch language and culture over details of Monet to various computer- and software-related problems. Always walking a few steps ahead of me, Albrecht guided my way through all the administrative and technical hurdles of the last few month of this project.

Finally, I want thank all my “old friends from home”. Despite being spread (almost) all over Europe, we stayed in touch, meat at various parties, spent fortunes on phone bills, and some of them even managed to visit me in Amsterdam. Not loosing contact to them has been vital for my Amsterdam-project. I am especially grateful to my most frequent (and most welcome) visitor that she has been concerned of and took care of my cultural life.

Insbesondere danke ich auch meiner Familie. Ohne Eure Geduld und Eure Unterstützung — nicht nur was mein leibliches Wohl angeht ;-) — wäre diese Arbeit nicht möglich gewesen.

Amsterdam, November 2002

Chapter 1

Introduction

Databases have not only become essential to business and science, but also begin to appear more and more in everyday life. Classically, databases are used to maintain internal business data about employees, clients, accounting, stock, and manufacturing processes. Furthermore, companies nowadays use them to present data to customers and clients on the World-Wide-Web. In science, databases store the data gathered by astronomers, by investigators of the human genome, and by biochemists exploring the medical properties of proteins, to name only a few examples. With home-PC's becoming more and more powerful — one can easily get 1 gigabyte (*GB*) of main-memory and up to 160 GB on a single commodity disk drive — database systems are beginning to appear as a common tool for various computer applications, much as spreadsheets and word processors did before them. Even in portable devices, such as PDAs or mobile phones, (small) databases are used for storing everything from contact information such as postal addresses and phone numbers to your digital family photo album. And smart-cards storing a persons medical history are just around the corner.

The power of databases comes from a body of knowledge and technology that has developed over several decades and is embodied in specialized software called a *database management system*, or *DBMS*, or colloquially a "database system". DBMSs are among the most complex types of software available. A DBMS is a powerful tool for creating and managing large amounts of data efficiently. First of all, it provides capabilities allowing safe and persistent storage of data over long periods of time. Furthermore, DBMSs provide powerful query[1] languages, such as for instance *SQL* (*Structured Query Language*), that allow both interactive users and application programs to access and modify the data stored in the database.

In this thesis, we assume the reader is familiar with the basic concepts of DBMSs and database query processing as described in good introductory literature, for instance [KS91, EN94, AHV95, GMUW02].

[1] A "query" is database lingo for a question about the data.

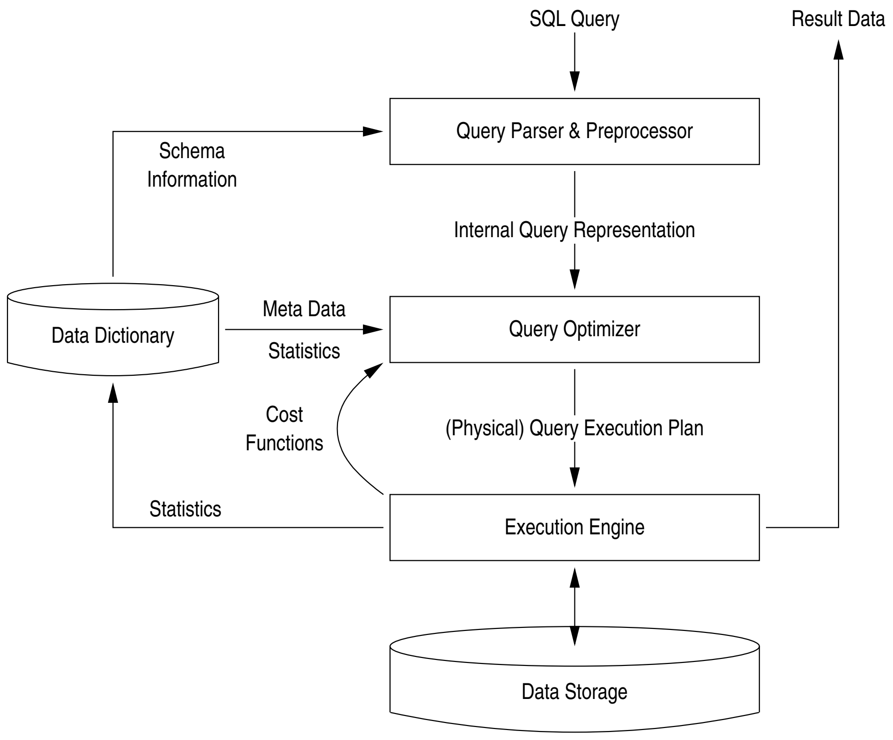


Figure 1.1: Query Processing Architecture

1.1 Query Processing

A major reason why database systems have become so popular and successful is the fact that users can formulate their queries in an almost “intuitive” way using declarative query languages such as SQL. This means, that users just have to care about *what* they want to know, but not *how* to retrieve this information from the data stored in the database. In particular, users do not need to have any sophisticated programming skills or knowledge about how the DBMS physically stores the data. All a user needs to know to formulate ad-hoc queries is the query language and the logical database schema.

Completely hidden from the user, a complex machinery within the DBMS then takes care of interpreting the user’s query and executing the right commands to provide the user with the requested answer. Figure 1.1 roughly sketches a typical database system’s query processing architecture.[2]

Briefly, query processing consists of the following steps. First, the query text is

[2]The complete architecture of nowadays DBMSs is obviously much more complex. Here, we focus on the parts that are related to the work in this thesis. We also omit details that are not specific to problems and techniques addressed in this work.

parsed, checked for syntactical and semantical correctness, and translated into an internal representation. In relational DBMSs, this representation is typically derived from the *relational algebra* and makes up some kind of *operator tree*. Due to certain properties of the relational algebra, such as commutativity and associativity of operators, each query can be represented by various operator trees. These trees are obviously equivalent in that they define the same query result. However, the order in which the various operators are applied may differ. Moreover, relational algebra provides equivalent alternatives for certain operator combinations. One of the most prominent examples is the following. Imagine a sequence where a cartesian product of two tables, say $\times(U, V)$, is followed by a selection σ applying a predicate θ on the newly combined table: $\sigma_\theta(\times(U, V))$. This sequence can alternatively be expressed as a single join operation $\bowtie$ on the two tables, i.e., $\sigma_\theta(\times(U, V)) \equiv \bowtie_\theta (U, V)$. Usually, database query processors apply some normalizations first, to provide a “clean” starting point for the subsequent tasks.

In a second step, the normalized operator tree has to be translated into a procedural program that the DBMS’s query engine can execute. We call such a program a *query execution plan* (*QEP*), or simply *query plan*. Usually, a declarative query can be translated into several equivalent QEPs. These QEPs may differ not only in the order of operator execution (see above), but also in the algorithms used for each single operator (e.g., *hash-join*, *merge-join*, and *nested-loop-join* are the most prominent join algorithms), and in the kind of access structures (such as indices) that are used. The DBMS’s query optimizer is in charge of choosing the “best” (or at least a “reasonably good”) alternative. The goal or objective (function) of this optimization depends on the application. Traditional goals are, e.g., to minimize the response time (for the first answer, or for the complete result), to minimize the resource consumption (like CPU time, network bandwidth or amount of memory), or to maximize the throughput, i.e., the number of queries that the system can answer per time. Other, less obvious objectives — e.g., in a mobile environment — may be to minimize the power consumption needed to answer the query or the on-line time being connected to a remote database server.

For the time being, we do not have to distinguish between these different objective functions. Hence, we use the common terminology in the database world, and call them the *execution costs* or simply *costs* of the QEP. Thus, query optimization means to find a QEP with minimal execution costs.

Conceptually, query optimization is often split into two phases. First, the optimizer determines the order in which the single operators are to be applied. This phase is commonly referred to as *logical optimization*. By definition, the final result size is the same for all equivalent QEPs of a given query. However, with different operator orders, the intermediate result sizes may vary significantly. Assuming that the execution costs of each operator, and hence of the whole QEP, are mainly determined by the amount of data that is to be processed, logical optimization usually aims at minimizing the total sum of all intermediate result sizes in a QEP. In a second phase, commonly called *physical optimization*, the query optimizer determines for each operator in the QEP, which of the available algorithms is to be used and whether existing access structures (e.g., indices) can/should be exploited. Physical optimization aims at actually minimizing the execution costs with respect to the given cost metric.

In practice, however, the implicit assumption that the best *physical plan* can be derived from the best *logical plan* usually does not hold. Hence, query optimization is often performed in a single phase combining both logical and physical optimization. A further discussion of this subject, especially the problems that arise due to the increased complexity of the optimization process, is beyond the scope of the thesis. The interested reader is referred to, e.g., [Cha98, Pel97, Waa00]. In this work, we focus on cost modeling and consider cost models independently from a particular optimization algorithm.

1.2 Cost Models

In order to find the desired QEP, optimizers need to assess and compare different alternatives with respect to their costs. Obviously, evaluating a QEP to measure its execution cost does not make sense. Hence, we need to find a way to *predict* the execution costs of a QEP *a priori*, i.e., without actually evaluating it.

This is where *cost models* come into play. Cost models can be seen as abstract images or descriptions of the real system. Providing a simplified view of the system, (cost) models help us to analyze and/or better understand how a system works, and hence, enable us to estimate query execution costs within this "idealized" abstract system without actually executing the query. The abstraction from the real system is captured in a set of assumptions made about the system, e.g., assuming uniform data distributions, independence of attribute values, constant cost (time) per I/O operation, no system contention, etc.. The degree of abstraction depends on the purpose the cost model is supposed to serve. In general, the more general assumptions are made, i.e., the more abstract the model is, the less adequate or accurate it is. The more detailed a model is, the more accurate it is. The most detailed, and hence most accurate, model is the system itself. However, not only the accuracy of a model is important, but also the time necessary to derive estimations using the model. Here, it usually holds, that evaluation time decreases with the increasing degree of abstraction. In other words, the less accurate a model is, the faster can estimations be evaluated. For instance, models based on *simulation* provide a very detailed image of the real system, and hence allow very accurate estimates. Evaluating the model, however, means running a simulation experiment, which might take even longer than running the real system. On the other hand, using more general assumptions makes the models simpler. Simple models might then be represented in closed mathematical terms. We call these models *analytical models*. Evaluating analytical models means simply evaluating (a set of) closed mathematical expressions. This is usually much faster than both evaluating simulation models and evaluating the real system. This trade-off between accuracy and evaluation performance is one of the most important factors to be considered when choosing or designing models for certain purposes.

In principle, query optimization does not require very accurate cost models that estimate the execution cost to the microsecond. The major requirement here is, that the costs estimated by the cost model generate the same order on the QEPs as the actual execution costs do. The "cheapest" of a set of candidate QEPs is still the very same QEP, even if the cost model is off by an order of magnitude for all plans (provided it is off in the same direction for all plans). We use the term *adequate* for such cost models that preserve the order among the QEPs but are not necessarily accurate. As the query optimization problem is in general NP-hard [IK84, CM95, SM97], finding *the best* plan is practically not possible in reasonable time. Hence, optimization strategies usually (try to) find *a reasonably good* plan.[3] This in turn means that is it not necessary to preserve the proper order of plans whose costs are "very similar"; actually, it is often not even necessary to distinguish plans with very similar costs at all.

Many traditional disk-based DBMSs make use of these properties. In a disk-based DBMS, access to secondary storage is the dominating cost factor. To keep cost models simple, they often neglect other cost factors and estimate only I/O costs. Sometimes, I/O costs are not even given in time needed to perform the required I/O operations, but simply the number of necessary I/O operations is estimated.

However, there are situations where these simplifications do not apply as more accurate costs are required. For instance, two plans may differ in a way that one requires more I/O operations while the other one requires more computation time. To compare such plans, we need to express their total costs (i.e., I/O + CPU) in a common unit (i.e., time). Also the objective function or the stopping criterion for the optimization process might require more accurate costs in terms of processing time: "Stop optimization when the cheapest plan found so far takes less than 1 second", or "Stop optimization when the total time spent on optimization has reached a certain fraction of the execution time of the cheapest plan found so far".

In this thesis, we consider the following three cost components. A more elaborate description is given in Chapter 2.

Logical Costs consider only the data distributions and the semantics of relational algebra operations to estimate intermediate result sizes of a given (logical) query plan.

Algorithmic Costs extend logical costs by taking also the computational complexity (expressed in terms of O-classes) of the algorithms into account.

Physical Costs finally combine algorithmic costs with system/hardware specific parameters to predict the total costs in terms of execution time.

Next to query optimization, cost models can serve another purpose. Especially algorithmic and physical cost models can help database developers to understand and/or predict the performance of existing algorithms on new hardware systems. Thus, they can improve the algorithms or even design new ones without having to run time and resource consuming experiments to evaluate their performance.

[3] A further discussion of optimization strategies is beyond the scope of this thesis.

1.3 Increasing Importance of Memory Access Costs

Looking at the hardware development over, the last two decades, we recognized two major trends. On the one hand, CPU speed keeps on following Moore's law [Moo65], i.e., it doubles about every 18 months. In other words, clock speeds increase by more than 50 percent per year, and there are no indications that this trend might significantly change in the foreseeable future. Concerning main-memory, the picture looks differently. While main-memory sizes and main-memory bandwidth almost keep up with the CPU development, main-memory access latency is increasingly staying behind, improving only about 1 percent per year. New DRAM standards like Rambus and SLDRAM continue concentrating on improving the bandwidth, but hardly manage to reduce the latency. Hence, the performance gap between CPU speed and memory latency that has grown significantly over the last two decades is expected to widen even more in the near future.

To bridge the gap, hardware vendors have introduced small but fast cache memories — consisting of fast but expensive SRAM chips — between the CPU and main-memory. Nowadays, cache memories are often organized in two or three cascading levels, with both their size and latency growing with the distance from the CPU. Cache memories can reduce the memory latency, only if the requested data is found in one of the cache levels. This mainly depends on the application's memory access pattern. Hence, it becomes the responsibility of software developers to design and implement algorithms that make optimal use of the very cache/memory architecture the respective application runs on.

Standard main-memory database technology has mainly ignored this hardware development. The design of algorithms and especially cost models is still based on the assumptions that did hold in the early 80's. Without I/O as dominant cost factor, costs are commonly reduced to CPU processing costs. Memory access, if not considered negligible compared to CPU costs, is assumed to be uniform.

With memory sizes in commodity hardware getting larger at a rapid rate, also in disk-based DBMSs many database processing tasks can more and more take place in main memory.[4] As the persistent data remains located on disk farms, initial data access still requires disk I/O. But once the query operands are identified and streamed into memory, large intermediate results, temporary data structures, and search accelerators will fit into main memory, significantly reducing the number of I/O operations. With powerful RAID systems and high-capacity I/O buses reducing the I/O bandwidth at a rate that roughly matches the improvements in CPU speed, even in disk-based DBMSs memory access is expected to become a cost factor that can no longer be ignored.

1.4 Research Objectives

Concerning cost modeling, both data volume estimations and complexity measures are independent of the underlying system and hardware architecture. However, the per-

[4] "... the typical computing engine may have one terabyte of main memory. "Hot" tables and most indexes will be main-memory resident." [BBC+98].

formance experienced in terms of time costs is heavily dependent on these parameters. In this thesis, we devote ourselves to the latter, also referred to as physical costs.

The research in this thesis is driven by three major questions:

Understanding: What is the impact of the increasing gap between CPU and memory access costs on main-memory database performance?

Modeling: Is it possible to predict memory access cost accurately, and if so, how should the respective physical cost functions be designed?

Improving: What can we learn from analyzing and modeling main-memory database performance on hierarchical memory systems, and how can we use this knowledge to improve main-memory database technology?

In more detail, the research problems and objectives addressed in this document are formulated as follows:

Understanding Predicting physical query processing costs requires in-depth insight in how database software and modern hardware do interact. Hence, the first problem is to identify which hardware-specific parameters determine the processing costs in a main-memory database, and therefore need to be reflected in the cost models. It is desirable to analyze various hardware platforms to identify the commonalities and differences of the performance characteristics.

Modeling Another open issue is how to acquire new cost models. Traditionally, physical cost functions highly depend on various parameters that are specific to the very DBMS's software (e.g., the algorithms used and the way they are implemented) and to the hardware the database systems is running on (such as disk access latencies, I/O bandwidth, memory access speed, CPU speed, etc.). Thus, physical cost functions are usually created "by hand" for each algorithm, each DBMS, and each platform individually. This approach is not only time-consuming, but also tends to be error-prone. Hence, the question is whether — and if so, how — the process of designing physical cost models can be simplified and/or automated. Moreover, the generic use of database software on a range of platforms requires an analysis of their portability. The problem is, that we need to find a proper set of parameters describing the hardware-specific features and to design a cost model that can use these parameters.

Improving The main task of performance modeling is to understand and describe the behavior of a given system consisting of certain hardware and software components. However, while analyzing the details, one often is confronted with a priori unknown bottlenecks that might have to be minimized or even eliminated to improve the performance. The final question addressed in this thesis is whether and how we can use both the cost models we created and the knowledge gained while developing them, to improve main-memory database technology.

1.5 Thesis Outline

The research objectives mentioned above are explored in detail in the remaining chapters.

In Chapter 2, we dedicate ourselves to preliminaries, briefly reviewing the role of performance models in database literature. We mainly focus on the different types of cost models, the various purposes they serve, and proposed techniques how to acquire cost models for a given system and purpose. Moreover, we give a concise overview of our main-memory DBMS prototype Monet, which we use as implementation and validation platform throughout this thesis.

In order to create database cost models, we need to know which parameters are relevant for the performance behavior of database algorithms. In Chapter 3, we first discuss the characteristics of state-of-the-art CPU, cache-, and main-memory architectures. Our research covers various hardware platforms, ranging form standard PC's over workstations to high-performance servers. For convenience, we introduce a unified hardware model, that gathers the performance-relevant characteristics of hierarchical memory systems — including CPU caches, main-memory, and disk systems — in a single framework. The unified hardware model provides the necessary abstraction to treat various hardware platforms equally on a qualitative level. Equipped with these technical details, we analyze the impact of various hardware characteristics on the performance of database algorithms. In particular, we show that on modern hierarchical memory systems (consisting of the main memory and one or more levels of caches) memory access must not be seen as "for free", not even as uniform, concerning costs. Hence, traditional cost models focusing on I/O and CPU costs are not suitable any more. To adequately predict database performance in the new scenario, where main memory access is (partly) replacing the formerly cost-dominating disk access, new cost models are required that respect the performance impact of hierarchical memory systems. The analysis results in a calibration tool that automatically derives the relevant hardware characteristics, such as cache sizes, cache miss latencies, and memory access bandwidths, from any hardware platform. The ability to quantify these hardware features lays the foundation for hardware independent database cost models.

Equipped with the necessary tools, we are ready to design hardware independent physical database cost models in Chapter 4. Focusing on data access costs, we develop a cost model that achieves hardware independence by using the hardware characteristics as parameters. The unified hardware model permits the creation of parameterized cost functions. Porting these functions to other systems or hardware platforms can then simply be done by filling in the new specific parameters as derived by our calibration tool. Moreover, we propose a generic approach that simplifies the task of creating cost functions for a plethora of database operations. For this purpose, we introduce the concept of data access patterns as a method to describe the data access behavior of database algorithms in an abstract manner. Based on this concept, we propose a novel generic technique to design cost models.

In Chapter 5, we demonstrate how to use the knowledge gained during our work on database performance modeling to design algorithms that efficiently exploit the performance potentials of contemporary hardware architectures.[5] Focusing on join algorithms in a main-memory scenario and pursuing our line of generic and portable solutions, we propose new cache-conscious algorithms that automatically adapt to new hardware platforms. In this context, our cost models serve a triple purpose. First, they prove valuable to model and hence understand the performance behavior of different algorithms in various hardware environments. Second, they enable us to design algorithms that can be tuned to achieve the best performance on various hardware platform. Tuning is done automatically at runtime, using the cost models and the parameters as measured by our calibration tool. And third, of course, our cost functions serve as input for cost-based query optimization.

The thesis is concluded in Chapter 6, which summarizes the contributions and discusses future research directions.

Much of the material presented in this thesis has been published in preliminary and condensed form in the following papers:

- P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, Edinburgh, Scotland, UK, September 1999.

  The paper analyzes the impact of modern hardware trends on database query performance. Exhaustive experiments on an SGI Origin2000 demonstrate that main-memory access forms a significant bottleneck with traditional database technology. Detailed analytical performance models are introduced to describe the memory access costs of some join algorithms. The insights gained are translated into guidelines for future database architecture, in terms of both data structures and algorithms. We discuss how vertically fragmented data structures optimize cache performance on sequential data access. Further, we present new radix algorithms for partitioned nested-loop- and hash-join. Detailed experiments confirm that these hardware-conscious algorithms improve the join performance by restricting random data access to the smallest cache size.

- S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000.

  This extended version of the previous paper has been re-published in the ”Best-of-VLDB 1999”collection. The paper provides a more detailed analysis of main-memory access cost on core database algorithms. Analytical models are presented that also cover the effects that occur due to CPU work and memory access overlapping each other. Moreover, we present a revised version of our partitioned hash-join algorithm. We found out that using perfect hashing instead of aiming at an average hash-bucket size of 4 tuples, improved the performance significantly by reducing the number of cache misses that occur while following the collision list. With this improvement, partitioned hash-join became superior to radix-join, which was faster in our initial experiments.

[5] Joint work with Peter Boncz; certain parts do overlap with parts of his Ph.D. thesis [Bon02].

- S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? — Dissecting CPU and Memory Optimization Effects. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 339–350, Cairo, Egypt, September 2000.

  In this paper, we show that CPU costs become distinctive, once memory access is optimized as proposed in our previous work. Exhaustive experimentation on various hardware platforms indicates that conventional database code is much too complex to be handled efficiently by modern high-performance CPUs. In turns out that especially function calls and branches make the code unpredictable for the CPU and thus hinder efficient use of the CPU internal parallel resources. We propose new coding techniques that enable better exploitation of the available resource. Experiments on various hardware platforms show that optimizing memory access and CPU resource utilization support each other, yielding a total performance improvement of up to an order of magnitude.

- S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, July 2002.

  This work discusses our work on analyzing, modeling, and improving memory access and CPU costs in a broader context. We provide refined cost models for our radix-based partitioned hash-join algorithms. Being parameterized by architecture-specific characteristics such as cache sizes and cache miss penalties, our models can be easily ported to various hardware platform. We introduce a calibration tool to automatically measure the respective hardware parameters.

- S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 191–202, Hong Kong, China, August 2002.

  In this paper, we present a generalized framework for our cost models. We provide a novel unified hardware model to describe performance relevant characteristics of hierarchical memory systems, hardware caches, main-memory, and secondary storage. Together with our calibration tool, this unified hardware model allows automatic porting of our cost models to various hardware platforms. To simplify the task of designing physical cost functions for various database algorithms, we introduce the concept of data access patterns. The data access behaviors of an algorithm is described in terms of simple combinations of basic patterns such as “sequential access” or “random access”. From this description, the detailed physical cost functions are derived automatically using the rules we developed in this work. The resulting cost functions estimate the number of accesses to each level of the memory hierarchy and score them with their respective latency. Respecting the features of both disk drives and modern DRAM-chips, we distinguish different latencies for sequential access and random access.

Chapter 2

Preliminaries

Models play a very important role in science and research. Usually, a model is seen as an abstract image or description of a certain part of “the real world” that helps us to analyze and/or better understand this part of the real world. As we saw in Chapter 1, database systems rely on cost models to do efficient and effective query optimization. In this chapter, we first discuss some fundamentals of cost models and informally introduce the terminology we use throughout this thesis. Then, we briefly review the role of database cost models in literature and discuss some approaches in more detail. The last part of this chapter gives a concise overview of our main-memory DBMS prototype Monet, which we use as implementation and validation platform throughout this thesis.

2.1 Cost Models

We indicated in Chapter 1 that different execution plans require different amounts of effort to be evaluated. The objective function for the query optimization problems assigns every execution plan a single non-negative value. This value is commonly referred to as *costs* in the query optimization business.

2.1.1 Cost Components

In the Introduction, we mentioned already briefly that we consider cost models to be made up of three components: logical costs, algorithmic costs, and physical costs. In the following, we discuss these components in more detail.

2.1.1.1 Logical Costs / Data Volume

The most important cost component is the amount of data that is to be processed. Per operator, we distinguish three data volumes: input (per operand), output, and temporary data. Data volumes are usually measured as cardinality, i.e., number of tuples. Often, other units such as number of I/O blocks, number of memory pages, or total size in bytes are required. Provided that the respective tuple sizes, page sizes, and block sizes are known, the cardinality can easily be transformed into the other units.

The amount of input data is given as follows: For the leaf nodes of the query graph, i.e., those operations that directly access base tables stored in the database, the input cardinality is given by the cardinality of the base table(s) accessed. For the remaining (inner) nodes of the query graph, the input cardinality is given by the output cardinality of the predecessor(s) in the query graph.

Estimating the output size of database operations — or more generally, their *selectivity* — is anything else but trivial. For this purpose, DBMSs usually maintain statistic about the data stored in the database. Typical statistics are

- cardinality of each table,
- number of distinct values per column,
- highest / lowest value per column (where applicable).

Logical cost functions use these statistics to estimate output sizes (respectively selectivities) of database operations. The simplest approach is to assume that attribute values are uniformly distributed over the attribute's domain. Obviously, this assumption virtually never holds for "real-life" data, and hence, estimations based on these assumption will never be accurate. This is especially severe, as the estimation errors compound exponentially throughout the query plan [IC91]. This shows, that more accurate (but compact) statistics on data distributions (of base tables as well as intermediate results) are required to estimate intermediate results sizes.

The importance of statistics management has led to a plethora of approximation techniques, for which [GM99] have coined the general term "*data synopses*". Such techniques range from advanced forms of *histograms* (most notably, *V-optimal histograms* including multidimensional variants) [Poo97, GMP97, JKM+98, IP99] over *spline synopses* [KW99, KW00], *sampling* [CMN99, HNSS96, GM98], and *parametric curve-fitting techniques* [SLRD93, CR94] all the way to highly sophisticated methods based on *kernel estimators* [BKS99] or *Wavelets* and other transforms [MVW98, VW99, LKC99, CGRS00].

A logical cost model is a prerequisite for the following two cost components. In this work, we do not analyze logical cost models in more detail, but we assume that a logical cost model is available.

2.1.1.2 Algorithmic Costs / Complexity

Logical costs only depend on the data and the query (i.e., the operators' semantics), but they do not consider the algorithms used to implement the operators' functionality. Algorithmic costs extend logical costs by taking the properties of the algorithms into account.

A first criterion is the algorithm's complexity in the classical sense of complexity theory. Most unary operator are in $O(n)$, like selections, or $O(n \log n)$, like sorting; n being the input cardinality. With proper support by access structures like indices or hash tables, the complexity of selection may drop to $O(\log n)$ or $O(1)$, respectively. Binary operators can be in $O(n)$, like a union of sets that does not eliminate duplicates, or, more often, in $O(n^2)$, as for instance join operators.

More detailed algorithmic cost functions are used to estimate, e.g., the number of I/O operations or the amount of main memory required. Though these functions require some so-called "physical" information like I/O block sizes or memory pages sizes, we still consider them algorithmic costs and not physical cost, as these informations are system specific, but not hardware specific. The standard database literature provides a large variety of cost formulas for the most frequently used operators and their algorithms. Usually, these formulas calculate the costs in term of I/O operations as this still is the most common objective function for query optimization in database systems. We refer the interested reader, e.g., to [KS91, EN94, AHV95, GMUW02].

2.1.1.3 Physical Costs / Execution Time

Logical and algorithmic costs alone are not sufficient to do query optimization. For example, consider two algorithms for the same operation, where the first algorithm requires slightly more I/O operations than the second, while the second requires significantly more CPU operations than the first one. Looking only at algorithmic costs, both algorithms are not comparable. Even assuming that I/O operations are more expensive than CPU operations cannot in general answer the question which algorithm is faster. The actual execution time of both algorithms depends on the speed of the underlying hardware. The physical cost model combines the algorithmic cost model with an abstract hardware description to derive the different cost factors in terms of time, and hence the total execution time. A hardware description usually consists of information such as CPU speed, I/O latency, I/O bandwidth, and network bandwidth. The next section discusses physical cost factors on more detail.

2.1.2 Cost Factors

In principle, physical costs are considered to occur in two flavors, *temporal* and *spatial*. Temporal costs cover all cost factors that can easily be related to execution time, e.g., by multiplying the number of certain events with there respective cost in terms of some time unit. Spatial costs contain resource consumptions that cannot directly (or not at all) be related to time. In the following, we briefly describe the most prominent cost factors of both categories.

2.1.2.1 Temporal Cost Factors

As indicated above, physical costs are highly related to hardware. Hence, it is only natural that we distinguish different temporal cost factors according to the respective hardware components involved.

Disk-I/O This is the cost of searching for, reading, and writing data blocks that reside on secondary storage, mainly on disk. In addition to accessing the database files themselves, temporary intermediate files that are too large to fit in main memory buffers and hence are stored on disk also need to be accessed. The cost of searching for records in a database file or a temporary file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. I/O costs are either simply measured in terms of the number of block-I/O operations, or in terms of the time required to perform these operations. In the latter case, the number of block-I/O operations is multiplied by the time it takes to perform a single block-I/O operation. The time to perform a single block-I/O operation is made up by an initial seek time (*I/O latency*) and the time to actually transfer the data block (i.e., block size divided by *I/O bandwidth*). Factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered across the disk affect the access cost. In the first case (also called *sequential I/O*), I/O latency has to be counted only for the first of a sequence of subsequent I/O operations. In the second case (*random I/O*), seek time has to be counted for each I/O operation, as the disk heads have to be repositioned each time.

Main-Memory Access These are the costs for reading data from or writing data to main memory. Such data may be intermediate results or any other temporary data produced/used while performing database operations.

Traditionally, memory access costs were ignored in database systems. The reason for this was, that they were completely overshadowed by the dominating I/O costs in disk-base systems. As opposed to I/O costs, memory access cost were considered uniform, i.e., independent of both the physical locality and the physical order of accesses. This assumption was mainly true on the hardware in the 80's. Hence, main-memory DBMSs considered memory access costs to be included in the CPU costs.

In this thesis, we demonstrate that due to recent hardware trends, memory access costs have become a highly significant cost factor. Furthermore, we show that memory access on modern hierarchical memory systems depicts similar cost-related characteristics as I/O, i.e., we need to consider both latency and bandwidth, and we need to distinguish between sequential and random access patterns.

Network Communication In centralized DBMSs, communication costs cover the costs of shipping the query from the client to the server and the query's result back to the client. In distributed, federated, and parallel DBMSs, communication costs additionally contain all costs for shipping (sub-)queries and/or (intermediate) results between the different hosts that are involved in evaluating the query.

Also with communication costs, we have a latency component, i.e., a delay to initiate a network connection and package transfer, and a bandwidth component, i.e., the amount of data that can be transfer through the network infrastructure per time.

CPU Processing This is the cost of performing operations such as computations on attribute values, evaluating predicates, searching and sorting tuples, and merging tuples for join. CPU costs are measured in either CPU cycles or time. When using CPU cycles, the time may be calculated by simply dividing the number of cycles by the CPU's clock speed. While allowing limited portability between CPUs of the same kind, but with different clock speeds, portability to different types of CPUs is usually not given. The reason is, that the same basic operations like adding two integers might require different amounts of CPU cycles on different types of CPUs.

Traditionally, CPU costs also cover the costs for accessing the respective data stored in main memory. However, we treat memory access costs separately.

Summarizing, we see that temporal cost are either caused by data access and/or data transfer (I/O, memory access, communication), or by data processing (CPU work).

2.1.2.2 Spatial Cost Factors

Usually, there is only one spatial cost factor considered in database literature: *memory size*. This cost it the amount of main memory required to store intermediate results or any other temporary data produced/used while performing database operations.

Next to not (directly) being related to execution time, there is another difference between temporal and spatial costs that stems from the way they share the respective resources. A simple example shall demonstrate the differences. Consider to operations or processes each of which consumes 50% of the available resources (i.e., CPU power, I/O-, memory-, and network bandwidth). Further, assume that when run one at a time, both tasks have equal execution time. Running both tasks concurrently on the same system (ideally) results in the same execution time, now consuming all the available resources. In case each individual process consumes 100% of the available resources, the concurrent execution time will be twice the individual execution time. In other words, if the combined resource consumption of concurrent tasks exceed 100%, the execution time extends to accomodate the excess resource requirements. With spatial cost factors, however, such "stretching" is not possible. In case two tasks together would require more than 100% of the available memory, they simply cannot be executed at the same time, but only after another.

2.1.3 Types of (Cost) Models

According to their degree of abstraction, (cost) models can be classified into two classes: *analytical models* and *simulation models*.

Analytical Models In some cases, the assumptions made about the real system can be translated into mathematical descriptions of the system under study. Hence, the result is a set of mathematical formulas. We call this an analytical model. The advantage of an analytical model is that evaluation is rather easy and hence fast. However, analytical models are usually not very detailed (and hence not very accurate). In order to translate them into a mathematical description, the assumptions made have to be rather general, yielding a rather high degree of abstraction.

Simulation Models Simulation models provide a very detailed and hence rather accurate description of the system. They describe the system in terms of (a) simulation experiment(s) (e.g., using event simulation). The high degree of accuracy is charged at the expense of evaluation performance. It usually takes relatively long to evaluate a simulation base model, i.e., to actually perform the simulation experiment(s). It is not uncommon, that the simulation actually takes longer than the execution in the real system would take.

Simulation models are usually used in scenarios where a very detailed analysis as close as possible to the real system is required, but the actual system in not (yet) available. The most prominent example is processor development. The design of new CPUs is evaluated via exhaustive simulation experiments, first, to ensure the correctness and analyze the (expected) performance. The reason is, that producing functional prototypes in an early stage of the development process would be to expensive.

In database query optimization, though it would appreciate the accuracy, simulation models are not feasible, as the evaluation effort is far to high. Query optimization requires that costs of numerous alternatives are evaluated and compared as fast as possible. Hence, only analytical cost models are applicable in this scenario.

2.1.4 Architecture and Evaluation of Database Cost Models

The architecture and evaluation mechanism of database cost models is tightly coupled to the structure of query execution plans. Due to the strong encapsulation offered by relational algebra operators, the cost of each operator, respectively each algorithm, can be described individually. For this purpose, each algorithm is assigned a set of *cost functions* that calculate the three cost components as described above. Obviously, the physical cost functions depend on the algorithmic cost functions, which in turn depend on the logical cost functions. Algebraic cost functions use the data volume estimations of the logical cost functions as input parameters. Physical cost functions are usually specializations of algorithmic cost functions that are parameterized by the hardware characteristics.

The cost model also defines how the single operator costs within a query have to be combined to calculate the total costs of the query. In traditional sequential DBMSs, the single operators are assumed to have no performance side-effects on each other. Thus, the cost of a QEP is the cumulative cost of the operators in the QEP [SAC+79]. Since every operator in the QEP is the root of a sub-plan, its cost includes the cost of its input operators. Hence, the cost of a QEP is the cost of the topmost operator in the QEP. Likewise, the cardinality of an operator is derived from the cardinalities of its inputs, and the cardinality of the topmost operator represents the cardinality of the query result.

In non-sequential (e.g., distributed or parallel) DBMSs, this subject is much more complicated, as more issues such as scheduling, concurrency, resource contention, and data dependencies have to considered. For instance, in such environments, more than one operator may be executed at a time, either on disjoint (hardware) resources, or (partly) sharing resources. In the first case, the total cost (in terms of time) is calculated as the maximum of the costs (execution times) of all operators running concurrently. In the second case, the operators compete for the same resources, and hence mutually influence their performance and costs. More sophisticated cost function and cost models are required here to adequately model this resource contention [LTS90, LST91, SE93, SYT93, LVZ93, ZZBS93, SHV96, SF96, GHK92].

2.2 Logical Cost Models / Estimation

Most DBMSs make certain assumptions on the underlying data in order to perform inexpensive estimations. Christodoulakis studied the implications of various common assumptions on the performance of databases [Chr83, Chr84]. The main set of assumptions studied by him are:

Uniformity of attribute values: All possible values of an attribute have the same frequency in the data distribution.

Attribute Independence: The data distributions of all attributes in a relation are independent of each other.

Uniformity of queries: Queries refer attribute values with equal frequencies.

Constant number of records per block: Each block of the file contains the same number of tuples.

Random placement: Each record of the file has the same probability to qualify in a query, regardless of its placement among the pages of secondary storage.

He also showed that the expected cost of a query estimated using these assumptions is an upper bound on the actual expected cost. He demonstrated that existing systems using these assumptions tend to utilize expensive query evaluation strategies and that non-uniformity, non-independence, and non-random placement could be exploited in database design in order to reduce the system cost. In addition to providing such extensive motivation for better estimation techniques, his work also pioneered in the usage of several mathematical techniques such as *Schur concavity* [MO79] in database performance evaluation.

The System-R optimizer, assumed that the underlying data is uniform and independent ([SAC$^+$79]). As a result, only the number of tuples and the lowest and highest values in each attribute are stored in the system catalogs, and it is assumed that all possible values between the two extremes occur with the same probability. Hence, very few resources are required to compute, maintain, and use these statistics. In practice, though, these assumptions rarely hold because most data tends to be non-uniform and has dependencies. Hence, the resulting estimates are often inaccurate. This was formally verified in the context of query result size estimation by Ioannidis and Christodoulakis in [IC91]. In their work they proved that the worst case errors incurred by the uniformity assumption propagate exponentially as the number of joins in the query increases. As a result, except for very small queries, errors may become extremely high, resulting in inaccurate estimates for result sizes and hence for the execution costs.

Several techniques have been proposed in the literature to estimate query result sizes, most of them contained in the extensive survey by Mannino, Chu, and Sager [MCS88]. The broad classes of various estimation techniques are described in the following sections.

2.2.1 Sampling-based Techniques

These techniques compute their estimates by collecting and processing random samples of the data, typically at query optimization time. There has been considerable amount of work done in sampling-based techniques for result size estimation [Ant92, ASW87, CMN99, GGMS96, HNSS96, HS92, HS95, LNS90, SN92, OR86, LS95]. Since these techniques do not rely on any precomputed information about the data, they are not affected by database updates and do not incur storage overheads. Another advantage of these techniques is their probabilistic guarantees on the accuracy of the estimates. Some of the undesirable properties of the sampling-based techniques are: (1) they incur disk I/Os and CPU overheads during query optimization, and (2) the information gathered is not preserved across queries and hence these techniques may incur the costs repetitively. When a quantity needs to be estimated once and with high accuracy in the presence of updates, the sampling technique works very well (e.g., by a query profiler). To overcome point (2), techniques for incremental maintenance of random samples have been developed in recent works [GMP97, GM98].

Another weak point of sampling is that the relations which are to be sampled have to be available. In other words, sampling can only be applied to base table or completely calculated intermediate results. Propagating samples through the operators of a complex query is generally not possible, especially with joins. These problems have been analyzed in detail in [CMN99, AGPR99, GGMS96].

2.2.2 Parametric Techniques

These techniques approximate the actual data distribution by a parameterized mathematical distribution, such as the uniform distribution [SAC$^+$79], multivariate normal distributions or Zipf distributions [Chr83]. The parameters for these distributions are obtained from the actual data distributions, and the accuracy of this approximation depends heavily on the similarity between the actual and parameterized distributions. The main advantage of this approach is the small storage overhead involved and the insignificant run-time costs. On the other hand, real data often does not resemble any simple mathematical distribution and hence such estimations may cause inaccuracies in estimates. Also, since the parameters are precomputed, this approach may incur additional errors if the database is updated significantly. Variants of this approach are the *algebraic* techniques, where the actual data distribution is approximated by a polynomial function. The coefficients of this function are determined using regression techniques [SLRD93]. A promising algebraic technique was proposed calling for adaptively approximating the distribution by a six-degree polynomial, whose coefficients are varied dynamically based on query feedback [CR94]. Some of the problems associated with the algebraic techniques are the difficulties in choosing the degree of the polynomial function and uniformly handling result size estimates for operators other than simple selection predicates. On the other hand, the positive results obtained in the work of Wei Sun et al. [SLRD93] on algebraic techniques indicates their potential applicability.

2.2.3 Probabilistic Counting Techniques

These techniques have been applied in the contexts of estimating the number of unique values in the result of projecting a relation over a subset of attributes ([GG82, FM85, SDNR96]). The technique for estimating the number of distinct values in a multiset, proposed by Flajolet and Martin [FM85] makes an estimate during a single pass through the data and uses a small amount of fixed storage. Shukla et al. applied this technique in estimating the size of multidimensional projections (the cube operator) [SDNR96]. Their experiments have shown that these techniques can provide more reliable and accurate estimates than the sampling-based techniques [SDNR96]. The applicability of these techniques to other operators is still an open issue.

2.2.4 Non-parametric (Histogram-based) Techniques

These techniques approximate the underlying data distribution using precomputed tabular information (histograms). They are probably the most common techniques used in practice (e.g., they are used in DB2, Informix, Ingres, Microsoft SQL-Server, Oracle, Sybase, Teradata). Since they are precomputed, they may incur errors in estimation if the database is updated and hence require regular re-computation.

Most of the work on histograms is in the context of single operations, primarily selections. Specifically, Piatetsky-Shapiro and Connell dealt with the effect of histograms on reducing the error for selection queries [PSC84]. They studied two classes of histograms: *equi-width* histograms and *equi-depth* (or *equi-height*) histogram [Koo80]. Their main result showed that equi-width histograms have a much higher worst-case and average errors for a variety of selection queries than equi-depth histograms. Muralikrishna and DeWitt [MD88] studied techniques for computing and using multi-dimensional equi-depth histograms. By building histograms on multiple attributes together, their techniques were able to capture dependencies between those attributes. Several other researchers have dealt with ”variable-width”histograms for selection queries, where the buckets are chosen based on various criteria [Koo80, KK85, MK88]. Kooi's thesis [Koo80] contains extensive information on using histograms inside an optimizer for general queries and the concept of variable-width histograms. The survey by Mannino, Chu, and Sager [MCS88] contains various references to work in the area of statistics on choosing the appropriate number of buckets in a histogram for sufficient error reduction. That work deals primarily with selections as well. Histograms for single-join queries have been minimally studied and then again without emphasis on optimality [Chr83, Koo80, MK88]. Probably the earliest efforts at studying optimality of histograms for result size estimation of join operators are those of Ioannidis and Christodoulakis [IC93, Ioa93]. They introduce two new classes of histograms, *serial* and *end-biased* histograms, and show that certain types of these classes, so called *V-optimal(F,F)* histograms, are optimal for worst-case errors of tree equality-join and selection queries. Practicality issues in computing the optimal histograms were not addressed in their work.

Some of the limitations of earlier work on histograms are as follows. First, they were mostly restricted to estimating the result sizes of a few operators such as selections and equi-joins. Second, barring the study by Ioannidis and Christodoulakis, most of the earlier work on histograms has been empirical in nature with almost no effort to identify optimal histograms. Finally, the computation techniques for the new classes of histograms proposed by [Ioa93] were too expensive to be of use in practice. Also, being restricted to predicates on single attributes, most of the earlier work on histograms did not deal with the effects of correlation between attributes from the same relation. Some of the work that did consider multiple attributes together [Ioa93] assumes that multi-dimensional histograms can be built in practice, but do not fully explore the practicality issues involved.

The work by Ioannidis and Poosala [IP95] marks probably the first effort to find a compromise between optimality and practicality of histograms. The work is continued and extended in [PIHS96] and [JKM$^+$98].

There is lots of on-going work on and around the use of histograms in database query evaluation. Some recent issues are new techniques how to construct and maintain histograms efficiently [GMP97, CMN98, MVW98, AC99], and the use of histograms for approximate query answering [PGI99].

2.3 I/O-based Cost Models

Physical cost functions belong to the core of proprietary code of a database vendor. Their design, accurate tuning, and alignment with all other database components requires high level of expertise and knowledge of both hardware and database components. Hence, the vendors keep their physical cost functions as precious secrets.

Early work on System-R [SAC$^+$79] uses a cost function balancing both factors I/O and CPU using a constant weight, a factor difficult to determine in practice. Moreover, given the discrepancy in I/O and CPU cost granularity, i.e., microseconds versus milliseconds, the former has become the prime factor in choosing a query execution plan.

A subsequent study on System-R* [ML86] identified that in addition to the physical and statistical properties of the input data streams and the computation of selectivity, modeling buffer utilization plays a key role in accurate estimation. This requires using different buffer pool hit ratios depending on the levels of indexes as well as adjusting buffer utilization by taking into account properties of join methods, e.g., a relatively pronounced locality of reference in an index scan for indexed nested loop join [GLSW93].

With I/O being the dominant cost factor, database research has developed various techniques to reduce the number of I/O operations needed to answer a query. Two of the most prominent of these techniques are in-memory buffers to cache frequently access pages of the database relations and indices to access a fraction of a table, e.g., as requested by a selection predicate, without the need to scan the whole table. While significantly improving database performance, these techniques make cost estimation more complicated, and it becomes more difficult to predict the exact number of I/O operations that will be needed to answer a query. A number of research works has been devoted to analyzing and predicting the impact of indices, buffer pools, and various buffer replacement strategies on the number of I/O operations, e.g., [CD85, SS86, Sac87, ML89, CS89, DYC95].

Numerous further I/O-based cost models appear in a database literature. However, in most cases, the cost models themselves a not the (primary) subject. Rather, they are just presented as necessary tools for query optimization. Though sharing some commonalities, most physical cost models are specific for the respective DBMS, its architecture, algorithms, data structures, and the hardware platform it runs on.

2.4 Main-Memory Cost Models

Relatively little work has been done on modeling of the performance cost of main-memory DBMSs (MM-DBMSs). Early work on the design of database machines provides hints on the interdependencies of algorithms and memory access [Ozk86, Su88, BF89], but this research track has long been abandoned. This can partly be attributed to a lack of need, as use of MM-DBMS techniques have since been restricted to areas like real-time database systems (e.g., telecom switching, financial trading) that required relatively simple queries; say a hash-lookup in a single table.

In recent database literature, mainly the work around two research prototypes, IBM's *office-by-example* (*OBE*) and HP's *Smallbase*, has dealt with the issue of query optimization and cost modeling in main-memory environments.

2.4.1 Office-By-Example (OBE)

Whang and Krishnamurthy [WK90] discuss query optimization techniques applied in IBM's research prototype office-by-example (OBE). OBE is a memory-resident domain relational calculus database system that extends the concepts of *query-by-example* (*QBE*). To enable cost-based query optimization, they present a complete cost model. Due to the assumption that the data being processed is resident in main memory, the traditional database cost factor, I/O access, becomes obsolete. Rather, CPU computation cost now becomes dominant. Modeling CPU costs, however, is very difficult as too many parameters, like the software design, the hardware architecture, and even programming styles, may affect the CPU computation costs. A detailed analysis of larger systems to count the CPU cycles is virtually impossible. The solution that Whang and Krishnamurthy propose is to use an approach using both experimental and analytical methods. First, they identify the system's *bottlenecks* using an execution analyzer / profiler. Only bottlenecks will be used to model the system's CPU cost.

To prevent the cost model from drifting frequently due to changes in the program, the bottlenecks are improved as much as possible with reasonable effort. The next step is then to find, by experiments, *relative weights* of different bottlenecks and to determine their *unit costs*. Finally, they develop comprehensive cost formulas based on these unit costs.

For OBE, Whang and Krishnamurthy identified the following bottlenecks and measured the respective unit costs:

1. evaluating the expressions involved in predicates (unit cost = C_1);
2. comparison operations needed to finally determine the outcome of predicates (unit cost = C_4);
3. retrieving a tuple from a (memory-resident) relation (unit cost = C_3);
4. unit operation in creating an index (creating an index on a relation of n tuples takes $n \log_2 n$ such unit operations; unit cost = C_4);
5. unit operation in the sorting needed to prepare a multi-column index (unit cost = C_5).

The most interesting result of their experiments was that evaluating expressions in predicates turned out to be by far the most expensive operation in OBE. While C_2 through C_5 are of approximately the same order, C_1 exceeds them by approximately an order of magnitude.

2.4.2 Smallbase

Listgarten and Neimat [LN96, LN97] classify main-memory cost models into three categories:

hardware-based A *hardware-based* cost model is constructed analogously to traditional I/O-based cost models. Instead of counting I/O operations, now CPU cycles are counted. While conceptually simple, this approach is difficult to implement. In addition to the problems mentioned in [WK90], Listgarten and Neimat point out that hardware policies like cache-replacement or pre-fetching need to be incorporated, which are hard to model. Further, portability would be very limited, as such policies vary between hardware architectures. However, once constructed, a hardware-based model is believed to be accurate and reliable.

application-based This second category matches the approach presented in [WK90]: costs are expressed in terms of a system's bottleneck costs. While being simpler to develop than hardware-based cost models, *application-based* cost models are less general. The bottlenecks found highly depend on the workload used to identify them, and hence may not sufficiently represent the costs of all types of queries. In principle, application-based models are easier to port than hardware-based models, by simply regenerating the profiles. However, this might not only result in different unit costs, but also in a different set of bottlenecks. In this case, the model itself changes, not just the instantiation, and hence the cost functions of all database operations need to be re-formulated in terms of the new set of bottlenecks.

engine-based The third type of cost models is a compromise between detailed but complex hardware-based models and simple but less general application-based models. An *engine-based* cost model is based on the costs of primitive operations provided by the (MM-)DBMS's execution engine.

Listgarten and Neimat propose a two-step process to construct engine-based cost models. First, the general model is created by identifying the base costs and expressing query processing costs in terms of these building blocks. Second, the model is instantiated by determining the relative values for the base costs, and then verified.

Step one requires detailed knowledge about the internals of the execution engine and is usually to be done by hand. In case of doubt about how detailed the model should be, they propose to make it as detailed as practically feasible. Simplifications or further refinements can be done during verification. For their Smallbase system, Listgarten and Neimat identified the following primitive costs:

- fetching a column or parameter value
- performing arithmetic operations
- performing boolean operations
- evaluating a comparison
- evaluating a like expression
- scanning a table, T-tree index, hash index, temporary table
- creating/destroying a T-tree index, hash index, temporary table
- sorting tuples
- selecting distinct tuples
- performing a join (nested loop join, merge join)

Dependencies of these costs on factors like tables size, data type, etc. are dealt with during the second step.

Step two is automated by developing a cost test program that instantiates and verifies the model. For each unit costs, two queries are provided whose execution costs differ in only that value (plus maybe further cost that are already known). Further, formulas that specify the dependency of each unit cost on the table size have to be specified. The cost test program than finds the respective parameters and verifies the model by running each pair of queries several times with varying table sizes and performing a least square regression on the difference in execution time of the pairing queries.

2.4.3 Remarks

The work described above, as well as other recent work on main-memory query optimization [LC86b, PKK+98], models the main-memory costs of query processing operators on the coarse level of procedure calls, using profiling to obtain some 'magical' constants. As such, these models do not provide insight in individual components that make up query costs, limiting their predictive value.

2.5 Cost Models for (Heterogeneous) Federated and Multi-Database Systems

From a cost modeling perspective, classical sequential, parallel, and distributed database management systems share the property that the whole system is developed by the same vendor. Hence, those developers in charge of the query optimizer have access to all specification details and the source code of the execution engine. Thus, they can exploit this detailed insight when designing cost models for query optimization. However, when it comes to global query optimization in heterogeneous federated and multi-database systems, the picture looks differently. Usually, the individual DBMSs gathered in such systems are off-the-shelf products made by one or more different vendors. This means that no detailed knowledge about these systems is available to the vendor of the federated/multi-DBMS. The participating DBMSs have to be treated as "black boxes", and thus, new techniques are required to acquire proper cost models for global query optimization. The following sections briefly present some approaches published in resent database literature.

2.5.1 Calibration

Pegasus Du, Krishnamurthy, and Shan [DKS92] propose a calibration-based approach to obtain cost models for the participating DBMSs. The costs of basic operators — such as sequential scan, index scan, index lookup, different join algorithms — are modeled as rather generic formulas. The cost of a sequential scan over a relation *R* evaluating a predicate *P*, e.g., is given as:

$$C_{seq\_scan}(R, P) = c_0 + c_1 \cdot ||R|| + c_2 \cdot ||R|| \cdot s(P)$$

with

$\|\|R\|\|$:	the number of tuples in relation R
$s(P)$:	the selectivity of predicate P
c_0:	the initialization cost for the select
c_1:	the cost to retrieve a single tuple and to evaluate P on it
c_2:	the cost to process a result tuple satisfying P

The authors assume, that statistical information about the data stored in the participating DBMSs as well as a (global) logical cost model to derive selectivity factors and intermediate result sizes are available.

The coefficients c_0, c_1, and c_2 are assumed to be functions depending on parameters such as data types, tuple sizes, number of clauses in a predicate, etc. (where appropriate). Further, costs are measured in elapsed time, and cover I/O as well as CPU costs.

In order to calibrate the respective coefficients for a given “black box” database system, the authors designed a special synthetic database and a set of queries whose run times are measured. The major problem that arises here, is that the whole calibration process has to be predictable. For instance, calibration does not make sense, if one does not know how the system will execute a given query (e.g., using which algorithm and which index, if any). Further, effects related to data placement, paginating, etc. have to be eliminated. The presented database and query set take care of these issues. Experiments with AllBase, DB2, and Informix show, that the proposed process derives quite accurate cost models in 80% of the cases.

IRO-DB Gardarin, Sha, and Tang [GST96] extend the calibration approach of Du, Krishnamurthy, and Shan [DKS92] for the object-oriented federated database system IRO-DB [GFF97]. The major extension required was to introduce a path traversal (“pointer-chasing”) operator and the respective cost formula. Further, cost parameters such as object size, collection size, projection size, and fan out needed to be regarded. Also, the calibration database and the query set are extended to meet the requirements of an object-oriented environment.

2.5.2 Sampling

Zhu and Larson [ZL94, ZL96, ZL98] propose a query sampling method. The key idea is as follows. It first groups local queries that can be performed on a local DBS in an MDBS into homogeneous classes, based on some information available at the global level in an MDBS such as the characteristics of queries, operand tables and the underlying local DBS. A sample of queries are then drawn from each query class and run against the user local database. The costs of sample queries are used to derive a cost model for each query class by multiple regression analysis. The cost model parameters are kept in the MDBS catalog and utilized during query optimization. To estimate the cost of a local query, the class to which the query belongs is first identified. The corresponding cost model is retrieved from the catalog and used to estimate the cost of the query. Based on the estimated local costs, the global query optimizer chooses a good execution plan for a global query.

2.5.3 Cost Vector Database

HERMES Adali, Candan, Papakonstantinou, and Subrahmanian [ACPS96] suggest to maintain a cost vector database to record cost information for every query issued to a local DBS. Cost estimation for a new query is based on the costs of similar queries. For each call to a local DBS, the cost vector registers the time to compute the first answer, the time to compute all the answer, the cardinality of the answer, and the type of predicates to which these values correspond to. Summary table are also generated off-line to avoid heavy burden on storage. To estimate the costs of a new sub-query, the sub-query is matched against the cost vector database and a kind of regression is applied. The approach is demonstrated as efficient for sources queried with similar sub-queries.

2.6 Main Memory Database Systems

During the mid-1980s falling DRAM prices seemed to suggest that future computers would have such huge main memories that most databases could entirely be stored in them. In such situations, it would be possible to eliminate all (expensive) I/O from DBMS processing. This seriously changes the architecture for a DBMS, as in a Main Memory DBMS (MMDBMS) there is no central role for I/O management.

An important question in a MMDBMS is how to do transactions and recovery in an efficient way. Some of the proposed algorithms [LC86b, Eic87], assume that a (small) stable subset of the main memory exists, a piece of memory whose content will not be lost in a power outage through a battery backup. These stable memories can be used to place, e.g., a redo log. Others do not assume stable memories, and still use I/O to write transaction information to stable storage. These algorithms hence do not eliminate I/O (e.g., “logical logging” [JSS93]), but minimize it, as the critical path in a MMDBMS transaction only needs to write the log; not data pages from the buffer manager.

The main asset of a MMDBMS is its unparalleled speed for querying and update. Information on design and implementation of basic database data structures and algorithms can be found in the overviews by Garcia-Molina and Salem [GMS92] and Eich [Eic89]. Some specific research has been done in index structures for main memory lookup [Ker89, LC86a, DKO$^+$84, AP92]. It turns out, that simple data structures like the binary AVL tree, called T-Tree, and simple bucket-chained hash outperform bread-and-butter disk-based structures like B-tree and linear hash, due to the fact that the only costs involved in index lookup and maintenance are CPU and memory access.

A specific problem in MMDBMS is query optimization. The lack of I/O as dominant cost factor means that it is much more difficult in a MMDBMS to model query costs, as they depend on fuzzy factors like CPU execution cost of a routine. Therefore, DBMS query optimization tends to make use of simple cost models that contain “hard” constants obtained by profiling [LN96, WK90]. One challenge in this area is to model the interaction between coding style, hardware factors like CPU and memory architecture and query parameters into a reliable prediction of main memory execution cost.

The end of popularity of MMDBMS techniques came in the early 1990s, when it became clear that not only DRAM sizes had grown, but also disk size, and problem sizes. MMDBMS were thereafter only considered of specific interest to real-time database applications, like, e.g., encountered in embedded systems or telephone switches. Still, main memory sizes in commodity computers continue to increase, and for those application areas whose problem sizes do not grow as fast, it holds that at a certain time they will fit in main memory. Recently, prominent database researchers concluded in the Asilomar workshop [BBC$^+$98] that MMDBMSs have an important future in such application areas.

Well known main memory systems are Smallbase [BHK$^+$86, LN96] developed by HP, the object-oriented AMOS [FR97] system, the parallel MMDBMS PRISMA [AvdBF$^+$92], and Dalí [JLR$^+$94, RBP$^+$98] by Bell Labs. Smallbase and Dalí have been reshaped into commercial products, under the names Times Ten [Tea99] and DataBlitz [BBG$^+$99], respectively. Their main focus is highly efficient support of OLTP DBMS functionality on small or medium-size data sets. Also, all main relational vendors (IBM, Microsoft, Oracle) are offering small-footprint “ultra-light” versions of their DBMS servers for use in mobile computing devices and web PDAs.

2.7 Monet

Our research goals as specified in Section 1.4 require a “real” DBMS to conduct empirical analysis and experimental validation. In principal, there are three options:

1. use a commercial main-memory DBMS, e.g., *Times-Ten* [Tea99], or *DataBlitz* [BBG$^+$98, BBG$^+$99];

2. use a fully-fledged commercial disk-based DBMS, such as IBM’s DB2, Microsoft’s SQL-Server, or Oracle;

3. use a main-memory DBMS research prototype with accessible source code, e.g., our own *Monet* system.

For this work, we consider the third option to be the most practical. Mainly the fact that we know the internal architectural details and the source code helps us to understand how the complex interaction between database software, compilers, operation system, and hardware does work. We can easily play around with compiler switches and add profiling hooks to gain the necessary insight for our modeling plan. Moreover, it is only the ability to modify data structures, algorithms, and coding techniques that enables us to validate the new techniques we propose. In this section, we give a concise introduction to Monet, focusing on the core features that are important in the given context. For an complete description of Monet, the interested reader is referred to [Bon02].

2.7.1 Design

Monet is a main-memory database kernel developed at CWI since 1994 [BK95], and commercially deployed in a Data Mining tool [KSHK97]. It is targeted at achieving high performance on *query-intensive* workloads, such as created by *on-line analytical processing* (*OLAP*) or *data mining* applications. Monet uses the *Decomposed Storage Model* (*DSM*) [CK85], storing each column of a relational table in a separate binary table, called a *Binary Association Table* (*BAT*). A BAT is represented in memory as an array of fixed-size two-field records [OID,value], or *Binary UNits* (*BUN*). The OIDs in the left column are unique per original relational tuple, i.e., they link all BUNs that

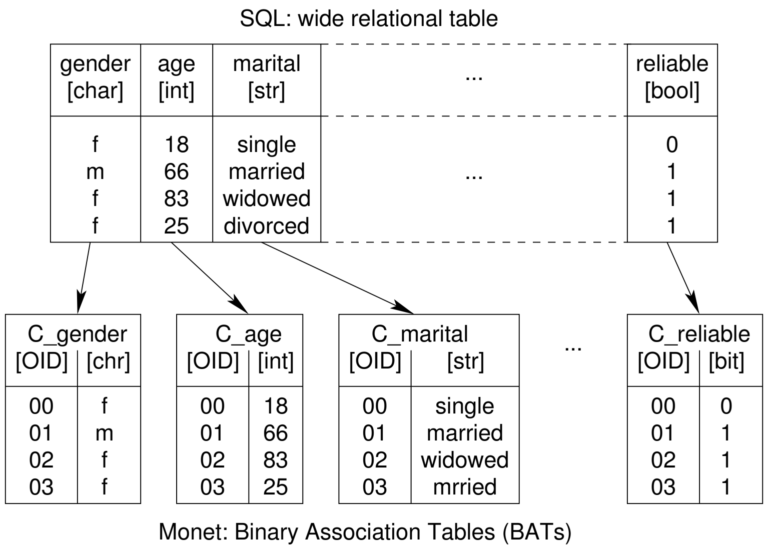


Figure 2.1: Vertically Decomposed Storage in BATs

make up an original relational tuple (cf., Figure 2.1). The major advantage of the DSM is that it minimizes I/O and memory access costs for column-wise data access, which occurs frequently in OLAP and data mining workloads [BRK98, BMK99, MBK02]. The BAT data structure is maintained as a dense memory array, without wasted space for unused slots, both in order to speed up data access (e.g., not having to check for free slots) and because all data in the array is used, which optimizes memory cache utilization on sequential access.

Most commercial relational DBMSs were designed in a time when OLTP was the dominant DBMS application, hence their storage structures, buffer management infrastructure, and core query processing algorithms remain optimized toward OLTP. In the architecture of Monet, we took great care that systems facilities that are only needed by OLTP queries do not slow down the performance of query-intensive applications. We shortly discuss two such facilities in more detail: buffer management and lock management.

Buffer management in Monet is done on the coarse level of a BAT (it is entirely loaded or not at all), hence the query operators always have direct access to the entire relation in memory. The first reason for this strategy is to eliminate buffer management as a source of overhead inside the query processing algorithms, which would result if each operator must continuously make calls to the buffer manager asking for more tuples, typically followed by copying of tuple data into the query operator. The second reason is that all-or-nothing I/O is much more efficient nowadays than random I/O (similarly to memory, I/O bandwidth follows Moore's law [Moo65], I/O latency does not).

In Monet, we chose to implement explicit transaction facilities, which provide the building blocks for ACID transaction systems, instead of implicitly building in transaction management into the buffer management. Monet applications use the explicit locking primitives to implement a transaction protocol. In OLAP and data mining, a simple transaction protocol with a very coarse level of locking is typically sufficient (a read/write lock on the database or table level). We can safely assume that all applications adhere to this, as Monet clients are front-end programs (e.g., an SQL interpreter, or a data mining tool) rather than end-users. The important distinction from other systems is hence that Monet separates lock management from its query services, eliminating all locking overhead inside the query operators.

As a result, a sequential scan over a BAT comes down to a very simple loop over a memory array with fixed-length records, which makes Monet's query operator implementations look very much like scientific programs doing matrix computations. Such code is highly suitable for optimization by aggressive compiler techniques, and does not suffer from interference with other parts of the system. In other words, Monet's algorithms are implemented directly on the physical storage structures, i.e., without any intermediate storage management layer. Thus it becomes feasible (1) to understand, how the algorithms, respectively their implementations, interact with the underlying hardware, and (2) to accurately model the algorithms' cost in detail.

2.7.2 Architecture and Implementation

Monet is designed as a MMDBMS kernel providing core database functionality and acting as a back-end for various front-end applications. The user-interfaces are provided via the front-end applications. Sample applications are among others an SQL front-end, an ODMG front-end, as well as OLAP and data mining tools. For communication between the front-end and the back-end, Monet provides an intermediate query language called *Monet Interpreter Language* (*MIL*) [BK99]. The core of MIL is made up by primitives that form a *BAT-algebra* similar to the relational algebra. Around this core, MIL has been developed as a computational complete procedural language giving full access to Monet's core functionality. Further characteristics of MIL are that (1) MIL programs are interpreted and (2) MIL operators are evaluated one at a time. The latter means that query evaluation in Monet follows a *bulk-processing* approach. As opposed to *pipelining*, this means that all intermediate results are fully materialized. Here, the Decomposed Storage Model offers another advantage, keeping intermediate results very "slim" and hence rather small. The crucial advantage of bulk-processing—next to its simple implementation—is the following. With the BAT-algebra primitives operating on binary tables only and a limited set of atomic data types, such as integer, float, character, string, etc., the number of type-combinations per operator is rather limited. We exploit this to overcome a disadvantage of the interpretation approach. Interpretation of algebra operators usually requires a type-switch in the innermost loops, as the actual type-binding is only known at runtime. Such type switches are typically realized by ADT-like function calls, and hence they are rather expensive. With the limited number of type combinations in MIL, we can provide a separate implementation of each operator for each case. To limit the coding effort, Monet is written in a macro language, from which C language implementations are generated. Per algorithm, only one template-implementation has to be implemented "by hand". At compile time, the macro language then expands the type-specific variants from these templates. The macros implement a variety of techniques, by virtue of which the inner loops of performance-critical algorithms like join are free of overheads like database ADT calls, data movement, and loop condition management. These techniques were either pioneered by our group (e.g., logarithmic code expansion [Ker89]) or taken from the field of high performance computing [Sil97]. Furthermore, Monet is implemented using aggressive coding techniques for optimizing CPU resource utilization [MBK00b] that go much beyond the usual MMDBMS implementation techniques [DKO+84]. In Chapter 5, we present some examples of these techniques.

2.7.3 Query Optimization

In Monet, we pursue a multi-level query optimization approach [MPK00]. Front-end applications can use *strategical optimization*, exploiting domain-specific knowledge to pre-optimize queries before sending them to the back-end. The MIL interpreter does some multi-query optimization in a *tactical optimization* phase. This phase removes common subexpressions of multiple queries, possibly sent by different front-ends, and takes care of introducing parallelism in case the Monet back-ends runs multi-threaded. Finally, the Monet kernel itself performs *operational optimization*. Operational optimization means that for each operator that is to be evaluated the kernel can choose the proper algorithm and its implementation just before executing it. Here, we exploit the fact that due to the bulk-processing approach, all operands are fully available before the operator has to be executed. Next to their sizes, MIL operators maintain some more properties of the BATs they generate, such as their data types and information about whether attribute values are sorted and/or unique. At runtime, the kernel examines the of the operands as well as the systems current state and applies heuristics to pick the most efficient algorithm and implementation for the given situation. For instance in case of a join, the kernel can currently choose between nested-loop join, hash-join, sort-merge join, and index-lookup join. In Chapter 5, we provide additional cache-conscious join algorithms and show how the cost models developed in Chapter 4 are used to find the most suitable algorithm.

Chapter 3

Cost Factors in MMDBMS: “No I/O” does not mean “Only CPU”

In this chapter, we describe those aspects of hardware technology found in custom computer systems that are most relevant for the performance of main-memory query execution. We identify ongoing trends, and outline their consequences for database architecture. In addition, we describe our *calibration tool* which extracts the most important hardware characteristics like cache size, cache line size, and cache latency from any computer system, and provide results for our benchmark platforms (modern SGI, Sun, Intel, and AMD hardware).

3.1 Commodity Computer Architecture

Focusing on main-memory processing, we discuss the major technical issues of CPUs, main-memory, and hardware caches which are relevant for database performance.

3.1.1 CPUs

Concerning CPUs, two aspects are of primary interest for this thesis. First of all, we need to have a closer look at how modern CPUs are designed and how they do work. Secondly, we briefly introduce hardware counters that help us to monitor certain events within the CPU and thus understand their impact on the performance of programs.

3.1.1.1 Design & Working Methods

While CPU clock frequency has been following Moore’s law (doubling every 18 months [Moo65]), CPUs have additionally become faster through parallelism *within* the processor. Scalar CPUs separate different execution stages for instructions, e.g.,

year	computer model	processor			memory	
		type	MHz	#par. units	STREAM/copy (bandwidth)	latency (ns)
1989	Sun 3/60	68020	20	1	6.5	
1990	Sun 3/80	68030	20	1	4.9	
1991	Sun 4/280	Sparc	17	1	9.6	160
1992	Sun ss10/31	superSparc I	33	3	42.9	
1993	Sun ss10/41	superSparc I	40	3	48.0	
1994	Sun ss20/71	superSparc II	75	3	62.5	870
1995	Sun Ultra1 170	ultraSparc I	167	5	225.2	225
1996	Sun Ultra2 2200	ultraSparc II	200	5	228.5	225
1996	SGI Power Chall.	R10000	195	5	172.7	610
1997	SGI Origin 2000	R10000	250	5	332.0	424
1998	SGI Origin 2000	R12000	300	5	336.0	404
1992	Intel PC	80486	66	1	33.3	
1993	Intel PC	Pentium	60	2	47.1	161
1994	Intel PC	Pentium	90	2	46.4	161
1995	Intel PC	Pentium	100	2	85.1	161
1996	Intel PC	Pentium	133	2	84.4	161
1996	Intel PC	PentiumPro	200	5	140.0	203
1997	Intel PC	PentiumII	300	5	188.2	145
1998	Intel PC	PentiumII	350	5	279.3	145
1998	Intel PC	PentiumII	400	5	304.0	145
1999	Intel PC	PentiumIII	600	5	379.2	135
2000	Intel PC	PentiumIII	733	5	441.9	135
1999	AMD PC	Athlon	500	9	373.5	217
2000	AMD PC	Athlon	800	9	387.9	217

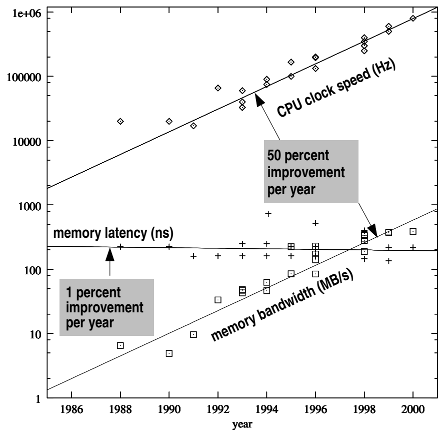


Figure 3.1: Trends in DRAM and CPU speed

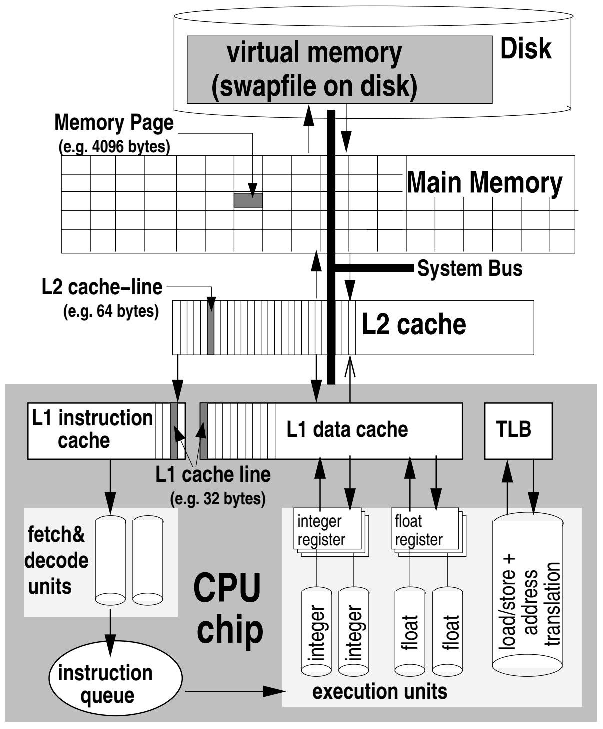


Figure 3.2: Modern CPU and Hierarchical Memory Architecture

allowing a computation stage of one instruction to be overlapped with the decoding stage of the next instruction. Such a *pipelined* design allows for inter-stage parallelism. Modern *super-scalar* CPUs add intra-stage parallelism, as they have multiple copies of certain (pipelined) units that can be active simultaneously. Although CPUs are commonly classified as either RISC (reduced instruction set computer) or CISC (complex instruction set computing), modern CPUs combine successful features of both. Figure 3.2 shows a simplified schema that characterizes how modern CPUs work: instructions that need to be executed are loaded from memory by a fetch-and-decode unit. In order to speed up this process, multiple fetch-and-decode units may be present (e.g., Intel's PentiumIII and AMD's Athlon have three, the MIPS R10000 has two). Decoded instructions are placed in an instruction queue, from which they are executed by one of various functional units, which are sometimes specialized in integer-, floating-point, and load/store pipelines. The PentiumIII, for instance, has two such functional units, the R10000 has five, and the Athlon has even nine. To exploit this parallel potential, modern CPUs rely on techniques like *branch prediction* to predict which instruction will be next before the previous has finished. Also, the modern cache memories are *non-blocking*, which means that a cache miss does not stall the CPU. Such a design allows the pipelines to be filled with multiple instructions that will probably have to be executed (a.k.a. *speculative execution*), betting on yet unknown outcomes of previous instructions. All this goes accompanied by the necessary logic to restore order in case of mispredicted branches. As this can cost a significant penalty, and as it is very important to fill all pipelines to obtain the performance potential of the CPU, much attention is paid in hardware design to efficient branch prediction. CPUs work with *prediction tables* that record statistics about branches taken in the past.

3.1.1.2 Hardware Counters

Detailed insight into the behavior of CPUs while processing application code is a prerequisite to understand, and eventually model, the performance of application programs. To aid this process, many modern CPUs provide so-called *hardware event counters* or *performance counters* that allow to monitor certain performance-related events that occur within the CPU while processing user code. Examples are the MIPS R10k/R12k series, Sun's UltraSPARC family, all Intel Pentium and Itanium CPUs, AMD's Athlons, DEC's Alphas, IBM's and Motorola's PowerPC's. Usually, each counter can only monitor one event at a time, however with multiple counters present, several events can be monitored concurrently. The number of counters per CPU varies from 2 (e.g., MIPS R10k) to 8 (e.g., Intel Pentium 4).

Like the number of counters, also the number and kind of events that can be monitored vary significantly between the different CPUs. We omit the details here and refer the interested reader to the respective product manuals. Typically, the set of events includes events like cache misses (both instruction and data), instructions decoded and executed, branches executed, branch mispredictions, etc.. We provide more information as required later when we use these features.

In contrary to software profiling as offered by certain compilers and/or profiling tools, using the hardware event counters has no impact on the execution performance.

Low-level access to the counters typically works via direct register access to select the events, start and stop monitoring, and finally read-out the results. More convenient high-level tools do exist but vary between hardware vendors and operating systems.

3.1.2 Main-Memory- & Cache-Systems

We now turn our attention to memory- and cache architectures. We explain the basic technical principles, discuss various aspects of memory access costs, and finally introduce a unified hardware model to be used in the remainder of this thesis.

3.1.2.1 Memory- & Cache-Architectures

Modern computer architectures have a *hierarchical memory system* as depicted in Figure 3.2. The main memory on the system board consists of *DRAM* chips (*Dynamic Random Access Memory*). While CPU speeds are increasing rapidly, DRAM access latency has hardly progressed through time. To narrow the exponentially growing performance gap between CPU speed and memory latency (cf., Figure 3.1), *cache memories* have been introduced, consisting of fast but expensive *SRAM* chips (*Static Random Access Memory*). SRAM cells are usually made-up from six transistors per memory bit, and hence, they consume a rather large area on the chips. DRAM cells require just a single transistor and a small capacitor to store a single bit. Thus, DRAMs can store much more data than SRAMs of equal (physical) size. But due to some leak current, the capacitor in DRAMs get discharged over time, and have to be recharged (*refreshed*) periodically to keep their information. These refreshes slowdown access.

The fundamental principle of all cache architectures is "*reference locality*", i.e., the assumption that at any time the CPU, respectively the program, repeatedly accesses only a limited amount of data (i.e., memory) that fits in the cache. Only the first access is "slow", as the data has to be loaded from main memory. We call this a *compulsory cache miss* (see below). Subsequent accesses (to the same data or memory addresses) are then "fast" as the data is then available in the cache. We call this a *cache hit*. The fraction of memory accesses that can be fulfilled from the cache is called *cache hit rate*; analogously, the fraction of memory accesses that cannot be fulfilled from the cache is called *cache miss rate*.

Cache memories are often organized in *multiple cascading levels* between the main memory and the CPU. They become faster, but smaller, the closer they are to the CPU. Originally, there was one level of typically 64 KB to 512 KB cache memory located on the system board. As the chip manufacturing processes improved, a small cache of about 4 KB to 16 KB got integrated on the CPU's die itself, allowing much faster access. The on-board is typically not replaced by the on-chip cache, but rather both make up a cache hierarchy, with the one on chip called *first level* (*L1*) cache and the one on board called *second level* (*L2*) cache. Recently, also the L2 cache has been integrated on the CPU's die (e.g., with Intel's Pentium III "Coppermine", or AMD's Athlon "Thunderbird"). On PC systems, the on-board cache has since disappeared, keeping two cache levels. On other platforms, e.g., workstations based on Compaq's (formerly DEC's) Alpha CPU, the on-board cache is kept as *third level* (*L3*) cache, next to the two levels on the die.

To keep presentations from becoming to complicated, we assume a typical system with two cache levels (L1 & L2) in most examples in the remainder of this work. However, our observations and results can easily be generalized to an arbitrary number of cascading cache levels in a straightforward way.

In practice, caches memories do not only cache the data used by an application, but also the program itself, more accurately, the instructions that are currently being executed. With respect to caching, there is one major difference between data and program. Usually, a program must not be modified while it is running, i.e., the caches may be read-only. Data, however, requires caches that also allow modification of the cached data. Therefore, almost all systems nowadays implement two separate L1 caches, a read-only one for instructions and a read-write one for data. The L2 cache, however, is usually a single "unified" read-write cache used for both instructions and data. Later in this thesis, we will see that instruction cache misses do not play a significant role in our scenario. Hence, we will not discuss instruction caches in more detail. Only where necessary, we will address them explicitly. Unless mentioned differently, we will refer to data caches simply as caches.

Caches are characterized by three major parameters: Capacity (C), Line Size (Z), and Associativity (A):

Capacity (*C*) A cache's capacity defines its total size in bytes. Typical cache sizes range from 8 KB to 8 MB.

Line Size (*Z*) Caches are organized in *cache lines*, which represent the smallest unit of transfer between adjacent cache levels. Whenever a cache miss occurs, a complete cache line (i.e., multiple consecutive words) is loaded from the next cache level or from main memory, transferring all bits in the cache line in parallel over a wide bus. This exploits spatial locality, increasing the chances of cache hits for future references to data that is ”closed to”the reference that caused a cache miss. Typical cache line sizes range from 16 bytes to 128 bytes.

Dividing the cache capacity by the cache line size, we get the *number of available cache lines* in the cache: $\# = C/Z$. Cache lines are often also called *cache blocks*. We use both terms as synonyms throughout this document.

Associativity (*A*) To which cache line the memory is loaded, depends on the memory address and on the cache's *associativity*. An *A-way set associative* cache allows to load a line in A different positions. If $A > 1$, some *cache replacement* policy chooses one from the A candidates. *Least Recently Used* (*LRU*) is the most common replacement algorithm. In case $A = 1$, we call the cache *direct-mapped*. This organization causes the least (virtually no) overhead in determining the cache line candidate. However, it also offers the least flexibility and may cause a lot of *conflict misses* (see below). The other extreme case are *fully associative* caches. Here, each memory address can be loaded to any line in the cache ($A = \#$). This avoids conflict misses, and only *capacity misses* (see below) occur as the cache capacity gets exceeded. However, determining the cache line candidate in this strategy causes a relatively high overhead that increases with the cache size. Hence, it is feasible only for smaller caches. Current PCs and workstations typically implement 2-way to 8-way set associative caches.

With multiple cache levels, we further distinguish two types: inclusive and exclusive caches. With *inclusive caches*, all data stored in L1 is also stored in L2. As data is loaded from memory, it gets stored in all cache levels. Whenever a cache line needs to be replaced in L1 (because a mapping conflict occurs or as the capacity is exceeded), its original content can simply be discarded as another copy of that data still remains in the (usually larger) L2. The new content is then loaded from where it is found (either L2 or main memory). The total capacity of an inclusive cache hierarchy is hence determined by the largest level. With *exclusive caches*, all cached data is stored in exactly one cache level. As data is loaded from memory, it gets stored only in the L1 cache. When a cache lines needs to be replaced in L1, its original content is first written back to L2. If the new content is then found in L2, it is moved from L2 to L1, otherwise, it is copied from main memory to L1. Compared to inclusive cache hierarchies, exclusive cache hierarchies virtually extend the cache size, as the total capacity becomes the sum of all levels. However, the “swap” of cache lines between adjacent cache levels in case of a cache miss also causes more “traffic” on the bus and hence increases the cache miss latency. We will analyze this in more detail in Section 3.3.

Cache misses can be classified into the following disjoint types [HS89]:

Compulsory The very first reference to a cache line always causes a cache miss, which is hence classified as a compulsory miss. The number of compulsory misses obviously depends only on the data volume and the cache line size.

Capacity A reference that misses in a fully associative cache is classified as a capacity miss because the finite sized cache is unable to hold all the referenced data. Capacity misses can be minimized by increasing the temporal and spatial locality of references in the algorithm. Increasing cache size also reduces the capacity misses because it captures more locality.

Conflict A reference that hits in a fully associative cache but misses in an A-way set associative cache is classified as a conflict miss. This is because even though the cache was large enough to hold all the recently accessed data, its associativity constraints force some of the required data out of the cache prematurely. For instance, alternately accessing just two memory addresses that “happen to be” mapped to the same cache line will cause a conflict cache miss with each access. Conflict misses are the hardest to remove because they occur due to address conflicts in the data structure layout and are specific to a cache size and associativity. Data structures would, in general, have to be remapped so as to minimize conflicting addresses. Increasing the associativity of a cache will decrease the conflict misses.

3.1.2.2 Memory Access Costs

We identify the following three aspects that determine memory access costs. For simplicity of presentation, we assume 2 cache levels in this section. Generalization to an arbitrary number of caches is straight forward.

Latency Latency is the time span that passes after issuing a data access until the requested data is available in the CPU. In hierarchical memory systems, the latency increases with the distance from the CPU. Accessing data that is already available in the L1 cache causes *L1 access latency* (λ_{L1}), which is typically rather small (1 or 2 CPU cycles). In case the requested data in not found in L1, an *L1 miss* occurs, additionally delaying the data access by *L2 access latency* (λ_{L2}) for accessing the L2 cache. Analogously, if the data is not yet available in L2, an *L2 miss* occurs, further delaying the access by *memory access latency* (λ_{Mem}) to finally load the data from main memory. Hence, the total latency to access data that is in neither cache is $\lambda_{\mathrm{Mem}} + \lambda_{\mathrm{L2}} + \lambda_{\mathrm{L1}}$. As L1 accesses cannot be avoided, we assume in the remainder of this thesis, that L1 access latency is included in the pure CPU costs, and regard only memory access latency and L2 access latency as explicit memory access costs. As mentioned above, all current hardware actually transfers multiple consecutive words, i.e., a complete cache line, during this time.

When a CPU requests data from a certain memory address, modern DRAM chips supply not only the requested data, but also the data from subsequent addresses. The data is then available without additional address request. This feature is called *Extended Data Output* (EDO). Anticipating sequential memory access, EDO reduces the effective latency. Hence, we actually need to distinguish two types of latency for memory access. *Sequential access latency* (λ^{s}) occurs with sequential memory access, exploiting the EDO feature. With random memory access, EDO does not speed up memory access. Thus, *random access latency* (λ^{r}) is usually higher than sequential access latency.

Bandwidth Bandwidth is a metric for the data volume (in megabytes) that can be transfered between CPU and main memory per second. Bandwidth usually decreases with the distance from the CPU, i.e., between L1 and L2 more data can be transfered per time than between L2 and main memory. We refer to the different bandwidths as *L2 access bandwidth* (β_{L2}) and *memory access bandwidth* (β_{Mem}), respectively. In conventional hardware, the memory bandwidth used to be simply the cache line size divided by the memory latency. Modern multiprocessor systems typically provide excess bandwidth capacity $\beta' \geq \beta$. To exploit this, caches need to be *non-blocking*, i.e., they need to allow more than one outstanding memory load at a time, and the CPU has to be able to issue subsequent load requests while waiting for the first one(s) to be resolved. Further, the access pattern needs to be sequential, in order to exploit the EDO feature as described above.

Indicating its dependency on sequential access, we refer to the excess bandwidth as *sequential access bandwidth* ($\beta^{\mathrm{s}} = \beta'$). We define the respective *sequential access latency* as $\lambda^{\mathrm{s}} = Z/\beta^{\mathrm{s}}$. For *random access latency* as described above, we define the respective *random access bandwidth* as $\beta^{r} = Z/\lambda^{r}$. For better readability, we will simply use plain λ and β (i.e., without ^{s} respectively ^{r}) whenever we refer to both sequential and random access without explicitly distinguishing between them.

On some architectures, there is a difference between read and write bandwidth, but this difference tends to be small. Therefore, we do not distinguish between read and write bandwidth in this article.

Address Translation For data access, logical virtual memory addresses used by application code have to be translated to physical page addresses in the main memory of the computer. In modern CPUs, a *Translation Lookaside Buffer* (*TLB*) is used as a cache for physical page addresses, holding the translation for the most recently used pages (typically 64). If a logical address is found in the TLB, the translation has no additional costs. Otherwise, a *TLB miss* occurs. The more pages an application uses (which also depends on the often configurable size of the memory pages), the higher the probability of TLB misses.

The actual *TLB miss latency* (l_{TLB}) depends on whether a system handles a TLB miss in hardware or in software. With software-handled TLB, TLB miss latency can be up to an order of magnitude larger than with hardware-handled TLB. Hardware-handled TLB fetches the translation from a fixed memory structure, which is just filled by the operating system. Software-handled TLB leaves the translation method entirely to the operating system, but requires trapping to a routine in the operating system kernel on each TLB miss. Depending on the implementation and hardware architecture, TLB misses can therefore be more costly even than a main-memory access. Moreover, as address translation often requires accessing some memory structure, this can in turn trigger additional memory cache misses.

We will treat TLBs just like memory caches, using the memory page size as their cache line size, and calculating their (virtual) capacity as *number_of_entries* $\times$ *page_size*. TLBs are usually fully associative. Like caches, TLBs can be organized in multiple cascading levels.

For TLBs, there is no difference between sequential and random access latency. Further, bandwidth is irrelevant for TLBs, because a TLB miss does not cause any data transfer.

3.1.2.3 Unified Hardware Model

Summarizing our previous discussion, we describe a computers memory hardware as a cascading hierarchy of N levels of caches (including TLBs). We add an index $i \in \{1, \ldots, N\}$ to the parameters described above to refer to the respective value of a specific level. The relation between access latency and access bandwidth then becomes $\lambda_{i+1} = Z_i/\beta_{i+1}$. To simplify the notation, we exploit the dualism that an access to level $i+1$ is caused a miss on level i. Introducing the *miss latency* $l_i = \lambda_{i+1}$ and the respective *miss bandwidth* $b_i = \beta_{i+1}$, we get $l_i = Z_i/b_i$. Each cache level is characterized by the parameters given in Table 3.1.[1] In Section 3.3, we will present a

[1]We assume that costs for L1 cache accesses are included in the CPU costs, i.e., λ_1 and β_1 are not used and hence undefined.

description	unit	symbol
cache name (level)	-	Li
cache capacity	[bytes]	C_i
cache block size	[bytes]	Z_i
number of cache lines	-	$\#_i = C_i/Z_i$
cache associativity	-	A_i
sequential access		
access bandwidth	[bytes/ns]	β^{s}_{i+1}
access latency	[ns]	$\lambda^{\mathrm{s}}_{i+1} = Z_i/\beta^{\mathrm{s}}_{i+1}$
miss latency	[ns]	$l^{\mathrm{s}}_i = \lambda^{\mathrm{s}}_{i+1}$
miss bandwidth	[bytes/ns]	$b^{\mathrm{s}}_i = \beta^{\mathrm{s}}_{i+1}$
random access		
access latency	[ns]	$\lambda^{\mathrm{r}}_{i+1}$
access bandwidth	[bytes/ns]	$\beta^{\mathrm{r}}_{i+1} = Z_i/\lambda^{\mathrm{r}}_{i+1}$
miss bandwidth	[bytes/ns]	$b^{\mathrm{r}}_i = \beta^{\mathrm{r}}_{i+1}$
miss latency	[ns]	$l^{\mathrm{r}}_i = \lambda^{\mathrm{r}}_{i+1}$

Table 3.1: Characteristic Parameters per Cache Level ($i \in \{1, \ldots, N\}$)[1]

system independent C program called *Calibrator* to measure these parameters on any computer hardware. We point out, that these parameters also cover the cost-relevant characteristics of disk accesses. Hence, viewing main memory (e.g., a database system's buffer pool) as cache for I/O operations, it is straight forward to include disk access in this hardware model. Where appropriate, we use level $N + 1$ as synonym for main memory respectively secondary storage.

Though the unified hardware model is convenient for our following analysis, it sometime makes real-life examples a bit hard to read. For this reason, we will also use symbolical indices like ”L1”, ”L2”, ”TLB”, ”Mem”, or ”Disk”instead of i to indicate the various levels of caches and memories.

3.2 The New Bottleneck: Memory Access

In this section, we demonstrate the severe impact of memory access costs on the performance of elementary database operations. Using a traceable example, we first gather some general observations. Then, we analyze the results in detail and develop an analytical performance model. Finally, we present the results of our experiment on a number of machines and discuss them in a broader context.

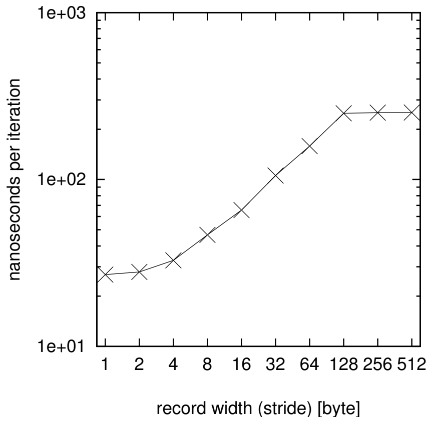


Figure 3.3: Sequential scan: Performance

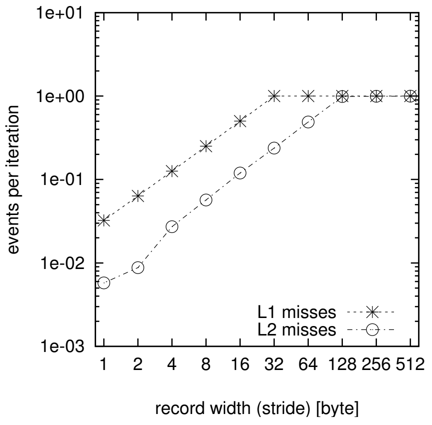


Figure 3.4: Sequential scan: Cache misses

3.2.1 Initial Example

As sample query, we use a simple aggregation (say, SELECT MAX(column) FROM table) on a one-byte attribute of an in-memory table. This query performs a sequential scan over the table. By varying the record width of the table, we vary the *stride*, i.e., the offset between two subsequently accessed memory addresses. We keep the cardinality of the table constant at 1,000,000 tuples. We use Monet to execute the experiment on an SGI Origin2000. This system uses the MIPS R10000 processor (250 MHz) with an L1 cache of 32KB (1024 lines of 32 bytes), and has an L2 cache of 4MB (32,768 lines of 128 bytes). The detailed hardware characteristics as derived by our Calibration Tool can be found in Section 3.3.

Figure 3.3 shows the results for various strides in terms of nanoseconds per iteration. We made sure that the table was in memory, but not in any of the memory caches, by first scanning the table in question, and then multiple times scanning some other table larger than the largest cache size.

3.2.2 General Observations

When the stride is small, successive iterations in the scan read bytes that are near to each other in memory, hitting the same cache line. The number of L1 and L2 cache misses is therefore low (cf., Figure 3.4)[2]. The L1 miss rate reaches its maximum of one miss per iteration as soon as the stride reaches the size of an L1 cache line (32 bytes). Only the L2 miss rate increases further, until the stride exceeds the size of an L2 cache line (128 bytes). Then, it is certain that every memory read is a cache miss. Performance cannot become any worse and stays constant.

[2]We used the hardware counters provided by the MIPS R10000 CPU to measure the number of cache misses.

3.2.3 Detailed Analysis

In the following, we present a detailed analysis of our experiment. Though we use the SGI Origin2000 as sample machine, we keep the models applicable to other systems as well by using a set of specific parameters to describe the respective hardware characteristics. See Section 3.1.2 for a detailed description of these parameters. In Section 3.3, we will present our Calibration Tool to measure the parameters.

In general, the execution costs per iteration of our experiment—depending on the stride s—can be modeled in terms of pure CPU costs (including data accesses in the on-chip L1 cache) and additional costs due to L2 cache accesses and main-memory accesses.

To measure the pure CPU costs—i.e., without any memory access costs—, we reduce the problem size to fit in L1 cache and ensure that the table is cached in L1 before running the experiment. This way, we observed $T_{\text{CPU}} = 24\text{ns}$ (6 cycles) per iteration for our experiment.

We model the costs for accessing data in the L2 cache and in main memory by scoring each access with the respective latency. As observed above, the number of L2 and main memory accesses (i.e., the number of L1 and L2 misses, respectively) depends on the access stride. With a stride s smaller than the cache line size Z, the average number of cache misses per iteration is $\mathbf{M}(s) = \frac{s}{Z}$. With a stride equal to or larger than the cache line size, a miss occurs with each iteration. In general, we get

$$\mathbf{M}_{\text{L}i}(s) = \left\{ \begin{array}{ll} \frac{s}{Z_{\text{L}i}}, & \text{if } s < Z_{\text{L}i} \\ 1, & \text{if } s \geq Z_{\text{L}i} \end{array} \right\} = \min\left\{\frac{s}{Z_{\text{L}i}}, 1\right\}, \quad i \in \{1, 2\} \tag{3.1}$$

with $\mathbf{M}_{\text{L}i}$ and $Z_{\text{L}i}$ ($i \in \{1, 2\}$) denoting the number of cache misses and the cache line sizes for each level, respectively. Figure 3.5 compares $\mathbf{M}_{\text{L1}}$ and $\mathbf{M}_{\text{L2}}$ to the measured number of cache misses.

We get the total costs per iteration—depending on the access stride—by summing the CPU costs, the L2 access costs, and the main-memory access costs:

$$T(s) = T_{\text{CPU}} + T_{\text{L2}}(s) + T_{\text{Mem}}(s) \qquad (\text{“model 1”})$$

with

$$T_{\text{L2}}(s) = \mathbf{M}_{\text{L1}}(s) * \lambda_{\text{L2}}, \qquad T_{\text{Mem}}(s) = \mathbf{M}_{\text{L2}}(s) * \lambda_{\text{Mem}},$$

where λ_x ($x \in \{\text{L2}, \text{Mem}\}$) denote the (cache) memory access latencies for each level, respectively. We measure the L2 and memory latency with our calibration tool presented in the next section (see Table 3.2). Figure 3.6 shows the resulting curve as “model 1”.

Obviously, this model does not match the experimental results. The reason is, that the R10000 processor is super scalar and can handle up to $q = 4$ active operations concurrently. Thus, the impact of memory access latency on the overall execution time may be reduced as (1) there must be four unresolved memory requests before the CPU stalls, and (2) up to q L1 or L2 cache lines may be loaded in parallel. In other words, operations may (partly) overlap. Consequently, their costs must not simply be

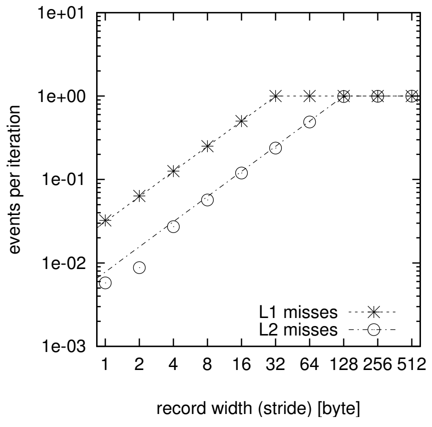


Figure 3.5: Seq. scan cache misses: Measured (points) and Modeled (lines)

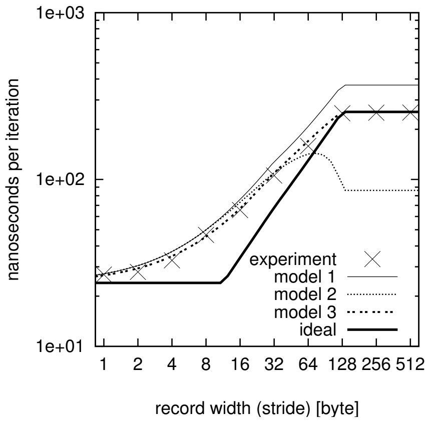


Figure 3.6: Seq. scan performance: Experiment and Models

added. Instead, we combine two cost components x and y, given the degree $o \in [0..1]$ they overlap, using the following function:

$$\mathbf{O}(o, x, y) = \max\{x, y\} + (1 - o) * \min\{x, y\} = x + y - o * \min\{x, y\}.$$

This overlap function forms a linear interpolation between the two extreme cases

- no overlap $(o = 0) \implies \mathbf{O}(0, x, y) = x + y,$ and
- full overlap $(o = 1) \implies \mathbf{O}(1, x, y) = \max\{x, y\}.$

Let o_1 and o_2 be the degrees of overlap for L2 access and main-memory access, respectively. Then, we get the total cost — considering overlap of CPU cost and memory access costs — as follows:

$$T = \mathbf{O}(o_1 * o_2, T_{\mathrm{CPU}}, T_{\mathrm{L2}} + T_{\mathrm{Mem}}).$$

The following consideration will help us to determine o_1 and o_2. In our experiments, we have a pure sequential memory access pattern. Up to a stride of 8 bytes, 4 subsequent memory references refer to the same 32-bytes L1 line, i.e., only one L1 line is loaded at a time, not allowing any overlap of pure calculation and memory access ($o_1 = o_2 = 0$). With strides between 8 and 32 bytes, o_1 linearly increases toward its maximum. The same holds for o_2 with strides between 32 and 128 bytes, as L2 lines contain 128 bytes on the R10000. Thus, we get

$$\begin{aligned} o_i(s) &= \max\left\{0, \min\left\{1, \frac{s - \frac{Z_{\mathrm{L}i}}{q}}{Z_{\mathrm{L}i} - \frac{Z_{\mathrm{L}i}}{q}}\right\}\right\} \\ &= \begin{cases} 0, & \text{if } s \le \frac{Z_{\mathrm{L}i}}{q} \\ \frac{s - \frac{Z_{\mathrm{L}i}}{q}}{Z_{\mathrm{L}i} - \frac{Z_{\mathrm{L}i}}{q}}, & \text{if } \frac{Z_{\mathrm{L}i}}{q} < s < Z_{\mathrm{L}i} \\ 1, & \text{if } s \ge Z_{\mathrm{L}i} \end{cases} \qquad (i \in \{1,2\}) \end{aligned}$$

Similarly, up to $q = 4$ cache lines can be loaded concurrently during a single latency period, reducing the effective latency per cache miss to $\frac{1}{q}$-th. Following the previous overlap considerations, we model the effective latency depending on the stride:

$$\begin{aligned} \lambda'_{\mathrm{L2}}(s) &= \lambda^{\min}_{\mathrm{L2}} + (1 - o_1(s)) * \left(\lambda_{\mathrm{L2}} - \lambda^{\min}_{\mathrm{L2}}\right), \qquad \lambda^{\min}_{\mathrm{L2}} = \frac{\lambda_{\mathrm{L2}}}{q} \\ \lambda'_{\mathrm{Mem}}(s) &= \lambda^{\min}_{\mathrm{Mem}} + (1 - o_2(s)) * \left(\lambda_{\mathrm{Mem}} - \lambda^{\min}_{\mathrm{Mem}}\right), \quad \lambda^{\min}_{\mathrm{Mem}} = \frac{\lambda_{\mathrm{Mem}}}{q}. \end{aligned}$$

Now, we can refine our model as follows:

$$T(s) = \mathbf{O}\left(o_1(s) * o_2(s), T_{\mathrm{CPU}}, T'_{\mathrm{L2}}(s) + T'_{\mathrm{Mem}}(s)\right) \qquad (\text{“model 2”})$$

with

$$T'_{\mathrm{L2}}(s) = \mathbf{M}_{\mathrm{L1}}(s) * \lambda'_{\mathrm{L2}}(s), \qquad T'_{\mathrm{Mem}}(s) = \mathbf{M}_{\mathrm{L2}}(s) * \lambda'_{\mathrm{Mem}}(s),$$

and T_{CPU}, $\mathbf{M}_{\mathrm{L1}}$, $\mathbf{M}_{\mathrm{L2}}$, λ'_{L2}, λ'_{Mem} as above.

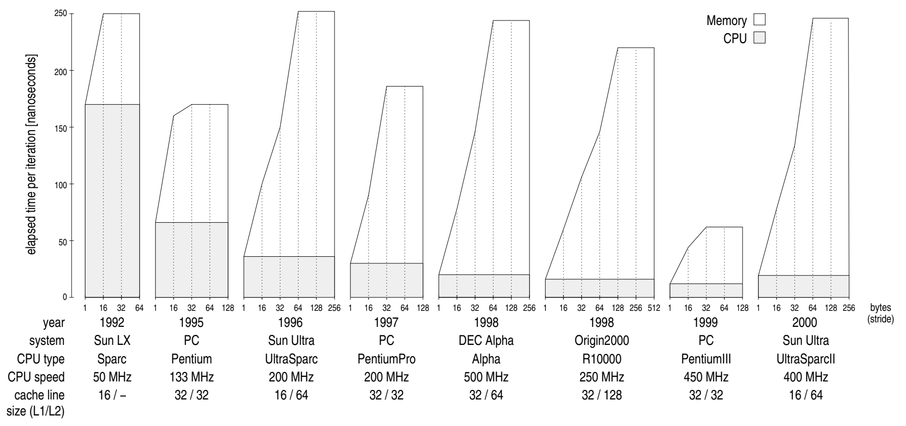
Figure 3.6 depicts the resulting curve as “model 2”. The curve fits the experimental results almost exactly for smaller strides up to $s = 32$. For larges strides, however, the modeled costs are significantly lower than the measured costs. When loading several cache lines concurrently we have to consider another limit: bandwidth. L2 bandwidth is large enough to allow $q = 4$ concurrent L1 loads within a single L2 latency period ($4 * 32$ bytes within 24ns (6 cycles), i.e., ~5GB/s). Memory bandwidth, however, is limited to 555MB/s.[3] Hence, loading four L2 lines ($4 * 128$ bytes) in parallel takes at least 880ns (220 cycles), or on average $\lambda^{\mathrm{bw}}_{\mathrm{Mem}} = 220$ns (55 cycles) per line.

Replacing $\lambda^{\min}_{\mathrm{Mem}}$ by $\lambda^{\mathrm{bw}}_{\mathrm{Mem}}$ in “model 2” yields our final “model 3”. As Figure 3.6 shows, “model 3” fits the experimental curve pretty well. In this scenario, the “ideal” performance of

$$T(s) = \max\{T_{\mathrm{CPU}}, T'_{\mathrm{L2}}(s), T'_{\mathrm{Mem}}(s)\}, \qquad (\text{“ideal”})$$

i.e., with $o_1 = o_2 = 1$, is not reached (cf., “ideal” in Fig. 3.6), because the whole memory bandwidth cannot be utilized automatically for smaller strides, i.e., when several memory references refer to a single L2 line.

[3]See Section 3.3, Table 3.2.



year	1992	1995	1996	1997	1998	1998	1999	2000
system	Sun LX	PC	Sun Ultra	PC	DEC Alpha	Origin2000	PC	Sun Ultra
CPU type	Sparc	Pentium	UltraSparc	PentiumPro	Alpha	R10000	PentiumIII	UltraSparcII
CPU speed	50 MHz	133 MHz	200 MHz	200 MHz	500 MHz	250 MHz	450 MHz	400 MHz
cache line size (L1/L2)	16 / –	32 / 32	16 / 64	32 / 32	32 / 64	32 / 128	32 / 32	16 / 64

Figure 3.7: CPU and memory access costs per tuple in a simple table scan

3.2.4 Discussion

The detailed analysis and the models derived, show how hardware specific parameters such as cache line sizes, cache miss penalties, and degree of CPU-inherent parallelism determine the performance of our scan experiment. We will now discuss the experiment in a broader context.

Figure 3.7 shows results of the above experiment on a number of popular workstations of the past decade. The X-axis shows the different systems ordered by their age, and per system the different strides tested. The Y-axis shows the absolute elapsed time for the experiments. For each system, the graph is split up to show which part of the elapsed time is spent waiting for memory (upper), and which part with CPU processing (lower, gray-shaded).

While all machines in Figure 3.7 exhibit the same pattern of performance degradation with decreasing data locality, Figure 3.7 clearly shows that the penalty for poor memory cache usage has dramatically increased in the last ten years. The CPU speed has improved by at least an order of magnitude, both through higher clock frequencies and through increased CPU-inherent parallelism. However, the memory cost trend in Figure 3.7 shows a mixed picture, and has clearly not kept up with the advances in CPU power. Consequently, while our experiment was still largely CPU-bound on the Sun from 1992, it is dominated by memory access costs on the modern machines (even the PentiumIII with fast memory is 75% of the time waiting for memory). Note that the later machines from Sun, Silicon Graphics and DEC actually have memory access costs that in absolute numbers are even higher than on the Sun from 1992. This can be attributed to the complex memory subsystem that comes with SMP architectures, resulting in a high memory latency. These machines do provide a high memory bandwidth—thanks to the ever growing cache line sizes[4]—but this does not solve the

[4]In one cache miss, the Origin2000 fetches 128 bytes, whereas the Sun LX fetches only 16; an improvement of factor 8.

latency problem if data locality is low. In fact, we must draw the sad conclusion that if no attention is paid in query processing to data locality, all advances in CPU power are neutralized due to the memory access bottleneck caused by memory latency.

The trend of improvement in bandwidth but standstill in latency [Ram96, SLD97] is expected to continue, with no real solutions in sight. The work in [Mow94] has proposed to hide memory latency behind CPU work by issuing *prefetch* instructions, before data is going to be accessed. The effectiveness of this technique for database applications is, however, limited due to the fact that the amount of CPU work per memory access tends to be small in database operations (e.g., the CPU work in our select-experiment requires only 4 cycles on the Origin2000). Another proposal [MKW+98] has been to make the caching system of a computer configurable, allowing the programmer to give a "cache-hint" by specifying the memory-access stride that is going to be used on a region. Only the specified data would then be fetched; hence optimizing bandwidth usage. Such a proposal has not yet been considered for custom hardware, however, let alone in OS and compiler tools that would need to provide the possibility to incorporate such hints for user-programs.

Our simple experiment makes clear why database systems are quickly constrained by memory access, even on simple tasks like scanning, that seem to have an access pattern that is easy to cache (sequential). The default physical representation of a tuple is a consecutive byte sequence (a "record"), which must always be accessed by the bottom operators in a query evaluation tree (typically selections or projections). The record byte-width of typical relational table amounts to some hundreds of bytes. Figure 3.7 makes clear that such large strides lead to worst-case performance, such that the memory access bottleneck kills all CPU performance advances.

To improve performance, we strongly recommend using *vertically fragmented* data structures. In Monet, we *fully* decompose relational tables on all columns, storing each in a separate Binary Association Tables (BAT). This approach is known in literature as the Decomposed Storage Model [CK85]. A BAT is represented in memory as an array of fixed-size two-field records [OID,value]—called Binary UNits (BUN)—where the OIDs are used to link together the tuples that are decomposed across different BATs. Full vertical fragmentation keeps the database records thin (8 bytes or less) and is therefore the key for reducing memory access costs (staying on the left side of the graphs in Figure 3.7). In Section 2.7, we presented specific implementation details of Monet.

3.2.5 Implications for Data Structures

In terms of data structures for query processing, we already noted from the simple scan experiment in Figure 3.7 that *full vertical table fragmentation* optimizes column-wise memory access to table data. This is particularly beneficial if the table is accessed in a sequential scan that reads a minority of all columns. Such table scans very often occur in both OLAP and Data Mining workloads. When record-oriented (i.e., non-fragmented) physical storage is used, such an access leads to data of the non-used columns being loaded into the cache lines, wasting memory bandwidth. In case of a vertically fragmented table, the table scan just needs to load the vertical fragments

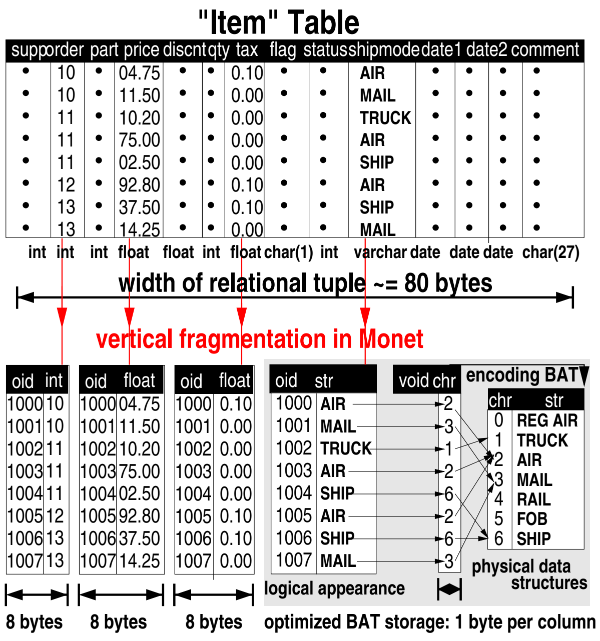

"Item" Table
supp order part price discnt qty tax flag status shipmode date1 date2 comment
int int int float float int float char(1) int varchar date date date char(27)
width of relational tuple ~= 80 bytes
vertical fragmentation in Monet
oid int
oid float
oid float
oid str
void chr
encoding BAT
chr str
REG AIR
TRUCK
AIR
MAIL
RAIL
FOB
SHIP
logical appearance
physical data structures
8 bytes
8 bytes
8 bytes
optimized BAT storage: 1 byte per column

Figure 3.8: Vertical Decomposition in BATs

pertaining to the columns of interest. Reading those vertical fragments sequentially achieves a 100% hit rate on all cache levels, exploiting optimal bandwidth on any hardware, including parallel memory access.

There are various ways to incorporate vertical fragmentation in database technology. In Monet, which we designed for OLAP and Data Mining workloads, vertical fragmentation is the basic building block of all physical storage, as Monet fully fragments all relations into Binary Association Tables (BATs) (see Figure 3.8). Flat binary tables are a simple set-oriented physical representation, that is not tied to a particular logical data model, yet is sufficiently powerful to represent, e.g., join indices [Val87]. Monet has successfully been used to store and query relational, object-oriented and network data structures, using this very simple data model and a small kernel of algebraic operations on it [BK99]. In Monet, we applied two additional optimizations that further reduce the per-tuple memory requirements in its BATs:

- *virtual-OIDs.* Generally, when decomposing a relational table, we get an identical system-generated column of OIDs in all decomposition BATs, which is *dense and ascending* (e.g., 1000, 1001, . . . , 1007). In such BATs, Monet computes the OID-values on-the-fly when they are accessed using positional lookup of the BUN, and avoids allocating the 4-byte OID field. This is called a “virtual-OID” or VOID column. Apart from reducing memory requirements by half, this optimization is also beneficial when joins or semi-joins are performed on

OID columns.[5] When one of the join columns is VOID, Monet uses positional lookup instead of, e.g., hash-lookup; effectively eliminating all join costs.

- *byte-encodings.* Database columns often have a low domain cardinality. For such columns, Monet uses fixed-size encodings in 1- or 2-byte integer values. This simple technique was chosen because it does not require decoding effort when the values are used (e.g., a selection on a string "MAIL" can be re-mapped to a selection on a byte with value 3). A more complex scheme (e.g., using bit-compression) might yield even more memory savings, but the decoding-step required whenever values are accessed can quickly become counterproductive due to extra CPU effort. Even if decoding would just cost a handful of cycles per tuple, this would more than double the amount of CPU effort in simple database operations, like a simple aggregation from Section 3.2.1, which takes just 2 cycles of CPU work per tuple.

Figure 3.8 shows that when applying both techniques, the storage needed for 1 BUN in the "shipmode" column is reduced from 8 bytes to just one. Reducing the stride from 8 to 1 byte significantly enhances performance in the scan experiment from Figure 3.7, eliminating all memory access costs.

Alternative ways of using vertical table fragmentation in a database system are to offer the logical abstraction of relational tables but employ physically fragmentation in *transposed files* [Bat79] on the physical level (like in NonStopSQL [CDH+99]), or to use vertically fragmented data as a search accelerator structure, similar to a B-tree. Sybase IQ uses this approach, as it automatically creates *projection indices* on each table column [Syb96]. In the end, however, all these approaches lead to the same kind and degree of fragmentation.

3.3 The Calibrator: Quantification of Memory Access Costs

In order to model memory access costs in detail, we need to know the characteristic parameters of the memory system, including memory sizes, cache sizes, cache line sizes, and access latencies. Often, not all these parameters are (correctly) listed in the hardware manuals. In the following, we describe a simple but powerful *calibration tool* to measure the (cache) memory characteristics of an arbitrary machine.

3.3.1 Calibrating the (Cache-) Memory System

The idea underlying our calibrator tool is to have a micro benchmark whose performance only depends on the frequency of cache misses that occur. Our calibrator is a simple C program, mainly a small loop that executes a million memory reads, repeatedly sweeping over an array stored in main memory. By changing the *stride* (i.e., the

[5] In Monet, the projection phase in query processing typically leads to additional "tuple-reconstruction" joins on OID columns that are caused by the fact that tuples are decomposed into multiple BATs.

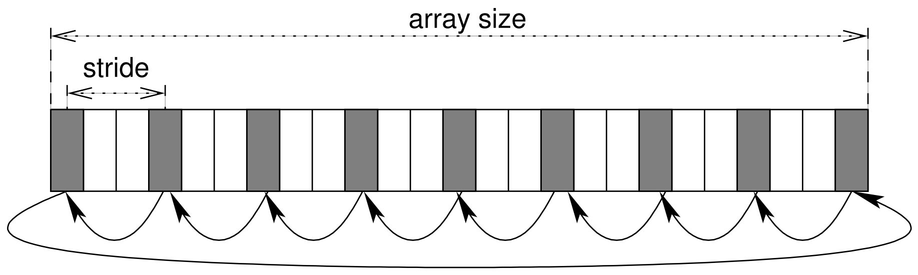


Figure 3.9: Calibration Tool: Walking “backward” through the memory array

offset between two subsequent memory accesses) and the *array size*, we force varying cache miss rates.

In principle, the occurrence of cache misses is determined by the array size. Accessing an array that fits into the L1 cache does not generate any cache misses once the data is loaded into the cache. Analogously, sweeping over an array that exceeds the L1 cache size, but still fits into L2, will cause L1 misses but no L2 misses. Finally, using an array larger than L2 causes both L1 and L2 misses.

The frequency of cache misses depends on the access stride and the cache line size. With strides equal to or larger than the cache line size, a cache miss occurs with every iteration. With strides smaller than the cache line size, a cache miss occurs only every n iterations (on average), where n is the ratio cache_line_size/stride. In this latter case, each miss causes a complete cache line to be loaded into the cache, providing the data for both the request that triggered the miss and the subsequent $n - 1$ requests that access data within the same cache line.

Thus, we can calculate the latency for a cache miss by comparing the execution time without misses to the execution time with exactly one miss per iteration. This approach only works, if memory accesses are executed purely sequential, i.e., we have to ensure that neither two or more load instructions nor memory access and pure CPU work can overlap. We use a simple pointer chasing mechanism to achieve this: the memory area we access is initialized such that each load returns the address for the subsequent load in the next iteration. Thus, super-scalar CPUs cannot benefit from their ability to hide memory access latency by speculative execution. Further, we need to avoid that the system can benefit from prefetching. *Prefetching* depicts a mechanism where CPUs do not only load the demanded cache line, but also some cache line ahead (i.e., the subsequent cache lines in memory) although they are not requested, yet. With a sequential “forward-oriented” memory access pattern, this technique allows to (partly) overlap CPU processing and memory accesses (even without speculative execution), and hence may reduce the effective memory access latency. To disable prefetching or at least make it “useless”, the calibration tool walks “backward” through the memory. Figure 3.9 illustrates this.

To measure the cache characteristics, we run our experiment several times, varying the stride and the array size. We make sure that the stride varies at least between

4 bytes and twice the maximal expected cache line size, and that the array size varies from half the minimal expected cache size to at least ten times the maximal expected cache size. In case the array is so big that the default 1 million iterations using the given stride do not cover the whole array, we increase the number of iterations accordingly.

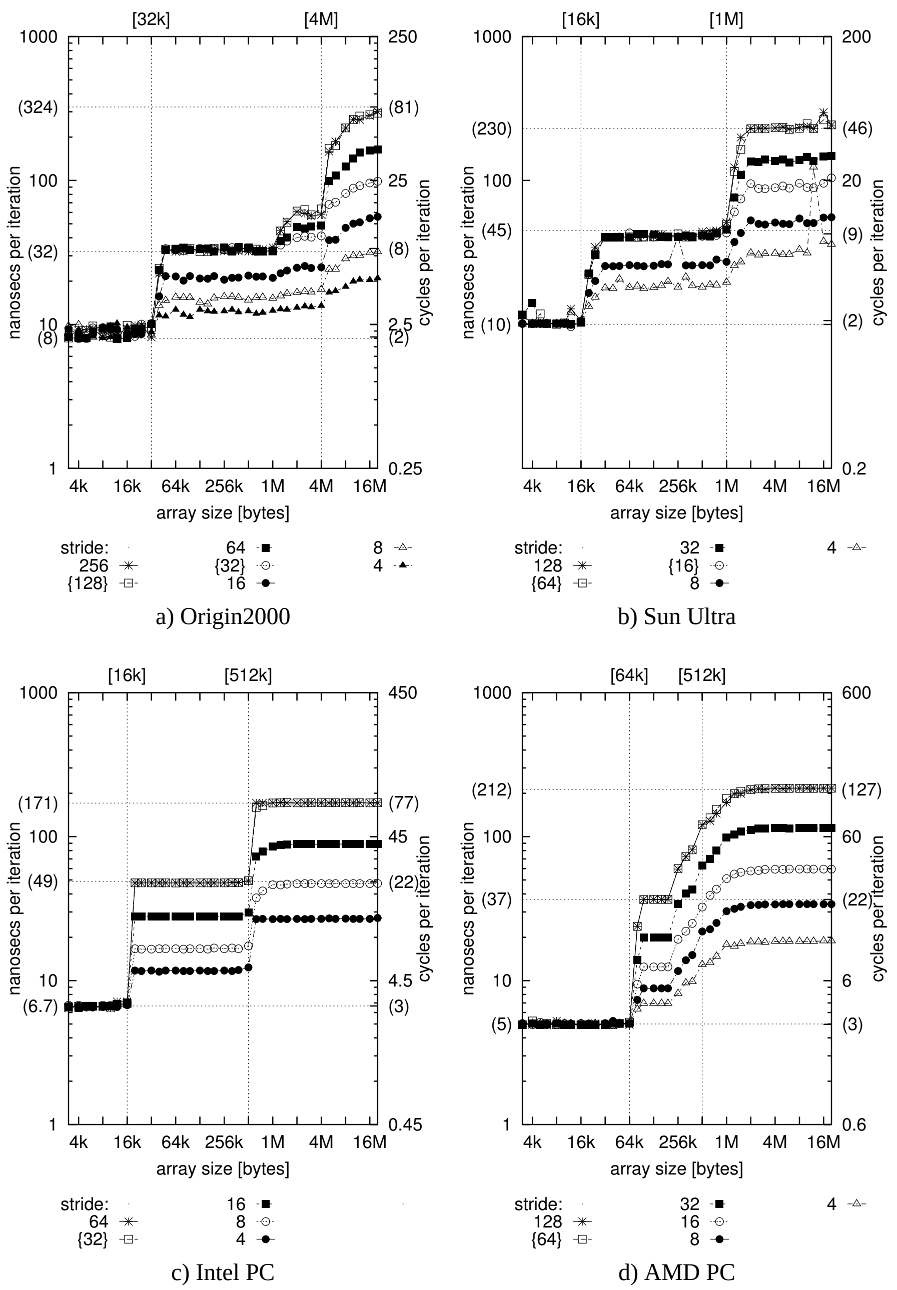
We run two experiments. In the first, the insert a delay of about 100 CPU cycles between to subsequent memory accesses. Thus, we give the whole cache-memory-system and the connecting bus some time to “calm down”. This tries to mimic a “once-a-while” kind of access. In the second experiment, we run without the delay, continuously issuing memory accesses. Comparing the two experiments, we find out, whether the cache and memory latencies differ between “once-a-while” and continuous memory access. To distinguish the results derived from both experiments, we call the first *cache miss latencies* and the latter *cache replace times* (derived from the fact that exclusive cache hierarchies actually do swap cache lines between adjacent cache levels; cf., Section 3.1.2).

Figure 3.10a depicts the resulting execution time (in nanoseconds) per iteration of the first experiment for different array sizes on an Origin2000 (MIPS R10000, 250 MHz = 4ns per cycle). Each curve represents a different stride. From this figure, we can derive the desired parameters as follows: Up to an array size of 32 KB, one iteration takes 8 nanoseconds (i.e., 2 cycles), independent on the stride. Here, no cache misses occur once the data is loaded, as the array completely fits in L1 cache. One of the two cycles accounts for executing the load instruction, the other one accounts for the latency to access data in L1. With array sizes between 32 KB and 4 MB, the array exceeds L1, but still fits in L2. Thus, L1 misses occur. In other words, the two steps in the curves at array sizes of 32 KB and 4 MB, respectively, indicate that there are two cache levels, L1 and L2, with sizes of 32 KB and 4 MB, respectively. The miss rate (i.e., the number of misses per iteration) depends on the stride (s) and the L1 cache line size (Z_{L1}). With $s < Z_{\mathrm{L1}}$, $\dfrac{s}{Z_{\mathrm{L1}}}$ L1 misses occur per iteration (or one L1 miss occurs every $\dfrac{Z_{\mathrm{L1}}}{s}$ iterations). With $s \geq Z_{\mathrm{L1}}$, each load causes an L1 miss. Figure 3.10a shows that the execution time increases with the stride, up to a stride of 32. Then, it stays constant. Hence, L1 line size is 32 byte. Further, L1 miss latency (i.e., L2 access latency) is 32ns − 8ns = 24ns, or 6 cycles. Similarly, when the array size exceeds L2 size (4 MB), L2 misses occur. Here, the L2 line size is 128 byte, and the L2 miss latency (memory access latency) is 324ns − 32ns = 292ns, or 73 cycles.

Analogously, Figures 3.10b through 3.10d show the results for a Sun Ultra (Sun UltraSPARC, 200 MHz = 5ns per cycle), an Intel PC (Intel PentiumIII, 450 MHz = 2.22ns per cycle), and an AMD PC (AMD Athlon, 600 MHz = 1.67ns per cycle). All curves show two steps, indicating the existence of two cache levels and their sizes.

The *random access memory bandwidth* for our systems, listed in Table 3.2, is computed from the cache line sizes and the latencies as follows:

$$\beta^{\mathrm{r}}_{\mathrm{Mem}} = \frac{Z_{\mathrm{L2}}}{\lambda_{\mathrm{Mem}} + \dfrac{Z_{\mathrm{L2}}}{Z_{\mathrm{L1}}} * \lambda_{\mathrm{L2}}}.$$



a) Origin2000

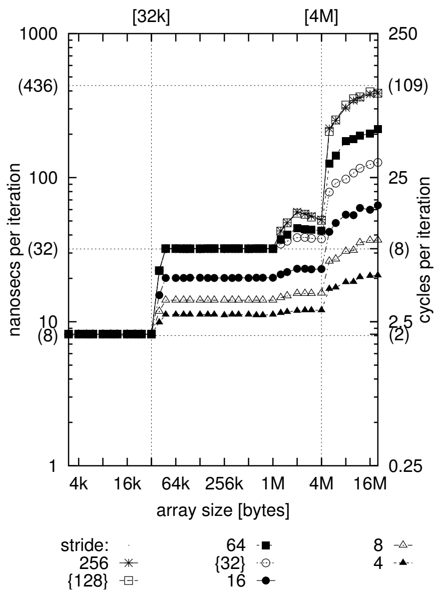
b) Sun Ultra

c) Intel PC

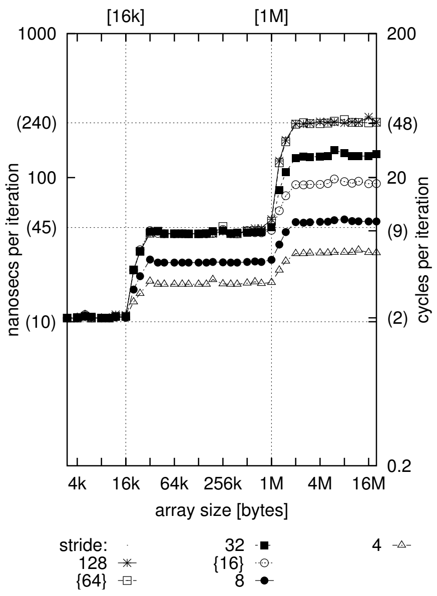
d) AMD PC

(Vertical grid lines indicate derived cache sizes, horizontal grid lines indicate derived latencies.)

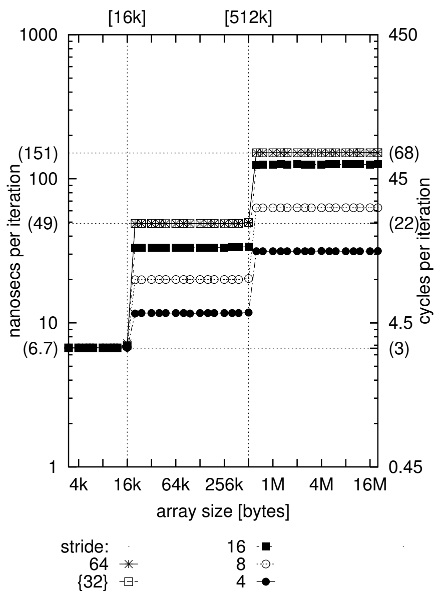
Figure 3.10: Calibration Tool: Cache sizes, line sizes, and miss latencies



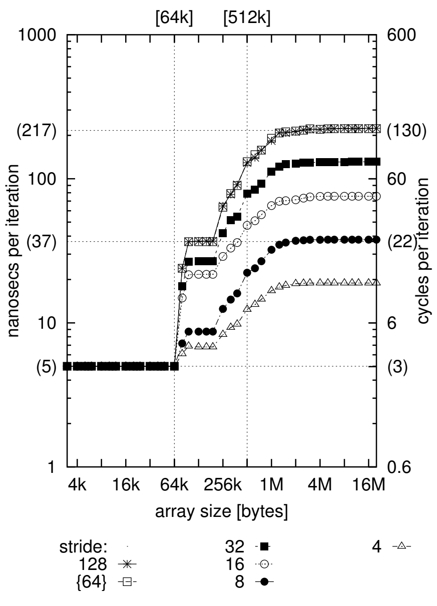
a) Origin2000



b) Sun Ultra



c) Intel PC



d) AMD PC

(Vertical grid lines indicate derived cache sizes, horizontal grid lines indicate derived latencies.)

Figure 3.11: Calibration Tool: Cache sizes, line sizes, and replace times

The *sequential access memory bandwidth* β^s is derived form the scan experiment of the previous section using a stride $s > Z_{L2}$.

Figures 3.11a through 3.11d depict the results of the second experiments for all four machines. As expected, the derived cache sizes and cache line sizes are identical to those from then first experiment. Comparing the replace times here with the latencies above, we observe the following. For L1 misses (L2 accesses), latency and replace time are equal on each of the four machines. All four machines use inclusive cache systems, hence we did not expect any other result. For L2 misses (i.e., main memory accesses), however, we see a different image. On the Origin2000, the Sun, and the AMD PC, replace time is higher than latency. On the Sun and the AMD PC, the difference are just about 3% to 4% (195ns vs. 188ns and 180ns vs. 175ns, respectively); on the Origin2000, however, it is 40% (406ns vs. 292ns). On the Intel PC, replace time is 20% less than latency (102ns vs. 123ns).

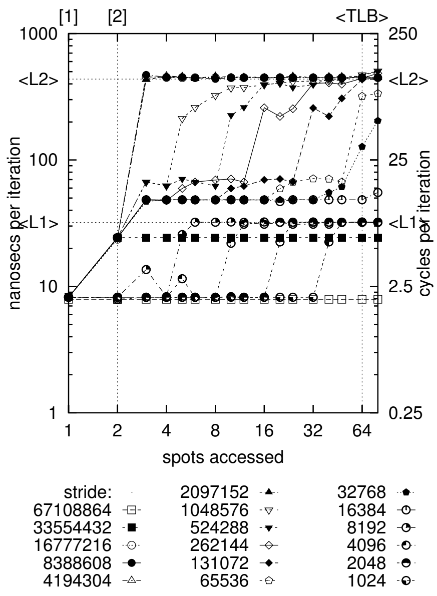
Cache Associativity Above, we forced capacity misses in order to measure cache sizes, line sizes and latencies. Now, to be able to measure the cache associativity, we need to force conflict misses. Assuming the usually only the lower bits of a memory address are used to determined the cache line to be used, we use rather large strides when walking through the memory array. Further using strides that are powers of 2, we ensure that the lower bits of subsequently accesses memory addresses are equal. Successively increasing the stride from 1024 through array size and varying the array size as above, conflict misses will occur as soon as the number of distinct spots accessed in the array exceeds the cache's associativity.

Figures 3.11a through 3.11d depict the respective results for all four machines. The X-axis now gives the number of spots accessed, i.e., array size divided by stride. Again, each curve represents a different stride. We can derive the following associativities. On the Origin2000, L1 is direct-mapped (1-way associative) and L2 is 2-way associative; on the Sun, L1 is direct-mapped (1-way associative) and L2 is 4-way associative; on the Intel PC, L1 and L2 are both 4-way associative; and on then AMD PC, L1 is 2-way associative and L2 is 4-way associative.

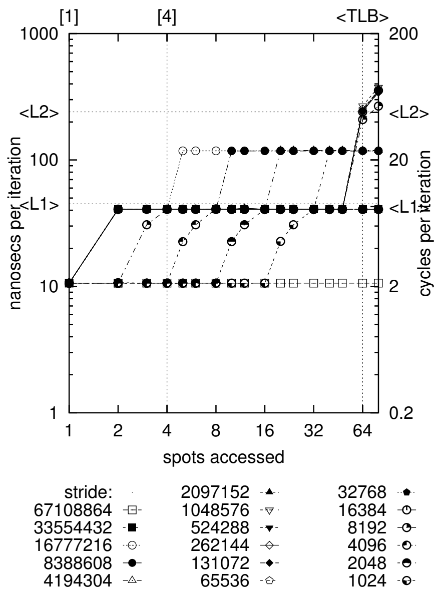
3.3.2 Calibrating the TLB

We use a similar approach as above to measure *TLB miss costs*. The idea here is to force one TLB miss per iteration, but to avoid any cache misses. We force TLB misses by using a stride that is larger than the system's page size, and by choosing the array size such that we access more distinct spots than there are TLB entries. Cache misses will occur at least as soon as the number of spots accessed exceeds the number of cache lines. We cannot avoid that. But even with less spots accessed, two or more spots might be mapped to the same cache line, causing conflict misses. To avoid this, we use strides that are not exactly powers of 2, but slightly bigger, shifted by L2 cache line size, i.e., $s = 2^x + Z_{L2}$.

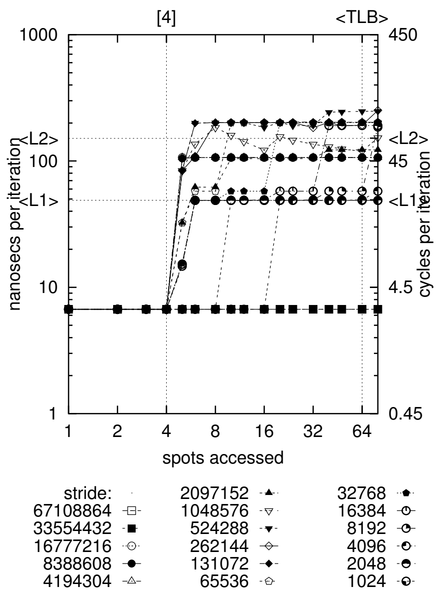
Figure 3.13 shows the results for four machines. Again, the X-axis gives the number of spots accessed, and each curve represents a different stride. From Figure 3.13a (Origin2000), e.g., we derive the following: Like above, we observe the base line of



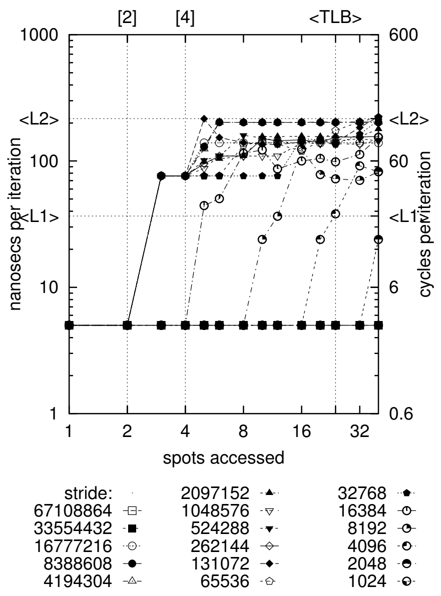
a) Origin2000



b) Sun Ultra



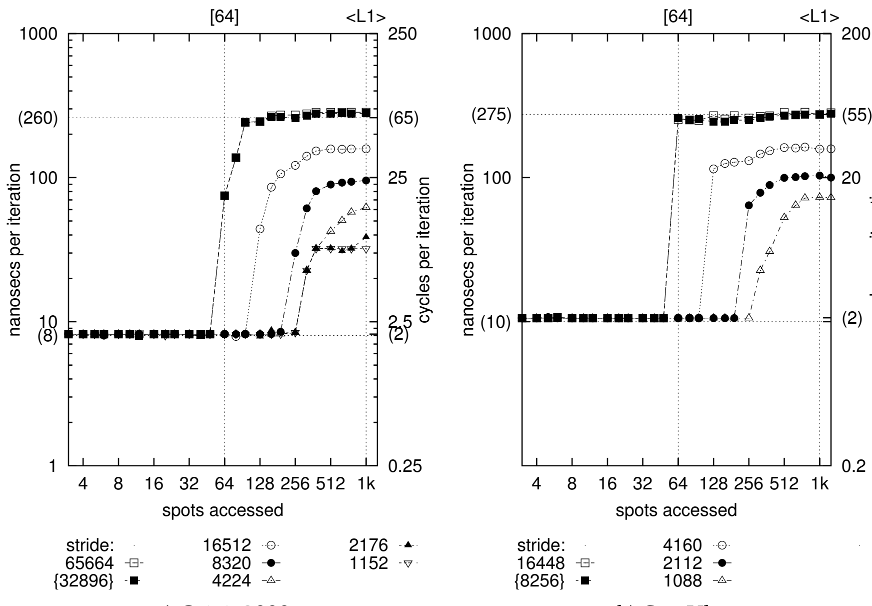
c) Intel PC



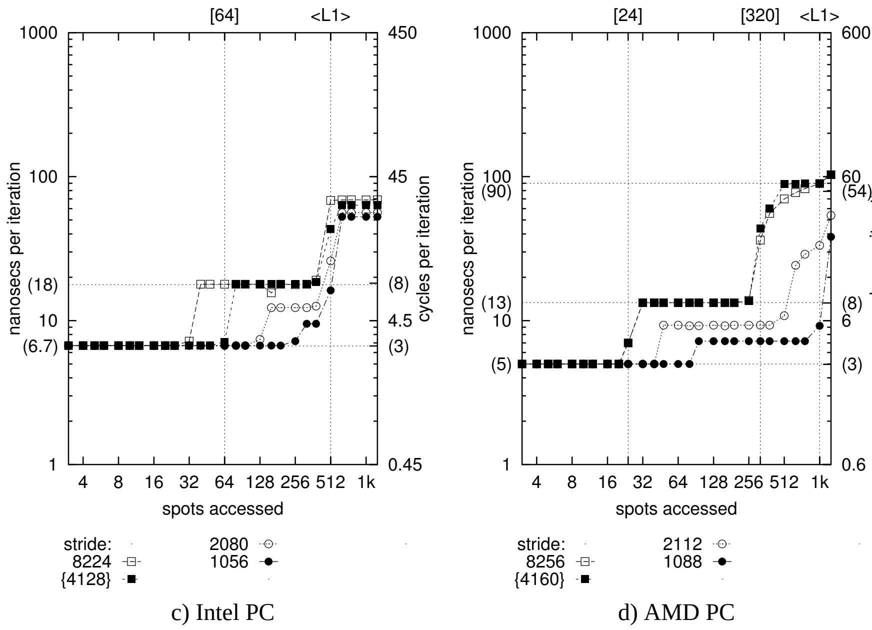
d) AMD PC

(Vertical grid lines indicate derived associativities, horizontal grid lines indicate cache miss latencies.)

Figure 3.12: Calibration Tool: Cache associativities



a) Origin2000 b) Sun Ultra



c) Intel PC d) AMD PC

(Vertical grid lines indicate derived number of TLB entries, horizontal lines indicate derived latencies.)

Figure 3.13: Calibration Tool: TLB entries and TLB miss costs

8 nanoseconds (i.e., 2 cycles) per iteration. The smallest number of spots where the performance decreases due to TLB misses is 64, hence, there must be 64 TLB entries. The decrease at 64 spots occurs with strides of 32 KB or more, thus, the page size is 32 KB. Further, TLB miss latency is 236ns − 8ns = 228ns, or 57 cycles. Figure 3.13d correctly reflects the Athlon's two TLB levels with 32 and 256 entries, respectively. The third step in the curves at 1024 spots is caused by L1 misses as L1 latency is 5 times higher than TLB latency on the Athlon. The same holds for the second step in the PentiumIII curves (Figure 3.13c) at 512 spots. On the Origin2000 and on the Sun, L1 misses also occur with more than 1024 spots accessed, but their impact is negligible as TLB latency is almost 10 times higher than L1 latency on these machines.

Due to their small size, TLBs are usually fully associative. Hence, we omit testing the TLBs' associativity.

3.3.3 Summary

Next to producing the graphs as depicted above, our calibration tool automatically analyzes the measured data and derives the desired parameters. The final output looks as follows (here: Origin2000):

```
CPU loop + L1 access:       8.18 ns =   2 cy
             ( delay:     401.74 ns = 100 cy )

caches:
level  size     linesize   associativity  miss-latency        replace-time
  1     32 KB   32 bytes        1-way     25.54 ns =   6 cy   23.94 ns =   6 cy
  2      4 MB  128 bytes        2-way    290.34 ns =  73 cy  405.52 ns = 101 cy

TLBs:
level #entries  pagesize  miss-latency
  1      64       32 KB   252.69 ns =  63 cy
```

Table 3.2 gathers the results for all four machines. The PCs have the highest L2 access latencies, probably as their L2 caches are running at only half the CPUs' clock speed. Main-memory access, however, is faster on the PCs than it is on the SGI and the Sun. The TLB miss latency of the PentiumIII and the Athlon (TLB_1) are very low, as their TLB management is implemented in hardware. This avoids the costs of trapping to the operating system on a TLB miss, that is necessary in the software controlled TLBs of the other systems. The TLB_2 miss latency on the Athlon is comparable to that on the R10000 and the UltraSPARC. The Origin2000 has the highest memory latency, but due to its large cache lines, it achieves better sequential memory bandwidth than the Sun and the Intel PC.

The calibration tool and results for a large number of different hardware platforms are available on our web site: `http://www.cwi.nl/~monet/`.

		SGI Origin2000	Sun Ultra	Intel PC	AMD PC
OS		IRIX64 6.5	Solaris 2.5.1	Linux 2.2.14	Linux 2.2.14
CPU		MIPS R10000	Sun UltraSPARC	Intel PentiumIII	AMD Athlon
CPU speed		250 MHz	200 MHz	450 MHz	600 MHz
main-memory size		16 * 4 GB	512 MB	512 MB	384 MB
L1 cache size	$\lVert L1 \rVert$	32 KB	16 KB	16 KB	64 KB
L1 cache line size	Z_{L1}	32 bytes	16 bytes	32 bytes	64 bytes
L1 cache lines	$\lvert L1 \rvert_{\text{L1}}$	1024	1024	512	1024
L2 cache size	$\lVert L2 \rVert$	4 MB	1 MB	512 KB	512 KB
L2 cache line size	Z_{L2}	128 bytes	64 bytes	32 bytes	64 bytes
L2 cache lines	$\lvert L2 \rvert_{\text{L2}}$	32,768	16,384	16,384	8192
TLB entries	$\lvert TLB \rvert$	64	64	64	32
TLB$_2$ entries	$\lvert TLB_2 \rvert$	-	-	-	256
page size	$\lVert Pg \rVert$	32 KB	8 KB	4 KB	4 KB
TLB size	$\lVert TLB \rVert$	2 MB	512 KB	256 KB	128 KB
TLB$_2$ size	$\lVert TLB_2 \rVert$	-	-	-	1 MB
L1 associativity	A_{L1}	1-way	1-way	4-way	2-way
L2 associativity	A_{L2}	2-way	4-way	4-way	4-way
L1 miss latency	l_{L1}	24 ns = 6 cy.	30 ns = 6 cy.	42 ns = 19 cy.	45 ns = 27 cy.
L1 replace time	rt_{L1}	24 ns = 6 cy.	30 ns = 6 cy.	42 ns = 19 cy.	45 ns = 27 cy.
L2 miss latency	l_{L2}	292 ns = 73 cy.	188 ns = 38 cy.	123 ns = 55 cy.	175 ns = 103 cy.
L2 replace time	rt_{L2}	406 ns = 101 cy.	195 ns = 39 cy.	102 ns = 46 cy.	180 ns = 108 cy.
TLB miss latency	l_{TLB}	252 ns = 63 cy.	265 ns = 53 cy.	11 ns = 5 cy.	8 ns = 5 cy.
TLB$_2$ miss latency	l_{TLB_2}	-	-	-	77 ns = 46 cy.
ran. acc. bandwidth	$\beta^{\text{r}}_{\text{Mem}}$	243 MB/s	194 MB/s	212 MB/s	271 MB/s
seq. acc. bandwidth	$\beta^{\text{s}}_{\text{Mem}}$	555 MB/s	244 MB/s	484 MB/s	670 MB/s

Table 3.2: Calibrated Performance Characteristics

3.4 Further Observations

In the remainder of this chapter, we discuss further aspects of main-memory access, such as parallel access and prefetching, and sketch some future hardware trends.

3.4.1 Parallel Memory Access

It is interesting to note that the calibrated latencies in Table 3.2 do not always confirm the suggested latencies in the sequential scan experiment from Figure 3.7. For the PentiumIII, the access costs per memory read of 52ns at a stride of 32 bytes, and 204ns at a stride of 128 bytes for the Origin2000, are considerably lower than their memory latencies (135ns resp. 424ns), whereas in the case of the Sun Ultra, the scan measurement at L2 line size almost coincides with the calibrated memory latency. The discrepancies are caused by *parallel memory access* that can occur on CPUs that feature both speculative execution and a non-blocking memory system. This allows a CPU to execute multiple memory load instructions in parallel, potentially enhancing memory bandwidth above the level of cache-line size divided by latency. Prerequisites for this technique are a bus system with excess transport capacity and a *non-blocking cache* system that allows multiple outstanding cache misses.

normal loop	**multi-cursor**	**prefetch**
for (**int** tot=i=0; i<N; i++) {	**for** (**int** tot$_0$ =tot$_1$ =i=0, C=N/2;	**for** (**int** tot=i=0; i<N; i++) {
	i<C; i++) {	#**prefetch** buf[i+32] **freq**=32
tot += buf[i];	tot$_0$+= buf[i];	tot += buf[i];
	tot$_1$+= buf[i+C];	
}	} **int** tot = tot$_0$+ tot$_1$;	}
5.88 cycles/addition	3.75 cycles/addition	3.88 $\rightarrow$ 2 cycles/addition

Figure 3.14: Three ways to add a buffer of integers, and costs per addition on the Origin2000

To answer the question what needs to be done by an application programmer to achieve these parallel memory loads, let us consider a simple programming loop that sums an array of integers. Figure 3.14 shows three implementations, where the leftmost column contains the standard approach that results in sequential memory loads into the buf[size] array. An R10000 processor can continue executing memory load instructions speculatively until four of them are stalled. In this loop, that will indeed happen if buf[i], buf[i+1], buf[i+2], and buf[i+3] are not in the (L2) cache. However, due to the fact that our loop accesses consecutive locations in the `buf` array, these four memory references request the same 128-byte L2 cache line. Consequently, no parallel memory access takes place. If we assume that this loop takes 2 cycles per iteration[6], we can calculate that 32 iterations cost 32*2 + 124 = 188 cycles (where 124 is the memory latency on our Origin2000); a total mean cost of 5.88 cycles per addition.

Parallel memory access can be enforced by having one loop that iterates two cursors through the buf[size] array (see the middle column of Figure 3.14). This causes 2 parallel 128 byte (=32 integer) L2 cache line fetches from memory per 32 iterations, for a total of 64 additions. On the R10000, the measured maximum memory bandwidth of the bus is 555MB/s, so fetching two 128-byte cache lines in parallel costs only 112 cycles (instead of 124 + 124). The mean cost per addition is hence 2 + 112/64 = 3.75 cycles.

It is important to note that parallel memory access is achieved only if the ability of the CPU to execute multiple instructions speculatively spans multiple memory references in the application code. In other words, the parallel effect disappears if there is too much CPU work between two memory fetches (more than 124 cycles on the R10000) or if the instructions are interdependent, causing a CPU stall before reaching the next memory reference. For database algorithms this means that random access operations like hashing will not profit from parallel memory access, as following a linked list (hash bucket chain) causes one iteration to depend on the previous; hence a

[6] As each iteration of our loop consists of a memory load (buf[i]), an integer addition (of "total" with this value), an integer increment (of i), a comparison, and a branch, the R10000 manual suggests a total cost of minimally 6 cycles. However, due to the speculative execution in the R10000 processor, this is reduced to 2 cycles on the average.

memory miss will block execution. Only sequential algorithms with CPU processing costs less than the memory latency will profit, like in the simple scan experiment from Figure 3.7. This experiment reaches optimal parallel bandwidth when the stride is equal to this L2 cache line size. As each loop iteration then requests one subsequent cache line, modern CPUs will have multiple memory loads outstanding, executing them in parallel. Results are summarized at the bottom of Table 3.2, showing the parallel effect to be especially strong on the Origin2000, the PentiumIII, and the Athlon. In other words, if the memory access pattern is *not* sequential (like in equi-join), the memory access penalty paid on these systems is actually much higher than suggested by Figure 3.7, but determined by the latencies from Table 3.2.

3.4.2 Prefetched Memory Access

Computer systems with a non-blocking cache can shadow memory latency by performing a memory fetch well before it is actually needed. CPUs like the R10000, the PentiumIII, the Athlon, and the newer SPARC Ultra2 models have special *prefetching instructions* for this purpose. These instructions can be thought of as memory load instructions that do not deliver a result. Their only side effect is a modification of the status of the caches. Mowry describes compiler techniques to generate these prefetching instructions automatically [Mow94]. These techniques optimize array accesses from within loops when most loop information and dependencies are statically available, and as such are very appropriate for scientific code written in FORTRAN. Database code written in C/C++, however, does not profit from these techniques as even the most simple table scan implementation will typically result in a loop with both a dynamic stride and length, as these are (dynamically) determined by the width and length of the table that is being scanned. Also, if table values are compared or manipulated within the loop using a function call (e.g., comparing two values for equality using a C function looked up from some ADT table, or a C++ method with late binding), the unprotected pointer model of the C/C++ languages forces the compiler to consider the possibility of side effects from within that function; eliminating the possibility of optimization.

In order to provide the opportunity to still enforce memory prefetching in such situations, the MipsPRO compiler for the R10000 systems of Silicon Graphics allows passing of explicit prefetching hints by use of pragma's, as depicted in the rightmost column of Figure 3.14. This pragma tells the compiler to request the next cache line once in every 32 iterations. Such a prefetch-frequency is generated by the compiler by applying loop unrolling (it unrolls the loop 32 times and inserts one prefetch instruction). By hiding the memory prefetch behind 64 cycles of work, the mean cost per addition in this routine is reduced to 2 + ((124-64)/32) = 3.88 cycles. Optimal performance is achieved in this case when prefetching two cache lines ahead every 32 iterations (#**prefetch** buf[i+64] **freq**=32). The 124 cycles of latency are then totally hidden behind 128 cycles of CPU work, and a new cache line is requested every 64 cycles. This setting effectively combines prefetching with parallel memory access (two cache lines in 128 cycles instead of 248), and reduces the mean cost per addition to the minimum 2 cycles; three times faster than the simple approach.

3.4.3 Future Hardware Features

In spite of memory latency staying constant, hardware manufacturers have been able to increase memory bandwidth in line with the performance improvements of CPUs, by working with ever wider lines in the L1 and L2 caches. As cache lines grew wider, buses also did. The latest Sun UltraII workstations, for instance, have a 64-byte L2 cache line which is filled in parallel using a 576 bits wide PCI bus (576 = 64*8 plus 64 bits overhead). The strategy of doubling memory bandwidth by doubling the number of DRAM chips and bus lines is now seriously complicating system board design. The Rambus [Ram96] memory standard eliminates this problem by providing an "protocol-driven memory bus". Instead of designating one bit in the bus for one bit of data transported to the cache line, this new technology serializes the DRAM data into packets using a protocol and sends these packets over a thin (16-bit) bus that runs at very high speeds (up to 800MHz). While this allows for continued growth in memory bandwidth, it does not provide the same perspective for memory latency, as Rambus still needs to access DRAM chips, and there will still be the relatively long distance for the signals to travel between the CPU and these memory chips on the system board; both factors ensuring a fixed startup cost (latency) for any memory traffic.

A radical way around the high latencies mandated by off-CPU DRAM systems is presented in the proposal to integrate DRAM and CPU in a single chip called IRAM (Intelligent RAM) [PAC+97]. Powerful computer systems could then be built using many such chips. Finding a good model for programming such a highly parallel systems seems one of the biggest challenges of this approach. Another interesting proposal worth mentioning here has been "smarter memory" [MKW+98], which would allow the programmer to give a "cache-hint" by specifying the access pattern that is going to be used on a memory region in advance. This way, the programmer is no longer obliged to organize his data structures around the size of a cache line. Instead, the cache adapts its behavior to the needs of the application. Such a configurable system is in some sense a protocol-driven bus system, so Rambus is a step in this direction. However, both configurable memory access and IRAM have not yet been implemented in custom hardware, let alone in OS and compiler tools that would be needed to program them usefully.

Recent developments concerning memory caches are to move the L2 cache closer to the CPU, either locate it on the same multi-chip module (e.g., Intel's first PentiumIII "Katmai", or AMD's first Athlon generation) or even include it on the CPU's die (e.g., Intel's latest PentiumIII "Coppermine", or AMD's latest Athlon "Thunderbird"). While reducing L2 latency — the L2 caches now operate at half or even full CPU speed — these trends do not reduce the memory latency. Further, on-chip caches are usually smaller than off-chip caches and hence provide even less potential to avoid memory accesses. Similarly, additional L3 caches — although increasing the total cache capacity — cannot reduce memory latency, but rather might even increase it due to an increased management overhead.

Concerning CPU technology, it is anticipated [Sem97] that the performance advances dictated by Moore's law [Moo65] will continue well into the millennium.

However, performance increase will also be brought by more parallelism within the CPU. The upcoming IA-64 architecture has a design called Explicitly Parallel Instruction Computing (EPIC) [ACM+98], which allows instructions to be combined in bundles, explicitly telling the CPU that they are independent. The IA-64 is specifically designed to be scalable in the number of functional units, so while newer versions are released, more and more parallel units will be added. This means that while current PC hardware uses less parallel CPU execution than the RISC systems, this will most probably change in the new 64-bit PC generation.

Summarizing, we have identified the following ongoing trends in modern hardware:

- CPU performance keeps growing with Moore's law for years to come.
- A growing part of this performance increase will come from parallelism within the CPU.
- New bus technology will provide sufficient growth in memory bandwidth.
- Memory latency will not improve significantly.

This means that the failure of current DBMS technology to properly exploiting memory and CPU resources of modern hardware [ADHW99, KPH+98, BGB98, TLPZT97] will grow worse. Modern database architecture should therefore take these new hardware issues into account. With this motivation, we investigate in the following new approaches to large main-memory equi-joins, that specifically aim at optimizing resource utilization of modern hardware.

Chapter 4

Generic Database Cost Models for Hierarchical Memory Systems

Accurate prediction of operator execution time is a prerequisite for database query optimization. Although extensively studied for conventional disk-based DBMSs, cost modeling in main-memory DBMSs is still an open issue. Recent database research has demonstrated that memory access is more and more becoming a significant—if not the major—cost component of database operations. If used properly, fast but small cache memories—usually organized in cascading hierarchy between CPU and main memory—can help to reduce memory access costs. However, they make the cost estimation problem more complex.

In this chapter, we propose a generic technique to create accurate cost functions for database operations. We identify a few basic memory access patterns and provide cost functions that estimate their access costs for each level of the memory hierarchy. The cost functions are parameterized to accommodate various hardware characteristics appropriately. Combining the basic patterns, we can describe the memory access patterns of database operations. The cost functions of database operations can automatically be derived by combining the basic patterns' cost functions accordingly.

To validate our approach, we performed experiments using our DBMS prototype Monet. The results presented here confirm the accuracy of our cost models for different operations.

Aside from being useful for query optimization, our models provide insight to tune algorithms not only in a main-memory DBMS, but also in a disk-based DBMS with a large main-memory buffer cache.

4.1 Related Work and Historical Development

Database cost models provide the foundation for query optimizers to derive an efficient execution plan. Such models consist of two parts: a logical and a physical component. The former is geared toward estimation of the data volumes involved. Usually, statistics about the data stored in the database are used to predict the amount of data that each operator has to process. The underlying assumption is that a query plan that has to process less data will also consume less resources and/or take less time to be evaluated. The logical cost component depends only on the data stored in the database, the operators in the query, and the order in which these operators are to be evaluated (as specified by the query execution plan). Hence, the logical cost component is independent of the algorithm and/or implementation used for each operator.

The problem of (intermediate) result size estimation has been intensively studied in literature (cf., Section 2.2). In this thesis, we focus on the physical cost component. Therefore, we assume a perfect oracle to predict the data volumes.

Given the data volumes, the physical cost component is needed to discriminate the costs of the various algorithms and implementations of each operator. The query optimizer uses this information to choose the most suitable algorithm and/or implementation for each operator.

Given the fact that disk-access used to be the predominant cost factor, early physical cost functions just counted the number of I/O operations to be executed by each algorithm [Gra93]. Any operation that loads a page from disk into the in-memory buffer pool or writes a page from the buffer back to disk is counted as an I/O operation. However, disk systems depict significant differences in cost (in terms of time) per I/O operation depending on the access pattern. Sequentially reading or writing consecutive pages causes less cost per page than accessing scattered pages in a random order. Hence, more accurate cost models discriminate between random and sequential I/O. The cost for sequential I/O is calculated as the data volume[1] divided by the I/O bandwidth. The cost for random I/O additionally considers the seek latency per operation.

With memory chips dropping in price while growing in capacity, main memory sizes grow as well. Hence, more and more query processing work is done in main memory, trying to minimize disk access as far as possible in order to avoid the I/O bottleneck. Consequently, the contribution of pure CPU time to the overall query evaluation time becomes more important. Cost models are extended to model CPU costs, usually in terms of CPU cycles (scored by the CPU's clock speed to obtain the elapsed time).

CPU cost used to cover memory access costs [LN96, WK90]. This implicitly assumes that main memory access costs are uniform, i.e., independent of the memory address being accessed and the order in which different data items are accessed. However, recent database research has demonstrated that this assumption does not hold (anymore) [ADHW99, BMK99]. With hierarchical memory systems being used, access latency varies significantly, depending on whether the requested data can be

[1] i.e., number of sequential I/O operations multiplied by the page size

found in (any) cache, or has to be fetch from main memory. The state (or contents) of the cache(s) in turn depends on the applications' access patterns, i.e., the order in which the required data items are accessed. Furthermore, while CPU speed is continuously experiencing an exponential growth, memory latency has hardly improved over the last decade.[2] Our detailed analysis of these issues in Section 3.2 comes to the conclusion that memory access has become a significant cost factor—not only for main memory databases—which cost models need to reflect.

In query execution, the memory access issue has been addressed by designing new cache-conscious data structures [RR99, RR00, ADHS01] and algorithms [SKN94, MBK00b]. On the modeling side, however, nothing has been published yet considering memory access appropriately.

4.2 Outline

In this chapter, we address the problem of how to model memory access costs of database operators appropriately. As it turns out to be quite complicated to derive proper memory access cost functions for various operations, we develope a new technique to automatically derive such cost functions. The basic idea is to describe the data access behavior of an algorithm in terms of a combination of basic access patterns (such as ”sequential”or ”random”). The actual cost function is then obtained by combining the patterns' cost functions (as derived in this chapter) appropriately. Using a unified hardware model that covers the cost-related characteristics of both main memory and disk access, it is straight forward to extend our approach to consider I/O cost as well. Gathering I/O and memory cost models into a single common framework is a new approach that simplifies the task of generating accurate cost functions.

Section 4.3 presents a simplified abstract representation of data structures and identifies a number of basic access patterns to be performed on such data structures. Equipped with these tools, we show how to specify the data access patterns of database algorithms by combining basic patterns. In Section 4.4, we derive the cost function for our basic access patterns and Section 4.5 provides rules how to obtain the cost functions of database algorithms from their representation introduced in Section 4.3. Section 4.7 contains some experimental results validating the obtained cost functions and Section 4.8 will draw some conclusions.

4.3 The Idea

Our recent work on main-memory database algorithms suggests that memory access cost can be modeled by estimating the number of cache misses **M** and scoring them with their respective miss latency l [MBK02]. This approach is similar to the one used for detailed I/O cost models. The hardware discussion in Section 3.1 shows, that

[2]Wider busses and raised clock speeds, such as with DDR-SDRAM or RAMBUS, help to keep memory bandwidth growing at almost the pace of CPU speed, however, these techniques do not improve memory access latency. See also Section 3.2.

also for main-memory access, we have to distinguish between sequential and random access patterns. However, in contrary to disk access, we now have multiple levels of cache with varying characteristics. Hence, the challenge is to predict the number and kind of cache misses *for all cache levels*. Our hypothesis is, that we can treat all cache levels individually, though equally, and calculate the total cost as the sum of the cost for all levels:

$$T_{\text{Mem}} = \sum_{i=1}^{N} (\mathbf{M}_i^{\text{s}} \cdot l_i^{\text{s}} + \mathbf{M}_i^{\text{r}} \cdot l_i^{\text{r}}). \tag{4.1}$$

With the hardware modeled as described in Section 3.1 and the hardware parameters measured by our calibration tool (see Section 3.3), the remaining challenge is to estimate the number and kind of cache misses per cache level for various database algorithms. The task is similar to estimating the number and kind of I/O operations in traditional cost models. However, our goal is to provide a generic technique for predicting cache miss rates of various database algorithms. Nevertheless, we want to sacrifice as little accuracy as possible to this generalization.

To achieve the generalization, we introduce two abstractions. Our first abstraction is a unified description of data structures. We call it *data regions*. The second are *basic data access patterns*. Both of them are driven by the goal to keep the models as simple as possible, but as detailed as necessary. Hence, we try to ignore any details that are not significant for our purpose (predicting cache miss rates) and only focus on the relevant parameters. The following paragraphs will present both abstractions in detail.

4.3.1 Data Regions

We model data structures as *data regions*. $\mathbb{D}$ denotes the set of data regions. A data region $R \in \mathbb{D}$ consists of $|R|$ *data items* of size $\underline{\overline{R}}$ (in bytes). We call $|R|$ the *length* of region R, $\underline{\overline{R}}$ its *width* and $||R|| = |R| \cdot \underline{\overline{R}}$ its *size*. Further, we define the *number of cache lines covered by* R as $|R|_Z = \lceil ||R||/Z \rceil$, and the *number of data items that fit in the cache* as $|C|_{\underline{\overline{R}}} = \left\lceil C/\underline{\overline{R}} \right\rceil$.

A (relational) database table is hence represented by a region R with $|R|$ being the table's cardinality and $\underline{\overline{R}}$ being the tuple size (or width). Similarly, more complex structures like trees are modeled by regions with $|R|$ representing the number of nodes and $\underline{\overline{R}}$ representing the size (width) of a single node.

4.3.2 Basic Access Patterns

Data access patterns vary in their referential locality and hence in their cache behavior. Thus, not only the cost (latency) of cache misses depend on the access pattern, but also the number of cache misses that occur. Each database algorithm describes a different data access pattern. This means, each algorithm requires an individual cost function to predict its cache misses. Deriving each cost function ”by hand”is not only exhaustive and time consuming, but also error-prone. Our hypothesis is that we only need to specify the cost functions of a few basic access patterns. Given these basic patterns and their cost functions, we could describe the access patterns of database operations as combinations of basic access patterns, and derive the resulting cost functions automatically.

In order to identify the relevant basic access patterns, we first have to analyze the data access characteristics of database operators. We classify database operations according to the number of operands.

Unary operators—such as, e.g., table scan, selection, projection, sorting, hashing, aggregation, or duplicate elimination—read data from one input region and write data to one output region. Data access can hence be modeled by two cursors, one for the input and one for the output. The input cursor traverses the input region sequentially. For table scan, selection, and projection, the output cursor also simply progresses sequentially with each output item. When building a hash table, the output cursor ”hops back and forth”in a non-sequential way. In practice, the actual pattern is not completely random, but rather depends on the physical order and attribute value distribution of the input data as well as on the hash function. In our case, i.e., knowing only the algorithm, but not the actual data, it is not possible to make more accurate (and usable) assumptions about the pattern described by the output cursor. Hence, we assume that the output region is accessed in a completely random manner. This assumption should not be too bad, as a ”good”hash function typically destroys any sorting order and tends/tries to level out skew data distributions.

Sort algorithms typically perform a more complicated data access pattern. In Section 4.7.2, we will present quick-sort as an example to demonstrate how such patterns can be specified as combinations of basic patterns. Aggregation and duplicate elimination are usually implemented using sorting or hashing. Thus, they incur the respective patterns.

Though also a unary operation, data partitioning takes a separate role. Again, the input region is traversed sequentially. However, modeling the output cursor’s access pattern as purely random is too simple. In fact, we can do better. Suppose, we want to partition the input region into *m* output regions. Then, we know that the access within each region is sequential. Hence, we model the output access as a *nested* pattern. Each region is a separate *local cursor*, performing a sequential pattern. A single *global cursor* hops back and forth between the regions. Similar to the hashing scenario described before, the order in which the different region-cursors are accessed—i.e., the global pattern—depends on the partitioning criterion (e.g., hash- or range-based) and the physical order and attribute value distribution of the input data. Again, it is not possible to model these dependencies in a general way without detailed knowledge about the actual data to process. Purely from the algorithm, we can only deduce a random order.

Concerning binary operations, we focus our discussion on join. The appropriate treatment of union, intersection and set-difference can be derived respectively. Binary operators have two inputs and a single output. In most cases, one input—we call it *left* or *outer* input—is traversed sequentially. Access to the other—*right* or *inner*—input depends on the algorithm and the data of the left input. A nested loop join performs a complete sequential traversal over the whole inner input for each outer data item. A merge join—assuming both inputs are already sorted—sequentially traverses the inner

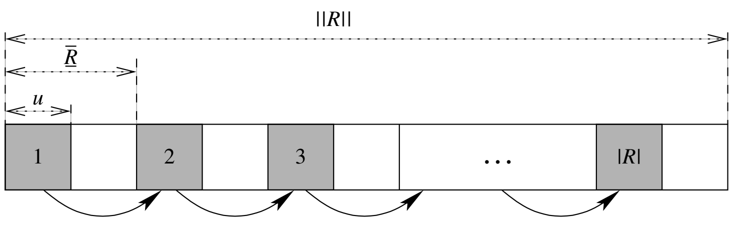


Figure 4.1: Single Sequential Traversal: s_trav(R, u)

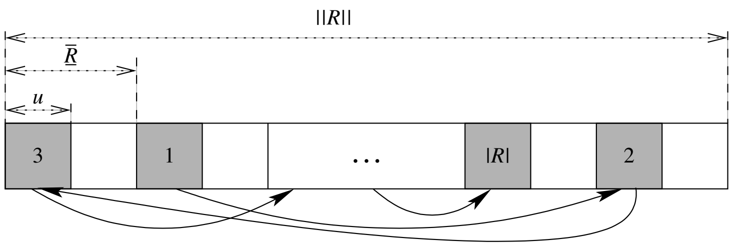


Figure 4.2: Single Random Traversal: r_trav(R, u)

input once while the outer input is traversed. A hash join—provided there is already a hash table on the inner input—performs an ”un-ordered”access pattern on the inner input's hash table. As discussed above, we assume a uniform random access.

From this discussion, we identify the following basic access patterns as eminent in the majority of relational algebra implementations. Let $R \in \mathbb{D}$ be a data region.

single sequential traversal: s_trav$(R[, u])$

A sequential traversal sequentially sweeps over R, accessing each data item in R exactly once. The optional parameter u gives the number of bytes that are actually used of each data item. If not specified, we assume that all bytes are used, i.e., $u = \overline{\underline{R}}$. If specified, we require $0 < u \leq \overline{\underline{R}}$. u is used to model the fact that an operator, e.g., an aggregation or a projection (either as separate operator or in-lined with another operator), accesses only a subset of its input's attributes. For simplicity of presentation, we assume that we always access u consecutive bytes. Though not completely accurate, this is a reasonable abstraction in our case.[3] Figure 4.1 shows a sample sequential traversal.

repetitive sequential traversal: rs_trav$(r, d, R, [, u])$

A repetitive sequential traversal performs r sequential traversals over R after another. d specifies, whether all traversals sweep over R in the same direction, or whether subsequent traversals go in alternating directions. The first case—*uni-directional*—is specified by $d =$ `uni`. The second case—*bi-directional*—is

[3]In case the u bytes are rather somehow spread across the whole item width $\overline{\underline{R}}$, say as k times u' bytes $(k \cdot u' = u)$, one can replace s_trav(R, u) by s_trav(R', u') with $\overline{\underline{R'}} = \overline{\underline{R}}/k$ and $|R'| = |R| \cdot k$.

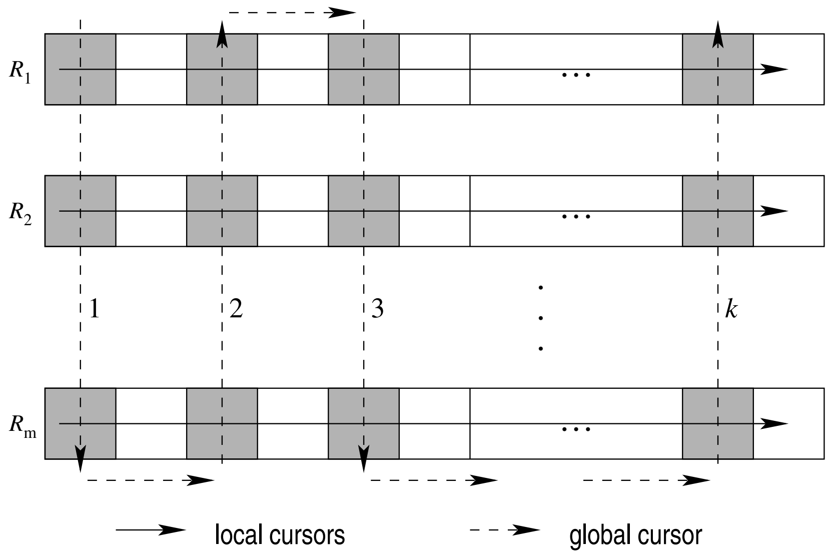


Figure 4.3: Interleaved Multi-Cursor Access: nest(R, m, s_trav(R, u), `seq`, `bi`)

specified by d = `bi`.

single random traversal: r_trav($R[, u]$)
Like a sequential traversal, a random traversal accesses each data item in R exactly once, reading or writing u bytes. However, the data items are not accessed in the order they are stored, but rather randomly. Figure 4.2 depicts a sample random traversal.

repetitive random traversal: rr_trav($r, R[, u]$)
A repetitive random traversal performs r random traversals over R after another. We assume that the permutation orders of two subsequent traversals are independent of each other. Hence, there is no point in discriminating uni-directional and bi-directional accesses, here. Therefore, we omit parameter d.

random access: r_acc($r, R[, u]$)
Random access hits r randomly chosen data items in R after another. We assume, that each data item may be hit more than once, and that the choices are independent of each other. Even with $r \geq |R|$ we do not require that each data item is accessed at least once.

interleaved multi-cursor access: nest($R, m, \mathcal{P}, O[, D]$)
A nested multi-cursor access models a pattern where R is divided into m (equal-sized) sub-regions. Each sub-region has its own local cursor. All local cursors perform the same basic pattern, given by $\mathcal{P}$. O specifies, whether the global cursor picks the local cursors randomly (O = `ran`) or sequentially (O = `seq`). In

the latter case, D specifies, whether all traversals of the global cursor across the local cursors use the same direction ($D = \texttt{uni}$), or whether subsequent traversals use alternating directions ($D = \texttt{bi}$). Figure 4.3 shows a sample interleaved multi-cursor access.

A similar idea has been used by Chou and DeWitt in their buffer management algorithm *DBMIN* [CD85]. DBMIN is based on a model for relational query behavior called *query locality set model* (*QLSM*). QLSM is founded on the observation that basic database operations (like scans, index scans, joins, etc.) could be characterized by a limited number of reference patterns to database pages. Chou and DeWitt propose three classes of reference patterns: sequential, random, and hierarchical. DBMIN exploits the information provided by QLSM to choose the most suitable pages replace strategy and estimate the proper buffer size to be used for each relation in a given query.

4.3.3 Compound Access Patterns

Database operations access more than one data region, usually at least their input(s) and their output. This means, they perform more complex data access patterns than the basic ones we introduced in the previous section. In order to model these complex patterns, we now introduce *compound data access patterns*. Unless we need to explicitly distinguish between basic and compound data access patterns, we refer to both as data access patterns, or simply patterns. We use $\mathbb{P}_b$, $\mathbb{P}_c$, and $\mathbb{P} = \mathbb{P}_b \cup \mathbb{P}_c$ to denote the set of basic access patterns, compound access patterns, and all access patterns, respectively. We require $\mathbb{P}_b \cap \mathbb{P}_c = \emptyset$.

Be $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ ($p > 1$) data access patterns. There are two principle ways to combine two or more patterns. Either the patterns are executed *one after the other* or they are executed *concurrently*. We call the first combination *sequential execution* and denote it by operator $\oplus : \mathbb{P} \to \mathbb{P}$; the second combination represents *concurrent execution* and is denoted by operator $\odot : \mathbb{P} \to \mathbb{P}$. The result of either combination is again a (compound) data access pattern. Hence, we can apply $\oplus$ and $\odot$ repeatedly to describe complex patterns. By definition, $\odot$ is commutative, while $\oplus$ is not. In case both $\odot$ and $\oplus$ are used to describe a complex pattern, $\odot$ has precedence over $\oplus$, i.e.,

$$\mathcal{P}_1 \odot \mathcal{P}_2 \odot \mathcal{P}_3 \oplus \mathcal{P}_4 \odot \mathcal{P}_5 \oplus \mathcal{P}_6 \;\equiv\; ((\mathcal{P}_1 \odot \mathcal{P}_2 \odot \mathcal{P}_3) \oplus (\mathcal{P}_4 \odot \mathcal{P}_5) \oplus \mathcal{P}_6).$$

We use bracketing to overrule these assumptions or to avoid ambiguity. Further, we use the following notation to simplify complex terms where necessary and appropriate:

$$\circledcirc \in \{\oplus, \odot\} : \quad \mathcal{P}_1 \circledcirc \ldots \circledcirc \mathcal{P}_p \;\equiv\; \circledcirc(P_1, \ldots, \mathcal{P}_p) \;\equiv\; \circledcirc|_{q=1}^{p}(\mathcal{P}_q).$$

Table 4.1 gives some examples how to describe the access patterns of some typical database algorithms as compound patterns. For convenience, some re-occurring compound access patterns are assigned a new name.

algorithm	pattern description	name
$W \leftarrow \mathit{select}(U)$	$\text{s\_trav}(U) \odot \text{s\_trav}(W)$	
$W \leftarrow \mathit{nested\_loop\_join}(U, V)$	$\text{s\_trav}(U) \odot \text{rs\_trav}(\|U\|, \texttt{uni}, V) \odot \text{s\_trav}(W)$	$=: \text{nl\_join}(U, V, W)$
$W \leftarrow \mathit{zick\_zack\_join}(U, V)$[a]	$\text{s\_trav}(U) \odot \text{rs\_trav}(\|U\|, \texttt{bi}, V) \odot \text{s\_trav}(W)$	
$V' \leftarrow \mathit{hash\_build}(V)$	$\text{s\_trav}(V) \odot \text{r\_trav}(V')$	$=: \text{build\_hash}(V, V')$
$W \leftarrow \mathit{hash\_probe}(U, V')$	$\text{s\_trav}(U) \odot \text{r\_acc}(\|U\|, V') \odot \text{s\_trav}(W)$	$=: \text{probe\_hash}(U, V', W)$
$W \leftarrow \mathit{hash\_join}(U, V)$	$\text{build\_hash}(V, V') \oplus \text{probe\_hash}(U, V', W)$	$=: \text{h\_join}(U, V, W)$
$\{U_j\}\|_{j=1}^{m} \leftarrow \mathit{cluster}(U, m)$	$\text{s\_trav}(U) \odot \text{nest}(\{U_j\}\|_{j=1}^{m}, m, \text{s\_trav}(U_j), \texttt{ran})$	$=: \text{part}(U, m, \{U_j\}\|_{j=1}^{m})$
$W \leftarrow \mathit{part\_nl\_join}(U, V, m)$	$\text{part}(U, m, \{U_j\}\|_{j=1}^{m}) \oplus \text{part}(V, m, \{V_j\}\|_{j=1}^{m})$ $\oplus\, \text{nl\_join}(U_1, V_1, W_1) \oplus \ldots \oplus \text{nl\_join}(U_m, V_m, W_m)$	
$W \leftarrow \mathit{part\_h\_join}(U, V, m)$	$\text{part}(U, m, \{U_j\}\|_{j=1}^{m}) \oplus \text{part}(V, m, \{V_j\}\|_{j=1}^{m})$ $\oplus\, \text{h\_join}(U_1, V_1, W_1) \oplus \ldots \oplus \text{h\_join}(U_m, V_m, W_m)$	

[a] nested-loop-join with alternating traversal direction on inner table, aka. "*boustrophedonism*"

Table 4.1: Sample Data Access Patterns ($U, V, V', W \in \mathbb{D}$)

Our hypothesis is, that we only need to provide an access pattern description as depicted in Table 4.1 for each operation we want to model. The actual cost function can then be created automatically, provided we know the cost functions for the basic patterns, and the rules how to combine them. To verify this hypothesis, we will now first estimate the cache miss rates of the basic access patterns and then derive rules how to calculate the cache miss rates of compound access patterns.

4.4 Deriving Cost Functions

In the following sections, N depicts the number of cache levels and i iterates over all levels: $i \in \{1, \ldots, N\}$. For better readability, we will omit the index i wherever we do not refer to a specific cache level, but rather to all or any.

4.4.1 Preliminaries

For each basic pattern, we need to estimate both sequential and random cache misses for each cache level. Given an access pattern $\mathcal{P} \in \mathbb{P}$, we describe the number of misses per cache level as pair

$$\vec{\mathbf{M}}_i(\mathcal{P}) = \langle \mathbf{M}_i^{\mathrm{s}}(\mathcal{P}), \mathbf{M}_i^{\mathrm{r}}(\mathcal{P}) \rangle \in \mathbb{N} \times \mathbb{N} \tag{4.2}$$

containing the number of sequential and random cache misses. Obviously, the random patterns cause only random misses, but no sequential misses. Consequently, we always set

$$\mathbf{M}_i^{\mathsf{s}}(\mathsf{T}) = 0 \qquad \text{for} \quad \mathsf{T} \in \{\mathsf{r\_trav}, \mathsf{rr\_trav}, \mathsf{r\_acc}\}.$$

Sequential traversals can achieve sequential latency (i.e., exploit full excess bandwidth), only if all the requirements listed in Section 3.1.2.2 are fulfilled. Sequential access is fulfilled by definition. The hardware requirements (non-blocking caches and super-scalar CPUs allowing speculative execution) are covered by the results of our calibration tool. In case these properties are not given, sequential latency will be the same as random latency. However, the pure existence of these hardware features is not sufficient to achieve sequential latency. Rather, the implementation needs to be able to exploit these features. Data dependencies in the code may keep the CPU from issuing multiple memory requests concurrently. It is not possible to deduce this information only from the algorithm without knowing the actual implementation. But even without data dependencies, multiple concurrent memory requests may hit the same cache line. In case the number of concurrent hits to a single cache line is lower than the maximal number of outstanding memory references allowed by the CPU, only one cache line is loaded at a time.[4] Though we can say how many subsequent references hit the same cache line (see below), we do not know how many outstanding memory references the CPU can handle without stalling.[5] Hence, it is not possible to automatically guess, whether a sequential traversal can achieve sequential latency or not. For this reason, we offer two variants of $\mathsf{s\_trav}$ and $\mathsf{rs\_trav}$. $\mathsf{s\_trav}^{\mathsf{s}}$ and $\mathsf{rs\_trav}^{\mathsf{s}}$ assume a scenario that can achieve sequential latency while $\mathsf{s\_trav}^{\mathtt{r}}$ and $\mathsf{rs\_trav}^{\mathtt{r}}$ do not. The actual number of misses is equal in both cases. However, in the first case, we get only sequential but no random misses, while the second case causes only random but no sequential misses:

$$\mathbf{M}_i^{\mathtt{r}}(\mathsf{T}^{\mathsf{s}}()) = \mathbf{M}_i^{\mathsf{s}}(\mathsf{T}^{\mathtt{r}}()) = 0 \quad \text{for} \quad \mathsf{T} \in \{\mathsf{s\_trav}, \mathsf{rs\_trav}\}.$$

Unless we need to explicitly distinguish between both variants, we will use $\mathsf{s\_trav}^x$ respectively $\mathsf{rs\_trav}^x$ to refer to both ($x \in \{\mathtt{s}, \mathtt{r}\}$). When describing the access pattern of a certain algorithm, we will use the variant that fits to the actual code.

4.4.2 Single Sequential Traversal

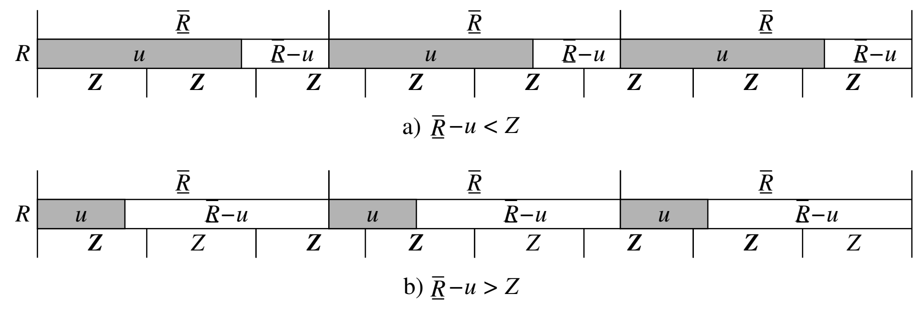
Be R a data region and $\mathcal{P} = \mathsf{s\_trav}^x(R, u)$ ($x \in \{\mathtt{s}, \mathtt{r}\}$) a sequential traversal over R. As mentioned above, we have

$$\mathbf{M}_i^{\mathtt{r}}(\mathsf{s\_trav}^{\mathsf{s}}(R, u)) = 0 \quad \text{and} \quad \mathbf{M}_i^{\mathsf{s}}(\mathsf{s\_trav}^{\mathtt{r}}(R, u)) = 0.$$

To calculate $\mathbf{M}_i^x(\mathsf{s\_trav}^x(R, u))$, we distinguish two cases: $\underline{\overline{R}} - u < Z$ and $\underline{\overline{R}} - u \geq Z$.

[4]For a more detailed discussion, we refer the interested reader to Section 3.2.

[5]Our calibration results can only indicate, whether the CPU can handle outstanding memory references without stalling, but not how many it can handle concurrently.



$\overline{\underline{R}}$
R
u
$\overline{\underline{R}}-u$
Z
a) $\overline{\underline{R}}-u < Z$
b) $\overline{\underline{R}}-u > Z$

Figure 4.4: Impact of $\overline{\underline{R}} - u$ on the Number of Cache Misses

Case $\overline{\underline{R}} - u < Z$. In this case, the gap between two adjacent accesses that is not touched at all is smaller than a single cache line. Hence, the cache line containing this gap is loaded to serve at least one of the two adjacent accesses (cf., Fig. 4.4a). Thus, during a sweep over R with $\overline{\underline{R}} - u < Z$ all cache lines coved by R have to be loaded, i.e.,

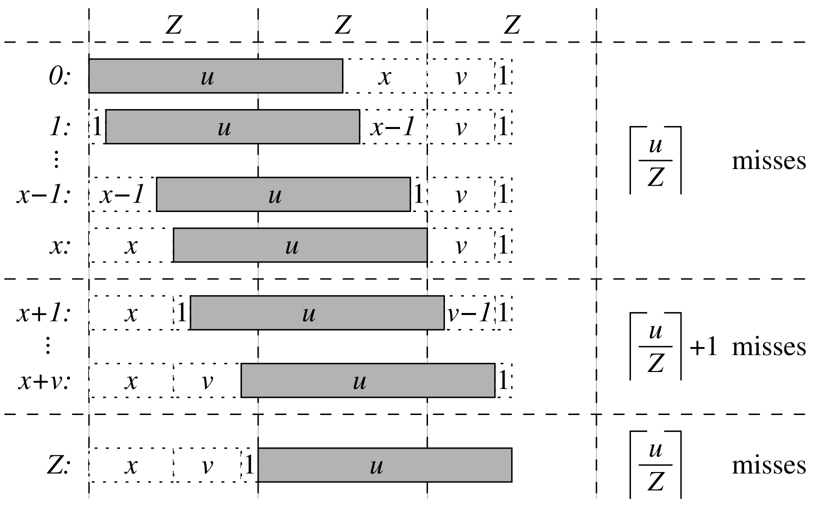
$$\mathbf{M}_i^x(\mathsf{s\_trav}^x(R,u)) = |R|_{Z_i}. \tag{4.3}$$

Case $\overline{\underline{R}} - u \geq Z$. In this case, the gap between two adjacent accesses that is not touched at all spans at least a complete cache line. Hence, not all cache lines coved by R have to be loaded during a sweep over R with $\overline{\underline{R}} - u \geq Z$ (cf., Fig. 4.4b). Further, no access can benefit from a cache line already loaded by a previous access to another spot. Thus, each access to an item in R requires at least $\left\lceil \frac{u}{Z} \right\rceil$ cache lines to be loaded. We get

$$\mathbf{M}_i^x(\mathsf{s\_trav}^x(R,u)) \geq |R| \cdot \left\lceil \frac{u}{Z_i} \right\rceil.$$

However, with $u > 1$ it may happen that — depending on the alignment of u within a cache line — one additional cache line has to be loaded per access. Figure 4.5 depicts such a scenario.

The actual alignment of each u in a sweep is determined by two parameters. First, it of course depends on the alignment of the first item in R, i.e., the alignment of R itself. Assuming a 1 byte access granularity, R can be aligned on Z places within a cache line. Second, $\overline{\underline{R}}$ determines whether all items in R are aligned equally, or whether their alignment changes throughout R. In case $\overline{\underline{R}}$ is a multiple of Z, all items in R are equally aligned as the first one. Otherwise, the alignment varies throughout R, but picking only $\frac{Z}{\gcd\{Z,\overline{\underline{R}}\}}$ out of the Z theoretically possible places. As we do not know anything about the alignment of R, there is no way of reasonably exploiting the information we just learned about the impact of $\overline{\underline{R}}$ on the alignment shift. Hence, all we can do is assuming that all Z possibilities occur equally often. All we need to do now, is count how many of these Z possibilities yield an additional cache miss. For


Z
Z
Z
0:
1:
x−1:
x:
x+1:
x+v:
Z:
u
x
v
1
x−1
v−1
$\left\lceil \frac{u}{Z} \right\rceil$ misses
$\left\lceil \frac{u}{Z} \right\rceil$ +1 misses
$\left\lceil \frac{u}{Z} \right\rceil$ misses

$$x = Z - (u \mathbin{\widehat{\text{mod}}} Z)$$

$$v = Z - 1 - x = (u-1) \bmod Z$$

Figure 4.5: Impact of Alignment on the Number of Cache Misses

convenience, we define

$$x \mathbin{\widehat{\text{mod}}} y = \begin{cases} y, & \text{if } x \bmod y = 0, \\ x \bmod y, & \text{else.} \end{cases}$$

When u is aligned on the first position (i.e., the first byte) in a cache line, no more than $\left\lceil \frac{u}{Z} \right\rceil$ cache lines have to be loaded to access whole u (cf., Fig. 4.5). In this case, the last $Z - (u \mathbin{\widehat{\text{mod}}} Z)$ bytes in the last cache line loaded for u are not used by u. Hence, shifting u's position by up to $Z - (u \mathbin{\widehat{\text{mod}}} Z)$ also does not require any additional cache miss. Only the remaining

$$\begin{aligned} & Z - 1 - (Z - (u \mathbin{\widehat{\text{mod}}} Z)) \\ &= (u \mathbin{\widehat{\text{mod}}} Z) - 1 \\ &= \begin{cases} Z - 1, & \text{if } u \bmod Z = 0 \\ (u \bmod Z) - 1, & \text{else} \end{cases} \\ &\overset{u>0}{=} (u-1) \bmod Z \end{aligned}$$

positions will yield one additional cache miss (cf., Fig. 4.5). Putting all pieces together, we get:

$$\mathbf{M}_i^x(\mathsf{s\_trav}^x(R,u)) = |R| \cdot \left(\left\lceil \frac{u}{Z_i} \right\rceil + \frac{(u-1) \bmod Z_i}{Z_i} \right). \tag{4.4}$$

Figure 4.6 demonstrates the impact of u on the number of cache misses. The points show the number of cache misses measured with various alignments. "$\mathsf{align} = 0$"and "$\mathsf{align} = -1$"make up the two extreme cases. In the first case, u is aligned on the first byte of a cache line; in the second case, u starts on the last byte of a cache line. "$\mathsf{average}$"depicts the average over all possible alignments. The dotted curve and the dashed curve represent Equations (4.5) and (4.3), respectively, which ignore u and assume that all $\underline{\overline{R}}$ bytes of each item are touched. The solid curve represents the identical Equations (4.4) and (4.6) which consider u. The graphs show, that u has a significant impact on the number of cache misses, and that our formulas correctly predict the average impact.

4.4.3 Single Random Traversal

Be R a data region and $\mathcal{P} = \mathsf{r\_trav}(R,u)$ a random traversal over R. As mentioned above, we have

$$\mathbf{M}_i^{\mathrm{s}}(\mathsf{r\_trav}(R,u)) = 0.$$

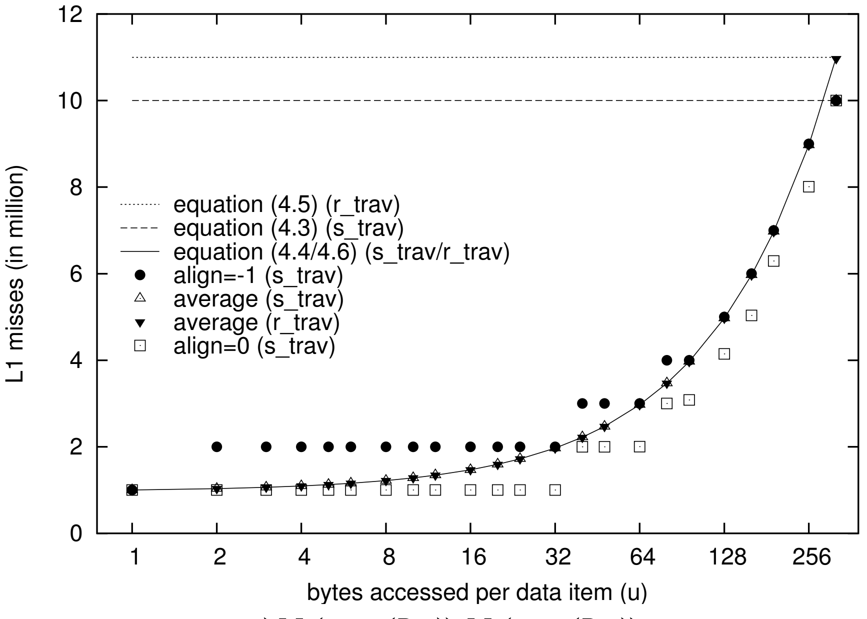
Like with sequential traversal, we distinguish two cases: $\underline{\overline{R}} - u < Z$ and $\underline{\overline{R}} - u \geq Z$.

Case $\underline{\overline{R}} - u < Z$. With the untouched gaps being smaller than cache line size, again all cache lines coved by R have to be accessed. Hence, $\mathbf{M}^{\mathrm{r}}(\mathcal{P}) \geq |R|_Z$. But due to the random access pattern, two locally adjacent accesses are not temporally adjacent. Thus, if $||R||$ exceeds the cache size, a cache line that serves two or more (locally adjacent) accesses may be replaced by another cache line before all accesses that require it actually took place. This in turn causes an additional cache miss, once the original cache line is accessed again. Of course, such additional cache misses only occur, once the cache capacity is exceeded, i.e., after $\min\left\{\#_i, |C_i|_{\underline{\overline{R}}}\right\}$ spots have been accessed. The probability that a cache line is removed from the cache although it will be used for another access increases with the size of R. In the worst case, each access causes an additional cache miss. Hence, we get

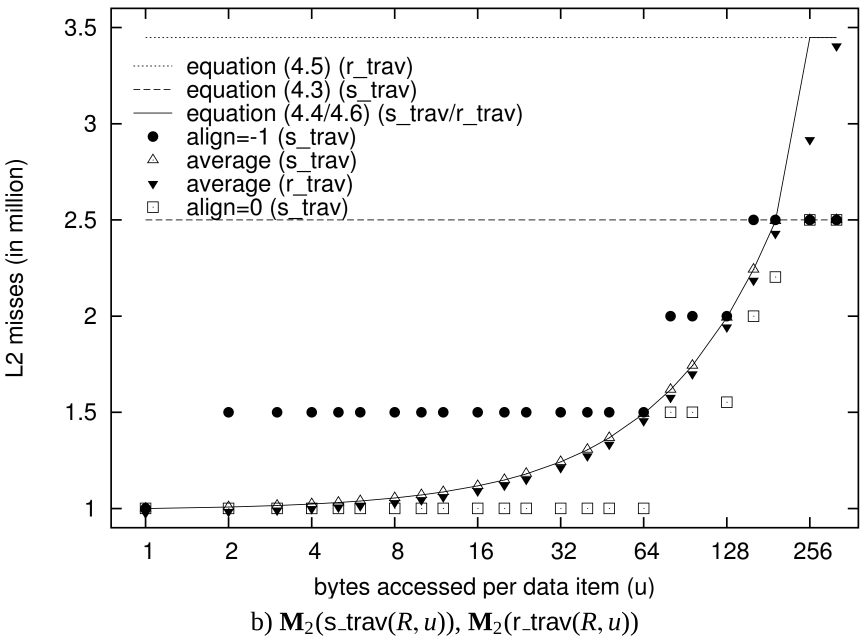
$$\begin{aligned}&\mathbf{M}_i^{\mathrm{r}}(\mathsf{r\_trav}(R,u)) \\ &\qquad = |R|_{Z_i} + \left(|R| - \min\left\{\#_i, |C_i|_{\underline{\overline{R}}}\right\}\right) \cdot \left(1 - \min\left\{1, \frac{C_i}{||R||}\right\}\right).\end{aligned} \tag{4.5}$$

Case $\underline{\overline{R}} - u \geq Z$. Each spot is touched exactly once, and as adjacent accesses cannot benefit from previously loaded cache lines, we get the same formula as for sequential access:

$$\mathbf{M}_i^{\mathrm{r}}(\mathsf{r\_trav}(R,u)) = |R| \cdot \left(\left\lceil \frac{u}{Z_i} \right\rceil + \frac{(u-1) \bmod Z_i}{Z_i} \right). \tag{4.6}$$



a) $\mathbf{M}_1$(s_trav(R, u)), $\mathbf{M}_1$(r_trav(R, u))



b) $\mathbf{M}_2$(s_trav(R, u)), $\mathbf{M}_2$(r_trav(R, u))

Figure 4.6: Impact of u and its Alignment on the Number of Cache Misses ($|R| = 1,000,000,\ \overline{\underline{R}} = 320\,\text{B},\ Z_1 = 32\,\text{B},\ Z_2 = 128\,\text{B}$)

4.4.4 Discussion

Comparing the final formulas for sequential and random traversals, we can derive the following relationships and invariants. Figure 4.7 visualizes some of the effects. The points represent the measured cache misses, while the lines represent the estimations of our formulas.

$$\begin{aligned}
\overline{\underline{R}} - u < Z_i \ \land\ \|R\| \le C_i \quad &\overset{(4.3,4.5)}{\Longrightarrow} \quad \mathbf{M}_i^x(\text{s\_trav}^x(R,u)) = \mathbf{M}_i^{\mathbf{r}}(\text{r\_trav}(R,u));\\
\overline{\underline{R}} - u < Z_i \ \land\ \|R\| > C_i \quad &\overset{(4.3,4.5)}{\Longrightarrow} \quad \mathbf{M}_i^x(\text{s\_trav}^x(R,u)) < \mathbf{M}_i^{\mathbf{r}}(\text{r\_trav}(R,u)).
\end{aligned}$$

With untouched gaps smaller than cache lines, random traversals cause as many misses as sequential traversals as long as R fits in the cache, but more, if R exceeds the cache (cf., Figure 4.7a vs. 4.7c & 4.7b vs. 4.7d).

$$\overline{\underline{R}} - u \ge Z_i \quad \overset{(4.4,4.6)}{\Longrightarrow} \quad \mathbf{M}_i^x(\text{s\_trav}^x(R,u)) = \mathbf{M}_i^{\mathbf{r}}(\text{r\_trav}(R,u)).$$

With untouched gaps larger than cache lines, random traversals cause as many misses as sequential traversals.

$$\begin{aligned}
\overline{\underline{R}} - u < Z_i \quad \overset{(4.3)}{\Longrightarrow} \quad &\mathbf{M}_i^x(\text{s\_trav}^x(R,u)) = \mathbf{M}_i^x(\text{s\_trav}^x(R',u'))\\
&\forall R', u' \text{ with } \|R'\| = \|R\| \ \land\ \overline{\underline{R'}} - u' < Z_i.
\end{aligned}$$

With untouched gaps smaller than cache lines, sequential traversals depend only on the size of R, but are invariant to varying item size (and hence number of items) and bytes touched per item (cf., Figure 4.7a & 4.7b).

$$\begin{aligned}
\overline{\underline{R}} - u < Z_i \ \land\ \|R\| \le C_i \quad \overset{(4.5)}{\Longrightarrow} \quad &\mathbf{M}_i^{\mathbf{r}}(\text{r\_trav}(R,u)) = \mathbf{M}_i^{\mathbf{r}}(\text{r\_trav}(R',u'))\\
&\forall R', u' \text{ with } \|R'\| = \|R\| \ \land\ \overline{\underline{R'}} - u' < Z_i;\\
\overline{\underline{R}} - u < Z_i \ \land\ \|R\| > C_i \quad \overset{(4.5)}{\Longrightarrow} \quad &\mathbf{M}_i^{\mathbf{r}}(\text{r\_trav}(R,u)) = \mathbf{M}_i^{\mathbf{r}}(\text{r\_trav}(R,u'))\\
&\forall u' \text{ with } \overline{\underline{R}} - u' < Z_i.
\end{aligned}$$

For random traversals, the invariance to item size holds only if R entirely fits in the cache (cf., Figure 4.7c & 4.7d).

$$\begin{aligned}
\overline{\underline{R}} - u \ge Z_i \quad \overset{(4.4/4.6)}{\Longrightarrow} \quad &\vec{\mathbf{M}}_i(\mathsf{T}(R,u)) = \vec{\mathbf{M}}_i(\mathsf{T}(R',u))\\
&\forall R' \text{ with } |R'| = |R| \ \land\ \overline{\underline{R'}} - u \ge Z_i,\\
&\mathsf{T} \in \{\text{s\_trav}^x, \text{r\_trav}\}.
\end{aligned}$$

With untouched gaps larger than cache lines, the number of misses of all traversals depend only on the number of items accessed and the number of bytes touched per item.

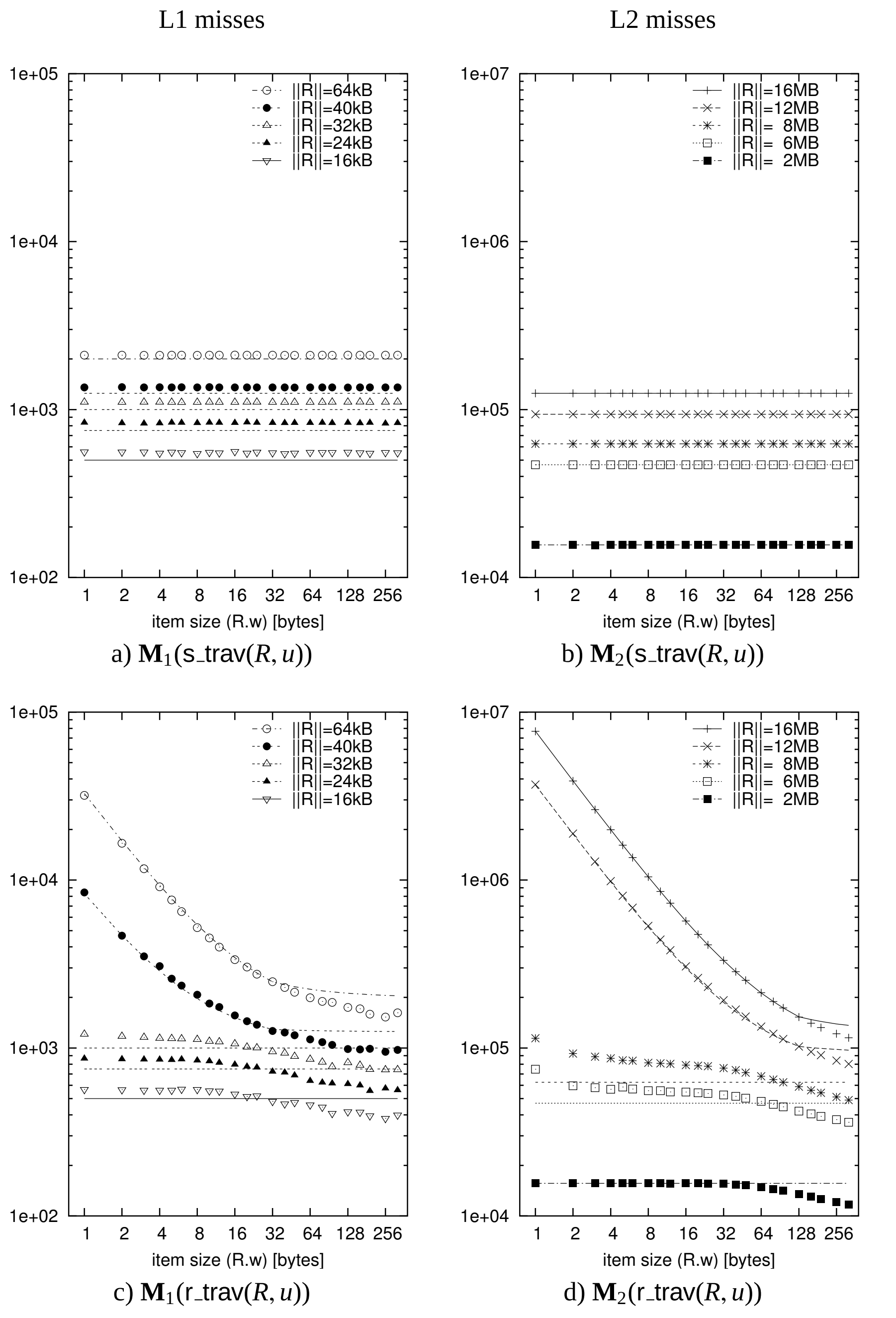


Figure 4.7: Impact of $||R||$ and $\overline{R}$ on the Number of Cache Misses
($u = \overline{R},\ C_1 = 32\,\text{kB},\ Z_1 = 32\,\text{B},\ C_2 = 8\,\text{MB},\ Z_2 = 128\,\text{B}$)

4.4.5 Repetitive Traversals

With repetitive traversals, cache re-usage comes into play. We assume initially empty caches.[6] Hence, the first traversal requires as many cache misses as estimated above. But the subsequent traversals may benefit from the data already present in the cache after the first access. We will analyze this in detail for both sequential and random traversals.

4.4.5.1 Repetitive Sequential Traversal

Be R a data region, $\mathcal{P} = \mathsf{rs\_trav}^x(r, d, R, u)$ a repetitive sequential traversal over R, and $\mathcal{P}' = \mathsf{s\_trav}^x(R, u)$ a single sequential traversal over R. Two parameters determine the caching behavior of a repetitive sequential traversal: the number $\mathbf{M}^x(\mathcal{P}')$ of cache lines touched during the first traversal and the direction d in which subsequent traversals sweep over R.

In case $\mathbf{M}^x(\mathcal{P}')$ is smaller than the total number of available cache lines, only the first traversal causes cache misses, loading all required data. All $r - 1$ subsequent traversals then just access the cache, causing no further cache misses.

In case $\mathbf{M}^x(\mathcal{P}')$ exceeds the number of available cache lines, the end of a traversal pushes the data read at the begin of the traversal out of the cache. If the next traversal then again starts at the begin of R, it cannot benefit from any data in the cache. Hence, with $d = \mathtt{uni}$, each sweep causes the full amount of cache misses. Only if a subsequent sweep starts where the previous one stopped, i.e., it traverses R in the opposite direction as its predecessor, it can benefit from the data stored in the cache. Thus, with $d = \mathtt{bi}$, only the first sweep causes the full amount of cache misses. The $r - 1$ remaining sweeps cause cache misses only for the fraction of R that does not fit into the cache.

In total, we get

$$\mathbf{M}_i^x(\mathsf{rs\_trav}^x(r, d, R, u)) \\ = \begin{cases} \mathbf{M}_i^x(\mathcal{P}'), & \text{if} \quad \mathbf{M}_i^x(\mathcal{P}') \le \#_i \\ r \cdot \mathbf{M}_i^x(\mathcal{P}'), & \text{if} \quad \mathbf{M}_i^x(\mathcal{P}') > \#_i \ \wedge \ d = \mathtt{uni} \\ \mathbf{M}_i^x(\mathcal{P}') + (r-1) \cdot (\mathbf{M}_i^x(\mathcal{P}') - \#_i), & \text{if} \quad \mathbf{M}_i^x(\mathcal{P}') > \#_i \ \wedge \ d = \mathtt{bi}. \end{cases} \tag{4.7}$$

4.4.5.2 Repetitive Random Traversal

Be R a data region, $\mathcal{P} = \mathsf{rr\_trav}(r, R, u)$ a repetitive random traversal over R, and $\mathcal{P}' = \mathsf{r\_trav}(R, u)$ a single random traversal over R. With random memory access, d is not defined, hence, we need to consider only $\mathbf{M}^{\mathrm{r}}(\mathcal{P}')$ to determine to which extend repetitive accesses can benefit from cached data.

When $\mathbf{M}^{\mathrm{r}}(\mathcal{P}')$ is smaller than the number of available cache lines, we get the same effect as above. Only the first sweep causes cache misses, loading all required data. All $r-1$ subsequent sweeps then just access the cache, causing no further cache misses.

[6] Section 4.5 will discuss how to consider pre-loaded caches.

In case $\mathbf{M}^{\mathrm{r}}(\mathcal{P}')$ exceeds the number of available cache lines, the most recently accessed data remains in the cache at the end of a sweep. Hence, there is a certain probability that the first accesses of the following sweep might re-use (some of) these # cache lines. This probability decreases as $\mathbf{M}^{\mathrm{r}}(\mathcal{P}')$ increases. We estimate the probability with $\#/\mathbf{M}^{\mathrm{r}}(\mathcal{P}')$.

Analogously to the sequential case, we get

$$\mathbf{M}^{\mathrm{r}}_i(\mathsf{rr\_trav}(r, R, u)) = \begin{cases} \mathbf{M}^{\mathrm{r}}_i(\mathcal{P}'), & \text{if} \quad \mathbf{M}^{\mathrm{r}}_i(\mathcal{P}') \le \#_i \\ \mathbf{M}^{\mathrm{r}}_i(\mathcal{P}') + (r-1)\cdot\left(\mathbf{M}^{\mathrm{r}}_i(\mathcal{P}') - \dfrac{\#_i}{\mathbf{M}^{\mathrm{r}}_i(\mathcal{P}')}\cdot \#_i\right), & \text{if} \quad \mathbf{M}^{\mathrm{r}}_i(\mathcal{P}') > \#_i. \end{cases} \tag{4.8}$$

4.4.6 Random Access

Be R a data region and $\mathcal{P} = \mathsf{r\_acc}(r, R, u)$ a random access pattern on R. As in Section 4.4.3, we have

$$\mathbf{M}^{\mathrm{s}}_i(\mathsf{r\_acc}(r, R, u)) = 0.$$

In contrary to a single random traversal, where each data item of R is touched exactly once, we do not know exactly, how many distinct data items are actually touched with random access. However, knowing that there are r independent random accesses to the $|R|$ data items in R, we can estimate the average/expected number $\mathbf{I}$ of distinct data items that are indeed touched. Be E the number of all different outcomes of picking r times one of the $|R|$ data items allowing multiple accesses to each data item. Further be E_j the number of outcomes containing exactly $1 \le j \le \min\{r, |R|\}$ distinct data items. If we respect ordering, all outcomes are equally likely to occur, hence, we have

$$\mathbf{I}(\mathsf{r\_acc}(r, R, u)) = \frac{\sum_{j=1}^{\min\{r,|R|\}} E_j(r, |R|)\cdot j}{E(r, |R|)}$$

with

$$\sum_{j=1}^{\min\{r,|R|\}} E_j(r, |R|) = E(r, |R|).$$

Calculating E is straight forward:

$$E(r, |R|) = |R|^r.$$

Calculating E_j turns out to be a bit more difficult. Be

$$\binom{x}{y} = \frac{x!}{(x-y)!\cdot y!}$$

the *binomial coefficient*, i.e., the number of ways of picking y unordered outcomes from x possibilities. Further be

$$\left\{ {x \atop y} \right\} = \frac{1}{y!}\cdot \sum_{j=0}^{y-1} (-1)^j \cdot \binom{y}{j} \cdot (y-j)^x$$

the *Stirling number of second kind*, i.e., the number of ways of partitioning a set of x elements into y nonempty sets [Sti30]. Then, we have

$$E_j(r, |R|) = \binom{|R|}{j} \cdot \left\{ r \atop j \right\} \cdot j!.$$

First of all, there are $\binom{|R|}{j}$ ways to choose j distinct data items from the available $|R|$ data items. Then, there are $\left\{ r \atop j \right\}$ ways to partition the r accesses into j groups, one for each distinct data item. Finally, we have to consider all $j!$ permutations to get equally likely outcomes.

Knowing the number $\mathbf{I}$ of distinct data items that are touched by $\mathsf{r\_acc}(r, R, u)$ on average, we can now calculate the number $\mathbf{C}$ of distinct cache lines touched. Again, we distinguish two cases, depending on the size of the (minimal) untouched gaps between two adjacent accesses.

Case $\underline{\overline{R}} - u \geq Z$. With the (minimal) untouched gaps larger than cache line size, no cache line is used by more than one data item. Following the discussion in Section 4.4.2, we get

$$\mathbf{C}_i(\mathsf{r\_acc}(r, R, u)) = \mathbf{I}(\mathsf{r\_acc}(r, R, u)) \cdot \left(\left\lceil \frac{u}{Z_i} \right\rceil + \frac{(u-1) \bmod Z_i}{Z_i} \right).$$

Case $\underline{\overline{R}} - u \geq Z$. With the (minimal) untouched gaps smaller than cache line size, (some) cache lines might be used by more than one data item. In case all $\mathbf{I}$ touched data items are pair-wise adjacent, we get

$$\check{\mathbf{C}}_i(\mathsf{r\_acc}(r, R, u)) = \left\lceil \frac{\mathbf{I}(\mathsf{r\_acc}(r, R, u)) \cdot \overline{\underline{R}}}{Z_i} \right\rceil.$$

However, if $\mathbf{I} \ll |R|$, the actual untouched gap might still be larger than cache line size, hence

$$\begin{aligned} &\hat{\mathbf{C}}_i(\mathsf{r\_acc}(r, R, u)) \\ &\qquad = \min \left\{ \mathbf{I}(\mathsf{r\_acc}(r, R, u)) \cdot \left(\left\lceil \frac{u}{Z_i} \right\rceil + \frac{(u-1) \bmod Z_i}{Z_i} \right), |R|_{Z_i} \right\}. \end{aligned}$$

$\check{\mathbf{C}}$ is more likely with large $\mathbf{I}$, while $\hat{\mathbf{C}}$ is more likely with small $\mathbf{I}$. Hence, we calculate the average $\mathbf{C}$ as a linear combination of $\check{\mathbf{C}}$ and $\hat{\mathbf{C}}$:

$$\begin{aligned} \mathbf{C}_i(\mathsf{r\_acc}(r, R, u)) = {} & \frac{\mathbf{I}(\mathsf{r\_acc}(r, R, u))}{|R|} \cdot \check{\mathbf{C}}_i(\mathsf{r\_acc}(r, R, u)) \\ & + \left(1 - \frac{\mathbf{I}(\mathsf{r\_acc}(r, R, u))}{|R|} \right) \cdot \hat{\mathbf{C}}_i(\mathsf{r\_acc}(r, R, u)). \end{aligned}$$

Knowing the number **C** of distinct cache lines touched, we can finally calculate the number of cache misses. With r accesses spread over **I** distinct data items, each item is touched $r/\mathbf{I}$ times on average. Analogously to Equation (4.8), we get ($\mathcal{P} = \mathsf{r\_acc}(r, R, u)$)

$$\mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_acc}(r,R,u)) = \begin{cases} \mathbf{C}_i(\mathcal{P}), & \text{if} \quad \mathbf{C}_i(\mathcal{P}) \le \#_i \\ \mathbf{C}_i(\mathcal{P}) + \left(\dfrac{r}{\mathbf{I}(\mathcal{P})} - 1\right) \cdot \left(\mathbf{C}_i(\mathcal{P}) - \dfrac{\#_i}{\mathbf{C}_i(\mathcal{P})} \cdot \#_i\right), & \text{if} \quad \mathbf{C}_i(\mathcal{P}) > \#_i. \end{cases} \tag{4.9}$$

4.4.7 Interleaved Multi-Cursor Access

Be $R = \{R_j\}|_{j=1}^{m}$ a data region divided into $m \le |R|$ sub-regions R_j with

$$\underline{\overline{R_j}} = \underline{\overline{R}} \qquad \text{and} \qquad k = |R_j| = \frac{|R|}{m}. \tag{$*$}$$

Further be $\mathcal{P} = \mathsf{nest}(R, m, \mathsf{T}([r,]R_j, u), O, D)$ with $\mathsf{T} \in \{\mathsf{s\_trav}^x, \mathsf{r\_trav}, \mathsf{r\_acc}\}$ an interleaved multi-cursor access. We inspect local random access ($\mathsf{T} \in \{\mathsf{r\_acc}, \mathsf{r\_trav}\}$) and local sequential access ($\mathsf{T} = \mathsf{s\_trav}^x$) separately.

4.4.7.1 Local Random Access

With $\mathsf{T} \in \{\mathsf{r\_acc}, \mathsf{r\_trav}\}$, $\mathcal{P}$ behaves like a single traversal $\mathcal{P}' = \mathsf{T}'([m \cdot r,]R, u)$. For $k = 1$ (i.e., $m = |R|$), the new order is the original global order, otherwise, it is the original local order, i.e., we get

$$\mathsf{T}' = \begin{cases} \mathsf{s\_trav}^x, & \text{if} \quad k = 1 \ \wedge \ O = \mathtt{seq} \\ \mathsf{T}, & \text{else} \end{cases}$$

and consequently

$$\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}([r,]R_j, u), O, D)) = \vec{\mathbf{M}}_i(\mathsf{T}'([m \cdot r,]R, u)).$$

4.4.7.2 Local Sequential Access

For $\mathsf{T} = \mathsf{s\_trav}^x$, we distinguish three cases:

- $\underline{\overline{R}} - u > Z,$
- $\underline{\overline{R}} - u \le Z \ \wedge \ m \cdot \left\lceil \frac{u}{Z} \right\rceil \le \#,$
- $\underline{\overline{R}} - u \le Z \ \wedge \ m \cdot \left\lceil \frac{u}{Z} \right\rceil > \#.$

Case $\underline{\overline{R}} - u > Z$. In this case, $\mathcal{P}$ means $k = |R_j|$ times traversing across all R_j in order O. Hence, each traversal performs m accesses (one to each R_j). The distance between adjacent accesses within each traversal is $||R_j|| = k \cdot \underline{\overline{R}}$. We describe these traversals by $\mathsf{T}'(R', u)$ where R' is a data region with

$$|R'| = m \quad \text{and} \quad \underline{\overline{R'}} = ||R_j||, \tag{†}$$

and

$$\mathsf{T}' = \begin{cases} \mathsf{r\_trav}, & \text{if} \quad O = \mathtt{ran} \\ \mathsf{T}, & \text{else.} \end{cases}$$

As the non-touched gap between adjacent accesses within each R_j is larger than a cache line ($\underline{\overline{R}} - u > Z$), no cache line is shared by two or more accesses. Thus, the total number of cache misses is the sum of the cache misses caused by the k traversals:

$$\begin{aligned}
&\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}(R_j, u), O, D)) \\
&= \quad \sum_{h=1}^{k} \vec{\mathbf{M}}_i(\mathsf{T}'(R', u)) \\
&= \quad k \cdot \vec{\mathbf{M}}_i(\mathsf{T}'(R', u)) \\
&\overset{(4.4/4.6)}{=} \begin{cases} \left\langle \quad k \cdot |R'| \cdot \left(\left\lceil \frac{u}{Z_i} \right\rceil + \dfrac{(u-1) \bmod Z_i}{Z_i} \right), 0 \right\rangle, & \text{if} \quad \mathsf{T}' = \mathsf{s\_trav}^{\mathsf{s}} \\ \left\langle 0, k \cdot |R'| \cdot \left(\left\lceil \frac{u}{Z_i} \right\rceil + \dfrac{(u-1) \bmod Z_i}{Z_i} \right) \quad \right\rangle, & \text{else} \end{cases} \\
&\overset{(4.4/4.6,*,†)}{=} \quad \vec{\mathbf{M}}_i(\mathsf{T}'(R, u)).
\end{aligned}$$

Case $\underline{\overline{R}} - u \leq Z \quad \wedge \quad m \cdot \left\lceil \frac{u}{Z} \right\rceil \leq \#$. With the non-touched gaps being smaller than cache line size ($\underline{\overline{R}} - u \leq Z$), adjacent accesses within each R_j might shared a cache line, and hence benefit from previous accesses. With one traversal across all R_j touching less cache lines than there are in total ($m \cdot \left\lceil \frac{u}{Z} \right\rceil \leq \#$), the subsequent traversal does not have to reload the shared cache lines. Hence, the total number of cache misses is just the sum of all local patterns. Though these are sequential, a global random pattern will avoid sequential latency. We take this into account when defining T' and get

$$\begin{aligned}
&\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}(R_j, u), O, D)) \\
&= \quad \sum_{j=1}^{m} \vec{\mathbf{M}}_i(\mathsf{T}'(R_j, u)) \\
&= \quad m \cdot \vec{\mathbf{M}}_i(\mathsf{T}'(R_j, u)) \\
&\overset{(4.3,*)}{=} \quad \vec{\mathbf{M}}_i(\mathsf{T}'(R, u))
\end{aligned}$$

with

$$\mathsf{T}' = \begin{cases} \mathsf{s\_trav}^{\mathrm{r}}, & \text{if} \quad O = \mathtt{ran} \\ \mathsf{T}, & \text{else.} \end{cases}$$

Case $\overline{\underline{R}} - u \leq Z \quad \wedge \quad m \cdot \left\lceil \frac{u}{Z} \right\rceil > \#$. With one traversal across all R_j touching more cache lines than there are in total ($m \cdot \left\lceil \frac{u}{Z} \right\rceil > \#$), only $h = \#/\left\lceil \frac{u}{Z} \right\rceil < m$ of the m shared cache lines remain in the cache for potential re-use. The number h' of cache lines that is actually re-used depends on O and D and is calculated similarly as for the repetitive traversals in Section 4.4.5:

$$h'_i = \begin{cases} 0, & \text{if } O = \mathtt{seq} \wedge D = \mathtt{uni} \\ h_i, & \text{if } O = \mathtt{seq} \wedge D = \mathtt{bi} \\ \dfrac{h_i}{m} \cdot h_i, & \text{if } O = \mathtt{ran} \end{cases}$$

$$h_i = \#_i / \left\lceil \frac{u}{Z_i} \right\rceil.$$

Hence, the total number of cache misses is the same as in the previous case, plus the $m - h'$ cache lines that have to be reloaded during all but the first traversal. These additional misses cause random latency, i.e., we get

$$\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}(R_j, u), O, D)) = \vec{\mathbf{M}}_i(\mathsf{T}'(R, u)) + \vec{\mathbf{X}}_i$$

with

$$\vec{\mathbf{X}}_i = \langle 0, (k-1) \cdot (m - h'_i) \rangle \tag{‡}$$

and T' as before.

4.4.7.3 Summary

Gathering the results from all the different cases discussed above, we get

$$\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}([r,]R_j, u), O, D))$$
$$= \begin{cases} \vec{\mathbf{M}}_i(\mathsf{T}'([m \cdot r,]R, u)) + \vec{\mathbf{X}}_i, & \begin{array}{l} \text{if} \quad \mathsf{T} = \mathsf{s\_trav}^x \\ \wedge \ \overline{\underline{R}} - u < Z_i \\ \wedge \ m \cdot \left\lceil \frac{u}{Z_i} \right\rceil > \#_i \end{array} \\ \vec{\mathbf{M}}_i(\mathsf{T}'([m \cdot r,]R, u)), & \text{else} \end{cases} \tag{4.10}$$

with $\vec{\mathbf{X}}_i$ as in (‡) and

$$\mathsf{T}' = \begin{cases} \mathsf{s\_trav}^x, & \text{if} \quad \mathsf{T} \in \{\mathsf{r\_acc}, \mathsf{r\_trav}\} \wedge O = \mathtt{seq} \wedge k = 1 \\ \mathsf{r\_trav}, & \text{if} \quad \mathsf{T} = \mathsf{s\_trav}^x \wedge O = \mathtt{ran} \wedge \overline{\underline{R}} - u > Z_i \\ \mathsf{s\_trav}^{\mathrm{r}}, & \text{if} \quad \mathsf{T} = \mathsf{s\_trav}^s \wedge O = \mathtt{ran} \wedge \overline{\underline{R}} - u \leq Z_i \\ \mathsf{T}, & \text{else.} \end{cases}$$

In other words, an interleaved multi-cursor access pattern causes at least as many cache misses as some simple traversal pattern on the same data region. However, it might cause random misses though the local pattern is expected to cause sequential misses. Further, if the cross-traversal requires more cache lines than available, $\mathbf{X}^{\mathrm{r}} = (k-1)\cdot(m-h'_i)$ additional random misses will occur.

4.5 Combining Cost Functions

Given the cache misses for basic patterns, we will now discuss how to derive the resulting cache misses of compound patterns. The major problem is to model cache interference that occurs among the basic patterns.

4.5.1 Sequential Execution

Be $\mathcal{P}_1,\ldots,\mathcal{P}_p \in \mathbb{P}$ $(p > 1)$ access patterns. $\oplus(\mathcal{P}_1,\ldots,\mathcal{P}_p)$ then denotes that $\mathcal{P}_{q+1}$ is executed after $\mathcal{P}_q$ is finished (cf., Sec. 4.3.3). Obviously, the patterns do not interfere in this case. Consequently, the resulting total number of cache misses is at most the sum of the cache misses of all p patterns. However, if two subsequent patterns operate on the same data region, the second might benefit from the data that the first one leaves in the cache. It depends on the cache size, the data sizes, and the characteristics of the individual patterns, how many cache misses may be saved this way.

To model this effect, we need to consider the contents or *state* of the caches. We describe the state of a cache as a set $\mathbf{S}$ of pairs $\langle R,\rho\rangle \in \mathbb{D}\times]0,1]$, stating for each data region R the fraction ρ that is available in the cache. For convenience, we omit data regions that are not cached at all, i.e., those with $\rho = 0$. In order to appropriately consider the caches' initial states when calculating the cache misses of a basic pattern $\mathcal{P} = \mathsf{T}([..,]R[,..]) \in \mathbb{P}_{\mathrm{b}}$, we define

$$\begin{aligned}&\vec{\mathbf{M}}_i(\mathbf{S}_i,\mathcal{P})\\ &\quad= \begin{cases}\langle 0,0\rangle\,, & \text{if} \quad \langle R,1\rangle \in \mathbf{S}_i\\ \vec{\mathbf{M}}_i(\mathcal{P}) - \left\langle 0, \dfrac{\rho\cdot|R|_{Z_i}}{\vec{\mathbf{M}}_i(\mathcal{P})}\cdot\rho\cdot|R|_{Z_i}\right\rangle, & \text{if} \quad \mathsf{T}\in\{\mathsf{r\_trav},\mathsf{rr\_trav},\mathsf{r\_acc}\}\\ & \quad\wedge\ \exists\rho\in]0,1[:\langle R,\rho\rangle\in\mathbf{S}_i\\ \vec{\mathbf{M}}_i(\mathcal{P}) & \text{else}\end{cases}\end{aligned} \tag{4.11}$$

with $\vec{\mathbf{M}}_i(\mathcal{P})$ as defined in Equations (4.3) through (4.10). In case R is already entirely available in the cache, no cache misses will occur during $\mathcal{P}$. In case only a fraction of R is available in the cache, there is a certain chance, that random patterns might (partially) benefit from this fraction. Sequential patterns, however, would only benefit if this fraction makes up the "head"of R. As we do not know whether this is true, we assume that sequential patterns can only benefit, if R is already entirely in the cache. For convenience, we write

$$\vec{\mathbf{M}}_i(\emptyset,\mathcal{P}) = \vec{\mathbf{M}}_i(\mathcal{P}) \qquad \forall\mathcal{P}\in\mathbb{P}.$$

Additionally, we need to know the caches' resulting states $\mathbf{S}(\mathcal{P})$ after a pattern $\mathcal{P}$ has been performed. For basic patterns $\mathcal{P} = \mathsf{T}([..,]R[,..]) \in \mathbb{P}_\mathrm{b}$, we define

$$\mathbf{S}_i(\mathcal{P}) = \left\{ \left\langle R, \min\left\{ \frac{C_i}{||R||}, 1 \right\} \right\rangle \right\}.$$

For compound patterns $\oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)$ with $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$, $p > 1$, we define

$$\mathbf{S}_i(\oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \mathbf{S}_i(\mathcal{P}_p).$$

Here, we assume that only that last data region (partially) remains in the cache. In case that R is smaller that the cache, (parts) of the previous data regions might also remain in the cache. However, we ignore this case here, and leave it for future research.

Equipped with these tools, we can finally calculate the number of cache misses that occur when executing patterns $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$, $p > 1$ sequentially, given an initial cache state $\mathbf{S}^0$:

$$\vec{\mathbf{M}}_i(\mathbf{S}_i^0, \oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \vec{\mathbf{M}}_i(\mathbf{S}_i^0, \mathcal{P}_1) + \sum_{q=2}^{p} \vec{\mathbf{M}}_i(\mathbf{S}_i(\mathcal{P}_{q-1}), \mathcal{P}_q). \tag{4.12}$$

4.5.2 Concurrent Execution

When executing two or more patterns concurrently, we actually have to consider the fact that they are competing for the same cache. The number of total cache misses will be higher than just the sum of the individual cache miss rates. The reason for this is, that the patterns will mutually evict cache lines from the cache due to alignment conflicts. To which extend such conflict misses occur does not only depend on the patterns themselves, but also on the data placement and details of the cache alignment. Unfortunately, these parameters are not know during cost evaluation.

Hence, we model the impact of the cache interference between concurrent patterns by dividing the cache among all patterns. Each individual pattern gets only a fraction of the cache according to its *footprint size*. We define a pattern's footprint size $\mathbf{F}$ as the number of cache lines that it potentially revisits.

With single sequential traversals, a cache line is never visited again once access has moved on to the next cache line. Hence, simple sequential patterns virtually occupy only one cache line a at time. Or in other words, the number of cache misses is independent of the available cache size. The same holds for single random traversals with $\underline{\overline{R}} - u \geq Z$. In all other cases, basic access patterns (potentially) revisit all cache lines covered by their respective data region. We define $\mathbf{F}$ as follows.

Be $\mathcal{P} = \mathsf{T}([..,]R, u) \in \mathbb{P}_\mathrm{b}$ a basic access pattern, then

$$\mathbf{F}_i(\mathcal{P}) = \begin{cases} 1, & \text{if} \quad \mathsf{T} = \mathsf{s\_trav}^x \\ 1, & \text{if} \quad \mathsf{T} = \mathsf{r\_trav} \ \wedge \ \underline{\overline{R}} - u \geq Z_i \\ |R|_{Z_i}, & \text{else.} \end{cases}$$

Be $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ $(p > 1)$ access patterns, then

$$\mathbf{F}_i(\oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \max\{\mathbf{F}_i(\mathcal{P}_1), \ldots, \mathbf{F}_i(\mathcal{P}_p)\},$$
$$\mathbf{F}_i(\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \sum_{q=1}^{p} \mathbf{F}_i(\mathcal{P}_q).$$

Further, we use $\vec{\mathbf{M}}_{i/\nu}$ with $\nu \geq 1$ to denote the number of misses with only $\frac{1}{\nu}$th of the total cache size available. To calculate $\vec{\mathbf{M}}_{i/\nu}$, we simply replace C and # by $\frac{C}{\nu}$ and $\frac{\#}{\nu}$, respectively, in the formulas in Sections 4.4 and 4.5.1. Likewise, we define $\mathbf{S}_{i/\nu}(\mathcal{P})$. We write $\vec{\mathbf{M}}_i = \vec{\mathbf{M}}_{i/1}$ and $\mathbf{S}_i = \mathbf{S}_{i/1}$.

With these tools at hand, we calculate the cache misses for concurrent execution of patterns $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ $(p > 1)$ given an initial cache state $\mathbf{S}^0$ as

$$\vec{\mathbf{M}}_{i/\nu}(\mathbf{S}_i^0, \odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \sum_{q=1}^{p} \vec{\mathbf{M}}_{i/\nu_q}(\mathbf{S}_i^0, \mathcal{P}_q) \tag{4.13}$$

with

$$\nu_q = \frac{\mathbf{F}(\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p))}{\mathbf{F}(\mathcal{P}_q)} \cdot \nu.$$

After executing $\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)$, the cache contains a fraction of each data region involved, proportional to its footprint size:

$$\mathbf{S}_i(\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \bigcup_{q=1}^{p} \mathbf{S}_{i/\nu_q}(\mathcal{P}_q)$$

with ν_q as defined before.

4.5.3 Query Execution Plans

With the techniques discussed in the previous sections, we have the basic tools at hand to also estimate the number and kind of cache misses of complete query plans, and hence to predict their memory access costs. The various operators in a query plan are combined in the same way the basic patterns are combined to form compound patterns. Basically, the query plan describes, which operators are executed one after the other and which are executed concurrently. Here, we view pipelining as concurrent execution of data-dependent operators. Hence, we can derive the complex memory access pattern of a query plan by combining the compound patterns of the operators as discussed above. Considering the caches' states as introduced before takes care of properly recognizing data dependencies.

4.6 CPU Costs

Next to memory access costs, we need to know the pure CPU processing costs in order to estimate the total execution costs. We now present a simple but effective method to acquire CPU costs using a calibration approach. As we focus on memory access costs in this work, a more detailed model of CPU costs is beyond the scope of this thesis and left for future work.

4.6.1 What to calibrate?

Calibrating CPU costs means to actually measure the costs — i.e., execution time — of all algorithms in a laboratory setting. A prerequisite for this approach is that we know the complexity of the algorithms in terms of input and output cardinalities. In other words, we assume we know the principle CPU cost functions of our algorithms. For instance, we know that the CPU cost of a scan-select can be described as

$$T_{\mathrm{CPU}} = c_0 + c_1 \cdot n + c_2 \cdot m$$

where c_0 represents the fix startup costs, n and m are the input and output cardinality, respectively, and c_1, c_2 represent the per tuple costs for processing input tuples and producing output tuples, respectively. As we are talking about our own algorithms, the assumption that we know such functions is reasonable. In case of doubt, we can use the *Software Testpilot* [KK93, AKK95] to experimentally derive these functions.

Obviously, these functions only depend on the algorithm itself. All implementations details like coding style, code optimizations, and compiler optimizations are covered by the constants c_i in the above formula. This approach implies that the constants c_i are indeed independent of the data volume. But this does hold for the pure CPU costs. The impact of data volume is already covered by the memory access costs. However, the c_i are typically not independent of other parameters like data types, and respective code optimizations in the algorithms. Here, again, our code expansion technique pays back. Within each expanded implementation variant of our algorithm, these parameters are constant, and thus, the c_i are indeed simple constants for each physical implementation.

4.6.2 How to calibrate?

Hence, calibrating the c_i can be done by some simple experiments. For instance, in our example case, the scan-select, we use the following experiments:

First, we need to measure the fix (start-up) costs c_0. The start-up costs cover all the "administrative"work that is done only once per algorithm. Such work contains among other

- parsing the MIL command;
- performing on-the-fly tactical/operational optimization, i.e., choosing the most adequate algorithm/implementation according to the current properties of the inputs and the current state of the system;
- overhead for calling the function that implements the operator, including possible instruction cache misses (i.e., memory access) to load the respective code and the necessary stack management;

- creating, initializing, and removing temporary data structures;
- creating the output/result data structure, usually a BAT.

We measure c_0 by measuring the time needed to execute the operation on empty inputs. Typically, the time needed for this task is so small, that the resolution of the timing function is just not small enough to ensure accurate and reliable measurements. We prevent such problems by repeating the empty call several times and measure the total time. The actual costs for a single call are then calculated by dividing the total time by the number of calls.

Once we know c_0, we can measure c_1 by performing the operation in question on a non-empty input that produces no output. In case of our example (select), for instance, this can be achieved by using a predicate the does not match any tuple of the input BAT. c_1 is then to be calculated by subtracting c_0 from the measured time and dividing the remainder by the cardinality of the input. Two things have to be considered. On the one hand, we would like to exclude any interference with memory access. One way to do this is to use an input table that is so small that all processing takes place in L1 cache. This requires an initial not-measured run to pre-load the cache. Again, we might need to measure several subsequent runs to ensure stable results. On the other hand, using too little input data might result in inaccurate times for the per tuple costs. The fix costs are likely to be dominant, as we might need to use really small inputs to be sure that all processing indeed is limited to L1. Using large inputs, however, implies that memory costs will be included in the measurement. But we do know the memory costs, and hence, we can easily subtract them from the measured times to get the pure CPU costs. We propose to use the second technique.

Actually, the second technique has another advantage. As we learned in Chapter 3, CPU costs and memory access costs may overlap. Hence, simply adding-up the CPU costs as calibrated without any memory access by the first technique and the memory costs as estimated by our models would result in too high overall costs. Accurately predicting the degree of overlap between CPU costs and memory access costs, however, depends on various parameters and is hence very difficult, if not impossible. Our second technique, however, implicitly considers the actual overlap by measuring only that part of the CPU costs that does not overlap with memory access costs. As we are interested in the CPU costs only to add them to the estimated memory access costs, and thus yield the total costs, the second technique makes-up a feasible solution.

Knowing both c_0 and c_1, we can measure c_2 in a third and final experiment. We modify the second experiment to produce output that is as big as the input. In case of our example (select), for instance, this can be achieved by using a predicate such that all input tuples do match. Subtracting fix costs, input processing costs (as measured before), and memory access costs from the measured time, we can calculate the output creation costs.

For other algorithms, the calibration procedure follows the same schema, though it gets more complex for more complex algorithms.

4.6.3 When to calibrate?

For the memory access cost models, we proposed to measure the required hardware parameters once when the DBMS is installed on a new (hardware-)system. Likewise, the CPU costs also need to be measured only once per system. However, as we need to measure the costs for each physical implementation of each algorithm, this might be a rather complex and long running task. One way to speed-up the installation process is to restrict this task (at installation time) to the most popular MIL operations and only their most important variants. The remaining costs can then be calibrated ”on-the-fly”only as soon as they are need by the system.

4.7 Experimental Validation

To validate our cost model, we compare the estimated costs with experimental results. We focus on characteristic operations, here. The data access pattern of each operation is a combination of several basic patterns. The operations are chosen so that each basic pattern occurs at least once. Extension to further operations and whole queries, however, is straight forward, as it just means applying the same techniques to combine access patterns and derive their cost functions.[7]

4.7.1 Setup

We implemented our cost functions and used our main-memory DBMS prototype Monet (see Section 2.7) as experimentation platform. We ran our experiments on an SGI Origin2000 and on an AMD PC. Table 4.2 lists the relevant hardware features of the machines. The cache characteristics are measured with our calibration tool. We use the CPU’s hardware counters to get the exact number of cache and TLB misses while running our experiments. Thus, we can validate the estimated cache miss rates. Validating the resulting total memory access cost (i.e., miss rates scored by their latencies) is more complicated, as there is no way to measure the time spent on memory access. We can only measure the total elapsed time, and this includes the (pure) CPU costs as well. Hence, we extend our model to estimate the total execution time T as sum of memory access time and pure CPU time

$$T = T_{\mathrm{Mem}} + T_{\mathrm{CPU}} \tag{4.14}$$

with T_{Mem} as in Equation (4.1). We calibrate T_{CPU} for each algorithm as described in the previous section.

4.7.2 Results

Figures 4.8 through 4.11 gather our experimental results. Each plot represents one algorithm. The cache misses and times measured during execution are depicted as points. The respective cost estimations are plotted as lines. Cache misses are depicted

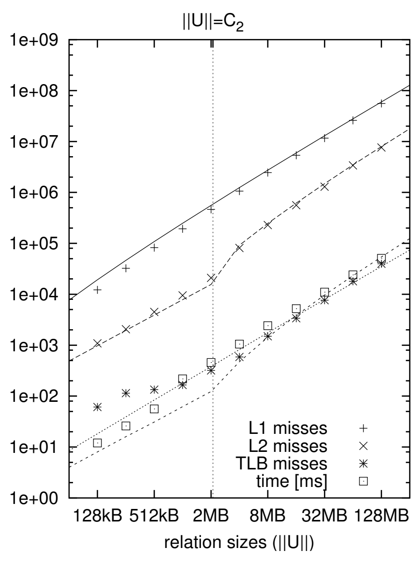
[7]We present more examples and validation in Chapter 5.

machine type		SGI Origin2000	AMD PC
OS		IRIX64 6.5	Linux 2.2.14
CPU		MIPS R10000	AMD Athlon
CPU speed		250 MHz	600 MHz
main-memory size		48 GB (4 GB local)	384 MB
cache & TLB levels	N	3	3
L1 cache capacity	C_1	32 KB	64 KB
L1 cache line size	Z_1	32 bytes	64 bytes
L1 cache lines	$\#_1$	1,024	1,024
L2 cache capacity	C_2	4 MB	512 KB
L2 cache line size	Z_2	128 bytes	64 bytes
L2 lines	$\#_2$	32,768	8,192
TLB entries	$\#_3$	64	32
page size	Z_3	16 KB	4 KB
TLB capacity ($\#_3 \cdot Z_3$)	C_3	1 MB	128 KB
TLB miss latency	$l_3^r = l_3^s$	228 ns = 57 cycles	8 ns = 5 cycles
sequential access			
L1 miss latency	l_1^s	10 ns = 2.5 cycles	20 ns = 12 cycles
L2 miss latency	l_2^s	180 ns = 45 cycles	71 ns = 43 cycles
L1 miss bandwidth	b_1^s	3052 MB/s	3052 MB/s
L2 miss bandwidth	b_2^s	555 MB/s	670 MB/s
random access			
L1 miss latency	l_1^r	24 ns = 6 cycles	45 ns = 27 cycles
L2 miss latency	l_2^r	406 ns = 101 cycles	180 ns = 108 cycles
L1 miss bandwidth	b_1^r	1272 MB/s	1356 MB/s
L2 miss bandwidth	b_2^r	243 MB/s	271 MB/s

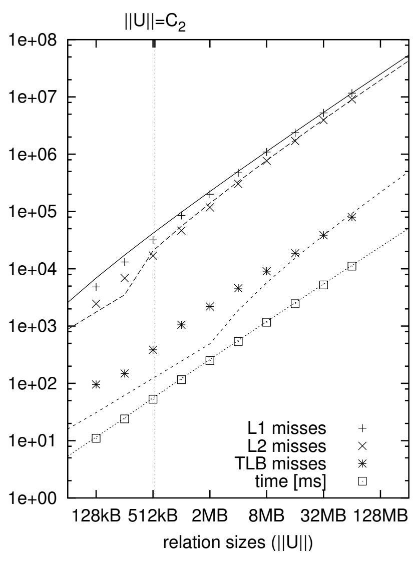
Table 4.2: Hardware Characteristics

in absolute numbers. Times are depicted in milliseconds. We will now discuss each algorithm in detail.

Quick-Sort Our first experiment is sorting. We use quick-sort to sort a table in-place. Quick-sort uses two cursors, one starting at the front and the other starting at the end. Both cursors sequentially walk toward each other swapping data items where necessary, until they meet in the middle. We model this as two concurrent sequential traversals, each sweeping over one half of the table: $\text{s\_trav}^s(U/2) \odot \text{s\_trav}^s(U/2)$. At the meeting point, the table is split in two parts and quick-sort recursively proceeds depth-first on each part. With n being the table's cardinality, the depth of the recursion



a) Origin2000



b) AMD PC

Figure 4.8: Measured (points) and Predicted (lines) Cache Misses and Execution Time of Quick-Sort

is $\log_2 n$. In total, we model the data access pattern of quick-sort as

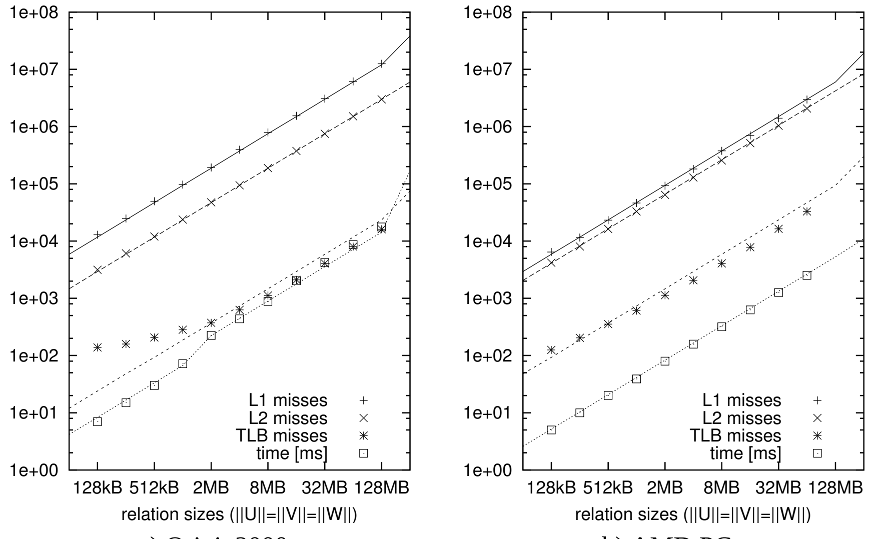
$$U \leftarrow \mathit{quick\_sort}(U) : \qquad \oplus|_{i=1}^{\log_2 U.n} \left(\oplus|_{j=i}^{\log_2 U.n} \left(\mathsf{s\_trav}^{\mathsf{s}}(U/2^j) \odot \mathsf{s\_trav}^{\mathsf{s}}(U/2^j) \right) \right).$$

We varied the table sizes from 128 KB to 128 MB and the tables contained randomly distributed (numerical) data. Figure 4.8 shows that the models accurately predict the actual behavior. Only the start-up overhead of about 100 TLB misses is not covered, but this is negligible. The step in the L2 misses-curve depicts the effect of caching on repeated sequential access: Tables that fit into the cache have to be loaded only once during the top-level iteration of quick-sort. Subsequent iterations operate on the cached data, causing no additional cache misses.

Merge-Join Our next candidate is merge-join. Assuming both operands are already sorted, merge-join simply performs three concurrent sequential patterns, one on each input and one on the output:

$$W \leftarrow \mathit{merge\_join}(U, V) : \qquad \mathsf{s\_trav}^{\mathsf{s}}(U) \odot \mathsf{s\_trav}^{\mathsf{s}}(V) \odot \mathsf{s\_trav}^{\mathsf{s}}(W).$$

Again, we use randomly distributed data and table sizes as before. In all experiments, both operands are of equal size, and the join is a 1:1-match. The respective



a) Origin2000 b) AMD PC

Figure 4.9: Measured (points) and Predicted (lines) Cache Misses and Execution Time of Merge-Join

results in Figure 4.9 demonstrate the accuracy of our cost functions. Further, we see that single sequential access is not affected by cache sizes. The costs are proportional to the data sizes.

Hash-Join While the previous operations perform only sequential patterns, we now turn our attention to hash-join. Hash-join performs random access to the hash-table, both while building it and while probing the other input against it. We model the data access pattern of hash-join as

$$W \leftarrow \mathit{hash\_join}(U, V) : \\ \qquad \mathsf{s\_trav}^{\mathsf{s}}(V) \odot \mathsf{r\_trav}(V') \oplus \mathsf{s\_trav}^{\mathsf{s}}(U) \odot \mathsf{r\_acc}(|U|, V') \odot \mathsf{s\_trav}^{\mathsf{s}}(W).$$

The plots in Figure 4.10 clearly show the significant increase in L2 and TLB misses, once the hash-table size $||V'||$ exceeds the respective cache size.[8] Our cost model correctly predicts these effects and the resulting execution time.

Partitioning One way to prevent the performance decrease of hash-join on large tables is to partition both operands on the join attribute and then hash-join the matching partitions [SKN94, MBK00a]. If each partition fits into the cache, no additional cache misses will occur during hash-join.

[8]The plots show no such effect for L1 misses, as all hash-tables are larger than the L1 cache, here.

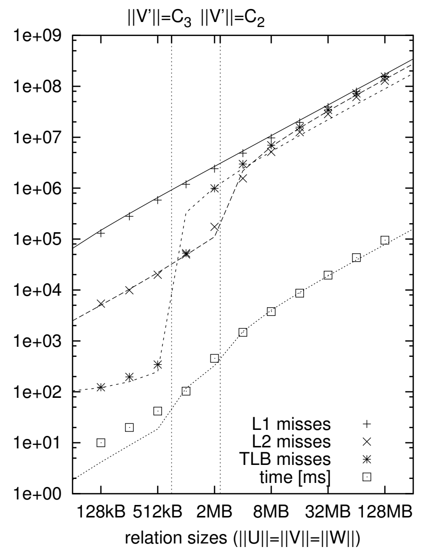
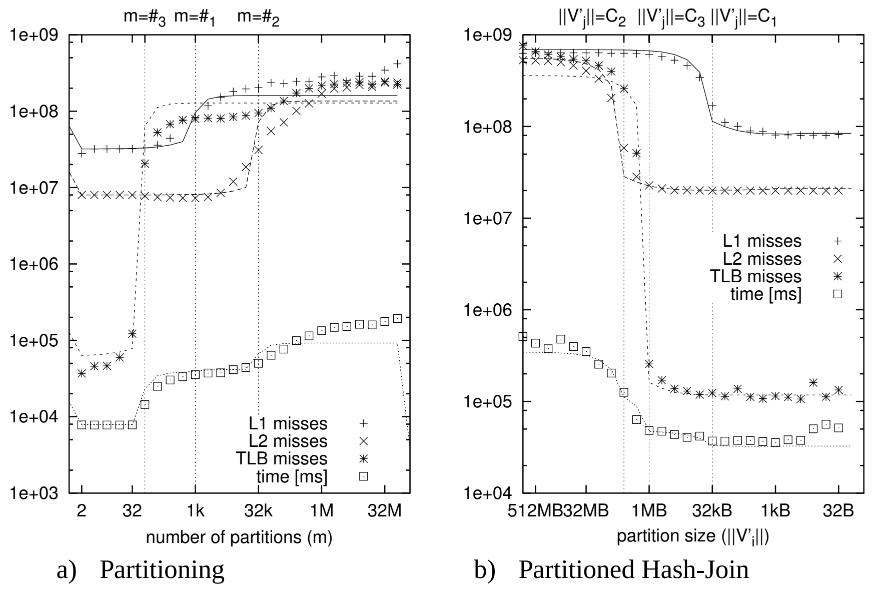


Figure 4.10: Measured (points) and Predicted (lines) Cache Misses and Execution Time of Hash-Join



a) Partitioning
($\|U\| = \|W\| = 512\,\mathrm{MB}$)

b) Partitioned Hash-Join
($\|U\| = \|V\| = \|W\| = 512\,\mathrm{MB}$)

Figure 4.11: Measured (points) and Predicted (lines) Cache Misses and Execution Time of Partitioning and Partitioned Hash-Join

Partitioning algorithms typically maintain a separate output buffer for each result partition. The input is read sequentially, and each tuple is written to its output partition. Data access within each output partition is also sequential. Hence, we model partitioning using a sequential traversal for the input and an interleaved multi-cursor access for the output:

$$\{U_j\}|_{j=1}^{m} \leftarrow \mathit{cluster}(U, m) : \qquad \mathsf{s\_trav}^{\mathsf{s}}(U) \odot \mathsf{nest}(\{U_j\}|_{j=1}^{m}, m, \mathsf{s\_trav}^{\mathsf{s}}(U_j), \mathtt{ran}).$$

The curves in Figure 4.11a demonstrate the effect we discussed in Section 4.4.7: The number of cache misses increases significantly, once the number of output buffers m exceeds the number of available cache blocks #. Though they tend to under estimate the costs for very high numbers of partitions, our models accurately predict the crucial points.

Partitioned Hash-Join Once the inputs are partitioned, we can join them by performing a hash-join on each pair of matching partitions. We model the data access pattern of partitioned hash-join as

$$\{W_j\}|_{j=1}^{m} \leftarrow \mathit{part\_hash\_join}(\{U_j\}|_{j=1}^{m}, \{V_j\}|_{j=1}^{m}, m) : \qquad \oplus\, |_{j=1}^{m}(\mathsf{hash\_join}(V_j, U_j, W_j)).$$

Figure 4.11b shows that the cache miss rates, and thus the total costs, decrease significantly, once each partition (respectively its hash-table) fits into the cache.

4.8 Conclusion

We presented a new generic approach to build generic database cost models for hierarchical memory systems.

We extended the knowledge base on analytical cost-models for query optimization with a strategy derived from our experimentation with main-memory database technology. The approach taken shows that we can achieve hardware-independence by modeling hierarchical memory systems as multiple level of caches. Each level is characterized by a few parameters describing its sizes and timings. This abstract hardware model is not restricted to main-memory caches. As we pointed out, the characteristics of main-memory access are very similar to those of disk access. Viewing main-memory (e.g., a database system's buffer pool) as cache for disk access, it is obvious that our approach also covers I/O. As such, the model presented provides a valuable addition to the core of cost-models for disk-resident databases as well.

Adaptation of the model to a specific hardware is done by instantiating the parameters with the respective values of the very hardware. Our *Calibrator*, a software tool to measure these values on arbitrary systems, is available for download from our web site `http://monetdb.cwi.nl`.

We identified a few key access patterns eminent in the majority of relational algebra implementations. The key patterns fulfill two major requirements: they are simple and they have a relevant impact on data access costs. For these basic patterns, we developed cost functions that estimate the respective access cost in terms of cache misses scored by there latency. To maintain hardware-independence, the functions are parameterized with the hardware characteristics.

We introduced two operators to combine simple patterns to more complex patterns and developed rules how to generate the respective cost functions.

With our approach, building physical costs function for database operations boils down to describing the algorithms' data access in a kind of "pattern language"as presented in Section 4.3.3. This task requires only information that can be derived from the algorithm. Especially, no knowledge about the hardware is needed, here. The detailed cost function are than automatically derived from the pattern descriptions.

Though focusing on data access costs, our model does not ignore CPU costs. We presented a simple but effective calibration approach that allows to automatically measure the CPU costs of each algorithm and its various implementations. An investigation as to whether and how CPU costs can be modeled in more details is left to future research.

Chapter 5

Self-tuning Cache-conscious Join Algorithms

In the past decade, the exponential growth in commodity CPUs' speed has far outpaced advances in memory latency. A second trend is that CPU performance advances are not only brought by increased clock rate, but also by increasing parallelism inside the CPU. Current database systems have not yet adapted to these trends, and show poor utilization of both CPU and memory resources on current hardware. We discussed these issues in detail in Chapter 3. In this chapter, we show how these resources can be optimized for large joins. We refine the partitioned hash-join with a new partitioning algorithm called radix-cluster, which is specifically designed to optimize memory access. The algorithms are designed to be tuned at runtime to achieve the optimal performance given the underlying hardware and the actual data to be processed. Tuning is done be means of just three parameters. We will demonstrate, how the cost models developed in Chapter 4 allow us to determine the optimal values for these parameters automatically at runtime. Finally, we investigate the effect of implementation techniques that optimize CPU resource usage. It turns out, that the full benefit of memory access optimization can only be achieved, if also the CPU resource usage is minimized. Exhaustive experiments on four different architectures show that large joins can be accelerated almost an order of magnitude on modern RISC hardware when both memory and CPU resources are optimized.

5.1 Introduction

Custom hardware—from workstations to PCs—has experienced tremendous performance improvements in the past decades. Unfortunately, these improvements are not equally distributed over all aspects of hardware performance and capacity. Figure 3.1 shows that the speed of commercial microprocessors has increased roughly 50% every year, while the access latency of commodity DRAM has improved by little more than 10% over the past decade [Mow94]. One reason for this is that there is a direct trade-off between capacity and speed in DRAM chips, and the highest priority has been for increasing capacity. The result is that from the perspective of the processor, memory is getting slower at a dramatic rate, making it increasingly difficult to achieve high processor efficiencies. Another trend is the ever increasing number of inter-stage and intra-stage parallel execution opportunities provided by multiple execution pipelines and speculative execution in modern CPUs. Current database systems on the market make poor use of these new features; studies on several DBMS products on a variety of workloads [ADHW99, BGB98, KPH$^+$98, TLPZT97] consistently show modern CPUs to be stalled (i.e., non-working) most of the execution time.

In this chapter, we show how large main-memory joins can be accelerated by optimizing memory and CPU resource utilization on modern hardware. These optimizations involve radical changes in database architecture, encompassing new data structures, query processing algorithms, and implementation techniques. Our findings are summarized as follows:

- *Memory access is a bottleneck to query processing.* We demonstrated in Section 3.2 that the performance of even simple database operations is nowadays severely constrained by memory access costs. For example, a simple in-memory table scan runs on Sun hardware from the year 2000 in roughly the same absolute time as on a Sun from 1992, now spending 95% of its cycles waiting for memory (see Section 3.2). It is important to note that this bottleneck affects database performance in general, not only main-memory database systems.

- *Data structures and algorithms should be tuned for memory access.* We discuss database techniques to avoid the memory access bottleneck, both in the fields of data structures and query processing algorithms. The key issue is to optimize the use of the various caches of the memory subsystem. We show how *vertical table fragmentation* optimizes sequential memory access to column data. For equi-join, which has a random access pattern, we refine partitioned hash-join with a new *radix-cluster* algorithm which makes its memory access pattern more easy to cache. Our experiments indicate that large joins can strongly benefit from these techniques.

- *Memory access costs can be modeled precisely.* Cache-aware algorithms and data structures must be tuned to the memory access pattern imposed by a query and hardware characteristics such as cache sizes and miss penalties, just like traditional query optimization tunes the I/O pattern imposed by a query to the size of the buffers available and I/O cost parameters. Therefore it is necessary to have models that predict memory access costs in detail. We apply the techniques presented in Chapter 4 to provide such detailed models for our partitioned hash-join algorithms. These models use an analytical framework that predicts the number of hardware events (e.g., cache misses and CPU cycles), and scores them with hardware parameters. The experiments in this chapter confirm both the usability and the accuracy of our generic cost models.

- *Memory optimization and efficient coding techniques boost each others effects.* CPU resource utilization can be optimized using implementation techniques known from high-performance computing [Sil97] and main-memory database systems [Ker89, BK99]. We observe that applying these optimizations in combination with memory optimizations yields a higher performance increase than applying them without memory optimizations. The same is also the case for memory optimizations: they turn out to be more effective on CPU-optimized code than on non-optimized code. Our experiments show that database performance can be improved by an order of magnitude applying both CPU and memory optimization techniques.

Our research group has studied large main-memory database systems for the past 10 years. This research started in the PRISMA project [AvdBF+92], focusing on massive parallelism, and is now centered around Monet: a high-performance system targeted to query-intensive application areas like OLAP and data mining (cf., Section 2.7). We use Monet as our experimentation platform.

5.1.1 Related Work

Database system research into the design of algorithms and data structures that optimize memory access, has been relatively scarce. Our major reference is the work by Shatdal et al. [SKN94], which shows that join performance can be improved using a main-memory variant of Grace Join, in which both relations are first hash-partitioned in chunks that fit the (L2) memory cache. There were various reasons that lead us to explore this direction of research further. First, after its publication, the observed trends in custom hardware have continued, deepening the *memory access bottleneck*. For instance, the authors list a mean performance penalty for a cache miss of 20-30 cycles in 1994, while a range of 100-300 is typical in 2002 (and rising). This increases the benefits of cache optimizations, and possibly changes the trade-offs. Another development has been the introduction of so-called level-one (L1) caches, which are typically very small regions on the CPU chip that can be accessed at almost CPU clock-speed. The authors of [SKN94] provide algorithms that are only feasible for the relatively larger off-chip L2 caches. Finally, this previous work uses standard relational data structures, while we argue, that the impact of memory access is so severe that vertically fragmented data structures should be applied at the physical level of database storage.

Though we consider memory-access optimization to be relevant for database performance in general, it is especially important for main-memory databases, a field that through time has received fluctuating interest within the database research community. In the 1980s [LC86a, LC86b, Eic89, Wil91, AP92, GMS92], when falling DRAM prices seemed to suggest that most data would soon be memory-resident, its popularity diminished in the 1990s, narrowing its field of application to real-time systems only. Currently, interest has revived into applications for small and distributed database systems, but also in high performance systems for query-intensive applications, like data mining and OLAP. In our research, we focus on this latter category. Example commercial systems are the Times-Ten product [Tea99], Sybase IQ [Syb96], and Compaq's Infocharger [Com98], which is based on an early version of Monet (cf., Section 2.7).

Past work on main-memory query optimization [LN96, WK90] models the main-memory costs of query processing operators on the coarse level of procedure calls, using profiling to obtain some 'magical' constants. As such, these models do not provide insight in individual components that make up query costs, limiting their predictive value. Conventional (i.e., non main-memory) cost modeling, in contrast, has I/O as the dominant cost aspect, making it possible to formulate accurate models based on the amount of predicted I/O work. Calibrating such models is relatively easy, as statistics on the I/O accesses caused during an experiment are readily available in a database system. Past work on main-memory systems was unable to provide such cost models on a similarly detailed level, for two reasons. First, it was difficult to model the interaction between low-level hardware components like CPU, Memory Management Unit, bus, and memory caches. Second, it was impossible to measure the status of these components during experiments, which is necessary for tuning and calibration of models. Modern CPUs, however, contain performance counters for events like cache misses, and exact CPU cycles [BZ98, ZLTI96, Yea96]. This enabled us to develop a new main-memory cost modeling methodology that first mimics the memory access pattern of an algorithm, yielding a number of CPU cycle and memory cache events, and then scores this pattern with an exact cost prediction (see Chapter 4). Therefore, the contribution of the algorithms, models, and experiments presented here is to demonstrate that detailed cost modeling of main-memory performance is both important and feasible.

5.1.2 Outline

In Section 5.2, we introduce the *radix-cluster* algorithm, which improves the partitioning phase in partitioned hash-join by trading memory access costs for extra CPU processing. We perform exhaustive experiments where we use CPU event counters to obtain detailed insight in the performance of this algorithm. First, we vary the partition sizes, to show the effect of tuning the memory access pattern to the memory cache sizes. Second, we investigate the impact of code optimization techniques for main-memory databases. These experiments show that improvements of almost an order of magnitude can be obtained by combining both techniques (cache tuning and code optimization) rather than by each one individually. Our results are fully explained by detailed models of both the partition (radix-cluster) and join phase of partitioned hash-join. We show how performance can exactly be predicted from hardware events like cache and TLB misses, and thus validate the generic cost modeling techniques we developed in the previous chapter. The design of our algorithms paired with the ability to accurately predict there performance build the foundation for automatically tune the algorithms at runtime to yield the efficient utilization of CPU and memory resources.

In Section 5.3, we present *radix-join* as an alternative to partitioned hash-join, and compare the performance of both algorithms.

In Section 5.4, we evaluate our findings and show how they support the choices we made back in 1994 when designing Monet, which uses full vertical fragmentation and

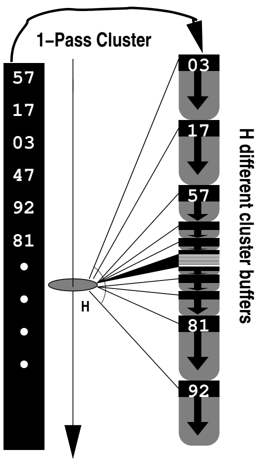


Figure 5.1: Straightforward cluster algorithm

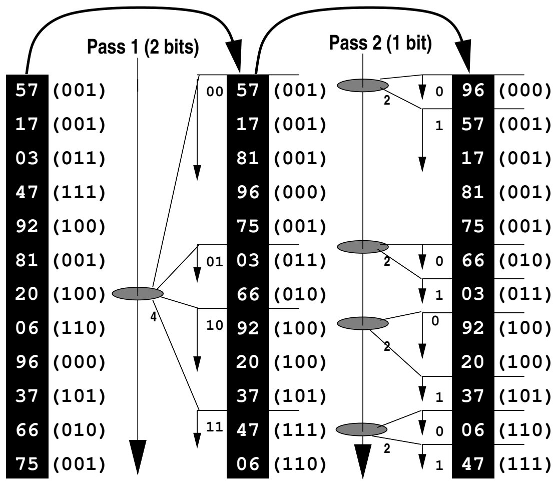


Figure 5.2: 2-pass/3-bit Radix Cluster (lower bits indicated between parentheses)

implementation techniques optimized for main memory to achieve high performance on modern hardware. We conclude with recommendations for future systems.

5.2 Partitioned Hash-Join

Shatdal et al. [SKN94] showed that a main-memory variant of Grace Join, in which both relations are first partitioned on hash-number into *H* separate *clusters*, that each fit the memory cache, performs better than normal bucket-chained hash join. This work employs a straightforward clustering-algorithm that simply scans the relation to be clustered once, inserting each scanned tuple in one of the clusters, as depicted in Figure 5.1. This constitutes a random access pattern that writes into *H* separate locations. If *H* is too large, there are two factors that degrade performance. First, if *H* exceeds the number of TLB entries[1] each memory reference will become a *TLB miss*. Second, if *H* exceeds the number of available cache lines (L1 or L2), *cache thrashing* occurs, causing the number of cache misses to explode.

As an improvement over this straightforward algorithm, we propose a clustering algorithm that has a memory access pattern that requires less random-access, even for high values of *H*.

[1]If the relation is very small and fits the total number of TLB entries times the page size, multiple clusters will fit into the same page and this effect will not occur.

5.2.1 Radix-Cluster Algorithm

The *radix-cluster* algorithm divides a relation U into H clusters using multiple passes (see Figure 5.2). Radix-clustering on the lower B bits of the integer hash-value of a column is achieved in P sequential passes, in which each pass clusters tuples on B_p bits, starting with the leftmost bits ($\sum_1^P B_p = B$). The number of clusters created by the radix-cluster is $H = \prod_1^P H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones. When the algorithm starts, the entire relation is considered one single cluster, and is subdivided into $H_1 = 2^{B_1}$ clusters. The next pass takes these clusters and subdivides each into $H_2 = 2^{B_2}$ new ones, yielding $H_1 * H_2$ clusters in total, etc.. Note that with $P = 1$, radix-cluster behaves like the straightforward algorithm.

For ease of presentation, we did not use a hash function in the table of integer values displayed in Figure 5.2. In practice, though, it is better to use such a function even on integers in order to ensure that all bits of the table values play a role in the lower bits of the radix number.

The interesting property of the radix-cluster is that the number of randomly accessed regions H_x can be kept low; while still a high overall number of H clusters can be achieved using multiple passes. More specifically, if we keep $H_x = 2^{B_x}$ smaller than the number of cache lines and the number of TLB entries, we totally avoid both TLB and cache thrashing.

After radix-clustering a column on B bits, all tuples that have the same B lowest bits in its column hash-value, appear consecutively in the relation, typically forming chunks of $|U|/2^B$ tuples (with $|U|$ denoting the cardinality of the entire relation). It is therefore not strictly necessary to store the cluster boundaries in some additional data structure; an algorithm scanning a radix-clustered relation can determine the cluster boundaries by looking at these lower B “radix-bits”. This allows very fine clusterings without introducing overhead by large boundary structures. It is interesting to note that a radix-clustered relation is in fact *ordered* on radix-bits. When using this algorithm in the partitioned hash-join, we exploit this property, by performing a merge step on the radix-bits of both radix-clustered relations to get the pairs of clusters that should be hash-joined with each other.

5.2.2 Quantitative Assessment

The radix-cluster algorithm presented in the previous section provides three tuning parameters:

1. the number of radix-bits used for clustering (B), implying the number of clusters $H = 2^B$,
2. the number of passes used during clustering (P),
3. the number of radix-bits used per clustering pass (B_p).

In the following, we present an exhaustive series of experiments to analyze the performance impact of different settings of these parameters. After establishing which parameter settings are optimal for radix-clustering a relation on B radix-bits, we turn our attention to the performance of the join algorithm with varying values of B. For both phases, clustering and joining, we investigate how appropriate implementations techniques can improve the performance even further. Finally, these two experiments are combined to gain insight in the overall join performance.

5.2.2.1 Experimental Setup

In our experiments, we use binary relations (BATs) of 8 bytes wide tuples and varying cardinalities ($|U|$), consisting of uniformly distributed random numbers. Each value occurs three times. Hence, in the join-experiments, the join hit-rate is three. The result of a join is a BAT that contains the [OID,OID] combinations of matching tuples (i.e., a join-index [Val87]). Subsequent tuple reconstruction is cheap in Monet, and equal for all algorithms, so just like in [SKN94] we do not include it in our comparison. The experiments were carried out on the machines presented in Section 3.3, an SGI Origin2000, a Sun Ultra, an Intel PC, and an AMD PC (cf., Table 3.2).

To analyze the performance behavior of our algorithms in detail, we break down the overall execution time into the following major categories of costs:

memory access In addition to memory access costs for data as analyzed above, this category also contains memory access costs caused by instruction cache misses.

CPU stalls Beyond memory access, there are other events that make the CPU stall, like branch mispredictions or other so-called resource related stalls.

divisions We treat integer divisions separately, as they play a significant role in our hash-join (see below).

real CPU This is the remaining time, i.e., the time the CPU is indeed busy executing the algorithms.

The four architectures we investigate, provide different hardware counters [BZ98] that enable us to measure each of these cost factors accurately. Table 5.1 gives an overview of the counters used. Some counters yield the actual CPU cycles spent during a certain event, others just return the number of events that occurred. In the latter case, we multiply the counters by the penalties of the events (as calibrated in Section 3.3). None of the architectures provides a counter for the pure CPU activity. Hence, we subtract the cycles spent on memory access, CPU stalls, and integer division from the overall number of cycles and assume the rest to be pure CPU costs.

In our experiments, we found that in our algorithms, branch mispredictions and instruction cache misses do not play a role on either architecture. The reason is that, in contrast to most commercial DBMSs, Monet's code base is designed for efficient main-memory processing. Monet uses a very large grain size for buffer management in its operators (an entire BAT), processing therefore exhibits much code locality during execution, and hence avoids instruction cache misses and branch mispredictions. Thus, for simplicity of presentation, we omit these events in our evaluation.

category	MIPS R10k	Sun UltraSPARC	Intel PentiumIII	AMD Athlon
memory access	L1_d_miss *6cy L2_d_miss*100cy TLB_miss *57cy L1_i_miss *6cy L2_i_miss *100cy	DC_miss[a] *6cy EC_miss[b] *39cy M_{TLB} *54cy stall_IC_miss	DCU_miss_ _outstanding M_{TLB} *5cy IFU_mem_stall ITLB_miss *32cy[c]	DC_refill_L2 *27cy DC_refill_sys *103cy L1_DTLB_miss *5cy L2_DTLB_miss*52cy IC_miss*27cy L1_ITLB_miss *5cy L2_ITLB_miss *52cy
CPU stalls	br_mispred *4cy	stall_mispred stall_fpdep	br_mispred *17cy[d] ILD_stalled resource_stalls[e] partial_rat_stalls	branches_mispred
divisions	$\|U\| * 2 * 35$cy	$\|U\| * 2 * 60$cy	cycles_div_busy	$\|U\| * 2 * 40$cy

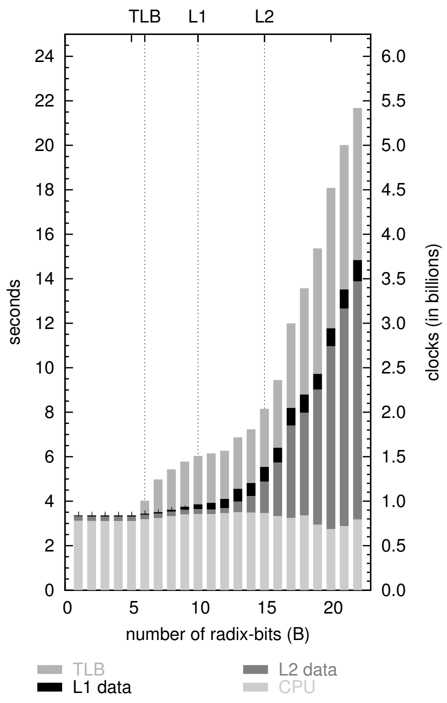
[a]DC_misses = DC_read - DC_read_hit + DC_write - DC_write_hit.
[b]EC_misses = EC_ref - EC_hit.
[c]Taken from [ADHW99].
[d]Taken from [ADHW99].
[e]This counter originally includes "DCU_miss_outstanding". We use only the remaining part after subtracting "DCU_miss_outstanding", here.

Table 5.1: Hardware Counters used for Execution Time Breakdown

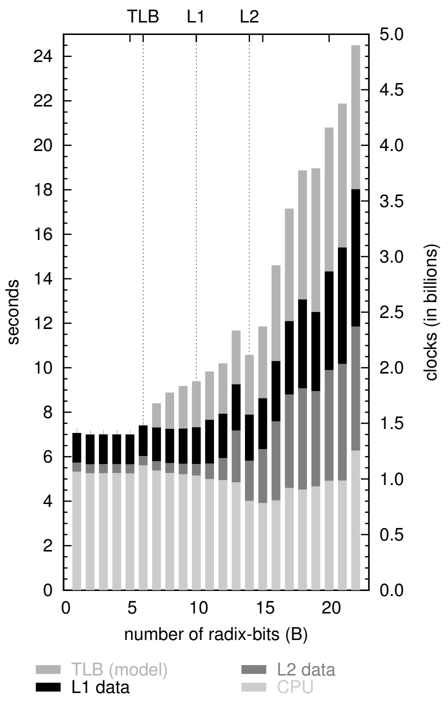
5.2.2.2 Radix Cluster

To analyze the impact of all three parameters (B, P, B_p) on radix clustering, we conduct two series of experiments, keeping one parameter fixed and varying the remaining two.

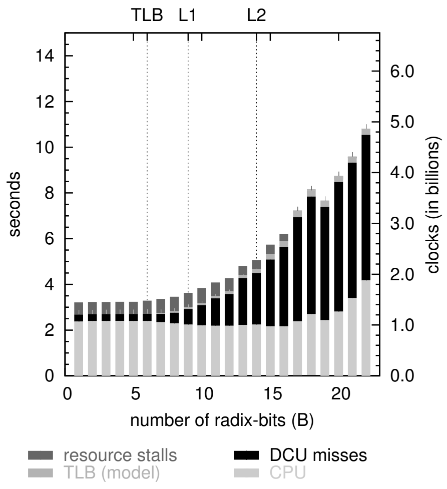
First, we conduct experiments with various numbers of radix-bits and passes, distributing the radix-bits evenly across the passes. Figure 5.3 shows an execution time breakdown for 1-pass radix-cluster ($|U| = 8$M) on each architecture. The pure CPU costs are nearly constant across all numbers of radix-bits, taking about 3 seconds on the Origin, 5.5 seconds on the Sun, 2.5 seconds on the PentiumIII, and a about 1.7 seconds on the Athlon. Memory and TLB costs are low with small numbers of radix-bits, but grow significantly with rising numbers of radix-bits. With more than 6 radix-bits, the number of clusters to be filled concurrently exceeds the number of TLB entries (64), causing the number of TLB misses to increase significantly. On the Origin and on the Sun, the execution time increases significantly due to their rather high TLB miss penalties. On the PentiumIII however, the impact of TLB misses is hardly visible due to its very low TLB miss penalty. The same holds for TLB$_1$ misses on the Athlon, while the impact of the more expensive TLB$_2$ misses is clearly visible. Analogously, the memory costs increase as soon as the number of clusters exceeds the number of L1 and L2 cache lines, respectively. Further, on the PentiumIII, "resource related stalls" (i.e., stalls due to functional unit unavailability) play a significant role. They make up one fourth of the execution time when the memory costs are low. When the memory



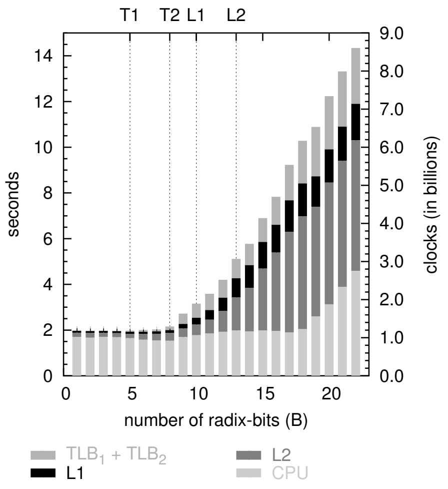
a) Origin2000



b) Sun Ultra



c) Intel PC



d) AMD PC

(Vertical grid lines indicate, where $H = 2^B$ equals $\#_{\text{TLB}}$, $\#_{\text{L1}}$, or $\#_{\text{L2}}$, respectively.)

Figure 5.3: Execution Time Breakdown of Radix-Cluster using one pass ($|U| = 8\text{M}$)

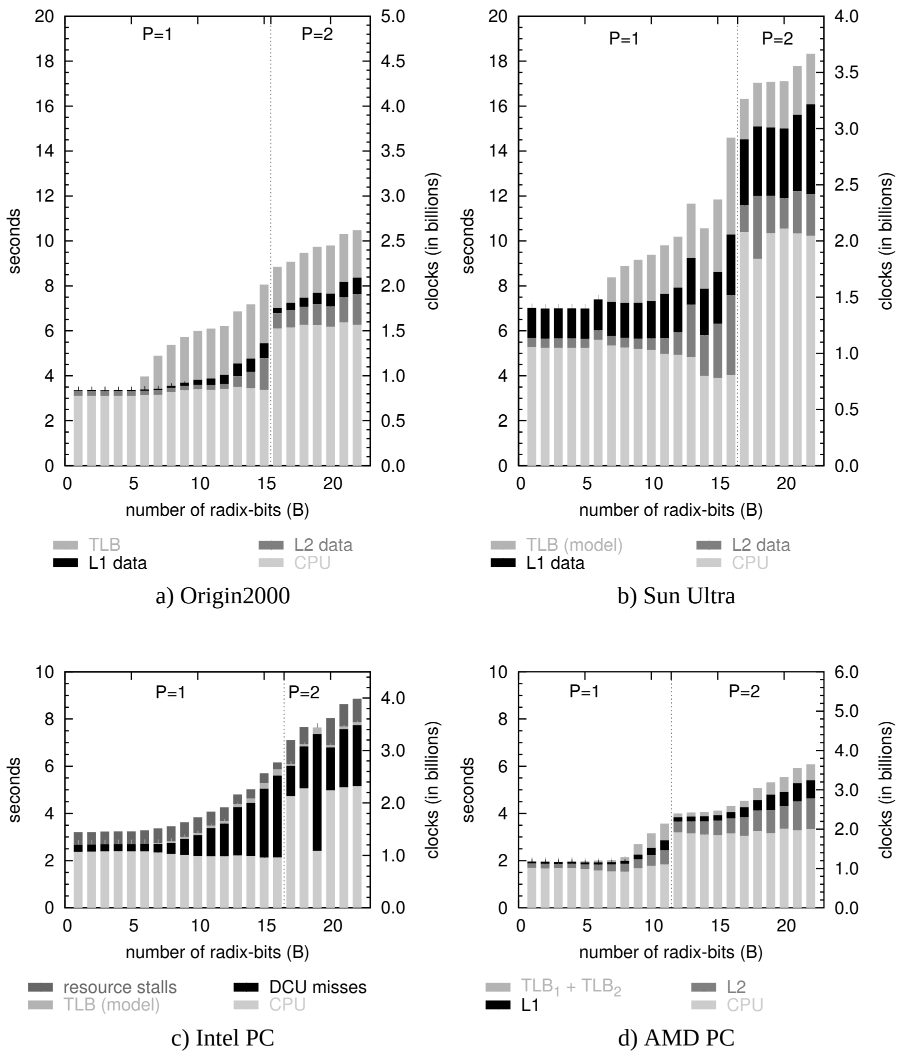


Figure 5.4: Execution Time Breakdown of Radix-Cluster using optimal number of passes ($|U|$ = 8M)

costs rise, the resource related stalls decrease and finally vanish completely, reducing the impact of the memory penalty. In other words, minimizing the memory access costs does not fully pay back on the PentiumIII, as the resource related stalls partly take over their part. The Athlon, however, does not seem to suffer from such “resource related stalls”.

Figure 5.4 depicts the breakdown for radix-cluster using the optimal number of

```
#define HASH(v) ((v≫7) XOR (v≫13) XOR (v≫21) XOR v)
typedef struct {
    int v1,v2; /* simplified binary tuple */
} bun;

radix_cluster(bun *dst[2^D], bun *dst_end[2^D]    /* output buffers for created clusters */
    bun *rel, bun *rel_end,                       /* input relation */
    int R, int D                                  /* radix and cluster bits */
){
  int M = (2^D - 1) ≪ R;
  for(bun*cur=rel; cur<rel_end; cur++) {
    int idx = (*hashFcn)(cur→v2)&M;                                    ||  int idx = HASH(cur→v2)&M;
    memcpy(dst[idx], cur, sizeof(bun));                                ||  *dst[idx] = *cur;
    if (++dst[idx]≥dst_end[idx]) REALLOC(dst[idx],dst_end[idx]);
  }
}
```

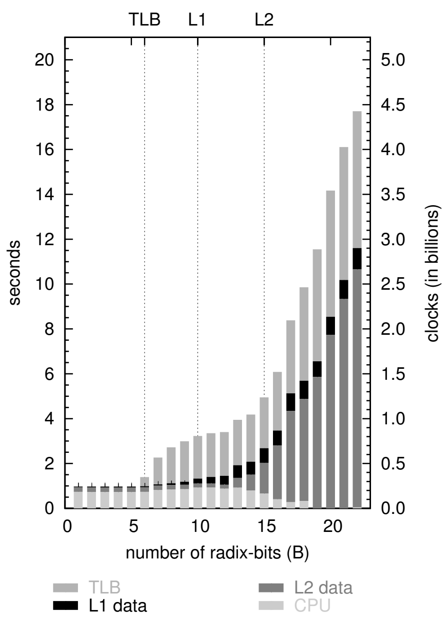
Figure 5.5: C language radix-cluster with annotated CPU optimizations (right)

passes. The idea of multi-pass radix-cluster is to keep the number of clusters generated per pass low—and thus the memory costs—at the expense of increased CPU costs. Obviously, the CPU costs are too high to avoid the TLB costs by using two passes with more than 6 radix-bits. Only with more than 15 radix-bits—i.e., when the memory costs exceed the CPU costs—two passes win over one pass. The only exception is the Athlon, where multi-pass radix-cluster benefits from the high clock speed, and hence, two passes outperform one pass already from 11 radix-bits onward.

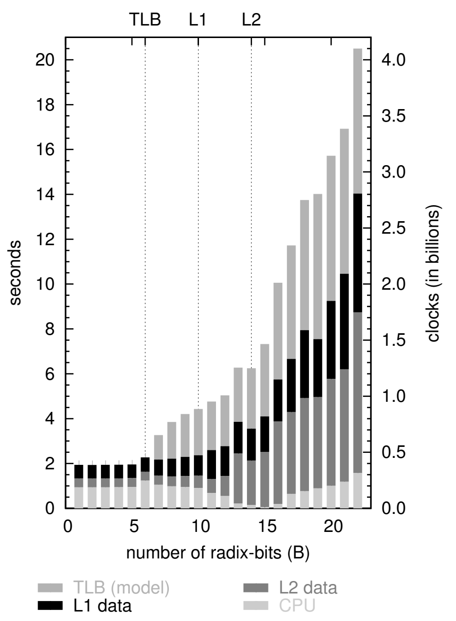
The only way to improve this situation is to reduce the CPU costs. Figure 5.5 shows the source code of our radix-cluster routine. It performs a single-pass clustering on the D bits that start R bits from the right (multi-pass clustering in $P > 1$ passes on $B = P * D$ bits is done by making subsequent calls to this function for pass $p = 1$ through $p = P$ with parameters $D_p = D$ and $R_p = (p - 1) * D$, starting with the input relation and using the output of the previous pass as input for the next). As the algorithm itself is already very simple, improvement can only be achieved by means of implementation techniques. We replaced the generic ADT-like implementation by a specialized one for each data type. Thus, we could inline the hash function and replace the `memcpy` by a simple assignment, saving two function calls per iteration.

Figure 5.6 shows the execution time breakdown for the optimized 1-pass radix-cluster. The CPU costs have reduced significantly, by almost a factor 4. Replacing the two function calls has two effects. First, some CPU cycles are saved. Second, the CPUs can benefit more from the internal parallel capabilities using speculative execution, as the code has become simpler and parallelization options more predictable. On the PentiumIII, the resource stalls have doubled, neutralizing the CPU improvement partly. We think the simple loop does not consist of enough instructions to fill the relatively long pipelines of the PentiumIII efficiently.

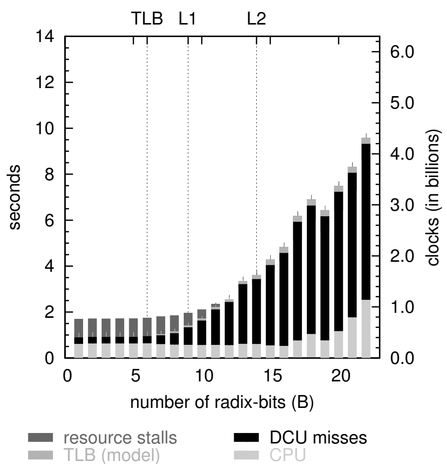
The results in Figure 5.7 indicate that with the optimized code, multi-pass radix-cluster is feasible already with smaller numbers of radix-bits. On the Origin, two passes win with more than 6 radix-bits, and three passes win with more than 13 radix-



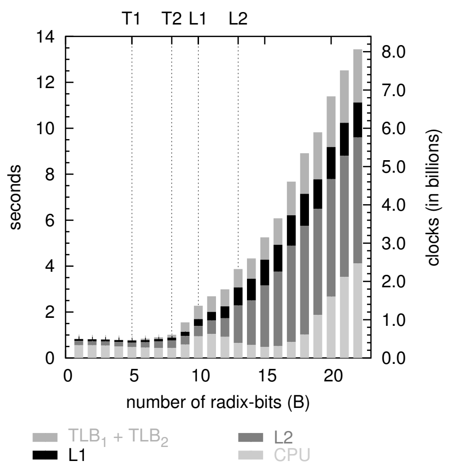
a) Origin2000



b) Sun Ultra



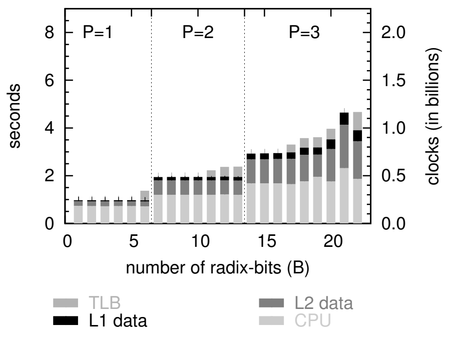
c) Intel PC



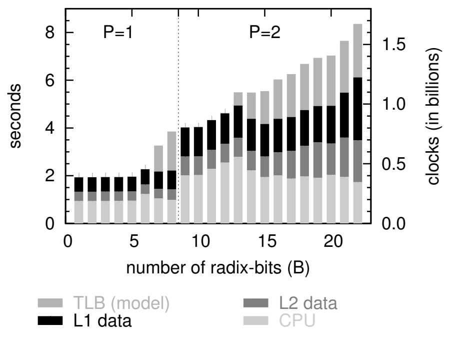
d) AMD PC

(Vertical grid lines indicate, where $H = 2^B$ equals $\#_{\mathrm{TLB}}$, $\#_{\mathrm{L1}}$, or $\#_{\mathrm{L2}}$, respectively.)

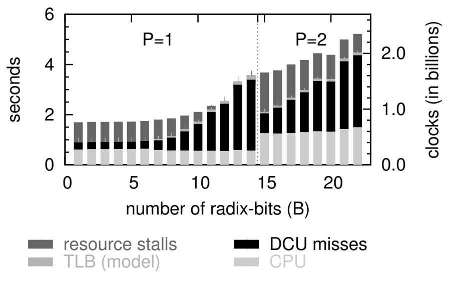
Figure 5.6: Execution Time Breakdown of optimized Radix-Cluster using one pass ($|U| = 8\mathrm{M}$)



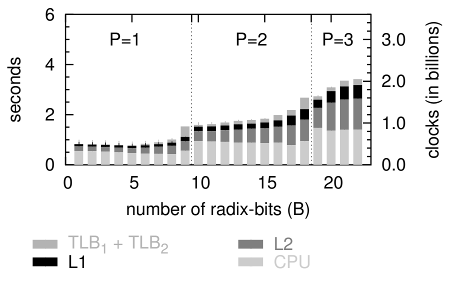
a) Origin2000



b) Sun Ultra



c) Intel PC



d) AMD PC

Figure 5.7: Execution Time Breakdown of optimized Radix-Cluster using optimal number of passes ($|U| = 8\text{M}$)

bits, thus avoiding TLB thrashing nearly completely. Analogously, the algorithm creates at most 512 clusters per pass on the AMD PC, avoiding L1 thrashing which is expensive due to the rather high L1 miss penalty on the Athlon. For the PentiumIII, however, the improvement is marginal. The severe impact of resource stalls with low numbers of radix-bits makes the memory optimization of multi-pass radix-cluster almost ineffective.

In order to effectively exploit the performance potential of our radix-cluster algorithm, we need to know the optimal number of passes to be used with a given number of radix-bits on a given machine. Obviously, we could give a rule of thumb for each machine we discussed here, like “Never use more than 6 bits (64 clusters) per pass on an Origin2000.” However, a more general approach is clearly desirable. Our proposal is to build a cost function for radix-cluster, that allows us to estimate and compare the performance for various numbers of passes. The best performance determines the optimal numbers of passes. We use the techniques we presented in Chapter 4 to create a cost function for radix-cluster estimating the total execution time as sum of pure CPU time and memory access time. With U describing the input data region, B denoting the requested number of radix-bits, and P being the number of passes, we get

$$T^{rc}(U,B,P) = T^{rc}_{\text{CPU}}(U,P) + T^{rc}_{\text{Mem}}(U,B,P).$$

The pure CPU costs are equal for all passes and independent of the number of bits used. For each pass, the pure CPU costs consist of a fixed start-up cost c^{rc}_0 and the per-tuple costs c^{rc}_1. We determine c^{rc}_0 and c^{rc}_1 using calibration as described in Section 4.6. Hence, we have

$$T^{rc}_{\text{CPU}}(U,P) = P \cdot (c^{rc}_0 + |U| \cdot c^{rc}_1).$$

The memory access cost is determined by the data access pattern. With each pass, radix-cluster sequentially reads the input and puts each tuple into one of $H_p = 2^{B_p}$ output clusters. Within each cluster, the access is sequential, but the clusters are accessed in a random order. Hence, we model the data access pattern of radix-cluster by

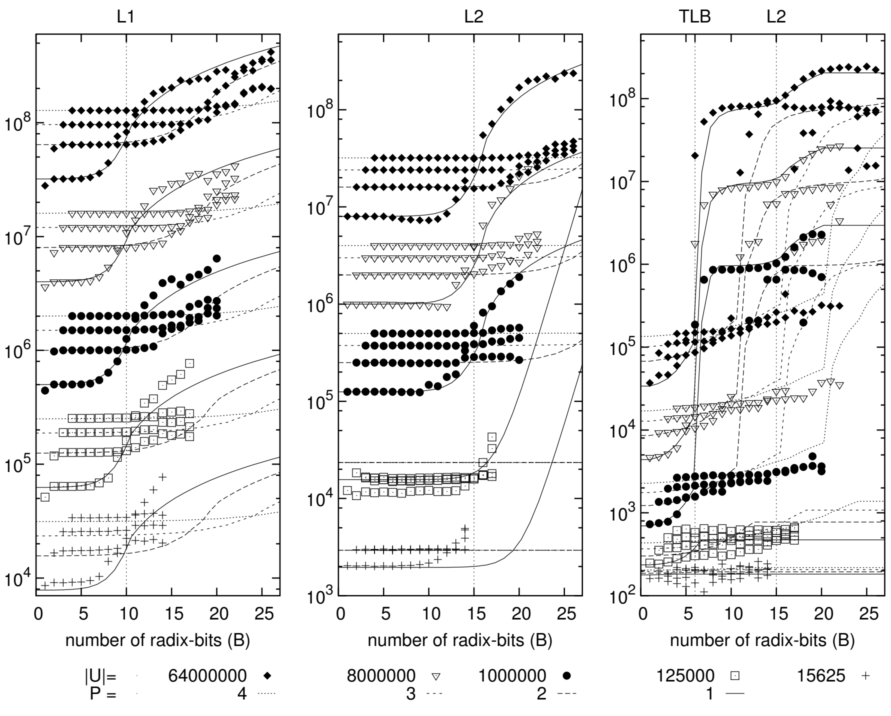
$$\{U_j\}|_{j=1}^{2^B} \leftarrow \mathit{radix\_cluster}(U,B,P) :$$
$$\oplus |_{p=1}^{P} \left(\mathsf{s\_trav^s}(U) \odot \mathsf{nest}\left(\{U_j\}|_{j=1}^{2^{B_p}}, 2^{B_p}, \mathsf{s\_trav^s}(U_j), \mathtt{ran}\right)\right).$$

$T^{rc}_{\text{Mem}}(U,B,P)$ is then calculated by estimating the number of cache misses and scoring them with their latency as described in Chapter 4.

Figure 5.8 compares the predicted events (lines) with the events observed during our experiments (points) on the Origin2000 for different cardinalities. The model accurately predicts the performance crucial behavior of radix-cluster, i.e., the steep raise in cache/TLB misses as soon as we generate more clusters per pass than there are cache lines respectively TLB entries. According to Figure 5.9, scoring the misses with their latency yields a reasonably accurate prediction of the total execution time on all architectures. The plots clearly reflect the impact of increasing cache and TLB misses on the execution time. Hence, our model provides sufficiently accurate information to determine the optimal number of passes for a given number of radix-bits. The effort needed to do so is quite limited. Consider the largest table in our experiments, containing $64,000,000$ tuples. In the worst case, we need to create $64,000,000$ clusters, each containing only a single tuple. Hence, we use at most 26 radix-bits. This means that theoretically at most 26 alternatives need to be compared. However, our experiments indicate, that we do not have to use more passes than we need to stay just within the smallest cache. With the 64-entry TLBs being the smallest caches in our case, there is no need to use less than 5 bits per pass, hence we can restrict the search to just $\left\lceil \frac{26}{5} \right\rceil = 6$ alternatives.

The question remaining is how to distribute the number of radix-bits over the passes. Of course, we could also use our cost model, here. However, with $\binom{B-1}{P-1}$ ways to distribute B radix-bits on P passes, the number of alternatives that need to be explored grows rapidly. For instance in our previous example, we need to distribute 26 bits on up to 6 passes, hence a total of

$$\sum_{j=1}^{6} \binom{26-1}{j-1} = 68,406$$



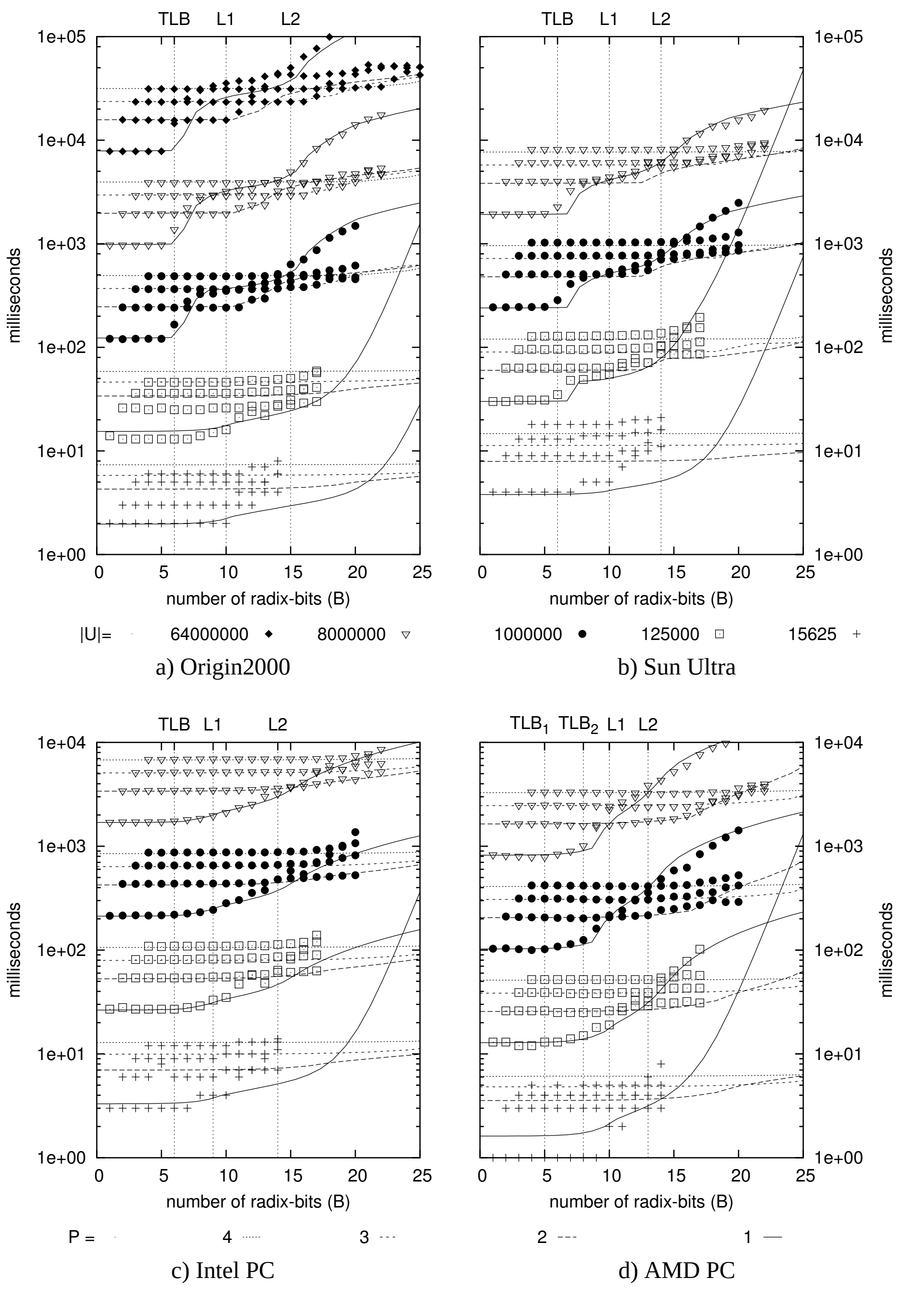
a) L1 misses b) L2 misses c) TLB misses

(Point types indicate cardinalities, line types indicate number of passes; vertical grid lines indicate, where the number of clusters created equals the number of TLB entries, L1, or L2 cache lines, respectively.)

Figure 5.8: Measured (points) and Modeled (lines) Events of Radix-Cluster (Origin2000)

candidates would need to be checked to find the best values for $P, B_1, \ldots, B_P$.

Alternatively, we consider the following empirical approach. We conducted another number of experiments, using a fix number of passes, but varying the number of radix-bits per pass. Figure 5.10 depicts the respective results for 4, 8, 12, 16, 20, and 24 radix-bits, using 2 passes. The x-axis shows $B + \frac{B_1}{5}$, hence, for each number of radix-bits ($B = B_1 + B_2$) there is a short line segment consisting of $B - 1$ points. The first (leftmost) point of each segment represents $B_1 = 1, B_2 = B - 1$, the last (rightmost) point represents $B_1 = B - 1, B_2 = 1$. The results show, that even distribution of radix-bits ($B_1 \approx B_2 \approx \frac{B}{2}$) achieves the best performance. Indeed, these results are quite intuitive, as even distribution of radix-bits minimizes the maximum number of bits per pass.



(Point types indicate cardinalities, line types indicate number of passes; vertical grid lines indicate, where the number of clusters created equals the number of TLB entries, L1, or L2 cache lines, respectively.)

Figure 5.9: Measured (points) and Modeled (lines) Performance of Radix-Cluster

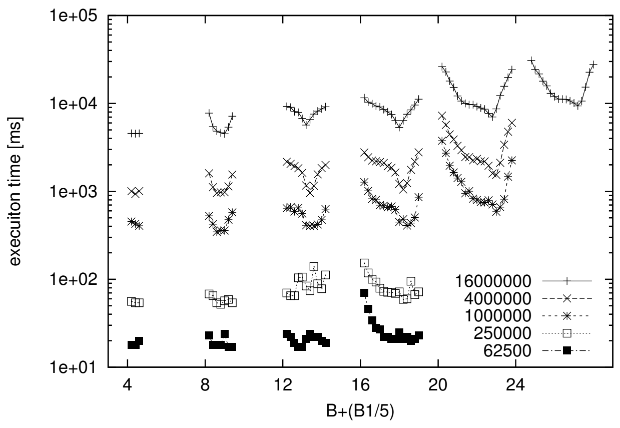


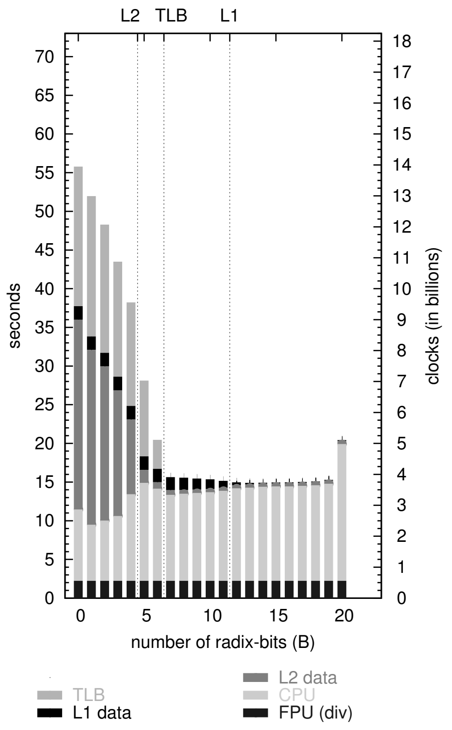
Figure 5.10: Bit-distribution for 2-pass Radix-Cluster

5.2.2.3 Isolated Partitioned Hash-Join Performance

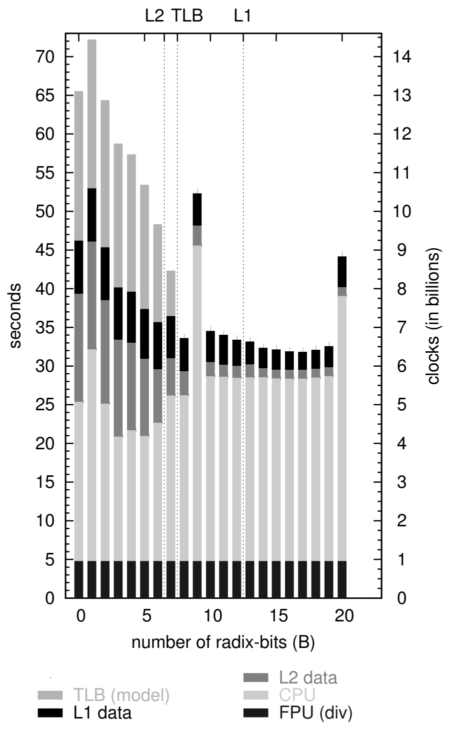
We now analyze the impact of the number of radix-bits on the pure join performance, not including the clustering costs. With 0 radix-bits, the join algorithm behaves like a simple non-partitioned hash-join.

The partitioned hash-join exhibits increased performance with increasing number of radix-bits. Figure 5.11 shows that this behavior is mainly caused by the memory costs. While the CPU costs are almost independent of the number of radix-bits, the memory costs decrease with increasing number of radix-bits. The performance increase flattens past the point where the entire inner cluster (including its hash table) consists of less pages than there are TLB entries (64). Then, it also fits the L2 cache comfortably. Thereafter, performance increases only slightly until the point that the inner cluster fits the L1 cache. Here, performance reaches its maximum. The fixed overhead by allocation of the hash-table structure causes performance to decrease when the cluster sizes get too small and clusters get very numerous. Again, the PentiumIII shows a slightly different behavior. TLB costs do not play any role, but “partial stalls” (i.e., stalls due to dependencies among instructions) are significant with small numbers of radix-bits. With increasing numbers of clusters, the partial stalls decrease, but then, resource stalls increase, almost neutralizing the memory optimization.

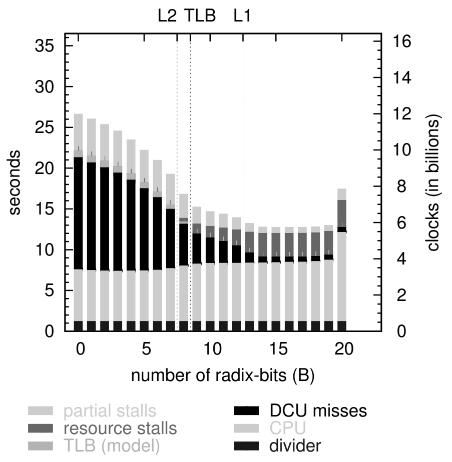
Like with radix-cluster, once the memory access is optimized, the execution of partitioned hash-join is dominated by CPU costs. Hence, we applied the same optimizations as above. We inlined the hash-function calls during hash build and hash probe as well as the compare-function call during hash probe and replaced two `memcpy` by simple assignments, saving five function calls per iteration. Further, we replaced the modulo division (“%”) for calculating the hash index by a bit operation (“&”). Figure 5.12 depicts the original implementation of our hash-join routine and the opti-



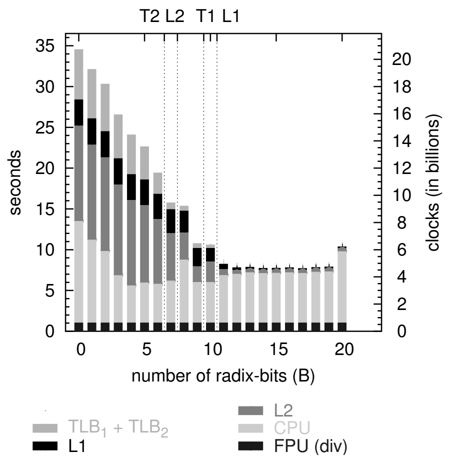
a) Origin2000



b) Sun Ultra



c) Intel PC



d) AMD PC

(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.11: Execution Time Breakdown of Partitioned Hash-Join ($|U| = |V| = 8M$)

```
hash_join(bun *dst, bun *end                                    /* start and end of result buffer */
    bun *outer, bun *outer_end, bun *inner, bun* inner_end,     /* inner and outer relations */
    int R                                                       /* radix bits */
){
  /* build hash table on inner */
  int pos=0, S=inner_end-inner, H=log2(S), N=2^H, M=(N-1)≪R;
  int next[S], bucket[N] = { -1 }; /* hash bucket array and chain-lists */
  for(bun *cur=inner; cur<inner_end; cur++) {
    int idx = ((*hashFcn)(cur→v2)≫R) % N;                   || int idx = HASH(cur→v2) & M;
    next[pos] = bucket[idx];
    bucket[idx] = pos++;
  }
  /* probe hash table with outer */
  for(bun *cur=outer; cur<outer_end; cur++) {
    int idx = ((*hashFcn)(cur→v2)≫R) % N;                   || int idx = HASH(cur→v2) & M;
    for(int hit=bucket[idx]; hit≥0; hit=next[hit]) {
      if ((*compareFcn)(cur→v2,inner[hit].v2)==0) {          || if ((cur→v2 == inner[hit].v2)) {
        memcpy(&dst→v1,&cur→v1, sizeof(int));                ||   dst→v1 = cur→v1;
        memcpy(&dst→v2,&inner[hit].v1, sizeof(int));         ||   dst→v2 = inner[hit].v1;
        if (++dst≥end) REALLOC(dst,end);
      }
    }
  }
}
```

Figure 5.12: C language hash-join with annotated CPU optimizations (right)

mizations we applied.

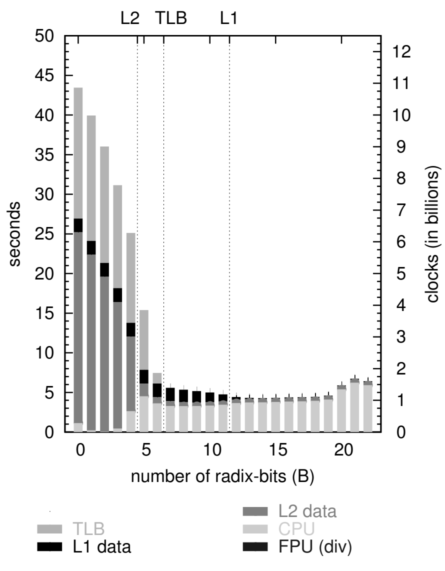
Figure 5.13 shows the execution time breakdown for the optimized partitioned hash-join. For the same reasons as with radix-cluster, the CPU costs are reduced by almost a factor 4 on the Origin and the Sun, by factor 3 on the PentiumIII, and by factor 2 on the Athlon. The expensive divisions have vanished completely. Additionally, the dependency stalls on the PentiumIII have disappeared, but the functional unit stalls remain almost unchanged, now taking about half of the execution time. It is interesting to note the 450 MHz PC outperforms the 250 MHz Origin on non-optimized code, but on CPU optimized code, where both RISC chips execute without any overhead, the PC actually becomes slower due to this phenomenon of resource stalls.

Like with radix-cluster, we will now create a cost function for partitioned hash-join that estimates the total execution time as sum of pure CPU time and memory access time. Let U and V describe the left (outer) and right (inner) input data region, respectively, and B denote the number of radix-bits that both U and V are clustered on. Further let W represent the output data region[2]

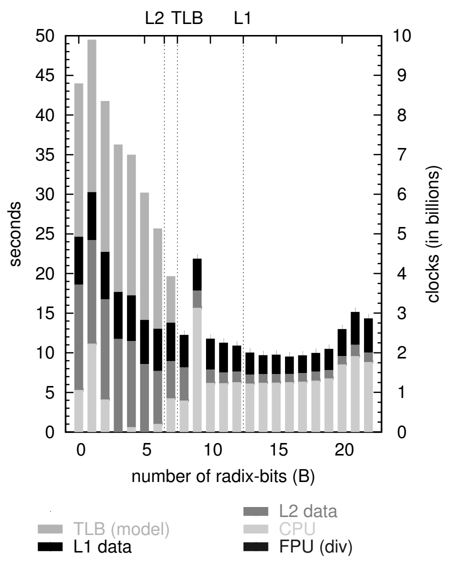
$$T^{phj}(U,V,B,W) = T^{phj}_{\text{CPU}}(U,V,B,W) + T^{phj}_{\text{Mem}}(U,V,B,W).$$

On each of the $H = 2^B$ pairs $\left\langle U_p, V_p \right\rangle$ of matching partitions, partitioned hash-join

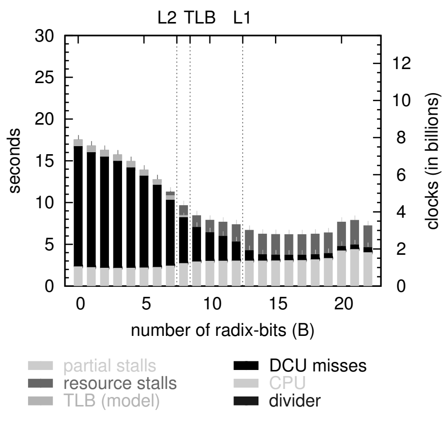
[2]Here, we use W to convey the results of a logical cost model, such as the estimates result size, to the physical cost model.



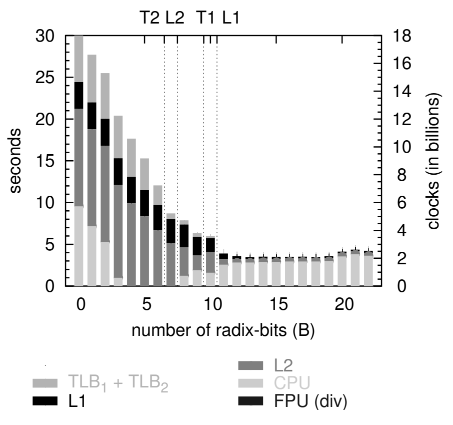
a) Origin2000



b) Sun Ultra



c) Intel PC



d) AMD PC

(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.13: Execution Time Breakdown: Optimized Partitioned Hash-Join ($|U| = 8\text{M}$)

performs a simple hash-join. The latter in turn consists of two phases, first building the hash table on the inner partition V_p, and then probing the outer partition U_p against the hash table. The pure CPU cost of the first phase is linear in the cardinality of V_p. The pure CPU cost of the second phase consists of two components. The first represents the actual hash lookup, and is in our case linear in the cardinality of U_p (i.e., independent of the size of the hash table). The second component reflects the creation of the actual result tuples, and is hence linear in the cardinality of W_p. Altogether, we get

$$\begin{aligned}
T^{hb}_{\mathrm{CPU}}(V_p) &= c^{hb}_0 + |V_p| \cdot c^{hb}_1 \\
T^{hp}_{\mathrm{CPU}}(U_p, W_p) &= c^{hp}_0 + |U_p| \cdot c^{hp}_1 + |W_p| \cdot c^{hp}_2 \\
T^{hj}_{\mathrm{CPU}}(U_p, V_p, W_p) &= T^{hb}_{\mathrm{CPU}}(V_p) + T^{hp}_{\mathrm{CPU}}(U_p, W_p) \\
T^{phj}_{\mathrm{CPU}}(U, V, B, W) &= \sum_{p=1}^{2^B} T^{hj}_{\mathrm{CPU}}(U_p, V_p, W_p)
\end{aligned}$$

We determine the respective cost constants using calibration as described in Section 4.6.

To obtain the memory access costs, we describe the memory access pattern of partitioned hash-join by combining the patterns of the single phases as follows:

$V'_p \leftarrow \mathit{hash\_build}(V_p)$:

$$\mathsf{s\_trav}^{\mathsf{s}}(V_p) \odot \mathsf{r\_trav}(V'_p)$$
$$=: \mathsf{build\_hash}(V_p, V'_p),$$

$W_p \leftarrow \mathit{hash\_probe}(U_p, V'_p)$:

$$\mathsf{s\_trav}^{\mathsf{s}}(U_p) \odot \mathsf{r\_acc}(|U_p|, V'_p) \odot \mathsf{s\_trav}^{\mathsf{s}}(W_p)$$
$$=: \mathsf{probe\_hash}(U_p, V'_p, W_p),$$

$W_p \leftarrow \mathit{hash\_join}(U_p, V_p)$:

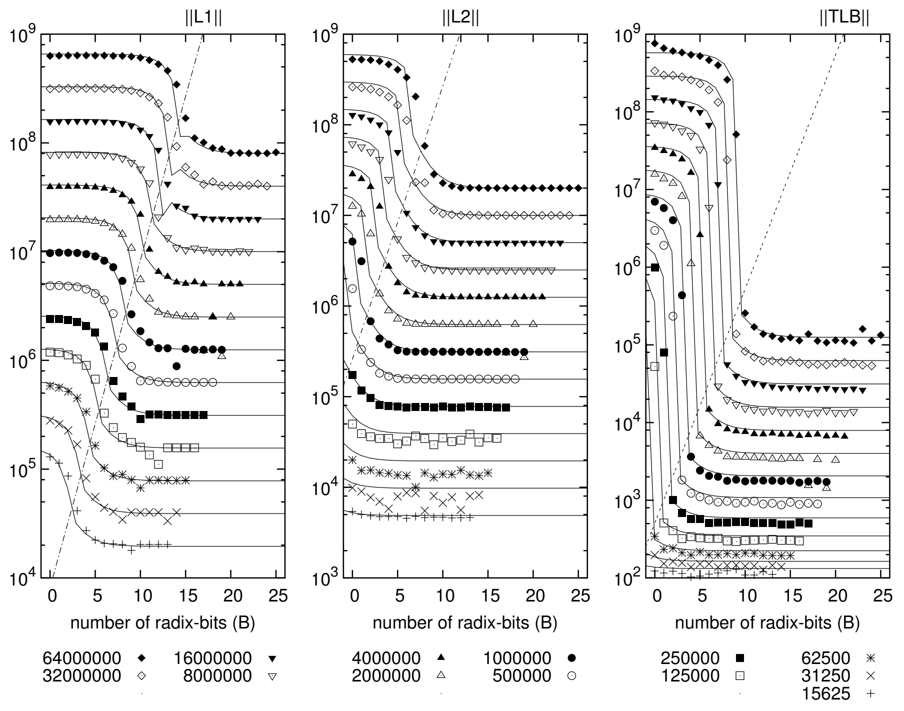
$$\mathsf{build\_hash}(V_p, V'_p) \oplus \mathsf{probe\_hash}(U_p, V'_p, W_p)$$
$$=: \mathsf{hash\_join}(U_p, V_p, W_p),$$

$\{W_p\}|^{2^B}_{p=1} \leftarrow \mathit{part\_hash\_join}(\{U_p\}|^{2^B}_{p=1}, \{V_p\}|^{2^B}_{p=1}, B)$:

$$\oplus|^{2^B}_{p=1}(\mathsf{hash\_join}(U_p, V_p, W_p)).$$

The actual cost functions are then derived using the techniques of Chapter 4.

Figure 5.14 compares the predicted events (lines) with the events observed during our experiments (points) on the Origin2000 for different cardinalities. The model accurately predicts the performance crucial behavior of partitioned hash-join, i.e., the significantly increased number of cache/TLB misses when using partition sizes that exceed the respective caches capacities. The plots in Figure 5.15 confirm that the estimated execution times match the actual performance on all architectures reasonably well.



a) L1 misses b) L2 misses c) TLB misses

(Point types indicate cardinalities; diagonal lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.14: Measured (points) and Modeled (lines) Events of Partitioned Hash-Join (Origin2000)

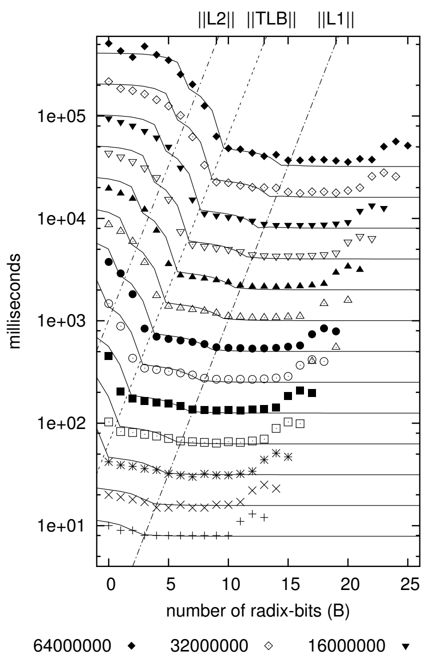
5.2.2.4 Overall Partitioned Hash-Join Performance

After having analyzed the impact of the tuning parameters on the clustering phase and the joining phase separately, we now turn our attention to the combined cluster and join costs. Radix-cluster gets cheaper for fewer radix-bits, whereas partitioned hash-join gets more expensive. Putting together the experimental data we obtained on both cluster- and join-performance, we determine the optimum number of B for given relation cardinality.

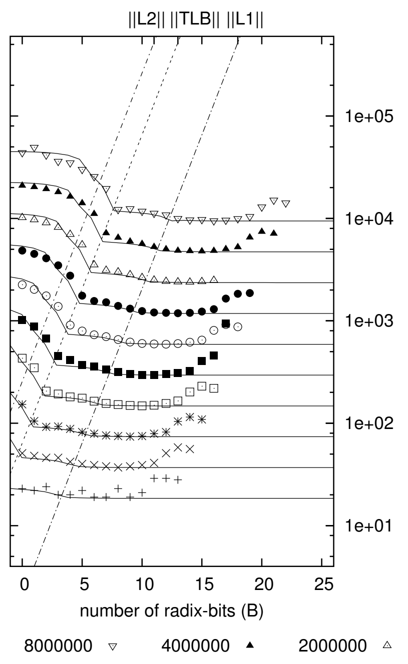
It turns out that there are three possible strategies, which correspond to the diagonals in Figure 5.15:

phash L2 partitioned hash-join on $B = log_2(|V'| * \overline{\underline{V'}}/||L2||)$ clustered bits, so the inner relation plus hash-table fits the L2 cache. This strategy was used in the work of Shatdal et al. [SKN94] in their partitioned hash-join experiments.

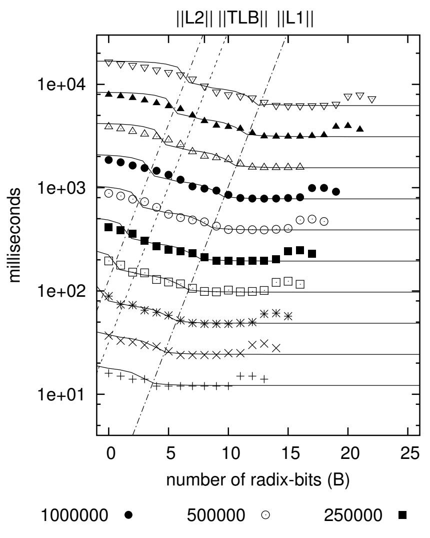
phash TLB partitioned hash-join on $B = log_2(|V'| * \overline{\underline{V'}}/||TLB||)$ clustered bits, so the



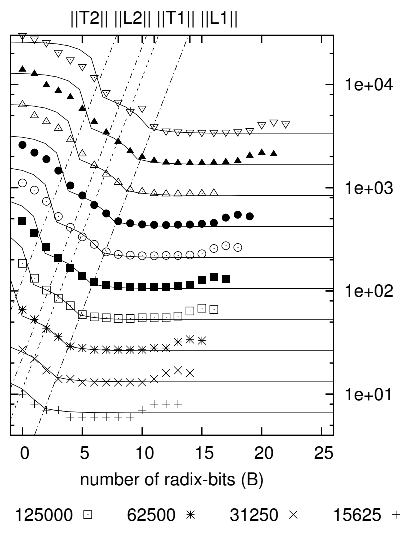
a) Origin2000



b) Sun Ultra



c) Intel PC



d) AMD PC

(Point types indicate cardinalities; diagonal lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.15: Measured (points) and Modeled (lines) Time of Partitioned Hash-Join

inner relation plus hash-table spans at most $|TLB|$ pages. Our experiments show a significant improvement of the pure join performance between phash L2 and phash TLB.

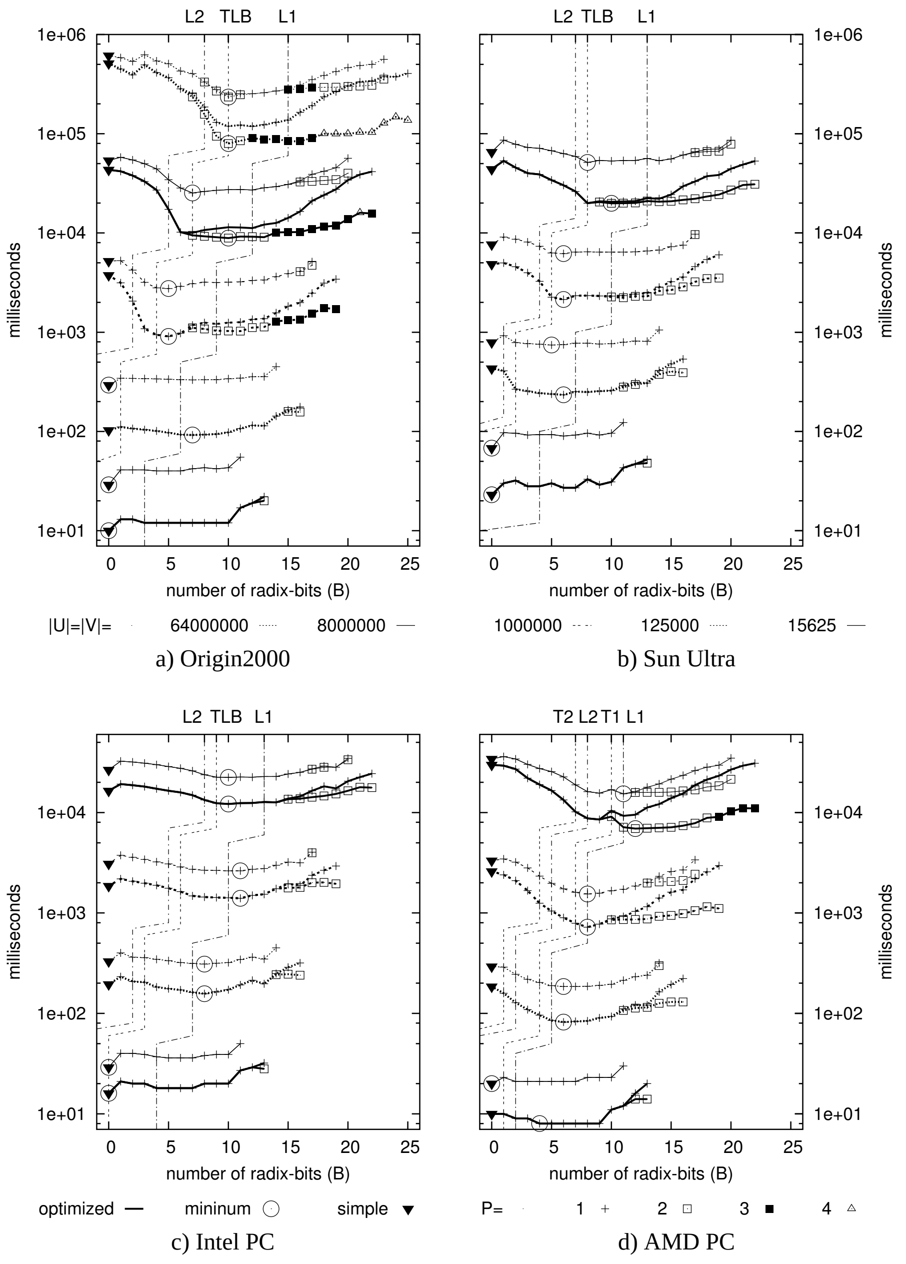
phash L1 partitioned hash-join on $B = log_2(|V'| * \overline{\underline{V'}}/||L1||)$ clustered bits, so the inner relation plus hash-table fits the L1 cache. This algorithm uses more clustered bits than the previous ones, hence it really needs the multi-pass radix-cluster algorithm (a straightforward 1-pass cluster would cause cache thrashing on this many clusters).

Figure 5.16 shows the overall performance for the original (thin lines) and the CPU-optimized (thick lines) versions of our algorithms, using 1-pass and multi-pass clustering. In most cases, phash TLB is the best strategy, performing significantly better than phash L2. On the Origin2000 and the Sun, the differences between phash TLB and phash L1 are negligible. On the PCs, phash L1 performs sightly better than phash TLB. With very small cardinalities, i.e., when the relations do not span more memory pages than there are TLB entries, clustering is not necessary, and the non-partitioned hash-join ("simple hash") performs best.

Using phash TLB seems to be a good choice the achieve reasonably good performance on all architectures without any optimization effort. However, our goal is to find and use the optimal strategy, also on architectures we have not evaluated here. Our cost models provide the necessary tools to achieve this goal. Using the cost models, we can—at runtime—estimate the costs of radix-cluster and partitioned hash-join for various numbers radix-bits (i.e., partition sizes). Thus, we can find the best setting for number of radix-bits and number of cluster passes to be used in the actual evaluation of the algorithms. Figure 5.17 compares the estimated performance (lines) with the results of our experiments (points) for 8M tuples. Our models tend to over estimate the "worst-case" costs, i.e., when using too large partitions for partitioned hash-join or too many partitions for radix-cluster. But in the crucial area around the optimal performance, they are rather accurate. Moreover, the models correctly predict the optimum in almost all cases. In case they do not, the cost of the estimated optimum only marginally differ from the actual optimum.

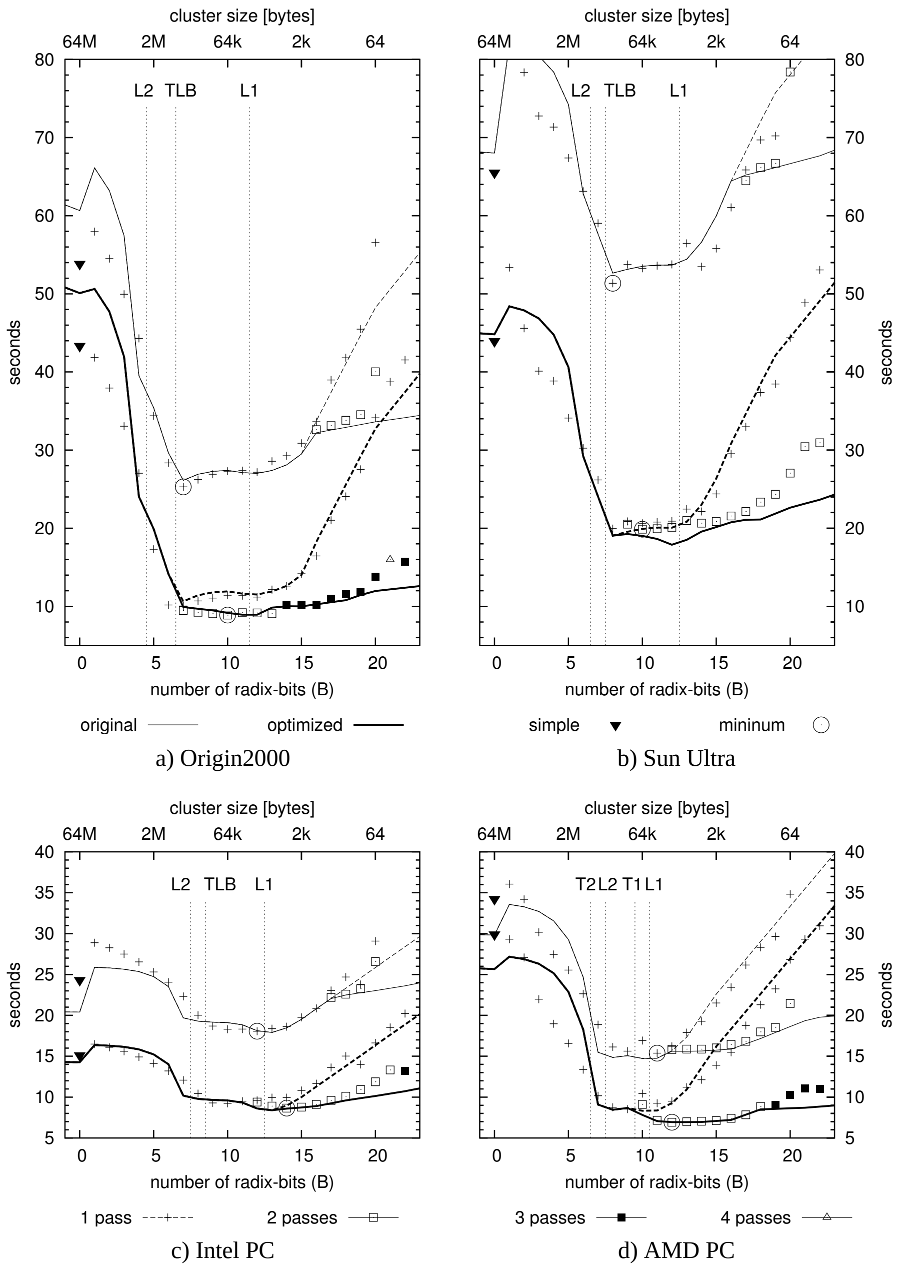
Further, the results in Figure 5.17 show that CPU and memory optimization support each other and *boost* their effects. The gain of CPU optimization for phash TLB is bigger than that for simple hash, and the gain of memory optimization for the CPU-optimized implementation is bigger than that for the non-optimized implementation. For example, for large relations on the Origin 2000, CPU optimization improves the execution time of simple hash by approximately a factor 1.25, whereas it yields a factor 3 with phash TLB. Analogously, memory optimization achieves an improvement of slightly less than a factor 2.5 for the original implementation, but more than a factor 5 for the optimized implementation. Combining both optimizations improves the execution time by almost a factor 10.

There are two reasons for the boosting effect to occur. First, modern CPUs try to overlap memory access with other useful CPU computations by allowing independent instructions to continue execution while other instructions wait for memory. In a



(Line types indicate cardinalities, point types indicate number of passes; diagonal lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.16: Overall Performance of Partitioned Hash-Join: non-optimized (thin lines) vs. optimized (thick lines) implementation



(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.17: Measured (points) and Modeled (lines) Overall Performance of Partitioned Hash-Join ($|U| = |V| = 8M$)

memory-bound load, much CPU computation is overlapped with memory access time, hence optimizing these computations has no overall performance effect (while it does when the memory access would be eliminated by memory optimizations). Second, an algorithm that allows memory access to be traded for more CPU processing (like radix-cluster), can actually trade more CPU for memory when CPU-cost are reduced, reducing the impact of memory access costs even more.

The Sun Ultra and the AMD PC achieve similar results like the Origin2000, although the absolute gains are somewhat smaller. With the Ultra, the CPU is so slow that trading memory for CPU less beneficial on this platform; with the AMD PC, the memory access costs are somewhat lower than on the Origin2000, thus offering less potential for improvements.

The overall effect of our optimizations on the PentiumIII is just over a factor 2. One cause of this is the low memory latency on the Intel PC, that limits the gains when memory access is optimized. The second cause is the appearance of the "resource-stalls", which surge in situations where all other stalls are eliminated (and both RISC architectures are really steaming). We expect, though, that future PC hardware with highly parallel IA-64 processors and new Rambus memory systems (that offer high bandwidth but high latencies) will show a more RISC-like performance on our algorithms.

5.3 Radix-Join

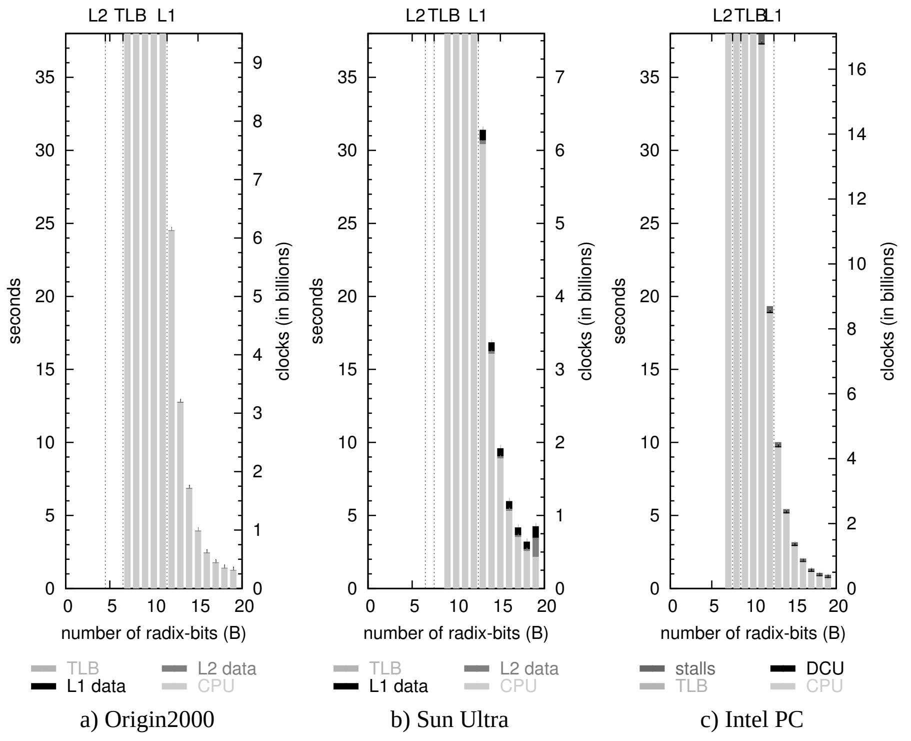
In this section, we present our *radix-join* algorithm as an alternative for the partitioned hash-join. Radix-join makes use of the very fine clustering capabilities of radix-cluster. If the number of clusters H is high, the radix-clustering has brought the potentially matching tuples near to each other. As cluster sizes are small, a simple nested loop is then sufficient to filter out the matching tuples. Radix-join is similar to hash-join in the sense that the number H should be tuned to be the relation cardinality $|U|$ divided by a small constant; just like the length of the bucket-chain in a hash-table. If this constant gets down to 1, radix-join degenerates to sort/merge-join, with radix-sort [Knu68] employed in the sorting phase.

5.3.1 Isolated Radix-Join Performance

Figure 5.18 shows the execution time breakdown for our radix-join algorithm ($|U|$ = 1M). On all three architectures[3], radix-join performs the better the more radix-bits are used, i.e., the smaller the clusters are. With increasing cluster size, execution time increases rapidly due to the nested-loop characteristic of radix-join. Only cluster sizes that fit into the L1 cache are reasonable. Hence, memory access costs are small, and the performance is dominated by CPU costs.

Like with radix-cluster and partitioned hash-join, we optimize our radix-join implementation (cf., Figure 5.19) in order to reduce the CPU costs. Figure 5.20 depicts the results for the optimized radix-join. For very small clusters, the optimizations

[3]Due to hardware problems, we have no results of radix-join on the AMD PC.



(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.18: Execution Time Breakdown of Radix-Join ($|U| = |V| = 1\text{M}$)

yield an improvement of factor 1.25 (on the PC) to 1.5 (on the Origin2000). For larger clusters, we observe similar improvements as with partitioned hash-join: factor 4.5 on the RISC architectures and factor 2.5 on the PC. We also note that functional unit stalls (resource stalls) become clearly visible on the PC, as soon as the cluster size reaches and exceeds L1 cache size.

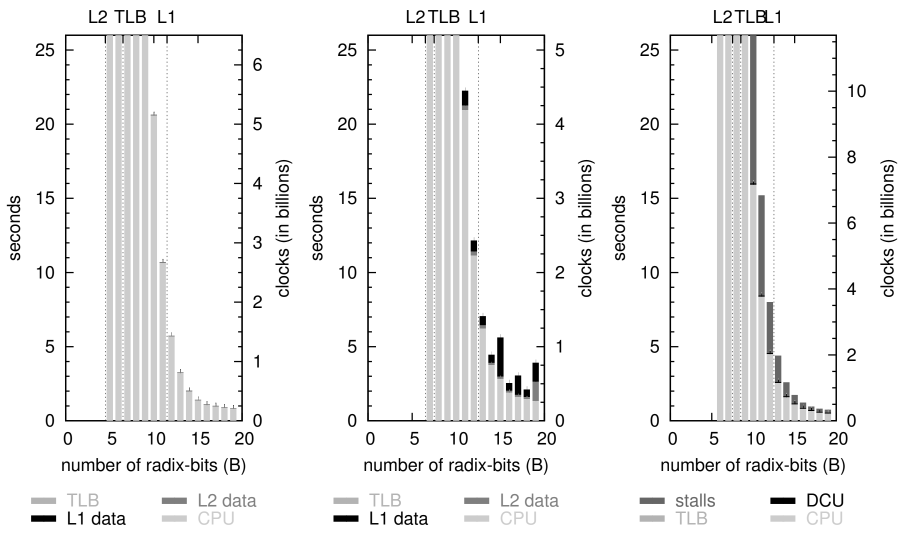
We follow our usual schema to model the cost of radix-join.

$$T^{rj}(U, V, B, W) = T^{rj}_{\text{CPU}}(U, V, B, W) + T^{rj}_{\text{Mem}}(U, V, B, W).$$

The total CPU costs are simply the sum of the CPU costs of executing a simple nested-loop-join on all matching pairs of partitions. A simple nested-loop-join incurs constant start-up cost (c_0^{nlj}), quadratic comparison cost (c_1^{nlj}), and linear result creation cost

```
nested_loop(bun *dst, bun *end                                   /* start and end of result buffer */
   bun *outer, bun *outer_end, bun *inner, bun* inner_end,       /* inner and outer relations */
){
   for(bun *outer_cur=outer; outer_cur < outer_end; outer_cur++) {
      for(bun *inner_cur=inner; inner_cur < inner_end; outer_cur++) {
         if ((*compareFcn)(outer_cur→v2,inner_cur→v2)==0) {  ||  if ((outer_cur→v2 == inner_cur→v2)) {
            memcpy(&dst→v1,&outer_cur→v1, sizeof(int));      ||     dst→v1 = outer_cur→v1;
            memcpy(&dst→v2,&inner_cur→v1, sizeof(int));      ||     dst→v2 = inner_cur→v1;
            if (++dst≥end) REALLOC(dst,end);
         }
      }
   }
}
```

Figure 5.19: C language nested-loop with annotated CPU optimizations (right)



a) Origin2000 b) Sun Ultra c) Intel PC

(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.20: Execution Time Breakdown of optimized Radix-Join ($|U| = |V| = 1\text{M}$)

(c_2^{nlj}).

$$T_{\text{CPU}}^{nlj}(U_p, V_p, W_p) = c_0^{nlj} + |U_p| \cdot |V_p| \cdot c_1^{nlj} + |W_p| \cdot c_2^{nlj}$$
$$T_{\text{CPU}}^{rj}(U, V, B, W) = \sum_{p=1}^{2^B} T_{\text{CPU}}^{nlj}(U_p, V_p, W_p)$$

We note that calibrating the CPU cost factors as described in Section 4.6 ensures that the resource stalls on the PC (see above) are properly included in the cost factors.

To get the memory access cost function for radix-join, we model its memory access pattern as follows.

$$\begin{aligned} W_p \leftarrow \textit{nested\_loop\_join}(U_p, V_p) :&\\ &\mathsf{s\_trav}^{\mathsf{s}}(U_p) \odot \mathsf{rs\_trav}^{\mathsf{s}}(|U_p|, \mathtt{uni}, V_p) \odot \mathsf{s\_trav}^{\mathsf{s}}(W_p)\\ &\qquad =: \mathsf{nl\_join}(U_p, V_p, U_p) \end{aligned}$$

$$\begin{aligned} \{W_p\}|_{p=1}^{2^B} \leftarrow \textit{radix\_join}(\{U_p\}|_{p=1}^{2^B}, \{V_p\}|_{p=1}^{2^B}, B) :&\\ &\oplus |_{p=1}^{2^B}(\mathsf{nl\_join}(U_p, V_p, W_p)). \end{aligned}$$

Figure 5.21 confirms the accuracy of our model (lines) for the number of L1, L2, and TLB misses on the Origin2000, and for the elapsed time on all architectures.

5.3.2 Overall Radix-Join Performance

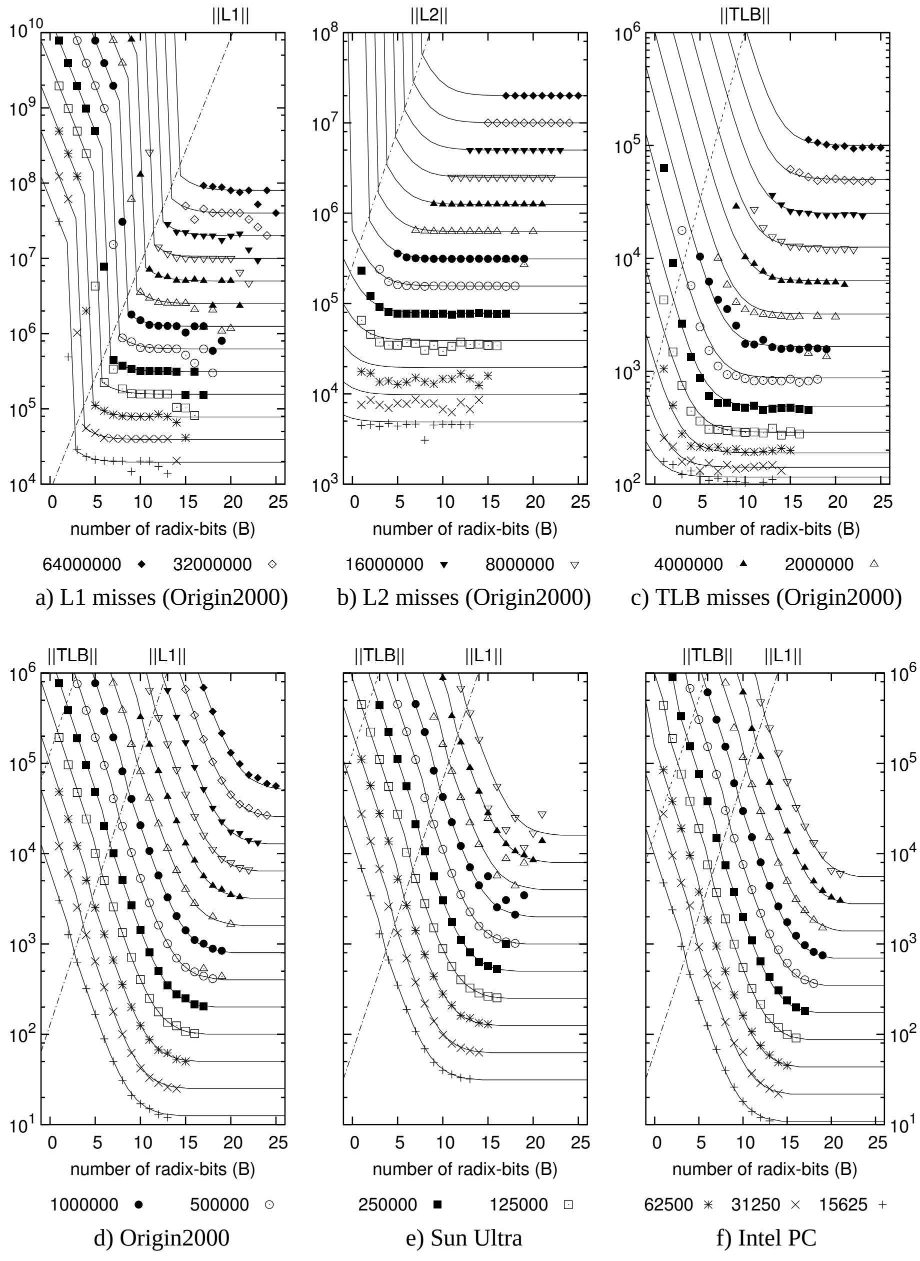
Figure 5.22 shows the overall performance of radix-join, i.e., including radix-cluster of both input relations. For larger clusters, the performance is dominated by the high CPU costs of radix-join. Only very small clusters with up to 8 tuples are reasonable, requiring multi-pass radix-clustering in most cases.

5.3.3 Partitioned Hash-Join vs. Radix-Join

Finally, Figure 5.23 compares the overall performance of partitioned hash-join and radix-join. With the original non-optimized implementation, the optimal performance of radix-join (circled points on thin lines) is almost the same as the optimal partitioned hash-join performance (circled points on thick lines). With code optimizations applied, optimal radix-join performance is never better than partitioned hash-join using the same number of radix-bits.

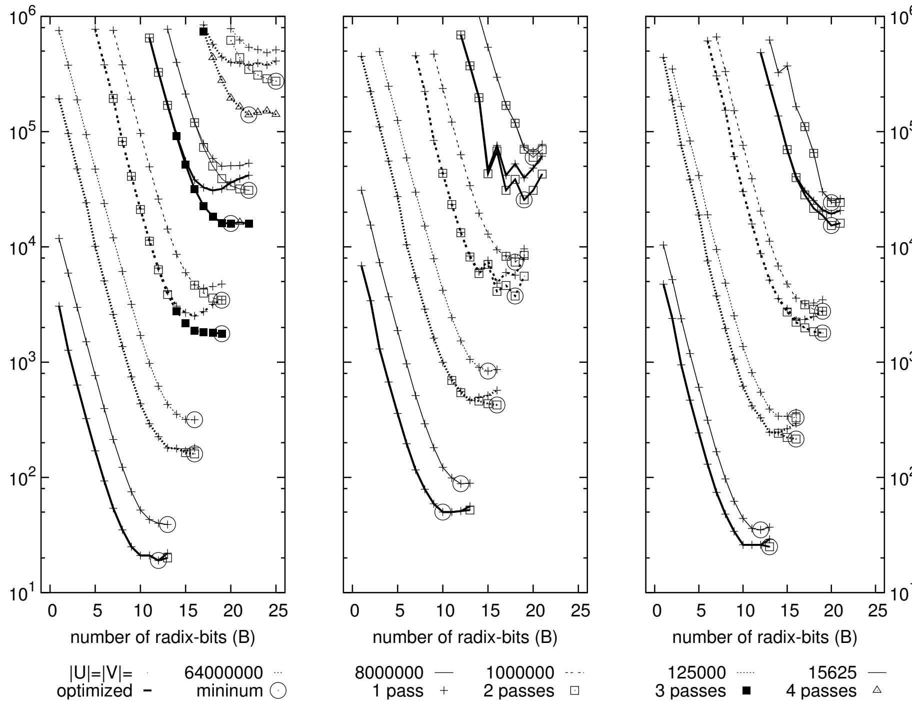
5.4 Evaluation

In the previous sections, we have demonstrated that performance of large equi-joins can be strongly improved by combining techniques that optimize memory access and CPU resource usage. As discussed in Section 3.4.3, hardware trends indicate that the effects of such optimizations will become even larger in the future, as the memory access bottleneck will deepen and future CPUs will have even more parallel resources. In the following, we discuss the more general implications of these findings to the field of database architecture.



(Events are in absolute numbers; times are in milliseconds. Point types indicate cardinalities; diagonal lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.21: Measured (points) and Modeled (lines) Events (Origin2000) Performance of Radix-Join



a) Origin2000 b) Sun Ultra c) Intel PC

(Times are in milliseconds. Line types indicate cardinalities, point types indicate number of passes.)

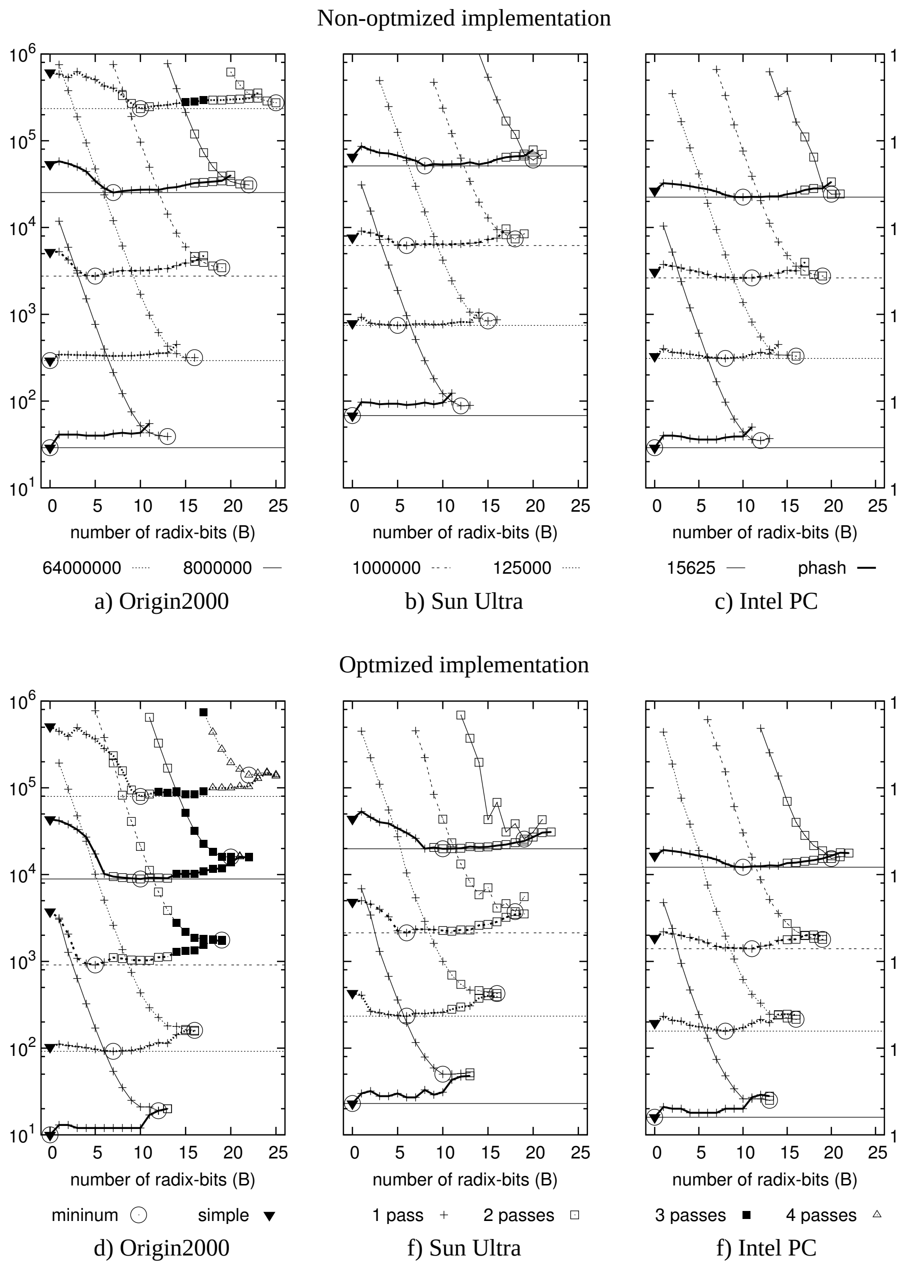
Figure 5.22: Overall Performance of Radix-Join: non-optimized (thin lines) vs. optimized (thick lines) implementation

5.4.1 Implications for Implementation Techniques

Implementation techniques strongly determine how CPU and memory are used in query processing, and have been the subject of study in the field of main-memory database engineering [DKO+84], where query processing costs are dominated by CPU processing. First, we present some rules of thumb, that specifically take into account the modern hardware optimization aspects, then we explain how they were implemented in Monet:

- *use the most efficient algorithm.* Even the most efficient implementation will not make a sub-optimal algorithm perform well. A more subtle issue is tuning algorithms with the optimal parameters.

- *minimize memory copying.* Buffer copying should be minimized, as it both wastes CPU cycles and also causes spurious main-memory access. As function



(Times are in milliseconds. Line types indicate cardinalities, point types indicate number of passes.)

Figure 5.23: Partitioned Hash-Join (thick lines) vs. Radix-Join (thin lines): non-optimized (top) and optimized (bottom)

calls copy their parameters on the stack, they are also a source of memory copying, and should be avoided in the innermost loops that iterate over all tuples. A typical function call overhead is about 20 CPU cycles.

- *allow compiler optimizations.* Techniques like memory prefetching, and generation of parallel EPIC code in the IA-64, rely on compilers to detect independence of certain statements. These compiler optimizations work especially well if the hotspot of the algorithm is one simple loop that is easily analyzable for the compiler. Again, performing function calls in these loops, force the compiler to assume the worst (side effects) and prevent optimizations from taking place. This especially holds in database code, where those function calls cannot be analyzed at compile time, since the database atomic type interface makes use of C dereferenced calls on a function-pointer looked up in an ADT table, or C++ late-binding methods.

As an example of correctly tuning algorithms, we discuss the (non-partitioned) hash-join implementation of Monet that uses a simple bucket-chained hash-table. In a past implementation, it used a default mean bucket chain length of four [BMK99], where actually a length of one is optimal (perfect hashing). Also, we had used integer division (modulo) by a prime-number (the number of hash buckets) to obtain a hash-bucket number, while integer division costs 40-80 cycles on current CPUs. Later, we changed the number of hash buckets to be a power of 2 (i.e., $N = 2^x$), and hence, we could replace the expensive modulo division by a much cheaper bit-wise AND with $N-1$. Such simple tuning made the algorithm more than 4 times faster.

In order to minimize copying, Monet does not do explicit buffer management, rather it uses virtual memory to leave this to the OS. This avoids having to copy tuple segments in and out of a buffer manager, whenever the DBMS accesses data. Monet maps large relations stored in a file into virtual memory and accesses it directly. Minimizing memory copying also means that pointer swizzling is avoided at all time by not having hard pointers and value-packing in any data representation.

Functions calls are minimized in Monet by applying *logarithmic code expansion* [Ker89]. Performance-critical pieces of code, like the hash-join implementation, are replicated in specific functions for the most commonly used types. For example, the hash-join is separated in an integer-join, a string-join, etc., and an ADT join (that handles all other types). The specific integer-join processes the table values directly as C integers, without calling a hash-function for hashing, or calling a comparison function when comparing two values. The same technique is applied for constructing the result relation, eliminating function calls for inserting the matching values in the result relation. To make this possible, the type-optimized join implementations require the result to have a fixed format: a join index containing OIDs (in Monet the result of joining two BATs is again a BAT, so it has a fixed binary format, and typical invocations produce a BAT with matching OID pairs). In this way, all function calls can be removed from an algorithm in the optimized cases. For the non-optimized cases, the (slower) but equivalent implementation is employed that uses ADT method calls for manipulating values. The Monet source code is kept small by generating both the optimized and ADT code instantiations with a macro package from one template algorithm. We refer to [BK99] for a detailed discussion of this subject.

5.4.2 Implications for Query Processing Algorithms

Our join experiments demonstrated that performance can strongly improve when algorithms that have a random memory access pattern are tuned, in order to ensure that the randomly accessed region does not exceed the cache size (be it L1, L2, or TLB). In the case of join, we confirmed results of Shatdal et al. who had proposed a partitioned hash-join such that each partition joined fits the L2 cache [SKN94], and showed that the beneficial effect of this algorithm is even stronger on modern hardware. Secondly, we introduced a new partitioning algorithm called *radix-cluster* that performs multiple passes over the data to be partitioned but earns back this extra CPU work with much less memory access costs when the number of partitions gets large.

We believe that similar approaches can be used to optimize algorithms other than equi-join. For instance, Ronström [Ron98] states, that a B-tree with a block-size equal to the L2 cache line size as a main-memory search accelerator, now outperforms the traditionally known-best main-memory T-tree search structure [LC86a]. As another example, memory cost optimizations can be applied to sorting algorithms (e.g., radix-cluster followed by quick-sort on the partitions), and might well change the trade-offs for other well-known main-memory algorithms (e.g., radix-sort has a highly cachable memory access pattern and is likely to outperform quick-sort).

Main-memory cost models are a prerequisite for tuning the behavior of an algorithm to optimize memory cache usage, as they allow to make good optimization decisions. Our work shows that such models can be obtained and how to do it. First, we show with our *calibration tool* how all relevant hardware characteristics can be retrieved from a computer system automatically. This calibrator does not need any OS support whatsoever, and should in our opinion be used in modern DBMS query optimizers. Secondly, we present a methodological framework that first characterizes the memory access pattern of an algorithm to be modeled in a formula that counts certain hardware events. These computed events are then scored with the calibrated hardware parameters to obtain a full cost model. This methodology represents an important improvement over previous work on main-memory cost models [LN96, WK90], where performance is characterized on the coarse level of a procedure call with “magical” cost factors obtained by profiling. We were helped in formulating this methodology through our usage of hardware event counters present in modern CPUs.

5.4.3 Implications for Disk Resident Systems

We think our findings are not only relevant to main-memory databases engineers. Vertical fragmentation and memory access costs have a strong impact on performance of database systems at a macro level, including those that manage disk-resident data. Nyberg et al. [NBC$^+$94] stated that techniques like software assisted disk-striping have reduced the I/O bottleneck; i.e., queries that analyze large relations (like in OLAP or Data Mining) now read their data faster than it can be processed. Hence the main performance bottleneck for such applications is shifting from I/O to memory access. We therefore think that, as the I/O bottleneck decreases and the memory access bottleneck increases, main-memory optimization of both data structures and algorithms—like described in this paper—will become a prerequisite to any DBMS for exploiting the power of custom hardware.

In Monet, we delegate I/O buffering to the OS by mapping large data files into virtual memory, hence treat management of disk-resident data as memory with a large granularity (a memory page is like a large cache line). This is in line with the consideration that disk-resident data is the bottom level of a memory hierarchy that goes up from the virtual memory, to the main memory through the cache memories up to the CPU registers (Figure 3.2). Algorithms that are tuned to run well on one level of the memory, also exhibit good performance on the lower levels.

5.5 Conclusion

We have shown what steps are taken in order to optimize the performance of large main-memory joins on modern hardware. To achieve better usage of scarce memory bandwidth, we recommend using vertically fragmented data structures. We refined partitioned hash-join with a new partitioning algorithm called radix-cluster, that prevents performance becoming dominated by memory latency (avoiding the memory access bottleneck). Exhaustive equi-join experiments were conducted on modern SGI, Sun, Intel, and AMD hardware. We formulated detailed analytical cost models that explain why this algorithm makes optimal use of hierarchical memory systems found in modern computer hardware and very accurately predict performance on all three platforms. Further, we showed that once memory access is optimized, CPU resource usage becomes crucial for the performance. We demonstrated, how CPU resource usage can be improved by using appropriate implementation techniques. The overall speedup obtained by our techniques can be almost an order of magnitude. Finally, we discussed the consequences of our results in a broader context of database architecture, and made recommendations for future systems.

Chapter 6

Summary and Outlook

Database systems pervade more and more areas of business, research, and everyday life. With the amount of data stored in databases growing rapidly, performance requirements for database management system increase as well. This thesis analyzes, why simply using newer and more powerful hardware is not sufficient to solve the problem. Quantifying the interaction between modern hardware and database software enables use to design performance models that make up vital tools for improving database performance.

6.1 Contributions

Focusing on main-memory database technology, this thesis describes research that aims at understanding, modeling, and improving database performance.

Hardware Trends and MMDBMS performance

We empirically analyzed how main-memory database software and modern hardware do interact. During our study, we found that starting with simple but representative experiments is both necessary and sufficient to identify the essential performance characteristics. Moreover, hardware counters as present in many contemporary CPUs have proved to be indispensable tools for tracking down the most performance-crucial events, such as cache misses, TLB misses, resource stalls, or branch mispredictions. Our experiments disclosed that memory access has become the primary performance bottleneck on various hardware platforms ranging from small of-the-shelf PCs to large high-performance servers. The reason for this can be found in the hardware trends of the last two decades. While CPU speed has been growing rapidly, memory access latency has hardly improved. Thus, the performance gap between CPU and memory speed did widen out. Ongoing and promised hardware development indicates that this trend will last for the foreseeable future. So far, database technology has not payed much attention to these trends; especially in cost modeling, main-memory access has completely been ignored.

Main-Memory Cost Modeling

Using the knowledge gained during our analysis, we developed precise physical cost models for core database algorithms. The main goal of these cost models is to accurately predict main-memory access costs. Furthermore, the new models provide two further innovations compared to traditional database cost models. First, we generalized and simplified the process of creating cost functions for arbitrary database operations. For this purpose, we introduced the concept of data access patterns. Instead of specifying the often complex cost functions for each algorithm "by hand", database software developers only need to describe the algorithms' data access behavior as simple combinations of a few basic access patterns. The actual cost functions can then be derived automatically by applying the rules developed in this thesis. The second innovation addresses the hardware dependency inherent to physical cost models. The principle idea is to have a single common cost model instead of individual cost models for each hardware platform. To achieve this goal, we introduced a novel unified hardware model for hierarchical memory systems. The hardware model allows an arbitrary number of hierarchies, such as several levels of CPU cache, main-memory, and secondary storage. For each level, it stores performance characteristic parameters such as size, access granularity, access latency, access bandwidth, and associativity. Thus, we could design a general cost model that is parameterized by the characteristics as represented in the hardware model. In order to have the cost model automatically instantiated on a new hardware platform, we developed a calibration tool that measures the respective parameters without human interaction. The "Calibrator" is freely available for download from our web site and has become a popular[1] tool to assess computer hardware not only within the database community.

Cache-Conscious Query Processing

Disappointed by the poor performance of standard database technology on supposedly powerful hardware, but enlightened with the insights gained during our analysis and modeling exercises, we developed new hardware-conscious algorithms and coding techniques. Focusing on equi-joins, we introduced radix-algorithms for partitioned hash-join. The basic idea is to reduce memory access costs by restricting random access patterns to the smallest cache size, and thus reduce the number of cache misses. This exercise demonstrates that our cost models — next to being fundamental for database query optimization — serve two further purposes. First, they help us to understand the details and thus enable us to design proper algorithms. Second, the query engine can use the cost functions to tune the algorithms at runtime.

Once memory access is optimized, CPU costs become dominant, mainly as standard database code is too complex to allow CPUs to efficiently exploit their internal parallel processing potentials. We presented coding techniques that achieve significant performance gains by reducing the number of function calls, branches, and data dependencies.

[1]more than 12,000 downloads in 2 years

We have performed extensive experimentation on a range of hardware platforms that confirmed both the accuracy and the portability of our cost model. Moreover, the experimental evaluations showed, that combining memory access and CPU optimization yields a performance gain of up to an order of magnitude.

6.2 Conclusion

The work presented in this thesis has demonstrated that current main-memory database technology needs to be adapted to achieve optimal performance on modern hardware. Next to data structures, algorithms, and coding techniques, cost models turned out to be a crucial factor that has not received much attention hitherto in main-memory context. Our analysis has shown, that understanding and modeling database performance in a main-memory dominated scenario is not only possible, but also a fundamental requirement for developing and tuning new techniques to improve performance.

While the main-memory dominated scenario looks similar to the classical I/O dominated scenario, but shifted one level up, traditionally successful solutions concerning cost models and algorithms do not work as efficient as one might expect. Small but significant details, such as the increased and now variable number of memory hierarchies, limited associativity and fixed size of hardware caches, and the fact that the cache replacement strategy is not under the control of the DBMS, require modified or completely new techniques. In this thesis, we explain how to solve these problems. Moreover, this thesis represents a new departure for database research and development into taking account of memory, cache, and processor characteristics, hitherto primarily the concern of optimizing compiler writers and operating system researchers.

6.3 Open Problems and Future Work

Like almost every research also the work presented here leaves some questions unanswered and even discovers new problems. Some of these open issues shall be mentioned here.

Portability. Concerning portability of our cost models, we focused on porting the models to various hardware platforms. Another aspect of portability is whether—and/or to which extend—our models can be successfully applied to other main-memory or even disk-based database systems. While the general framework should be flexible enough to cover the needs of such systems, the main question is whether it will achieve the same accuracy as we experienced in our experiments with Monet. Especially the capabilities to cope with various buffer pool implementations and buffer management strategies in order to accurately predict I/O costs need to be evaluated.

Basic Access Patterns. In the current set of basic access patterns, the interleaved multi-cursor pattern plays a kind of special role, being more complex than the other basic patterns. From an aesthetic point of view, it might be worth to investigate, whether it could be replaced by a compound pattern made up of a proper combination of simple sequential and random patterns that reflects the same knowledge about the pattern performed by algorithms doing partitioning or clustering.

On the other hand, new algorithms or other systems, might need more or different basic access patterns. Similar to the interleaved multi-cursor access pattern, there might be algorithms that allow to deduce more detailed properties of their access pattern(s) than simply being sequential or random. Within the framework we presented, it is simple to add new basic access patterns together with their respective basic cost functions.

In the current setup, basic access patterns are limited to the knowledge and properties that we can derive from the algorithms without knowing anything about the actual data that eventually is to be processed. A possible way to enrich this knowledge, and hence to improve the accuracy of the accompanying basic cost functions would be to combine or parameterize the basic access patterns with some information about the data distributions. The problem is that the cost functions need to be provided statically while the actual data distributions are only known at runtime. A solution might be to use a coarse classification of data distributions and provide cost functions for each class. At runtime, the actual data distribution is matched with one of the classes and the respective cost function can be used.

Logical Costs / Volume Estimation. Focusing on physical cost models, we assumed a "perfect oracle" for estimating intermediate result sizes in this thesis. In practice, any of the numerous techniques for volume, respectively selectivity, estimation proposed in literature (cf., Section 2.2) could pair with our physical cost models to make up a complete database cost model. Recall, logical costs, and hence their models, are independent of the physical design of a DBMS. However, especially in case of Monet, main-memory processing and the decomposed storage model might offer new opportunities to efficiently build and maintain synopses such as histograms or samples. These opportunities need to be explored.

An other idea derives from the bulk-processing approach of Monet. As all intermediate results are materialized, it is simple to create synopses directly from the exact intermediate results with little extra effort. These synopses can then be used for volume estimation of future queries that contain the same subexpressions. By using the exact intermediate results to build the synopses, this approach eliminates the problem that the estimation error grows exponentially throughout the query plan. Two obvious questions remain open to be answered by future research: (1) How can the synopses in this scenario efficiently be updated in case the underlying base tables get updated? (2) How can these synopses be used for volume estimation of queries that do not contain exactly the same subexpressions, but "similar" ones, e.g., a selection predicate on the same attribute(s) but with (slightly) different parameters?

Bibliography

[AC99] A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 181–192, Philadephia, PA, USA, June 1999.

[ACM$^+$98] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 227–237, Barcelona, Spain, June 1998.

[ACPS96] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 137–148, Montreal, QC, Canada, June 1996.

[ADHS01] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 169–180, Rome, Italy, September 2001.

[ADHW99] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where does time go? In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 266–277, Edinburgh, Scotland, UK, September 1999.

[AGPR99] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 275–286, Philadephia, PA, USA, June 1999.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, USA, 1995.

[AKK95] F. Andres, F. Kwakkel, and M. L. Kersten. Calibration of a DBMS Cost Model With the Software Testpilot. In *Conference on Information*

Systems and Management of Data, number 1006 in Lecture Notes in Computer Science, pages 58–74, Bombay, India, November 1995.

[Ant92] G. Antoshenkov. Random Sampling from pseudo-ranked B+ Trees. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 375–382, Vancouver, BC, Canada, August 1992.

[AP92] A. Analyti and S. Pramanik. Fast Search in Main Memory Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 215–224, San Diego, CA, USA, June 1992.

[ASW87] M. M. Astrahan, M. Schkolnick, and K.-Y. Whang. Approximating the Number of Unique Values of an Attribute Without Sorting. *Information Systems*, 12(1):11–15, 1987.

[AvdBF$^+$92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut. PRISMA/DB: A Parallel Main Memory Relational DBMS. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 4(6):541–554, December 1992.

[Bat79] D. S. Batory. On Searching Transposed Files. *ACM Transactions on Database Systems (TODS)*, 4(4):531–544, December 1979.

[BBC$^+$98] P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. The Asilomar Report on Database Research. *ACM SIGMOD Record*, 27(4):74–80, December 1998.

[BBG$^+$98] J. Baulier, P. Bohannon, S. Gogate, S. Joshi, C. Gupta, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, A. Silberschatz, and S. Sudarshan. DataBlitz: A High Performance Main-Memory Storage Manager. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, page 701, New York, NY, USA, August 1998.

[BBG$^+$99] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, S. Seshadri, A. Silberschatz, S. Sudarshan, M. Wilder, and C. Wei. DataBlitz Storage Manager: Main Memory Database Performance for Critical Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 519–520, Philadephia, PA, USA, June 1999.

[BF89] H. Borall and P. Faudemay, editors. *Database Machines, Sixth International Workshop, IWDM ’89*, number 368 in Lecture Notes in Computer Science, Deauville, France, June 1989. Springer.

[BGB98] L. A. Barroso, K. Gharachorloo, and E. D. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the International Symposium on Computer Architecture*, pages 3–14, Barcelona, Spain, June 1998.

[BHK+86] A. Brown, T. Hirata, A. Koehler, K. Vishwanath, J. Ng, M. Pechulis, M. Sikes, D. Singleton, and J. Veazey. Data Base Management for HP Precision Architecture Computers. *Hewlett-Packard Journal*, 37(12):33–48, December 1986.

[BK95] P. A. Boncz and M. L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proceedings Basque International Workshop on Information Technology*, San Sebastian, Spain, July 1995.

[BK99] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, October 1999.

[BKS99] B. Blohsfeld, D. Korus, and B. Seeger. A Comparison of Selectivity Estimators for Range Queries on Metric Attributes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 239–250, Philadephia, PA, USA, June 1999.

[BMK99] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, Edinburgh, Scotland, UK, September 1999.

[Bon02] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[BRK98] P. A. Boncz, T. Rühl, and F. Kwakkel. The Drill Down Benchmark. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 628–632, New York, NY, USA, August 1998.

[BZ98] R. Berrendorf and H. Ziegler. PCL - The Performance Counter Library. Technical Report FZJ-ZAM-IB-9816, ZAM, Forschungzentrum Jülich, Germany, October 1998.

[CD85] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 127–141, Stockholm, Sweden, August 1985.

[CDH+99] J. Clear, D. Dunn, B. Harvey, M. Heytens, P. Lohman, A. Mehta, M. Melton, H. Richardson, L. Rohrberg, A. Savasere, R. Wehrmeister, and M. Xu. NonStopSQL/MX primitives for knowledge discovery. In *Proceedings of the International Conference on Knowledge Discovery* *and Data Mining (KDD)*, pages 425–429, San Diego, CA, USA, August 1999.

[CGRS00] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing Using Wavelets. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 111–122, Cairo, Egypt, September 2000.

[Cha98] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 34–43, Seattle, WA, USA, June 1998.

[Chr83] S. Christodoulakis. Estimating Block Transfers and Join Sizes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 40–50, San Jose, CA, USA, May 1983.

[Chr84] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. *ACM Transactions on Database Systems (TODS)*, 9(2):163–186, June 1984.

[CK85] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 268–279, Austin, TX, USA, May 1985.

[CM95] S. Cluet and G. Moerkotte. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 54–67, Prague, Czechia, January 1995.

[CMN98] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Sampling for Histogram Construction: How much is enough? In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 436–447, Seattle, WA, USA, June 1998.

[CMN99] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On Random Sampling over Joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 263–274, Philadephia, PA, USA, June 1999.

[Com98] Compaq Corp. Whitepaper. *Infocharger*, January 1998.

[CR94] C. M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 161–172, Minneapolis, MN, USA, May 1994.

[CS89] I. R. Casas and K. C. Sevcik. A Buffer Management Model For Use In Predicting Overall Database System Performance. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 463–469, Los Angeles, CA, USA, February 1989.

[DKO$^+$84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–8, Boston, MA, USA, June 1984.

[DKS92] W. Du, R. Krishnamurthy, and M. C. Shan. Query Optimization in Heterogeneous DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 277–291, Vancouver, BC, Canada, August 1992.

[DYC95] A. Dan, P. S. Yu, and J.-Y. Chung. Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability. *The VLDB Journal*, 4(1):127–154, January 1995.

[Eic87] M. H. Eich. A Classification and Comparison of Main Memory Database Recovery Techniques. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 332–339, Los Angeles, CA, USA, February 1987.

[Eic89] M. H. Eich. Main Memory Database Research Directions. In *Database Machines. 6th International Workshop*, pages 251–268, Deauville, France, June 1989.

[EN94] E. Elmasri and S. B. Navathe. *Fundamentals of Database Sytems*. Benjamin/Cummings, Redwood City, CA, USA, 2nd edition, 1994.

[FM85] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182–209, October 1985.

[FR97] G. Fahl and T. Risch. Query Processing Over Object Views of Relational Data. *The VLDB Journal*, 6(4):261–281, November 1997.

[GFF97] G. Gardarin, B. Finance, and P. Fankhauser. Federating Object-Oriented and Relational Databases: The IRO-DB Experience. In *Proceedings of the IFCIS International Conference on Cooperative Information Systems (CoopIS)*, pages 2–13, Kiawah Island, SC, USA, June 1997.

[GG82] E. Gelenbe and D. Gardy. The Size of Projections of Relations Satisfying a Functional Dependency. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 325–333, Mexico City, Mexico, September 1982.

[GGMS96] S. Ganguly, P. B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal Sampling for Skew-Resistant Join Size Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 271–281, Montreal, QC, Canada, June 1996.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 9–18, San Diego, CA, USA, June 1992.

[GLSW93] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. Query Optimization in the IBM DB2 Family. *IEEE Data Engineering Bulletin*, 16(4):4–18, December 1993.

[GM98] P. B. Gibbons and Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 331–342, Seattle, WA, USA, June 1998.

[GM99] P. B. Gibbons and Y. Matias. Synopsis Data Structures for Massive Data Sets. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 909–910, Baltimore, MD, USA, January 1999.

[GMP97] P. B. Gibbons, Y. Matias, and V. Poosala. Fast Incremental Maintenance of Approximate Histograms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 466–475, Athens, Greece, September 1997.

[GMS92] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 4(6):509–516, December 1992.

[GMUW02] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, Englewood Cliffs, NJ, USA, 2002.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GST96] G. Gardarin, F. Sha, and Z. H. Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, and Object-Oriented Federated Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 378–389, Bombay, India, September 1996.

[HNSS96] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and Cost Estimation for Joins Based on Random Sampling. *Journal of Computer and System Sciences*, 52(3):550–569, June 1996.

[HS89] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers (TOC)*, 38(12):1612–1630, December 1989.

[HS92] P. J. Haas and A. N. Swami. Sequential Sampling Procedures for Query Size Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 341–350, San Diego, CA, USA, June 1992.

[HS95] P. J. Haas and A. N. Swami. Sampling-Based Selectivity Estimation for Joins using Augmented Frequent Value Statistics. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 522–531, Taipei, Taiwan, March 1995.

[IC91] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 268–277, Denver, CO, USA, May 1991.

[IC93] Y. E. Ioannidis and S. Christodoulakis. Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results. *ACM Transactions on Database Systems (TODS)*, 18(4):709–748, December 1993.

[IK84] T. Ibaraki and T. Kameda. On the Optimal Nesting for Computation N-Relational Joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, September 1984.

[Ioa93] Y. E. Ioannidis. Universality of serial histograms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 256–267, Dublin, Ireland, August 1993.

[IP95] Y. E. Ioannidis and V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 233–244, San Jose, CA, USA, May 1995.

[IP99] Y. E. Ioannidis and V. Poosala. Histogram-Based Approximation of Set-Valued Query-Answers. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 174–185, Edinburgh, Scotland, UK, September 1999.

[JKM+98] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal Histograms with Quality Guarantees. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 275–286, New York, NY, USA, August 1998.

[JLR+94] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A High Performance Main Memory Storage Manager. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 48–59, Santiago, Chile, September 1994.

[JSS93] H. V. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from Main-Memory Lapses. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 391–404, Dublin, Ireland, August 1993.

[Ker89] M. L. Kersten. Using Logarithmic Code-Expansion to Speedup Index Access and Maintenance. In *Proceedings of the International Conference on Foundation on Data Organization and Algorithms (FODO)*, pages 228–232, Paris, France, October 1989.

[KK85] N. Kamel and R. King. A Model of Data Distribution Based on Texture Analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 319–325, Austin, TX, USA, May 1985.

[KK93] M. L. Kersten and F. Kwakkel. Design and implementation of a DBMS Performance Assessment Tool. In *Proceedings of the International Workshop on Database and Expert Systems Application (DEXA)*, pages 265–276, Prague, Czechia, September 1993.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, MA, USA, 1968.

[Koo80] R. P. Kooi. *The Optimization of Queries in Relational Databases*. PhD thesis, Case Western Reserver University, Cleveland, OH, USA, September 1980.

[KPH$^+$98] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proceedings of the International Symposium on Computer Architecture*, pages 15–26, Barcelona, Spain, June 1998.

[KS91] H. Korth and A. Silberschatz. *Database Systems Concepts*. McGraw-Hill, Inc., New York, San Francisco, Washington, DC, USA, 1991.

[KSHK97] M. L. Kersten, A. P. J. M. Siebes, M. Holsheimer, and F. Kwakkel. Research and Business Challenges in Data Mining Technology. In *Proceedings Datenbanken in Büro, Technik und Wissenschaft*, pages 1–16, Ulm, Germany, March 1997.

[KW99] A. C. König and Gerhard Weikum. Combining Histograms and Parametric Curve Fitting for Feedback-Driven Query Result-size Estimation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 423–434, Edinburgh, Scotland, UK, September 1999.

[KW00] A. C. König and G. Weikum. Auto-Tuned Spline Synopses for Database Statistics Management. In *Proceedings of the International Conference on Management of Data (COMAD)*, Pune, India, December 2000.

[LC86a] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 294–303, Kyoto, Japan, August 1986.

[LC86b] T. J. Lehman and M. J. Carey. Query Processing in Main Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 239–250, Washington, DC, USA, May 1986.

[LKC99] J.-H. Lee, D.-H. Kim, and C.-W. Chung. Multi-dimensional Selectivity Estimation Using Compressed Histogram Information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 205–214, Philadephia, PA, USA, June 1999.

[LN96] S. Listgarten and M.-A. Neimat. Modelling Costs for a MM-DBMS. In *Proceedings of the International Workshop on Real-Time Databases, Issues and Applications (RTDB)*, pages 72–78, Newport Beach, CA, USA, March 1996.

[LN97] S. Listgarten and M.-A. Neimat. Cost Model Development for a Main Memory Database System. In A. Bastavros, K.-J. Lin, and S. H. Son, editors, *Real-Time Database Systems: Issues and Applications*, chapter 10, pages 139–162. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[LNS90] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical Selectivity Estimation through Adaptive Sampling. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–11, Atlantic City, NJ, USA, May 1990.

[LS95] Y. Ling and W. Sun. An Evaluation of Sampling-Based Size Estimation Techniques for Selections in Database Systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 532–539, Taipei, Taiwan, March 1995.

[LST91] H. Lu, M.-C. Shan, and K.-L. Tan. Optimization of Multi-Way Join Queries for Parallel Execution. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 549–560, Barcelona, Spain, September 1991.

[LTS90] H. Lu, K.-L. Tan, and M.-C. Shan. Hash-Based Join Algorithms for Multiprocessor Computers. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 198–209, Brisbane, Australia, August 1990.

[LVZ93] R. S. G. Lanzelotte, P. Valduriez, and M. Zaït. On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces. In

Proceedings of the International Conference on Very Large Data Bases (VLDB), pages 493–504, Dublin, Ireland, August 1993.

[MBK00a] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000.

[MBK00b] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? — Dissecting CPU and Memory Optimization Effects. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 339–350, Cairo, Egypt, September 2000.

[MBK02] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, July 2002.

[MCS88] M. V. Mannino, P. Chu, and T. Sager. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 20(3):192–221, September 1988.

[MD88] M. Muralikrishna and D. J. DeWitt. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 28–36, Chicago, IL, USA, June 1988.

[MK88] B. Muthuswamy and L. Kerschberg. A Detailed Database Statistics Model for Realtional Query Optimization. In *ACM Annual Conference*, pages 439–448, Denver, CO, USA, October 1988.

[MKW$^+$98] S. McKee, R. Klenke, K. Wright, W. Wulf, M. Salinas, J. Aylor, and A. Batson. Smarter Memory: Improving Bandwidth for Streamed References. *IEEE Computer*, 31(7):54–63, July 1998.

[ML86] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Loc Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 84–95, Washington, DC, USA, May 1986.

[ML89] L. F. Mackert and G. M. Lohman. Index Scans Using a Finite LRU Buffer: A Validated I/O Model. *ACM Trans. on Database Sys.*, 14(3):401–424, March 1989.

[MO79] A. W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and Its Applications*. Academic Press, New York, 1979.

[Moo65] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[Mow94] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Computer Science Department, Stanford, CA, USA, March 1994.

[MPK00] S. Manegold, A. Pellenkoft, and M. L. Kersten. A Multi-Query Optimizer for Monet. In *Proceedings of the British National Conference on Databases (BNCOD)*, number 1832 in Lecture Notes in Computer Science, pages 36–51, Exeter, United Kingdom, July 2000.

[MVW98] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-Based Histograms for Selectivity Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 448–459, Seattle, WA, USA, June 1998.

[NBC+94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 233–242, Minneapolis, MN, USA, May 1994.

[OR86] F. Olken and B. Rotem. Simple Random Sampling from Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 160–169, Kyoto, Japan, August 1986.

[Ozk86] E. Ozkarahan. *Database machines and database management*. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[PAC+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, March 1997.

[Pel97] A. Pellenkoft. *Probabilistic and Transformation based Query Optimization*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, November 1997.

[PGI99] V. Poosala, V. Ganti, and Y. E. Ioannidis. Approximate Query Answering using Histograms. *IEEE Data Engineering Bulletin*, 22(4):5–14, December 1999.

[PIHS96] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 294–305, Montreal, QC, Canada, June 1996.

[PKK+98] J. H. Park, Y. S. Kwon, K. H. Kim, S. Lee, B. D. Park, and S. K. Cha. Xmas: An Extensible Main-Memory Storage System for High-Performance Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 578–580, Seattle, WA, USA, June 1998.

[Poo97] V. Poosala. *Histogram-based Estimation Techniques in Database Systems*. PhD thesis, University of Wisconsin Madison, Madison, WI, USA, February 1997.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 256–276, Boston, MA, USA, June 1984.

[Ram96] Rambus Technologies, Inc. *Direct Rambus Technology Disclosure*, 1996. http://www.rambus.com/docs/drtechov.pdf.

[RBP+98] R. Rastogi, P. Bohannon, J. Parker, A. Silberschatz, S. Seshadri, and S. Sudarshan. Distributed Multi-Level Recovery in Main-Memory Databases. *Distributed and Parallel Databases*, 6(1):41–71, January 1998.

[Ron98] M. Ronström. *Design and Modeling of a Parallel Data Server for Telecom Applications*. PhD thesis, Linköping University, Linköping, Sweden, 1998.

[RR99] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 78–89, Edinburgh, Scotland, UK, September 1999.

[RR00] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 475–486, Dallas, TX, USA, May 2000.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 23–34, Boston, MA, USA, May 1979.

[Sac87] G. M. Sacco. Index Access with a Finite Buffer. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 301–309, Brighton, England, UK, September 1987.

[SDNR96] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramaswamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 522–531, Bombay, India, September 1996.

[SE93] J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *Proceedings of the International Conference* *on Parallel and Distributed Information Systems (PDIS)*, pages 84–92, San Diego, CA, USA, January 1993.

[Sem97] Sematech. *National Roadmap For Semiconductor Technology: Technology Needs*, 1997. http://www.itrs.net/ntrs/publntrs.nsf.

[SF96] M. Spiliopoulou and J.-C. Freytag. Modelling Resource Utilization in Pipelined Query Execution. In *Proceedings of the European Conference on Parallel Processing (EuroPar)*, pages 872–880, Lyon, France, August 1996.

[SHV96] M. Spiliopoulou, M. Hatzopoulos, and C. Vassilakis. A Cost Model for the Estimation of Query Execution Time in a Parallel Environment Supporting Pipeline. *Computers and Artificial Intelligence*, 14(1):341–368, 1996.

[Sil97] Silicon Graphics, Inc., Mountain View, CA. *Performance Tuning and Optimization for Origin2000 and Onyx2*, January 1997.

[SKN94] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 510–512, Santiago, Chile, September 1994.

[SLD97] SLDRAM Inc. *SyncLink DRAM Whitepaper*, 1997. http://www.sldram.com/Documents/SLDRAMwhite970910.pdf.

[SLRD93] W. Sun, Y. Ling, N. Rishe, and Y. Deng. An Instant and Accurate Size Estimation Method for Joins and Selection in a Retrieval-Intensive Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 79–88, Washington, DC, USA, May 1993.

[SM97] W. Scheufele and G. Moerkotte. On the Complexity of Generating Optimal Plans with Cross Products. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 238–248, Tucson, AZ, USA, May 1997.

[SN92] S. Seshadri and J. F. Naughton. Sampling Issues in Parallel Database Systems. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, number 580 in Lecture Notes in Computer Science, pages 328–343, Vienna, Austria, March 1992.

[SS86] G. M. Sacco and M. Schkolnick. Buffer Management in Relational Database Systems. *ACM Transactions on Database Systems (TODS)*, 11(4):473–498, December 1986.

[Sti30] J. Stirling. *Methodus differentialis, sive tractatus de summation et interpolation serierum infinitarium*. London, 1730. English translation by J. Holliday, The Differential Method: A Treatise of the Summation and Interpolation of Infinite Series. 1749.

[Su88] S. Su. *Database Computers, principles, architectures & techniques*. McGraw-Hill, Inc., New York, San Francisco, Washington, DC, USA, 1988.

[Syb96] Sybase Corp. Whitepaper. *Adaptive Server IQ*, July 1996.

[SYT93] E. J. Shekita, H. C. Young, and K.-L. Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 479–492, Dublin, Ireland, August 1993.

[Tea99] Times-Ten Team. In-Memory Data Management for Consumer Transactions The Times-Ten Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 528–529, Philadephia, PA, USA, June 1999.

[TLPZT97] P. Trancoso, J. L. Larriba-Pey, Z. Zhang, and J. Torellas. The Memory Performance of DSS Commericial Workloads in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 250–260, San Antonio, TX, USA, January 1997.

[Val87] P. Valduriez. Join Indices. *ACM Transactions on Database Systems (TODS)*, 12(2):218–246, June 1987.

[VW99] J. S. Vitter and M. Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 193–204, Philadephia, PA, USA, June 1999.

[Waa00] F. Waas. *Principles of Probabilistic Query Optimization*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, November 2000.

[Wil91] A. Wilschut. *Parallel Query Execution in a Main-Memory Database System*. PhD thesis, Universiteit Twente, Enschede, The Netherlands, April 1991.

[WK90] K.-Y. Whang and R. Krishnamurthy. Query Optimization in a Memory-Resident Domain Relational Calculus Database System. *ACM Transactions on Database Systems (TODS)*, 15(1):67–95, March 1990.

[Yea96] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[ZL94] Q. Zhu and P.-Å. Larson. A Query Sampling Method of Estimating Local Cost Parameters in a Multidatabase System. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 144–153, Houston, TX, USA, February 1994.

[ZL96] Q. Zhu and P.-Å. Larson. Developing Regression Cost Models for Multidatabase Systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 220–231, Miami Beach, FL, USA, December 1996.

[ZL98] Q. Zhu and P.-Å. Larson. Solving Local Cost Estimation Problem for Global Query Optimization in Multidatabase Systems. *Distributed and Parallel Databases*, 6(4):373–421, October 1998.

[ZLTI96] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proc. of the Supercomputing '96 Conf.*, Pittsburgh, PA, USA, November 1996.

[ZZBS93] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel Query Processing in DBS3. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 103–109, San Diego, CA, USA, January 1993.

Curriculum Vitae

Education

PhD in Computer Science (expected): 1997–2002
University of Amsterdam (The Netherlands)

MSc in Computer Science: 1988–1994
Technical University Clausthal (Germany)

High School: 1979–1988
Friedrich-Wilhelm-Schule & Oberstufengymnasium Eschwege (Germany)

Professional Experience

Researcher with CWI Amsterdam (The Netherlands): 1997 – present
I hold a permanent research position at the Centre for Mathematics and Computer Science (CWI) in the database group of Prof. Martin L. Kersten. My research includes database cost models, query optimization, and database algorithms, mainly focusing on main-memory database systems.

Researcher with Humboldt-University Berlin (Germany): 1995 – 1997
I worked in the database group of Prof. Johann-Christoph Freytag on query evaluation and optimization in parallel database system.

Compulsory Military Service with German Air Force: 1994 – 1995
I spent my compulsory military service with a computing center of the German Air Force. My main task was system administration of two Siemens mainframes (H90 and 7.580i) running BS 2000.

Publications

Journals

[1] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, July 2002.

[2] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000.

Conferences and Workshops

[1] A. R. Schmidt, S. Manegold, and M. L. Kersten. Integrated Querying of XML Data in RDBMSs. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Melbourne, FL, USA, March 2003. Accepted for publication.

[2] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 191–202, Hong Kong, China, August 2002.

[3] M. L. Kersten, S. Manegold, P. A. Boncz, and N. J. Nes. Macro- and Micro-Parallelism in a DBMS. In *Proceedings of the European Conference on Parallel Processing (EuroPar)*, number 2150 in Lecture Notes in Computer Science, pages 6–15, Manchester, UK, August 2001.

[4] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? — Dissecting CPU and Memory Optimization Effects. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 339–350, Cairo, Egypt, September 2000.

[5] S. Manegold, A. Pellenkoft, and M. L. Kersten. A Multi-Query Optimizer for Monet. In *Proceedings of the British National Conference on Databases (BNCOD)*, number 1832 in Lecture Notes in Computer Science, pages 36–51, Exeter, United Kingdom, July 2000.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, Edinburgh, Scotland, UK, September 1999.

[7] S. Manegold and F. Waas. Integrating I/O processing and Transparent Parallelism — Toward Comprehensive Query Execution in Parallel Database Systems. In A. Dogaç, M. T. Özsu, and Ö. Ulusoy, editors, *Current Trends in Data Management Technology*, chapter 8, pages 130–152. Idea Group Publishing, Hershey, PA, USA, January 1999.

[8] S. Manegold, F. Waas, and M. L. Kersten. Transparent Parallelism in Query Execution. In *Proceedings of the International Conference on Management of Data (COMAD)*, pages 217–234, Hyderabad, India, December 1998.

[9] S. Manegold and J. K. Obermaier. Efficient Resource Utilization in Shared-Everything Environments. In *Proceedings of the International Workshop on Issues and Applications of Database Technology (IADT)*, pages 209–216, Berlin, Germany, July 1998.

[10] S. Manegold and F. Waas. Thinking Big in a Small World — Efficient Query Execution on Small-scale SMPs. In Jonathan Schaeffer, editor, *Proceedings of the International Symposium on High Performance Computing Systems and Applications (HPCS)*, chapter 14, pages 133–146. Kluwer Academic Publishers, Edmonton, AL, Canada, May 1998.

[11] S. Manegold, F. Waas, and D. Gudlat. In Quest of the Bottleneck - Monitoring Parallel Database Systems. In *Proceedings of the European PVM-MPI Users' Group Meeting*, number 1332 in Lecture Notes in Computer Science, pages 277–284, Cracow, Poland, November 1997.

[12] S. Manegold, J. K. Obermaier, and F. Waas. Load Balanced Query Evaluation in Shared-Everything Environments. In *Proceedings of the European Conference on Parallel Processing (EuroPar)*, number 1300 in Lecture Notes in Computer Science, pages 1117–1124, Passau, Germany, August 1997.

[13] S. Manegold. Efficient Data-Parallel Query Processing in Shared-Everything Environments. In *Technologie für Parallele Datenbanksysteme — Bericht zum Workshop*, München, Germany, February 1997. Technische Universität München, Institut für Informatik. In German.

Technical Reports

[1] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. Technical Report INS-R0203, CWI, Amsterdam, The Netherlands, March 2002.

[2] S. Manegold, A. Pellenkoft, and M. L. Kersten. A Multi-Query Optimizer for Monet. Technical Report INS-R0002, CWI, Amsterdam, The Netherlands, January 2000.

[3] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join On Modern Hardware. Technical Report INS-R9912, CWI, Amsterdam, The Netherlands, October 1999.

[4] S. Manegold, F. Waas, and M. L. Kersten. On Optimal Pipeline Processing in Parallel Query Optimization. Technical Report INS-R9805, CWI, Amsterdam, The Netherlands, February 1998.

[5] S. Manegold and J. K. Obermaier. Efficient Resource Utilization in Shared-Everything Environments. Technical Report INS-R9711, CWI, Amsterdam, The Netherlands, December 1997.

[6] S. Manegold, J. K. Obermaier, and F. Waas. Flexible Query Optimization in Parallel Database Systems (Working Paper). Technical Report HUB-IB-80, Humboldt-Universität zu Berlin, Institut für Informatik, Berlin, Germany, February 1997. In German.

[7] S. Manegold, J. K. Obermaier, and F. Waas. Load Balanced Query Evaluation in Shared-Everything Environments (Extended Version). Technical Report HUB-IB-70, Humboldt-Universität zu Berlin, Institut für Informatik, Berlin, Germany, September 1996.

[8] S. Manegold, J. K. Obermaier, F. Waas, and J.-C. Freytag. Load Balanced Query Evaluation in Shared-Everything Environments. Technical Report HUB-IB-61, Humboldt-Universität zu Berlin, Institut für Informatik, Berlin, Germany, June 1996.

[9] S. Manegold, J. K. Obermaier, F. Waas, and J.-C. Freytag. Data Threaded Query Evaluation in Shared-Everything Environments. Technical Report HUB-IB-58, Humboldt-Universität zu Berlin, Institut für Informatik, Berlin, Germany, April 1996.

List of Tables and Figures

List of Symbols

#	number of cache lines	49
$\odot$	concurrent execution	78
$\oplus$	sequential execution	78
$\binom{x}{y}$	binomial coefficient	88
$\left\{ {x \atop y} \right\}$	Stirling number of second kind	89
$\lvert R \rvert$	length of data region R [number of data items]	74
$\underline{\overline{R}}$	width of data region R (size data items) [in bytes]	74
$\lvert\lvert R \rvert\rvert$	size of data region R [in bytes]	74
$\lvert x \rvert_y$	$= \lceil x/y \rceil$	74
A	cache associativity	49
β	cache access bandwidth [in bytes/ns]	49
b	cache miss bandwidth [in bytes/ns]	49
B	total number of radix-bits	110
B_p	number of radix-bits per pass	110
$\mathbf{C}$	number of distinct cache lines touched by $\mathsf{r\_acc}(r, R, u)$	89
C	cache capacity [in bytes]	49
$\mathbb{D}$	set of data regions	74
E	number of all different outcomes of picking r times one of the $\lvert R \rvert$ data items allowing multiple accesses to each data item	88
E_j	number of outcomes containing exactly $1 \leq j \leq \min\{r, \lvert R \rvert\}$ distinct data items	88

$\mathbf{F}$	footprint of access pattern	94
H	total number of clusters or partitions	109
H_p	number of clusters per pass	110
$\mathbf{I}$	number of distinct data items touched by $\mathsf{r\_acc}(r, R, u)$	88
$_i$	indicate cache level	49
Li	cache name (level)	49
λ	cache access latency [in ns]	49
l	cache miss latency [in ns]	49
$\mathbf{M}$	number of cache misses	51
$\vec{\mathbf{M}}$	$= \langle \mathbf{M}^{\mathrm{s}}, \mathbf{M}^{\mathrm{r}} \rangle$, number of cache misses caused by both sequential and random access	79
nest	interleaved multi-cursor access	77
N	number of cache levels	49
$\mathbf{O}$	overlap function	52
o	degree of overlap	52
$\mathcal{P}$	an access pattern	78
$\mathbb{P}_{\mathrm{b}}$	set of basic access patterns	78
$\mathbb{P}_{\mathrm{c}}$	set of compound access patterns	78
$\mathbb{P}$	$= \mathbb{P}_{\mathrm{b}} \cup \mathbb{P}_{\mathrm{c}}$, set of all access patterns	78
P	number of radix-cluster passes	110
r_acc	random access	77
rr_trav	repetitive random traversal	77
rs_trav	repetitive sequential traversal	76
r_trav	single random traversal	77
$^{\mathtt{r}}$	indicate random access	49
R	a data region	74
$\mathbf{S}$	cache state	93
s_trav	single sequential traversal	76

^{s}	indicate sequential access	49
s	access stride	51
T	execution time [in ns]	51
U, V, W	data regions	79
$\overset{\rightrightarrows}{\mathbf{X}}$	additional cache misses with interleaved multi-cursor access	92
Z	cache block size [in bytes]	49

Samenvatting

Databank management systemen (DBMS) zijn bekend als belangrijke software producten voor het opslaan en bewerken van gegevens in de zakelijke en wetenschappelijke wereld. Databanken worden bijvoorbeeld ingezet om de gegevens van medewerkers en klanten te beheren, om boekhouding en opslag te ondersteunen, of om productieprocessen te volgen. Wetenschappers gebruiken databanken om data van hun experimenten op te slaan en te analyseren. Ook in het dagelijks leven worden databanken steeds vaker aangetroffen. Privé computers (PC's) worden steeds krachtiger — 1 gigabyte (GB) werkgeheugen en 160 GB opslagruimte op een enkele standaard harde schijf zijn niet alleen beschikbaar, maar ook betaalbaar — en dus worden databank systemen steeds vaker ingezet als standaard onderdeel van verschillende computer applicaties, net zoals tabel- en tekstverwerkingsprogramma's reeds eerder. Zelfs in draagbare apparatuur zoals PDA's en mobiele telefoons worden (kleine) databanken gebruikt om bijvoorbeeld adressen, telefoonnummers of een digitaal foto album op te slaan.

Het grote succes van databank management systemen is met name gebaseerd op twee punten. (1) Gebruikers kunnen hun informatieaanvragen op een intuïtieve manier formuleren in zogenoemde beschrijvende (declaratieve) talen zoals SQL. Dit betekent, dat men alleen moet vertellen *wat* men wil weten maar niet *hoe* men de informatie uit de opgeslagen data moet halen. Noch programmeerkunsten noch inzicht in de manier hoe het systeem de data fysiek opgeslagen heeft zijn dus vereist om databanken te kunnen gebruiken. Helemaal onzichtbaar voor de gebruiker gaat het DBMS de beste manier uitzoeken, om de aangevraagde informatie te leveren. Wij noemen dit proces vraagverwerking en -optimalisatie. (2) Na decennia van onderzoek en ontwikkeling heeft de DBMS technologie een heel stabiele en efficiënte status bereikt. In de meeste commerciële DBMS producten is deze technologie evenwel gebaseerd op de eigenschappen van de hardware van ruim twee decennia geleden. Implementatie en optimalisatie van DBMS modulen is voornamelijk gericht op het minimaliseren van het aantal van operaties die gegevens van de harde schijf aflezen of daarheen wegschrijven. Pas daarna worden ook de processorkosten (d.w.z. de rekentijd) geoptimaliseerd. De kosten voor het schrijven en lezen van data, die al in het werkgeheugen beschikbaar is, werden geheel genegeerd, omdat het werkgeheugen net zo snel als de processoren was en de kosten voor het schrijven en lezen van data in het werkgeheugen onafhankelijk waren van de locatie waar de data in het werkgeheugen staat.

De ontwikkeling van computer hardware tijdens de afgelopen twee decennia vertoont een beeld van opmerkelijke verandering. Met name de snelheid van processoren volgt de wet van Moore, d.w.z. door het verhogen van én de processorkloksnelheid én de parallelliteit binnen de processor verdubbelt de snelheid iedere 18 maanden. Voor het werkgeheugen geldt de wet van Moore slechts gedeeltelijk. De *bandbreedte* — het vermogen hoeveel data maximaal per tijdeenheid uit het werkgeheugen gelezen of naar het werkgeheugen geschreven kan worden — groeit wel bijna net zo snel als de snelheid van de processoren maar de tijd die nodig is om een enkele byte uit het geheugen te halen of daarheen te schrijven — wij noemen dit *toegangstijd* — is bijna niet veranderd. Om het uit-elkaar-lopen van processorsnelheid en geheugentoegangstijd te beperken, hebben hardware producenten zo genoemde cache geheugens geïntroduceerd. Deze snelle maar dure (en dus kleine) geheugenmodules worden, inmiddels in meerdere "verdiepingen", (logisch[1]) tussen de processor en het werkgeheugen geplaatst en kunnen zo de toegangstijd verlagen, als de aangevraagde data al in één van de cache geheugens zit. Wij noemen dit een hiërarchisch geheugensysteem.

In dit proefschrift wordt bestudeerd welke invloed deze veranderingen in de hardware situatie op de prestatie van databank systemen heeft en hoe databank technologieën moeten worden aangepast om rekening te houden met de gewijzigde hardware situatie.

Het onderzoek is gesplitst in vier gedeeltes. De eerste twee hoofdstukken geven een introductie tot databank systemen, met name vraagverwerking en de rol van kostenmodellen in vraagoptimalisatie.

Hoofdstuk 3 bestudeert hoe de prestatie van met name werkgeheugen-gebaseerde databank systemen bepaald is door de eigenschappen van de hardware, zoals processorsnelheid, aantal, grootte en snelheid van de cache geheugens, toegangstijd en bandbreedte van het werkgeheugen. In ons onderzoek gebruiken wij zogenoemde *hardware event counters* die in alle huidige processoren beschikbaar zijn en waarmee wij de frequentie van gebeurtenissen zoals *cache misses*, *TLB misses*, *resource stalls* en *branch mispredictions* kunnen meten. Met behulp van eenvoudige maar representatieve experimenten tonen wij aan, dat de toegang tot data in het werkgeheugen inderdaad inmiddels een grote beperking voor de prestatie van databank systemen oplevert. Standaard databank technologie is helaas niet geschikt om de beschikbare cache geheugens optimaal te kunnen benutten. Onze resultaten tonen ook aan, dat kostenmodellen voor vraagoptimalisatie de kosten voor de toegang tot data in het werkgeheugen niet langer mogen negeren, omdat de toegang tot data in het geheugen nu niet meer "gratis" is en varieert, afhankelijk van in welk cache niveau de data misschien al beschikbaar is. Verder ontwikkelen wij in hoofdstuk 3 een programma, de "*Calibrator*", waarmee wij de (voor ons) belangrijke hardware eigenschappen, zoals aantal, grootte en snelheid van de cache geheugens, grootte van cache regels, toegangstijd en bandbreedte van het werkgeheugen, op verschillende computersystemen automatisch kunnen meten.

Het doel van hoofdstuk 4 is om modellen te ontwikkelen, die de kosten (d.w.z. de tijd die ervoor nodig is) voor de toegang tot data in het geheugen van databank algoritmen kunnen voorspellen. Het idee is om het aantal kostgerelateerde gebeurtenis-

[1]Fysiek worden cache geheugens tegenwoordig meestal in de processorchips geïntegreerd.

sen, met name cache en TLB misses, van databank algoritmen te voorspellen en deze met de kosten per gebeurtenis te vermenigvuldigen. Verder introduceert onze aanpak twee andere innovaties ten opzichte van traditionele databank kostenmodellen. Ten eerste wordt het proces, om kostenfuncties voor willekeurige databank algoritmen te creëren, gegeneraliseerd en vereenvoudigd. Voor dit doel introduceren wij het concept van datatoegangspatronen. Hierdoor moeten de meestal ingewikkelde kostenfuncties niet meer "handmatig" voor ieder algoritme worden gecreëerd. In plaats daarvan kunnen de (complexe) datatoegangspatronen van databank algoritmen worden beschreven als combinatie van eenvoudige basispatronen zoals "van sequentiële volgorde" of "van toevallige volgorde". De echte kostenfuncties kunnen vanuit deze beschrijvingen automatisch worden gegenereerd. Hiervoor ontwikkelen wij in hoofdstuk 4 de kostenfuncties voor onze basispatronen en geven voorschriften hoe deze volgens de beschrijvingen tot complexe functies moeten worden gecombineerd. De tweede innovatie gaat over de hardware afhankelijkheid van fysieke kostenmodellen. Het idee is om maar één gemeenschappelijk kostenmodel nodig te hebben, in plaats van vele modellen waarvan elk maar één bepaalde hardware architectuur beschrijft. Om dit te bereiken introduceren wij een nieuw uniform hardware model voor hiërarchische geheugensystemen. Dit model bewaart voor iedere cachelaag en het werkgeheugen de eigenschappen, zoals grootte, bandbreedte en toegangstijd. De kostenfuncties zijn ontworpen om deze parameters te gebruiken. Voor een nieuw of veranderd hardware platform moeten dus geen kostenfuncties worden veranderd. Slechts de nieuwe hardware parameters moeten worden gemeten (met behulp van de Calibrator) en worden ingevoegd in de kostenfuncties.

In hoofdstuk 5 onderzoeken wij hoe wij de ervaringen uit hoofdstuk 3 en de modellen uit hoofdstuk 4 kunnen gebruiken om databank algoritmen te creëren, die het vermogen van huidige computers beter benutten. Wij gebruiken de equi-join als voorbeeld en ontwikkelen nieuwe radix-algoritmen voor de gepartitioneerde hash-join. Het idee is om het aantal cache misses te verkleinen door toegangpatronen met een toevallige volgorde te beperken tot de kleinste cache. Wij gebruiken de kostenmodellen om de algoritmen automatisch te kunnen aanpassen voor verschillende hardware systemen. Verder presenteren wij implementatie technieken om de processorkosten te optimaliseren door het aantal functie aanroepen, vertakkingen en data afhankelijkheden te verkleinen.

Het zesde hoofdstuk is een samenvatting van de belangrijkste punten en geeft mogelijke richtingen voor vervolgonderzoek.

SIKS Dissertatiereeks

1998-01	Johan van den Akker (CWI/UvA), *DEGAS - An Active, Temporal Database of Autonomous Objects.*
1998-02	Floris Wiesman (UM), *Information Retrieval by Graphically Browsing Meta-Information.*
1998-03	Ans Steuten (TUD), *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective.*
1998-04	Dennis Breuker (UM), *Memory versus Search in Games.*
1998-05	E.W. Oskamp (RUL), *Computerondersteuning bij Straftoemeting.*
1999-01	Mark Sloof (VU), *Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products.*
1999-02	Rob Potharst (EUR), *Classification using decision trees and neural nets.*
1999-03	Don Beal (UM), *The Nature of Minimax Search.*
1999-04	Jacques Penders (UM), *The practical Art of Moving Physical Objects.*
1999-05	Aldo de Moor (KUB), *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems.*
1999-06	Niek J.E. Wijngaards (VU), *Re-design of compositional systems.*
1999-07	David Spelt (UT), *Verification support for object database design.*
1999-08	Jacques H.J. Lenting (UM), *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.*
2000-01	Frank Niessink (VU), *Perspectives on Improving Software Maintenance.*
2000-02	Koen Holtman (TUE), *Prototyping of CMS Storage Management.*
2000-03	Carolien M.T. Metselaar (UvA), *Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.*
2000-04	Geert de Haan (VU), *ETAG, A Formal Model of Competence Knowledge for User Interface Design.*
2000-05	Ruud van der Pol (UM), *Knowledge-based Query Formulation in Information Retrieval.*
2000-06	Rogier van Eijk (UU), *Programming Languages for Agent Communication.*
2000-07	Niels Peek (UU), *Decision-theoretic Planning of Clinical Patient Management.*
2000-08	Veerle Coup (EUR), *Sensitivity Analysis of Decision-Theoretic Networks.*
2000-09	Florian Waas (CWI/UvA), *Principles of Probabilistic Query Optimization.*
2000-10	Niels Nes (CWI/UvA), *Image Database Management System Design Considerations, Algorithms and Architecture.*
2000-11	Jonas Karlsson (CWI/UvA), *Scalable Distributed Data Structures for Database Management.*
2001-01	Silja Renooij (UU), *Qualitative Approaches to Quantifying Probabilistic Networks.*
2001-02	Koen Hindriks (UU), *Agent Programming Languages: Programming with Mental Models.*
2001-03	Maarten van Someren (UvA), *Learning as problem solving.*
2001-04	Evgueni Smirnov (UM), *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets.*

2001-05	Jacco van Ossenbruggen (VU), *Processing Structured Hypermedia: A Matter of Style.*
2001-06	Martijn van Welie (VU), *Task-based User Interface Design.*
2001-07	Bastiaan Schonhage (VU), *Diva: Architectural Perspectives on Information Visualization.*
2001-08	Pascal van Eck (VU), *A Compositional Semantic Structure for Multi-Agent Systems Dynamics.*
2001-09	Pieter Jan 't Hoen (RUL), *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes.*
2001-10	Maarten Sierhuis (UvA), *Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design.*
2001-11	Tom M. van Engers (VU), *Knowledge Management: The Role of Mental Models in Business Systems Design.*
2002-01	Nico Lassing (VU), *Architecture-Level Modifiability Analysis.*
2002-02	Roelof van Zwol (UT), *Modelling and searching web-based document collections.*
2002-03	Henk Ernst Blok (UT), *Database Optimization Aspects for Information Retrieval.*
2002-04	Juan Roberto Castelo Valdueza (UU), *The Discrete Acyclic Digraph Markov Model in Data Mining.*
2002-05	Radu Serban (VU), *The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents.*
2002-06	Laurens Mommers (UL), *Applied legal epistemology; Building a knowledge-based ontology of the legal domain.*
2002-07	Peter Boncz (CWI/UvA), *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications.*
2002-08	Jaap Gordijn (VU), *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas.*
2002-09	Willem-Jan van den Heuvel (KUB), *Integrating Modern Business Applications with Objectified Legacy Systems.*
2002-10	Brian Sheppard (UM), *Towards Perfect Play of Scrabble.*
2002-11	Wouter C.A. Wijngaards (VU), *Agent Based Modelling of Dynamics: Biological and Organisational Applications.*
2002-12	Albrecht Schmidt (CWI/UvA), *Processing XML in Database Systems.*
2002-13	Hongjing Wu (TUE), *A Reference Architecture for Adaptive Hypermedia Applications.*
2002-14	Wieke de Vries (UU), *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems.*
2002-15	Rik Eshuis (UT), *Semantics and Verification of UML Activity Diagrams for Workflow Modelling.*
2002-16	Pieter van Langen (VU), *The Anatomy of Design: Foundations, Models and Applications.*
2002-17	Stefan Manegold (CWI/UvA), *Understanding, Modeling, and Improving Main-Memory Database Performance.*

ISBN 90 6196 517 9