Vrije Universiteit Amsterdam
Faculty of Sciences

Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Marcin Żukowski
Student number: 1289012

Parallel Query Execution in Monet on SMP Machines

Master's Thesis
Specialization: COMPUTER SCIENCE

Supervisors (Vrije Universiteit):
Prof. Henri Bal
Dr. Peter A. Boncz (CWI)

Supervisor (Warsaw University):
Dr. hab. Jerzy Tyszkiewicz

July 2002

The research described in this thesis was performed during the author's internship contract at Centrum voor Wiskunde en Informatica – National Research Institute for Mathematics and Computer Science in the Netherlands, within the theme Data Mining and Knowledge Discovery, a subdivision of the research cluster Information Systems.

This thesis is ready to be marked.

Date: Author's signature:

This thesis is ready to be verified by the second reader.

Date: Supervisor's signature:

Abstract

Parallel execution is a popular method of increasing performance of database management systems. This thesis describes our work on extending Monet – a novel DBMS for query-oriented applications – with parallel query execution in shared-everything environment. We discuss parallel implementations of relational algebra operators adapted to the Monet decomposed storage model and CPU-optimized execution, concentrating on parallelizing cache-conscious query processing algorithms. It turns out that this can be done with mixed success, depending on the used hardware platform. For better analysis of our benchmark results we present a Calibrator tool, that allows generic detection of various features of hierarchical memory systems present in modern computer architecture. We extend this tool to obtain characteristics of the memory subsystem in SMP environment. The obtained results are used to present possible optimization strategies for both sequential and parallel algorithms.

Keywords:

parallel query execution, main-memory DBMS, decomposed storage model, SMP architecture

Contents

Chapter 1

Introduction

Monet is a novel Database Management System (DBMS) [8], designed specifically for query-intensive applications such as *on-line analytical processing*, *data-mining* and multimedia retrieval. Its main design goals are adaptation to specific requirements of these applications and exploitation of the full potential of modern hardware.

Monet uses the *decomposed storage model* [13] optimizing data access cost for typical query-intensive access patterns. Its design as a main-memory DBMS and its *column-at-a-time* data processing model (as opposed to a *tuple-at-a-time* model) allow it to obtain good performance by using algorithms designed to optimize CPU efficiency and minimize memory-access costs introduced by hierarchical memory system.

In this thesis we improve Monet's performance further by extending its parallel features. Although parallel and distributed DBMSs have been studied extensively in the past [26], the Monet-specific features introduce new challenges. Firstly, its partitioned data model brings new problems not existent in traditional relational systems. Moreover, most recent research concentrates on the *shared-nothing* architecture, while we choose *SMP* machines as our hardware platform. We believe that with the dropping prices of such systems and introduction of new families of multi-threaded CPUs, SMPs will become a typical solution either as standalone systems or as nodes in large-scale shared-nothing architectures. It makes SMP parallelism a logical first step towards fully parallel system. This leads to stating first objective of this thesis:

Objective 1: *parallelize query execution in Monet on off-the-shelf SMP machines*

Obtaining maximum performance from commodity hardware requires adapting to its complex architecture. Parameters like cache memory sizes can be used for tuning algorithms to increase their efficiency. To allow application portability, an automatic and platform independent way to obtain detailed hardware characteristic is necessary. Such a tool can be also used to analyze algorithms performance. This leads to the second objective of this thesis:

Objective 2: *create a generic tool for obtaining SMP characteristics*

We present our work in these areas in the following chapters. In Chapter 2, we give a general introduction to different aspects of presented research. We describe general DBMS technology, modern hardware (especially parallel architectures) and most important Monet features. In Chapter 3, we describe our work on parallelizing the fundamental query processing operators from relational algebra in Monet. Moreover, we explain how to combine given algorithms to obtain parallel execution of full SQL queries. In Chapter 4, we present tools to detect various characteristics of modern hardware. We describe obtained information for our test platforms

and use it to explain results of our performance benchmarks. Moreover, we discuss possible strategies of modifying programs for better performance. Finally, in Chapter 5 we conclude this thesis and present possible research areas for the future.

Chapter 2

Preliminaries

This thesis connects knowledge from the different areas including parallel databases, modern computer architecture and the Monet DBMS. Since we do not expect the reader to be familiar with all of them, we start with a short 'walk-around'. We present a general overview and, where appropriate, give a rationale for the choices made for our research environment.

The structure of this chapter is as follows. Section 2.1 presents a general overview on database technology. Then, in Section 2.2 we proceed with description of the current hardware, focusing on parallel systems. These two areas are connected in Section 2.3 where we present parallel and distributed databases. Finally, Section 2.4 presents most important aspects (for our research) of the Monet DBMS.

2.1. Database Introduction

This section provides a short introduction to database research. We describe the motivation behind database technology development, DBMS architecture, SQL language and its theoretical background. More detailed information can be found in literature overviews, e.g. [35]. Readers familiar with this field can skip it.

2.1.1. DBMS Technology

Data management is an important part of most computer programs. In the past, applications used their private implementations of data access routines. This approach required extra effort during software development and additionally tied application to physical data representation. Moreover, applications routines were usually limited. Therefore, specialized technology for data management was introduced.

Database Management System (DBMS) is a software layer allowing data storage and manipulation. Applications can use it as a *"black-box"*, concentrating on *what* to do with the data and not *how*. It greatly improves software design process and its maintenance. Moreover, multiple DBMSs use the same interface, but achieve different performance in various tasks. It allows choosing the best solution for a given problem.

Being application-independent, a DBMS can concentrate on safe and efficient data storage, update and retrieval. DBMSs usually provide many useful additional features, including optimized query execution, automatic parallelization, concurrency control, index structures, data backup and recovery, security and authentication control.

Most DBMSs work in a *client-server* model. Since various clients can work on the same database, it enables cooperation between multiple users and applications. However, concur-

rent updates introduce the problem of *data consistency*, which is tackled with transaction management. Moreover, centralized solution puts extra requirements on server processing and storage capabilities.

2.1.2. OLTP and Query-Intensive Applications

DBMSs were at first used mostly for *on-line transactional processing* (OLTP) applications. Such systems in typical case execute big number of queries, yet each of them is relatively simple. Typical uses include lookup of data about specified customer, changing user's password on a Web site, adding new product to a list of ordered items or deletion of data about retired employee. Each of these queries follows the *tuple-at-a-time* access pattern.

Completely different access pattern is present in the *query-intensive applications*, including *on-line analytical processing* (OLAP), *data-mining* and *decision support systems*. Relations in such applications often contain hundreds of attributes, while typical queries usually access only few of them. An example may be a request to group all insurance company clients according to their age and calculate probability of insurance claims caused by car accidents. Such a query follows the *column-at-a-time* pattern.

Most DBMSs are optimized for the OLTP usage. Monet DBMS, presented later in this chapter, adapts data storage and execution strategies to query-intensive applications requirements.

2.1.3. Relational DBMSs

Various database paradigms have been presented in the past. Most important nowadays are *relational* and *object-oriented* databases. New trends in both hardware and software development have led to introducing new areas like *self-describing* databases (e.g. XML), solutions for multimedia, streaming applications and document storage.

In this thesis we concentrate on query execution in *relational DBMSs* (RDBMSs). It is *de facto* standard nowadays — most important DBMSs (e.g. Oracle, PostgreSQL) belong to this family.

The Relational Data Model

Data in the relational model is stored in structures described by a *schema*. It consists of multiple *relations*. A schema provides detailed information on relation structure: *attributes* , their domain types, data constraints etc. It may additionally contain management properties like definition of indices (data structures improving performance) or access rights. Each relation consists of collection of elements called *tuples*. Since the set of relation's attributes is fixed, and number of tuples is volatile, relations are usually visualized as two-dimensional *tables*, limited horizontally and unlimited vertically, where *columns* represent attributes and *rows* represent tuples.

Figure 2.1 shows a relational schema for a database storing information about customers and their orders. It consists of two tables:

Person with attributes: person identifier, name and city

Order with attributes: order identifier, person identifier, order value and postage cost.

Person

p_id	p_name	p_city
111	Blue	Amsterdam
222	Black	Warsaw
333	White	Amsterdam
444	Green	London
555	Yellow	Warsaw

Order

o_id	p_id	o_value	o_postage
11111	555	100	20
22222	333	200	20
33333	222	150	30
44444	444	200	20
55555	555	280	50
66666	111	150	40
77777	111	300	30
88888	555	120	20
99999	222	250	30

Figure 2.1: Example relational schema with tables Person and Order

```
SELECT
  SUM(o_value) AS sum_value, p_city
FROM
  Person, Order
WHERE
  o_value<=240 AND Person.p_id=Order.p_id
GROUP BY
  p_city
ORDER BY
  SUM(o_value) DESC
```

Result

sum_value	p_city
370	Warsaw
350	Amsterdam
200	London

Figure 2.2: Simple SQL query and its result

The SQL Language

As for the data management, RDBMSs use the *Structured Query Language* for interaction with the clients. SQL statements, called *queries*, are defined by the SQL standard (previously SQL92 [5], now SQL3). The most important SQL query types are `SELECT`, `INSERT`, `UPDATE` and `DELETE` queries. Since we concentrate on query-intensive applications, consisting mainly of data retrieval queries, we concentrate on the `SELECT` query type.

Figure 2.2 presents a simple SQL query and its output. This query asks for a sum of order values for each city, taking into account only orders with the value smaller or equal to 240.

Algebraic Operators

Our example can also be presented using a sequence of *relational algebra operators* [12]. Conceptually, each of them takes one or two source relations and returns a new relation as a result.

First we present a grammar for such an algebra:

```
<RELATION> ::= <OPERATOR> | table(IDENT)
<OPERATOR> ::= project({<EXPR:any>}*, <RELATION>)
             | select(<EXPR:bool>, <RELATION>)
             | join(<EXPR:bool>, <RELATION>, <RELATION>)
             | join(<ATTR>, <RELATION>, <RELATION>)
             | aggregate({<ATTR>}*, {<EXPR:any>}*, <RELATION>)
             | sort({<ATTR>}*, <RELATION>)
<EXPR:T>   ::= <ATTR> | IDENT.<ATTR> | CONST:T | IDENT({<EXPR:any>}*):T
<ATTR>     ::= IDENT
```

Above description uses BNF extended with:

- {X}* means a *list* of zero or more elements X
- Y:Z means element Y of type Z

We introduce these operators in more details. We assume that each operator input is a relation with attributes having unique names. For the operators creating new attributes we will define their naming:

project(*ExprList, rel*) if the expression list contains only source attributes, it returns input relation *vertically* limited to a specified collection of attributes, preserving their names. Its formal notation is:

$$\Pi_{attr_1,\ldots,attr_n}(rel)$$

In the *generalized projection*, the expressions may contain arithmetic functions (working on the source attributes). It introduces new columns explicitly specifying their names. It is denoted as:

$$\Pi_{Expr_1\ [\mathbf{AS}\ name_1],\ldots,Expr_n\ [\mathbf{AS}\ name_n]}(rel)$$

In our example query "`SELECT SUM(o_value) AS sum_value, p_city`" defines that we project sums of orders (giving it a new name) and a city name.

select(*boolExpr, rel*) returns input *horizontally* limited to tuples that match a given boolean formula. It is denoted as:

$$\sigma_{Expr}(rel)$$

It preserves the attribute set of a relation. In our example "`WHERE o_value<=240`" specifies that we take only orders with value smaller or equal to 240.

join(*boolExpr, leftRel, rightRel*) returns a subset of cartesian product of two input relations, limited to tuples that match boolean formula given. It is denoted as:

$$leftRel \bowtie_{Expr} rightRel = \sigma_{Expr}(leftRel \times rightRel)$$

Result relation contains attributes from both inputs. If the attribute name sets are not disjoint, they are additionally extended with the relation identifier as a prefix.

join(*attr, leftRel, rightRel*) is a special kind of join called *natural-join*. It is most common join style in typical queries, used to combine information from other relations. It is denoted as:

$$leftRel \bowtie_{attr} rightRel = leftRel \bowtie_{leftRel.attr=rightRel.attr} rightRel$$

In our example "`FROM Person, Order WHERE Person.p_id=Order.p_id`" defines that we perform join over Person and Order, returning only tuples in which person IDs in both tables are equal.

aggregate(*attrList, funcList, rel*) can be seen as two operations. First we locate all *groups* of tuples with the same combination of grouping attributes values. Secondly, we execute all *aggregate functions* on each group. Typical aggregates include **count()**, **sum()**, **avg()**, **min()** and **max()**. Note that attribute collection may be empty, then aggregates are performed for the whole relation. The result includes all grouping attributes and corresponding results of aggregates, named using identifiers consisting on function name and its position in the list (e.g. `sum3`). Aggregation is denoted as:

$$_{attr_1,\ldots,attr_n}\mathcal{G}_{func_1,\ldots,func_m}$$

In our example, "`SELECT SUM(o_value)`" and "`GROUP BY p_city`" define that we perform summing of order value for each city.

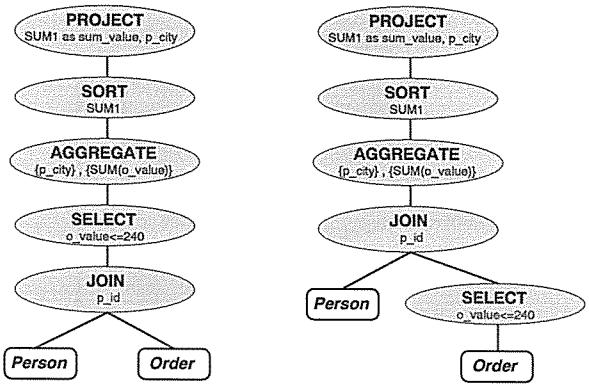

Figure 2.3: Two query trees for the example query

sort(*attrList, rel*) returns a permutation of tuples in the input relation with respect to the value of a given sequence of sort attributes. This operator is not a part of the "traditional" relational algebra, which is *set-based*, hence lacks the concept of tuple ordering. Still, most real-life systems provide it. Sorting does not change the attribute set of a relation. In our example, "`ORDER BY SUM(o_value)`" defines that we expect result to be sorted descending with the sum of orders.

Note that our algebra does not allow using expressions as parameters for **aggregate** and **sort** operators. This is due to the fact, that the same results can be obtained first by performing a proper **project** step, followed by **aggregate** or **sort** using its output.

We concentrate on described operators in Chapter 3 where we present their parallel versions.

Query Trees

Any SQL query can be expressed as a tree of relational operators. In such a *query-tree* the root node is a top-level operator which returns the final result, inner nodes are various relational operators, and leaves represent physical tables. The tree shape determines query execution – it is traversed bottom-up, calculating each node's operator result. Often, multiple mappings of an SQL query to a query-tree exist. Figure 2.3 presents two possible execution trees for our example query. While their outputs will be the same, the second tree will probably have smaller resource requirements, since preliminary order selection is performed before the join. It reduces input and output of selection, but more importantly, it reduces the input of join operator.

In Chapter 3 we will show how parallel implementations of single relational operators can be efficiently combined to allow parallel execution of full queries.

Any SQL query can also be presented as a relational algebra expression. For example, the second query tree from Figure 2.3 can be formally denoted as

$$\Pi_{SUM(o\_value)\ as\ sum\_value,p\_city}({}_{p\_city}\mathcal{G}_{SUM(o\_value)}(Person \bowtie_{p\_id} (\sigma_{o\_value<=240}(Order))))$$

2.2. Modern Hardware and Parallelism

CPU speeds, memory and disk sizes grow exponentially, with about 50% improvement a year, according to *Moore's law* (stated by Gordon Moore in 1965 [31]). However, this improvement

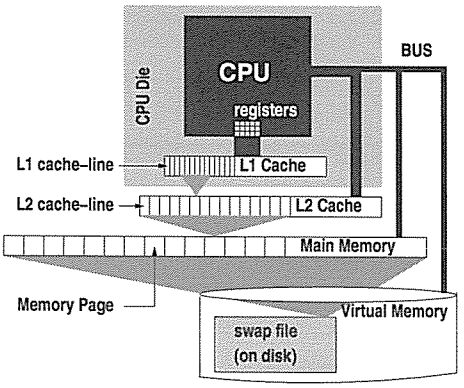

Figure 2.4: Hierarchical Memory System (courtesy of Peter Boncz)

is not equally balanced between different hardware characteristics – progress in some of them is much slower. For example, memory and disk latency times improve at the rate of ca. 1% and 10% a year, respectively. Detailed analysis of this fact with description of its impact on software efficiency will be presented in Chapter 4.

Due to this situation, some assumptions used by the designers of many "classical" algorithms do not hold anymore. A typical example is a memory access. RAM – *random-access memory* – used to mean that accessing any area in the memory takes exactly the same amount of time. Nowadays, with an introduction of the hierarchical memories (multiple cache levels) it is not the case anymore. Therefore, new *cache-conscious* algorithms have been introduced lately [20, 11]. In this thesis, we will adapt our solutions for parallel database processing to these hardware developments.

2.2.1. Hierarchical Memory Model

With the fast development in CPU speeds, the relative performance of the DRAM memory chips (which develop much slower) rapidly decreases. To accelerate access to the data additional *cache memories* made of SRAM have been introduced. First they were placed on the motherboard, now they often are a part of a CPU chip. Figure 2.4 presents a hierarchical memory scheme found in the modern CPUs.

The memory performance in this model may be influenced by multiple hardware parameters:

capacity Sizes of memory chips at different levels can be used as a tuning information for many algorithms

cache line size and number Caches can differ not only in size, but also in partitioning strategy. Each cache-memory is divided into *cache lines*. Their size and count give total cache size. These values can have great influence on other factors – big cache lines usually increase memory latency, yet big number of cache lines may result in less cache misses.

latency In general, *latency* is an amount of time needed to transfer a single byte from sender to receiver. For our purposes, the most important are memory latencies. Since the memory system in modern CPUs is hierarchical, we can define various latencies. *L1 latency* (l_{L1}) is the time needed by CPU to access data from L1 cache. When CPU tries to access data which is not in L1, *L1 miss* occurs. The data has to be transferred from

L2 to L1 – the time consumed is called *L2 latency* (l_{L2}). If the data is not in L2 then an (*L2 miss*), it has to be transferred from main memory, adding *memory latency* (l_{Mem}). Therefore, access to data which is not in any cache costs $l_{L1} + l_{L2} + l_{Mem}$. One should notice that memory is not transferred in single bytes, but in full cache-lines.

cache associativity Modern caches usually keep a few (e.g. 4) possible positions for each memory address. When a new cache line is fetched, its hash value is calculated by taking a few bits from the physical address, and an LRU (*least-recently used*) strategy is used to find final position in the cache. This strategy can be exploited by algorithms, for example to increase the probability of reusing cached data.

address translation An important phase of memory access is translating an address from application's virtual memory area into physical addresses in computer's main memory. To speed up this mechanism modern CPUs contain a *translation lookahead buffer* (TLB) – a kind of cache for recently used translations. Usually it consist of 64 entries. When the logical address is not found in the TLB (*TLB miss*), an address translation has to be performed. It can be either done in hardware (e.g. Athlon, Pentium 4) or in software via operating system (e.g. Origin2000, Sun Ultra). The former case is usually fast, yet the software approach can even exceed 50 CPU cycles.

throughput In general, *throughput* is as amount of data transferred from sender to receiver in a specified period of time. For our purposes, we concentrate on the main memory throughput, additionally checking this factor for the caches. Throughput strongly depends on the latency. However, in many cases it can provide better performance than expected by simply dividing cache line size by latency, because many cache lines can be transferred in parallel.

Note that the time-related values may be measured using either real-time units (seconds) or relatively to the CPU frequency (cycles). We emphasize the importance of the latter, to better realize the imbalance in the various hardware parts development. If we look at the performance values in seconds, we see they are improving. However, measuring them in cycles often shows a dramatical decrease, leading to wasting a lot of CPU power.

2.2.2. Parallelism

Parallel processing and data distribution is a widely used technique for problems with growing computational and storage needs. It may be realized at different levels of the computer architecture:

- inter-node parallelism – connecting multiple computers via network, usually with message passing as the only way of communication
- intra-node parallelism – installing multiple processing units in one machine, sharing all resources
- intra-processor parallelism – exploiting multiple execution units and simultaneous multithreading features of modern CPUs

This taxonomy differs from traditional ones by the last category. Low-level parallelism available inside CPUs is a relatively new phenomenon, introducing new problems and challenges. We believe that with the progress in this area, designing algorithms exploiting these features will become important.

The advantages of parallel systems over sequential ones include:

speedup Ratio of the parallel execution time comparing to a sequential program. System designers usually try to achieve *linear* speedup (equal to the number of computing units).

scale-up Ability to solve the same problem on bigger data in the same time.

replication Using multiple copies of the same data parallel systems offer better *failure resistance* – failure of one node does not result in a loss of continuity of work. Moreover, replication allows increased *data availability* – clients can choose the nearest copy for better performance. It is especially important in wide-area networks (e.g. GRID systems [18]).

Parallel and distributed systems also face a number of challenges not present in sequential systems:

hardware scalability A *scalable* system offers unlimited (or big) number of nodes resulting in speedup/scale-up close to linear. Other aspects should not be strongly influenced by the number of nodes. However, some architectures limit the number of processors used, due to the hardware problems. Moreover, performance of parts of the system (e.g. disks) may be degraded due to access interference of concurrently executing tasks.

algorithm scalability The performance of parallel algorithms often suffers from overheads: communication costs, unparallelizable code costs (*Amdahl's law* [4]) and bad load balancing caused by uneven data distributions and resources. Moreover, algorithms may be inherently sequential, and hence hard or even impossible to parallelize.

consistency control With the introduction of a data replication, sophisticated *consistency protocols* often have to be used, to guarantee the ACID properties of transactions.

cost Many parallel architectures require specialized hardware facilities (e.g. high-speed networks [7]), usually significantly increasing overall cost.

software development Designing and programming parallel systems is much more complex than sequential ones.

In the following sections we discuss these aspects of different parallel hardware platforms in more details.

2.2.3. Parallel Features of Modern CPUs

Current *scalar* CPUs execute instructions by separating different operational stages (like fetching, decoding, executing) into a *processing pipeline*. This way, some of the execution stages can be overlapped (*inter-stage parallelism*). Moreover, thanks to constantly increasing number of transistors on a chip, CPU vendors incorporate multiple execution pipelines in processors. In such *super-scalar* CPUs the same execution stage can be active for more than one instruction, which results in the *intra-stage parallelism*. To fully exploit the potential of the inherent parallelism CPUs deploy few techniques [3]:

- non-blocking caches : cache misses do not result in blocking a specified cache unit. Instead, the request is forwarded to the higher-level cache (or main-memory) and the cache can work on other requests
- out-of-order execution : many consecutive instructions can be executed at the same time, provided the input of one does not depend on the output of another

- speculative execution with branch prediction : after meeting conditional branch instruction the CPU "guesses" the value of the predicate and follows the appropriate instruction sequence. If the "guess" was incorrect, the pipeline has to be flushed and the proper instruction sequence has to be executed. To increase efficiency of branch prediction CPUs are incorporated with *prediction tables* that record predicates output in the past.

Another idea for parallelism inside the CPUs were *single-instruction multiple-data* (SIMD) [17] instructions. They allow performing simple operations on many elements at once. Modern off-the-shelf CPUs are equipped with different SIMD instruction sets, e.g. MMX, 3dNOW and SSE.

Parallel capabilities of CPUs can be further extended with introduction of *simultaneous multi-threading* (SMT) [39]. In this technology, chips are equipped with multiple execution units, having separate resources like registers, pipeline flushing mechanisms, subroutine return prediction. Some resources stay shared, like branch target buffer and translation lookaside buffer (extended with per-context identifiers). Also L1 and L2 caches stay shared. This leads to a truly parallel execution model, in which many independent threads can work simultaneously.

In a few years SMT may become a standard for off-the-shelf CPUs (Intel Pentium 4 Xeon CPUs already use this technology under the name of *hyper-threading*), giving them full-fledged parallel features. This is an additional reason for concentrating on SMP machines in this thesis.

2.2.4. Shared-Everything Machines

In the *shared-everything* (SE) model, all processors have access to common memory (the same address space), disks and input/output devices. System memory can be used as a communication medium. Two hardware architectures use this model: *symmetric multiprocessor* (SMP) and *cache-coherent non-uniform memory access* (ccNUMA) machines.

In SMP machines, one node contains multiple processing units. Since all processors share storage facilities (memory, disks), the problem of distributing data does not occur. It is also easy to program and perform load balancing. However, SMP performance may be decreased by resource interference – multiple tasks concurrently accessing e.g. disk may influence each other's performance. Moreover, SMP machines face one important problem – low scalability. Because of limited bandwidth of the system bus, the cost of cache-coherence protocols and inadequate memory-chips speeds[1] SMP systems can be scaled up to only 8-32 processors. Still, important chip vendors continue research on processors working in an SMP architecture (e.g. AMD's Athlon MP or IBM's Power4 CPUs). We choose the SMP architecture as a hardware platform for our research.

The ccNUMA (or simply NUMA) architecture can be seen as a hybrid of SMP and distributed machine. It consists of many SMP nodes connected via fast network. Running processes share the same address space. However, the memory access is *non-uniform* – accessing data allocated at the same node is much faster than transferring it from another node. NUMA has both advantages and disadvantages over SMP machines. The biggest advantage is better scalability – systems can even scale up to hundreds of CPUs (e.g. 512 in SGI Origin 3800). As for disadvantages, communication overhead results in increased difficulty of developing efficient programs. Moreover, the cost of NUMA machine is usually high due to the specialized equipment used.

[1] some of these issues are addressed in Chapter 4

2.2.5. Shared-Nothing Systems

The *shared-nothing* (SN) architecture consists of multiple nodes with private memory, disks and input/output devices, connected via interconnection network.

For many applications, SN systems obtain performance as good as SE machines. Additionally, SN architecture has one important advantage over SE – scalability. Using off-the-shelf machines with little extra hardware, it is possible to assemble systems consisting of thousands of nodes at a relatively small price. Moreover, SN offers multiple storage facilities, leading to *scale-up* and possibility of data replication. SN drawbacks include harder software implementation due to complicated machine model, slower communication and additional costs of consistency protocols.

For large-scale computing, SN architecture is now the most popular solution. It can be used to create *hierarchical* systems, where each node of upper-level system is in fact a separate parallel system. One typical combination is the SN/SE model, in which each SN node is an SMP machine. The other is SN/SN, consisting of highly-integrated nodes, connected internally with very fast network like Myrinet [7]. Such a hierarchy can be extended even further. Probably the most complex example of the hardware used for parallel and distributed programming are the *GRID* systems [18]. These comprise of nodes spread all over the world, each consisting of multiple machines, each of which can be a parallel system. It introduces architecture with enormous complexity, resulting in a new challenge for system designers.

2.3. Parallel DBMSs

Parallel databases have been studied intensively over the last two decades [26, 32]. In this section, we describe benefits and problems of parallel DBMS (PDBMS), parallel query execution and the relations between PDBMS and hardware.

2.3.1. Benefits and Problems

A PDBMS may improve its performance thanks to *speedup* and increase their capabilities thanks to *scale-up*. Moreover, data replication allows improved failure resistance and increased data availability. Another issue, especially important in wide-area systems, is transparent access to data distributed over multiple nodes.

Another benefit comes from the fact that database queries are expressed using the standardized language SQL. Since the DBMS maintains full control over *how* the query is executed, parallelization can be introduced in *transparent* and *automatic* way. Moreover, a relatively small number of possible relational operators allows preparing highly optimized solutions for generic queries. This differentiate PDBMSs from the general parallel programming, where automatic parallelization is often impossible.

PDBMSs share typical problems of parallel systems (Section 2.2), extended with:

data skew Query execution time for various datasets (even with the same size) may greatly differ, because some values can occur more often than the others. It may influence load balancing and makes prediction of operator costs harder.

concurrency control Preserving data consistency in PDBMSs requires new algorithmic solutions. It is especially difficult in a distributed DBMS, with data partitioned over multiple nodes and often replicated.

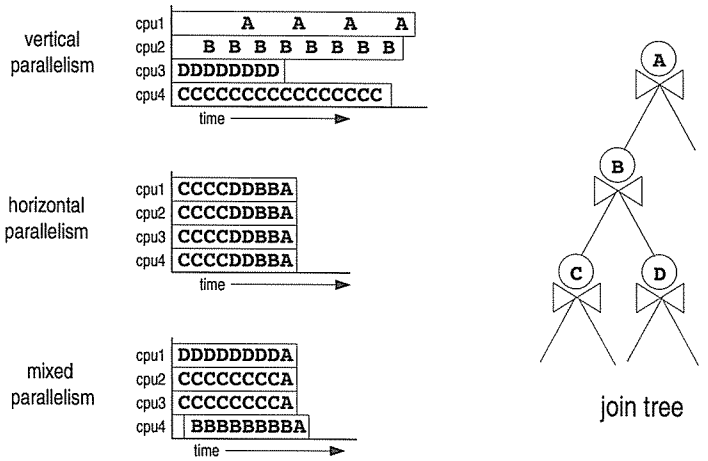

Figure 2.5: Parallel query execution strategies (courtesy of Peter Boncz)

Since many of these problems are application related, some of the recent work in DBMS parallelization concentrates on domain-specific research, including data mining [41], decision support systems [37] and geographical information systems (GIS) [34].

2.3.2. Parallel Query Execution

There are various mechanisms for parallel query execution:

intra-operator parallelization Parallel execution of a single algebraic operator. It is especially useful in the query-intensive applications where one operator can take minutes or hours to execute.

intra-query (inter-operator) parallelization Some branches in an operator tree may be executed in parallel. Moreover, partial results can be re-used.

inter-query parallelization Running multiple transactions as independent threads is a relatively easy solution, provided by most DBMS vendors. It is useful especially for typical OLTP usage. Moreover, for applications running many similar queries (like data-mining) it is possible to re-use partial results between them.

In a query tree, operators can be classified as *blocking* and *non-blocking*. For blocking operators, the output can be returned only after all the input tuples have been read (e.g. *sort* operator). Non-blocking operators can return tuples right after reading it from input (e.g. projection). For some operators, this classification depends on algorithm used. For example, *grouping by hashing* (Section 3.4) is blocking, while *grouping by sorting* may be non-blocking given sorted input. Moreover, some algorithms require an initial blocking phase, but then proceed in a non-blocking way, e.g. *build-probe hash join* [33].

This classification allows for different strategies of distributing work between processors, presented in Figure 2.5. Non-blocking operators bring *vertical parallelism* – each processor is given an operator, executing query in a *pipelined* way. However, since various operators usually differ in execution cost, it often leads to non-optimal performance, since some CPUs may be stalled while awaiting for the data. In the *horizontal parallelism*, each operator is executed by all processors. It usually obtains a good performance, however, it requires full materialization of partial results. The mixed approach [40] allows combining these two strategies, resulting in similar performance and minimizing cost of materialization.

2.3.3. Exploiting Parallel Hardware

One of the early ideas for introducing parallelism in DBMSs were *database machines*, e.g. DIRECT [14] and PRISMA [6]. They used hardware designed for database purposes, equipped with specialized elements, like on-disk CPUs or sorting processors. However, due to a high cost of this solution and because of low portability and short life-cycle of such a DBMS hardware, this approach has been abandoned in middle eighties.

Nowadays, most of the research on PDBMSs concentrates on the shared-nothing architecture. Distributed databases [32] consist of both wide-area systems using traditional networks and highly-integrated *cluster* solutions, using specialized fast communication methods. Although the general *shared-nothing* paradigm stays the same, different communication speeds may influence algorithms used.

As for shared-everything machines, they are often used as nodes in distributed databases. Recent research in this area concentrates on analyzing the differences between DBMS and typical scientific applications requirements, and its impact on performance of SMP systems [22, 16]. Moreover, some research is done on the new architectural trends, especially SMT technology [25]

More detailed overview of the influence of parallel hardware architecture on DBMSs can be found in [36].

2.4. Monet

The Monet DBMS [9, 8] is a research project in the database group of the Centrum voor Wiskunde en Informatica (CWI) – National Research Institute for Mathematics and Computer Science in the Netherlands. It is designed to fully exploit modern hardware features for work with query-intensive applications. As such it powers the commercial data mining tools of the CWI spin-off *Data Distilleries*. It has been also successfully adapted to other areas, including geographical information systems (GIS), XML manipulation and multimedia retrieval. Because of its flexibility, it was also a base for other experiments, e.g. *scalable distributed data structures* [21].

In this section we present the most important features of Monet, relevant for the remainder of this thesis.

2.4.1. Design Goals

The most important design objectives of Monet are:

- *providing efficient DBMS for query-intensive applications* – Monet is specifically designed to obtain good performance in applications like on-line analytical processing and data mining.

- *exploiting modern hardware features* – Monet aims to achieving maximum performance on current hardware architecture. It is realized thanks to CPU optimized code and cache-conscious algorithms.

- *supporting multiple data models and query languages* – Monet is designed to allow mapping various data models (e.g. XML, objects) onto its native storage format. Moreover, thanks to introducing intermediate execution layer – MIL – it allows translation and execution of different front-end languages.

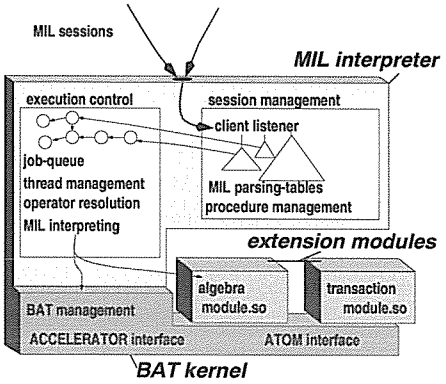

Figure 2.6: Monet software architecture (courtesy of Peter Boncz)

- *extensibility to new domains* – Monet may be extended with new data types, accelerator structures and language primitives.

2.4.2. Architecture

Monet is designed as a *main-memory database management system* (MMDBMS). While persistent data is stored in files, all intermediate results and temporary relations are kept in the main memory. To allow using large-volume data, Monet relies on the operating system's virtual memory mechanisms. Such an approach proves to be effective in the applications Monet is designed for. Since most OLAP-like queries access only few attributes from a relational table, even data few times bigger than available memory can be processed efficiently.

Another important design decision in Monet is the use of the *decomposed storage model* [13]. In this strategy, all attributes of a relational table are stored in separate *binary* relations, in Monet called *Binary Association Tables* – BATs. Section 2.4.3 describes this approach in more details.

Monet provides only *back-end* functionality for different *front-end* applications. This is made possible by an intermediate execution layer – *Monet Interpreter Language* (MIL), described in Section 2.4.5.

The Monet architecture is presented on Figure 2.6. Three important parts can be identified:

- *BAT kernel* – a low-level layer, offering basic data structures and management functions.
- *MIL interpreter* – a layer for parsing and executing MIL queries. Provides support for multiple concurrent transactions.
- *extension modules* – Monet is equipped with rich set of external routines, providing additional functionalities, new data types, index structures etc. They can consist of both binary routines as well as of MIL programs.

The multi-layer architecture allows performing optimizations at different levels. Front-end applications can use *strategical* optimizations to exploit domain-specific knowledge. The MIL interpreter supports the *tactical* optimization phase by removing common subexpressions, introducing parallelism, etc. Monet kernel can perform *operational* optimization – choosing different algorithms for performing relational operators.

Additionally, multi-level query execution allows reducing requirements on the kernel. Some expensive operations executed by many DBMSs in every situation, in Monet can be performed

on explicit demand. For example, the Monet kernel does not provide full transaction management. Instead the `trans` module introduces MIL extensions providing this functionality when necessary.

To illustrate possibilities of multi-layer solution, Monet is equipped with an SQL front-end (developed by Niels Nes). It allows using Monet as an RDBMS server, providing transparent relational data mapping and SQL to MIL translation.

2.4.3. Binary Relations

BATs used in Monet consist of two columns: *head* and *tail*. Aligned values from both columns are placed together in a memory as *binary units* (BUNs). A BAT descriptor consists of BUN heap (continuous memory area) information as well as the head and tail column descriptors. Figure 2.7 presents the implementation of BAT.

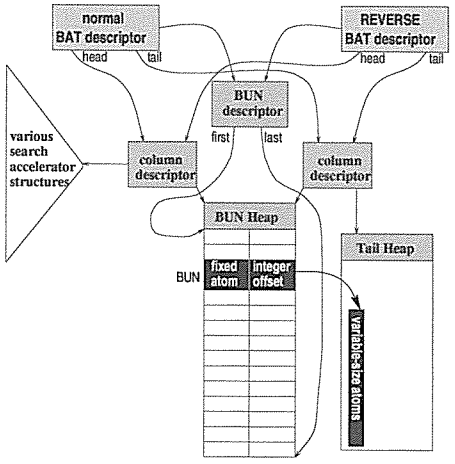

Figure 2.7: The BAT data structure (courtesy of Peter Boncz)

Each column consists of tuples of fixed-size data types (e.g. integers, chars). Variable-size data types (like strings) can be stored by saving an offset to an external heap structure in a BUN. Additionally, there is a special column type VOID (virtual-OID). It is a virtual data type for dense and ascending integer values. Such data is often found in tables translated from the relational model into Monet's BATs. This type saves memory and allows speeding up several operations.

The advantages of using binary partitioning in Monet include:

- mapping flexibility – different *front-end* data structures can be mapped onto BATs. Figure 2.8 presents a translation of the relational scheme from Figure 2.1 onto binary fragmented model. Similar mapping schemes can be applied not only for typical tabular data, but also for complex structures, including XML and graphs.
- small memory requirements – only a relevant data are loaded into main memory.
- efficient processing – while processing BATs all fetched data is (usually) processed, resulting in good I/O performance. Moreover, processing continuous memory blocks minimizes number of cache misses.

However, binary fragmentation also brings problems. In particular, it incurs the cost of data defragmentation. The result of one operator can consist of tuples belonging to one of the

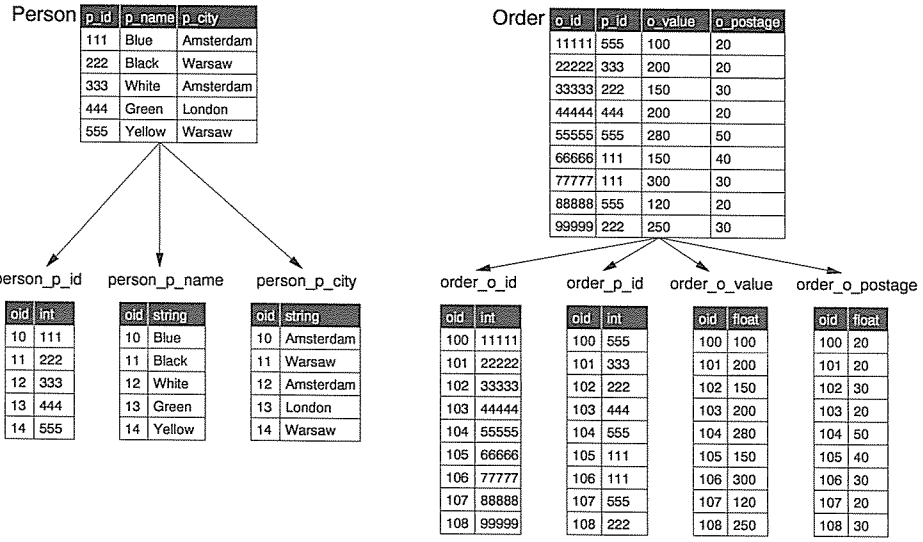

Figure 2.8: Mapping relational scheme onto BATs

relational table attributes, while the following operator may wish to use another one. In such a situation an extra attribute projection is necessary. Still, thanks to the VOID columns, and storing all attributes of the relation in aligned BATs, this operation can be done efficiently in most cases. In Section 3.7 we will present efficient projection algorithms suited to modern hardware characteristics.

2.4.4. CPU and Memory Optimized Code

Monet adapts its query execution strategies to current hardware characteristics and typical OLAP query schemes. For this purpose it uses the following optimization ideas:

- simple data structures – all data is stored in BATs, which are consecutive memory areas, seen by the program as an array. Accessing such a structure is extremely effective. As for accelerator (index) structures, the main memory access model makes structures like B-trees and linear hashing (efficient in disk DBMS) less effective than simpler T-trees (variant of AVL) and bucket-chained direct hashing.

- aggressive code expansion – Monet uses different routines for the same operations on different inputs. Looking at the source relations data types, as well as some special information stored in BAT descriptors (e.g. `sorted` or `dense` fields) the proper function implementation can be chosen. Code expansion is available thanks to using *Mx* tool [23]. With a little programmer's effort some routines are expanded to hundreds of different versions. Note, that such approach is impossible in traditional RDBMS, where there is unlimited number of combinations of input data types. Monet approach, resulting in a simple code, also enables good optimization on the compilation level, using e.g. *loop-unrolling*, *memory prefetching* and even using specialized vector instructions of modern CPUs (MMX etc.).

- full data materialization – Monet execution model consists of a series of consecutive algebraic operators, with each of them fully materializing its result. Combined with code expansion, it allows only constant number of decisions to be made during query execution. It minimizes branch misprediction problem and therefore improves overall performance

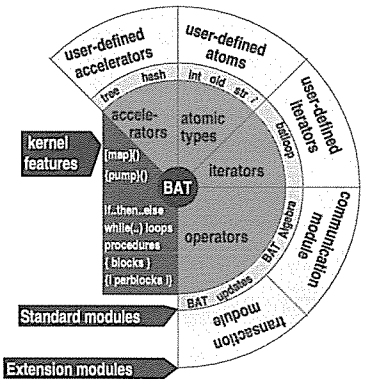

Figure 2.9: The structure of MIL (courtesy of Peter Boncz)

on modern super-scalar computers. However, it puts additional requirements on available memory sizes.

- cache-aware algorithms – Monet developers presented few efficient database algorithms, adapting to current memory and CPU limitations, e.g. radix-cluster join [27], radix decluster [8]

Together with the binary fragmentation strategy, CPU optimization introduces a new problem. Since for each tuple a small number of instructions are performed, and usually only data that has to be processed is loaded from memory, Monet operators can be characterized with a low *instructions per byte* ratio. This puts higher requirements on memory bandwidth. As we present in Chapter 4, during parallel execution it is even more important, introducing scalability problems.

2.4.5. The MIL Language

Monet Interpreter Language (MIL) [10] is a procedural language, giving full access to Monet core functionality. Such an approach allows relatively easy extension with new front-ends (simply by writing top-level-language to MIL translator).

The structure of MIL is presented in Figure 2.9. Its most important features include:

- BAT algebra: programs in MIL are expressed using *column-wise algebra*. It contains features similar to traditional relational algebra, adapted for vertical fragmentation.

- control structures: MIL contains most typical features of structural programming languages, including procedures, loops and comparison statements.

- block structures: script can be divided into blocks, creating new (nested) variable scopes. A *sequential block*, denoted by $\{..\}$, specifies that enclosed statements should be run consecutively. A *parallel block*, denoted by $\{||..||\}$, runs statements inside the block in parallel. Mixed combinations of both are possible.

- command overloading: MIL allows multiple commands/procedures implementations for different parameters.

- simple data types: the only data types supported by MIL are atomic values (of fixed or variable size) and BATs (possibly nested).

- iterators: these are MIL constructs that execute a MIL statement for a number of tuples in given relation.

Below we present the list of most important MIL primitives:

select (bat[H,T] A, str f, ...p_i...):bat[H,T] – returns a subset of tuples from A for which function f with a given tail of a tuple and specified parameters p_i returns **true**

select (bat[H,T] A, T lo, T hi):bat[H,T] – version optimized for range-select. Returns all tuples in which $lo <= tail <= hi$.

join (bat[H_1,T_1] A, bat[T_1,T_2] B):bat[H_1,T_2] – returns result of join of A and B where the tail in A is equal to the head in B.

reverse (bat[H,T] A):bat[T,H] – returns a *reverse view* of A – result's head and tail columns are switched. This is a zero-cost operation in MIL – new *BAT descriptor* is created, but BUN heap remains the same (Figure 2.7).

mirror (bat[H,T] A):bat[H,H] – returns a *mirror view* of A – result relation consists of two columns identical to A's head. It is also zero-cost operation.

mark (bat[H,T] A, *oid* v):bat[H,*void*] – returns a *pivot* table, in which A's tail column is replaced by a special *void* column, containing ascending values, starting with v. It is a zero-cost operation.

slice (bat[H,T] A, *int lo*, *int hi*):bat[H,T] – returns a *vertical* slice of an A, taking tuples with indices from *lo* to *hi*. It is a zero-cost operation.

group (bat[*oid*,T] A):bat[*oid*,*oid*] – performs *grouping* over A's tail value, returning for each A's tuple its head and group identifier. Groups are identified by ID of one of the belonging tuples – it allows fast lookup of the corresponding attribute values.

derive (bat[*oid*,T_1] A, bat[*oid*,T_2] B):bat[*oid*,*oid*] – takes *grouped* A and performs sub-grouping according to the tail values in B. The result tail values are the new group identifiers.

sort (bat[H,T] A):bat[H,T] – returns a permutation of A sorted according to the head value.

refine (bat[*oid*,T_1] A, bat[*oid*,T_2] B):bat[*oid*,*oid*] – refines ordering of *tail-sorted* A by sub-sorting on tail values of B, where A and B head columns match 1-1.

pump operation, denoted as **{**f**}(bat[***void*,*oid*] $EXTENT$, **BAT[***void*,*oid*] $GROUPING$, **BAT[***void*,T] $COLUMN$**):bat[void,**R] – performs f aggregate function (e.g. `SUM()`) over values from $COLUMN$ divided into disjoint groups identified by corresponding tail value in $GROUPING$. The result consists of a BAT with head values from $EXTENT$'s head and tail containing results of f for group identified by $EXTENT$'s tail.

multi-join map or **multiplex** denoted as [f](**bat[**H,T_1,**]** A_1, ..., **bat[**H,T_n**]** A_n**): bat[**H,R**]** – performs *implicit* equi-join over head columns of given BATs, and executes f for all possible combinations of tail values. The result contains source head value and the result of f. It is also possible to pass non-BAT variables and constants, as long as at least one of the parameters is still a BAT.

```
t1 := order_o_value.select(int(nil),240);   # select:  [order_oid, order_o_value]
t2 := t1.mirror.join(order_p_id);           # project: [order_oid, order_p_id]
t3 := t2.join(person_p_id.reverse);         # join:    [order_oid, person_oid]
t4 := t3.reverse.mark(oid(0)).reverse;      # mark:    [void, person-oid]
t5 := t4.join(person_p_city);               # project: [void, person_p_city]
t6 := group(t5);                            # group:   [void, group-oid]
t7 := t6.tunique.mark.reverse;              # extent:  [void2, unique-group-oid]
t8 := t3.mark(oid(0)).reverse;              # mark:    [void, order_oid]
t9 := t8.join(order_o_value);               # project: [void, order_o_value]
t10:= {sum}(t7, t6, t9);                    # sum:     [void2, localsum]
t12:= t7.join(t5);                          # project: [void2, person_p_city]
[printf]("| %6d | %10s |\n", t10, t12);     # print result
```

Figure 2.10: MIL translation of the example SQL query

batloop denoted as $RelA$@[N]**batloop** *mil* – an example of an *iterator*. It performs sequential scan over $RelA$ executing *mil* statement for every tuple. If [N] is specified, the work is divided among N independent threads. Inside the mil statement the tuple head and tail can be referred to by using `$h` and `$t`, respectively.

For programming convenience MIL allows using special notation, allowing *quasi* object-oriented style – a function call $\texttt{f}(p_1,p_2,\ldots,p_n)$ is equivalent to $p_1.\texttt{f}(p_2,\ldots,p_n)$.

Figure 2.10 presents a possible translation of SQL query described in Figure 2.2 into MIL. It uses most of the described MIL primitives. As can be seen, a MIL program contains more statements than corresponding relational formula. This is due to the extra projections necessary due to the binary data partitioning.

2.4.6. Parallelization in Monet

Although Monet is not yet a full-fledged parallel DBMS, it contains the necessary features to realize different types of parallel query execution.

An important Monet's idea is its multi-threaded architecture. Monet kernel maintains a collection of available threads, and distributes work between them. This element is sufficient to create *inter-query parallelism.*

The MIL language supports execution of parallel blocks and parallel iterators. Additionally, the `lock` module extends Monet with the implementation of traditional locks and semaphores. These features allow *inter-operator parallelism.* Using a *MIL Squeezer* tool [29], Monet can already detect possibly parallel paths in a query graph. For some queries it gives significant performance improvement.

To allow distributed computation, Monet gives access to a socket layer. MIL scripts can communicate with different hosts, sending data or MIL statements (remote code execution).

However, Monet lacks support for *intra-operator parallelism.* Since its main application area are query-intensive problems, it is often the case that single algebraic operators consume most query execution time. Therefore, in section 3 we will present parallelization strategies for most important algebraic operators.

2.5. Summary

In this chapter we have presented the preliminaries required for the remainder of this thesis. In Section 2.1 we have described the idea of the DBMS, relational data model, SQL language and algebraic operators. In Section 2.2 we have presented features of modern hardware,

concentrating on the hierarchical memory system, inherent parallelism of modern CPUs and shared-everything and shared-nothing architectures. In Section 2.3 we gave an overview of the parallel database technology, concentrating on strategies for parallel query execution and adaptation to hardware architectures. Finally, in Section 2.4 we have described most important aspects of the Monet DBMS – decomposed storage model, CPU and memory optimized execution and MIL language.

Chapter 3

Operator parallelization

3.1. Introduction

In Section 2.1.3 we described the possibility of representing a typical database query as a sequence of *algebraic operators*. The Monet SQL front-end first rewrites a query into a sequence of such operators. Then each algebraic operator is translated into equivalent sequential MIL code. In this chapter, we present our work on parallel versions of such translations.

First we present general ideas behind this translation process. Then we proceed with the algebraic operators – for each we describe its mapping to sequential MIL code, then we present a parallel version of it, and finally we give performance test results.

Pivot Relations

In our translations we assume that each operator creates a *pivot*. A pivot is a `[void,oid]` BAT, storing in a head *densely ascending* identifiers of the produced relation (e.g. 0,1,2,...), while in a tail keeping identifiers of the operator's source. In effect a pivot tells for each tuple in a result relation which tuples in the underlying relation produced it. Pivots are crucial in the MIL query processing. Note that its use of a `void` column instead of an `oid` minimizes storage cost and enables fast value lookup.

Horizontal Parallelism

As we described in Section 2.3, there are two basic strategies for parallel query execution: *horizontal* and *vertical* parallelism. In this thesis we concentrate on the horizontal approach, as it allows good load-balancing between CPUs. Its additional cost – full result materialization – is present in Monet anyway.

Data Fragmentation and Defragmentation

In an SMP environment data is available to all CPUs without the cost of transmission present in shared-nothing architectures. Therefore it can be divided between CPUs in any convenient way. Since Monet's BATs are continuous memory areas, two general fragmentation strategies are possible:

slice – a BAT is horizontally *sliced* into multiple chunks, without looking at the stored values. In Monet it can be done using zero-cost `slice()` operator.

partition – a BAT is divided according to some criteria basing on the values of stored data. An example is `rangesplit()` operator – it tries to return balanced chunks containing data from disjoint domain ranges.

We concentrate on the sliced solution, since the cost of partitioning is usually too high.

During parallel operator execution some presented algorithms produce partial results (usually one for each input slice or CPU). In such a situation creating a single pivot requires an extra phase of data combining. In Section 3.8 we will discuss situations where it is possible to skip this phase.

Before performing data slicing, we usually calculate proper slice boundaries using MIL procedure `make_bounds(size, NSLICES)`. It divides given size into balanced ranges, returning an `[int,int]` BAT containing beginning and end positions of them.

Concurrency control

The algorithms we present usually work using the *divide and conquer* strategy – after data partitioning, the concurrently running threads do not interact with each other until they finish their work. In cases when synchronization is necessary we use two primitives from the `lock` module: `lock_set()` and `lock_unset()`, working as typical semaphore instructions. In descriptions of our algorithms we usually assume that semaphore-type variable `lck` is available.

Hardware Platforms

For the benchmarks we have used two hardware platforms:

- 4-CPU Pentium III (Katmai core) with 550 MHz clock and 1GB memory
- 2-CPU Athlon MP with 1400 MHz clock and 1GB memory

In most cases we only present results for the Pentium machine. Results for the Athlon are similar, yet the speedup is often better since this architecture suffers less from memory starvation[1].

3.2. Expression Evaluation

We begin our description of algebraic operator translations into MIL with the **project** operator. However, in this section we concentrate on only one part of it – expression evaluation. The other part – attribute projection – will be described in Section 3.7.

In our algebra, expression evaluation is used to perform arithmetic functions over existing relation attributes, resulting in new relation columns.

Sequential Version

In general, every expression can be presented as a tree. Leafs in such a tree are relation attributes or constants, while inner nodes are arithmetic functions. It can be automatically translated into MIL using the simple mapping:

```
f(EXPR$1, ..., EXPR$N) => [f](EXPR$1, ..., EXPR$N);
```

In this mapping, as in the others presented in this chapter, we use capital letters to distinguish production rules from MIL statements. Figure 3.1 presents example SQL query, its projection expression as a tree and equivalent MIL code.

[1] as presented in Section 4.2

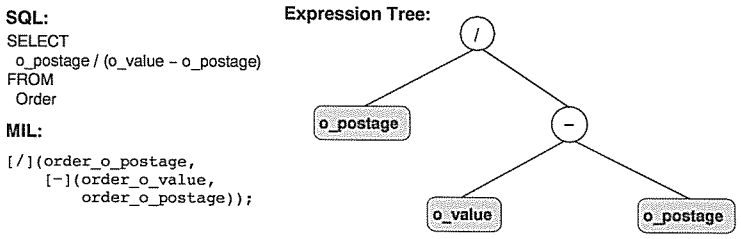

Figure 3.1: An example expression tree

Parallel Version

In our algebra, expression evaluation is not used as a standalone operator. Its result is usually an input for some other operator. Therefore, it is possible to leave the output of this operator *fragmented*. The parallel version of our example, running with `NSLICES` slices and `NCPUS` threads is simple:

```
bounds := make_bounds(order_o_postage.count, NSLICES);
res    := new(bat,int,NSLICES);
bounds@[NCPUS]batloop {
  sl_postage := order_o_postage.slice($h,$t);
  sl_value   := order_o_value.slice($h,$t);
  sl_res     := [/](sl_postage, [-](sl_value, sl_postage));
  lck.lock_set();
  res.insert(sl_res,$h);
  lck.lock_unset();
}
```

In the resulting `res` relation the head contains partial results, and the tail defines corresponding position in the input relation. Note that this operation does not create a new pivot, since for each source tuple there is exactly one tuple in the result. Therefore, obtained partial results can be used directly as an input for the next operator[2].

Benchmarks

We have conducted benchmarks using the `EXTENDEDPRICE` attribute from the `LINEITEM` relation from standard TPC-H [38] benchmark with a scaling factor (SF) equal to 1. It consists of ca. 6 million tuples. We have executed different arithmetic functions on it, to see the impact of the computation cost on the absolute performance and scalability.

The results are presented in Figure 3.2. It shows execution of both sequential program and parallel version using 1,2 and 4 threads. The top two diagrams present execution time, while the bottom ones show the speedup of the parallel version. As we see, since parallel version does not introduce any (significant) extra cost, the execution time using 1 thread is the same as for sequential version. Moreover, with growing cost of the function, the speedup increases. The reason for that – varying *instruction-per-byte* ratio – will be explained in Section 4.3.2

3.3. Selection

The selection operator returns a subset of the relation tuples that satisfy given predicate. In our algebra it is denoted as **select(*boolExpr, rel*)**.

[2]assuming that it uses the same number of slices

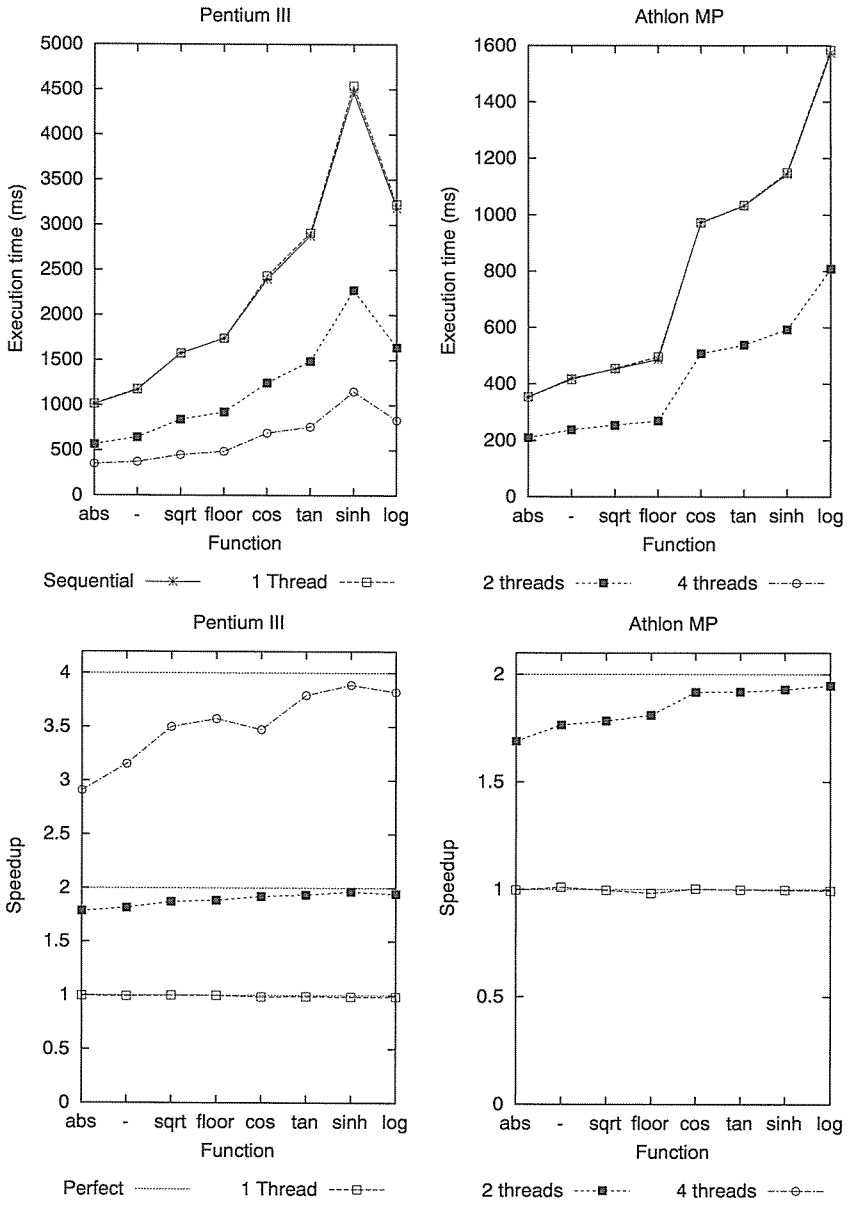

Figure 3.2: Performance of expression evaluation on 4-CPU 550MHz Pentium III and 2-CPU 1400MHz Athlon MP: execution time and speedup

Sequential Version

SQL uses boolean expressions to define tuples to be selected. Such an expression can be translated into MIL using the strategy presented in the previous section. If we use this solution, execution of such a query will result in an additional boolean attribute specifying the output of our expression for each tuple. Then we can use the MIL `uselect()` primitive to select all IDs of tuples for which expression returned true. Such a strategy can be defined using the following mapping:

```
SELECT(EXPR) => boolResult := EXPR;
                pivot := boolResult.uselect(true).mark.reverse;
```

This mapping produces a new pivot table, in which the head defines new tuple ordering and the tail is the tuple position in the source relation.

Described algorithm works well for general queries, however, it can be often optimized. MIL provides two primitives for fast tuple selection: `select(bat,val)`, which returns tuples with tail equal to `val`, and `select(bat,lo,hi)`, which returns tuples such that `lo<=tail<=hi`. The `uselect` versions of these primitives return only head values for faster execution and smaller memory footprint. These primitives can increase performance e.g. by exploiting special BAT properties and existing index structures.

Let us compare two versions of the MIL translation of the "`SELECT * FROM Rel WHERE a1<=100 AND a2+a3>=200`" SQL query:

```
#multiplex version
boolResult :=
  [and](
    [<=](rel_a1, 100),
    [>=](
  [+](rel_a2, rel_a3),
  200));
pivot := boolResult.uselect(true)
         .mark.reverse;
```

```
#optimized version
tmp1 := rel_a1.uselect(nil,100);
tmp2 := [+](rel_a2,rel_a3);
tmp3 := tmp2.uselect(200,nil);
tmp4 := tmp1.kintersect(tmp3);


pivot := tmp4.mark.reverse;
```

As we see, the two strategies can be combined. The `kintersect` primitive returns intersection of two BATs looking only at their head values.

Parallel Version

For parallel execution of **select** we use a simple *divide-and-conquer* strategy. It executes the following steps:

1. sequential: divide the source relation's size into equal ranges using `make_bounds()` procedure. We assume execution time for each tuple to be the same, hence it should result in good load-balancing.
2. parallel (each CPU):
   - take the next available range boundaries
   - perform `slice` on all source relations (zero-cost operation)
   - perform sequential `select` on private slices to obtain partial pivot
   - store the BAT identifier of the obtained pivot in a common result collection, together with information about its final position (sum of sizes of previously saved results) in the result

3. sequential: create result relation with the size of the sum of sizes of partial results

4. parallel: insert all partial results at the proper position in the final relation

The parallel version of our example working with NCPUS threads and using NSLICES slices would look like this:

```
bounds:=make_bounds(rel_a1.count, NSLICES); #step 1
parres:=new(bat,int);
curpos:=0;
bounds@[NCPUS]batloop {                     #step 2
  sl_a1:=rel_a1.slice($h,$t);
  sl_a2:=rel_a2.slice($h,$t);
  sl_a3:=rel_a3.slice($h,$t);
  boolResult :=
    [and](
      [<=](sl_a1, 100),
      [>=](
        [+](sl_a2, sl_a3),
        200));
  ppiv:=boolResult.uselect(true);
  lck.lock_set();
  slices.insert(ppiv, curpos);
  curpos:=curpos + ppiv.count;
  lck.lock_unset()
}
result:=new(oid, void, curpos);             #step 3
slices@[NCPUS]batloop {                     #step 4
  result.insert_into($h,$t);
}
pivot:=result.mark(0).reverse;
```

This strategy faces a problem of an additional data copying phase (step 4). This is due to the fact that Monet can only work with relations stored in a continuous memory area. Since select output size in not known beforehand, it is impossible to place partial results directly in final relation.

Benchmark Results

As a source relation we have taken LINEITEM.PARTKEY relation from standard TPC-H benchmark with a scaling factor (SF) equal to 1. It consists of 6 millions tuples from domain of 200 thousands possible values. We performed a simulation of the following SQL query:

```
    SELECT partkey
    FROM Lineitem
    WHERE (partkey>=100000-BOUND) and (partkey<=100000+BOUND)
=>  # generic translation
    v1:=[<=](lineitem_partkey,100000+BOUND);
    v2:=[>=](lineitem_partkey,100000-BOUND);
    v3:=[and](v1,v2);
    v:=v3.uselect(true)
      .mark.reverse;
=>  # optimized translation
    v:=lineitem_partkey.uselect(100000-BOUND,100000+BOUND)
      .mark.reverse;
```

We used a changing BOUND value to examine the influence of number of selected tuples on the performance. We present two translations – the first is a generic translation using

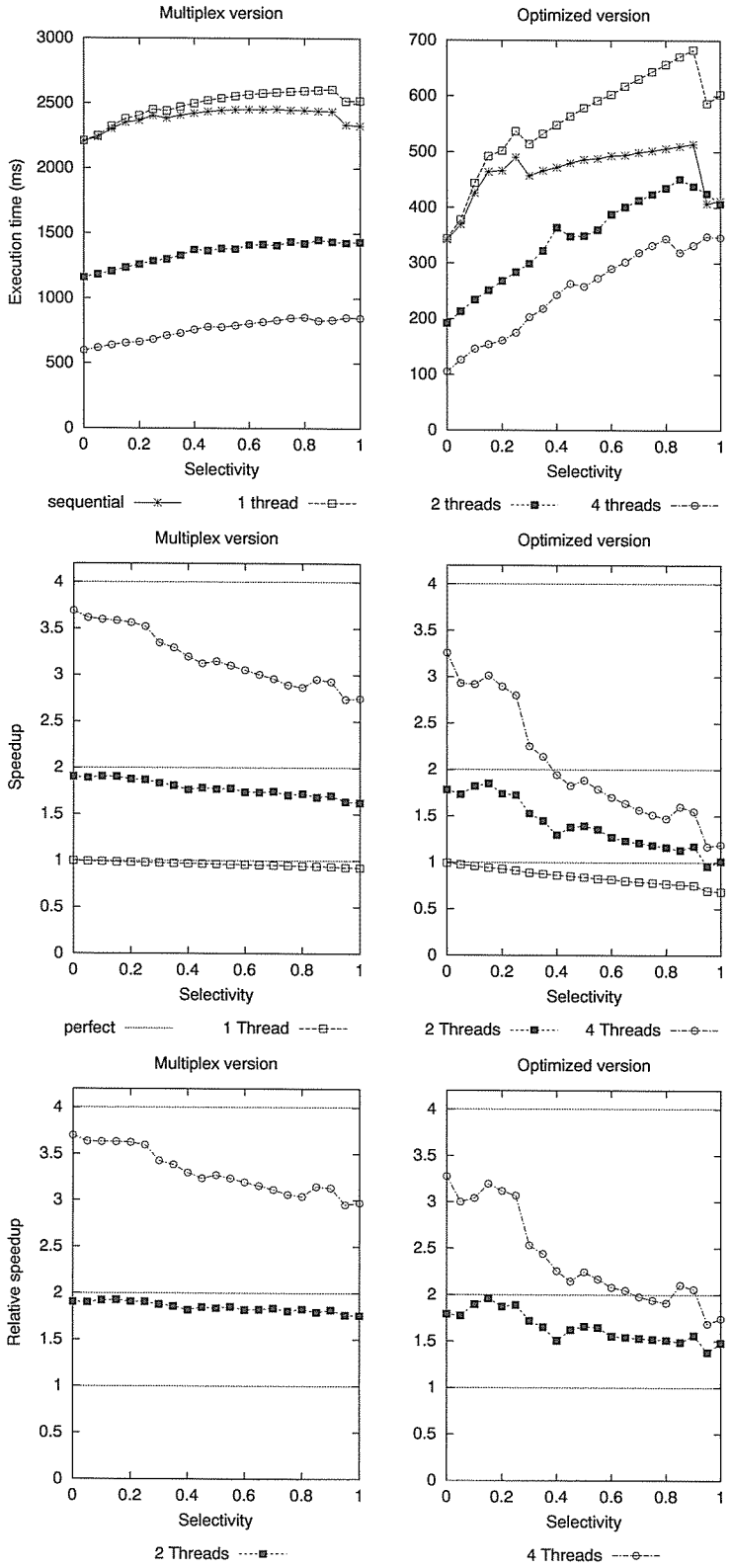

Figure 3.3: Performance of multiplex and optimized selection on 4-CPU 550MHz Pentium III: execution time, speedup and relative speedup

multiplex primitive, while the other is the optimized Monet function performing range-select. Figure 3.3 presents the results. The top diagrams present the execution time of a sequential program, and parallel version running using 1, 2 and 4 CPUs, one execution thread for each CPU. We performed tests with the parallel algorithm running on a single thread to see what is the overhead of the data copying. The diagrams in the middle shows parallel speedup comparing to the sequential version. The bottom diagrams present speedup of the parallel algorithm executed on 2 and 4 CPUs with respect to its single-threaded execution.

The obtained results clearly show that extra data copying influences overall performance. The single-threaded parallel algorithm significantly slows down with the huge number of selected tuples. Moreover, the bottom diagrams present an interesting phenomenon. Although the number of tuples additionally copied is the same for different number of CPUs, 2 and 4-threaded benchmarks slow down (relatively) with growing result size. In Chapter 4 we will show that the reason for that is the bad performance of the memory subsystem, especially low scalability of memory writing.

Comparing the performance of two presented versions, we see that the optimized version obtains much better absolute performance. There are two reasons for that: it performs only one data scan and it uses a CPU-wise optimized operator, which has all range-evaluation code inlined, whereas the multiplex version performs a C routine call for each tuple. However, if we look at the speedup, we see that optimized version has much worse scalability. This is because it is more *memory-bound*, hence the hardware limitations influence it more. Another issue is that the optimized version is more sensitive to the change of selectivity. It is because for the non-optimized version the relative cost of `uselect()` is lower, due to a more expensive execution of the multiplex primitive.

Additional experiments showed that the influence of extra data copying depends on the complexity of the selection formula. For queries with a simple predicate, the overhead observed was even larger than presented. For queries with multiple predicates[3], its cost was better amortized, resulting in much better speedup.

The problem of 0extra data copying in some cases could be solved by leaving the result partitioned (as we did for the expression evaluation). However, since **select** introduces a new tuple ordering, the solution is not as straightforward. In Section 3.8 we will show how this problem can be solved in an efficient way.

3.4. Aggregation

The relational operator **aggregate(*attrList, funcList, rel*)** returns results of *aggregate functions* performed over the source relation divided into groups with the same values of attributes specified. These *grouping attributes* are also returned.

Conceptually, aggregation can be divided into two steps. The first is *grouping* – the process of finding tuples sharing the same values of grouping attributes. The second phase is the *aggregate evaluation* – performing aggregate functions for groups found.

Two special cases of an aggregation are possible. When grouping attributes are not specified, the aggregate functions are performed over the *entire* source relation, without grouping phase. When aggregate functions are not specified, only the unique combinations of grouping attributes are selected. It is equivalent to the SQL statement "`SELECT DISTINCT attr1,...,attrN FROM relation`".

In a traditional RDBMS the implementation of `GROUP BY` and following aggregation is relatively straightforward. Two general strategies are possible:

[3] an example is presented in Section 4.3.3

- *grouping by sorting* – can be used when the relation given is sorted on the attributes that take part in grouping, or by performing sorting before grouping. Having a sorted relation, it is possible to simply scan it, detecting changes in the proper attributes values and calculating the aggregate result. With this approach, the results for one group can be returned immediately after moving to another one.

- *grouping by hashing* – this strategy is used when the given relation is not sorted, and the cost of sorting is expected to be too high. In such a situation, for each combination of grouping attributes the *hash-value* is calculated. For each distinct value the system keeps in a *hash-table* all the necessary information for aggregate implementation. With this approach, the results can only be returned when the whole relation has been scanned. It is preferred when the expected number of groups is relatively small, making size of the hash-table fit into main memory or even the cache.

In query-intensive applications, *grouping by hashing* is the preferred solution as groupings are usually performed over attributes with relatively small cardinality. In this thesis we concentrate on parallelization of this strategy.

Sequential Version

The Monet DBMS, with its full binary-fragmentation strategy, requires dividing the **aggregate** into three phases: grouping, aggregate evaluation and grouping attribute projection.

For the *grouping*, we need to perform a translation of the source relation IDs into the corresponding group IDs. At first we use the `group()` operator on the first grouping attribute. For the remaining columns we use the `derive()` function to change the group identifiers within previously found groups. As a result we obtain a `BAT[void,oid]` where the head value corresponds to the source tuples identifiers, and the tail is a group identifier (ID of one of the belonging tuples).

For the *aggregate evaluation* we use obtained `grouping` table, and exploit the *pump* MIL construct `{f}(extent, grouping, column)`. The `column` is a `bat[void,any]` storing source attribute values. The `extent` is a `bat[void,oid]` containing new result IDs in a head and unique group identifier in a tail. For each aggregate function it returns a new relation with head from the `extent` and the result of the function for corresponding tuples in the tail.

The final step is the projection of values of grouping attributes, using the `extent` table.

In general, **aggregate** can be mapped into MIL using the following production rules:

```
AGGREGATE( rel, (a1,...,aN), (f1,...,fN) ) =>
    grouping := GROUP( rel, a1,...,aN );
    extent := grouping.tunique.mark.reverse;
    GRP_EVAL(rel, f1,...,fN) );
    GRP_PROJECT( rel, a1, ..., aN) );
GROUP( rel, a1, t)          => DERIVE( rel, rel_a1.group(), t )
DERIVE( rel, g, a, t) )     => DERIVE( rel, g.derive(rel_a), t ))
DERIVE( rel, g, nil ) )     => g
GRP_EVAL( rel, f1(a), t ) ) => result_f1_a := {f1}( extent, grouping, rel_a );
                               GRP_EVAL( rel, t )
GRP_EVAL( rel, nil ) )      =>
GRP_PROJECT (rel, a, t ) )  => result_a := extent.join(rel_a) ;
                               GRP_PROJECT( rel, g, t )
GRP_PROJECT (rel, a ) )     =>
```

Parallel Version

For the parallelization of aggregation, the special features of various aggregate functions have to be taken into account. Depending on the possibility of calculating global value of $F()$ using results calculated for relation R divided into slices R_i, they can be classified into three categories [19]:

distributive – there is a function $G()$ for which $F(R) = G(F(R_i))$. For $F()$ being one of `SUM()`, `MIN()` and `MAX()` we simply use $G = F$. For F=`COUNT()` we can take G=`SUM()`.

algebraic – there is a function $G()$ (possibly multi-valued) and a function $H()$ such that $F(R) = H(G(R_i))$. As an example, for F=`AVG()` we can use $G()$ calculating sum and count for slices, and $H()$ that sums both components, and divides the results for final output

holistic – it is impossible to use constant-size storage for each sub-aggregate to count the total result. An example of such a function is `MEDIAN()`.

In general SQL queries include only `SUM()`, `COUNT()`, `MIN()`, `MAX()` and `AVG()`. For the last one we have to calculate extra sub-aggregates, while for the rest we simply use the G function presented.

The translation of an aggregation into parallel MIL code can be described by the following algorithm:

1. sequential: divide the source relation's size into equal ranges
2. sequential: prepare a global relation `tot_ext[oid,oid]` for partial results of grouping
3. sequential: prepare global relations `tot_aggrX[oid,any]` for partial results of aggregates
4. parallel (each CPU):
   (a) take the next available range boundaries
   (b) perform `slice` on all source relations (grouping attributes as well as aggregates inputs)
   (c) calculate *local* `grouping` and `extent` using strategy from sequential version. For `mark()` in `extent` calculation use lower range boundary to assure that it is unique. Add `extent` to `tot_ext`.
   (d) calculate *local* aggregate values. They are of type `[void,any]` where head is the same as in `extent`. Add them to the proper `tot_aggrX`.
5. sequential: perform `tot_ext` unification by projecting tail to grouping attribute values, and performing sequential grouping. It results in a BAT where the head value is an `oid` present in partial aggregates and the tail is one of the heads that share grouping attribute values.
6. parallel/sequential: for each table with partial aggregates perform sequential aggregation, using global group identifiers. Independent aggregates can be performed in parallel.
7. parallel/sequential: project grouping attributes.

As an example we present a parallel version of the query "`SELECT MIN(a1),AVG(a2) FROM rel GROUP BY g1,g2`", running with `NCPUS` threads using `NSLICES` slices:

```
bounds:=make_bounds(rel_a1.count, NSLICES);   #step 1
tot_ext:=new(oid,oid,NSLICES);                #step 2
tot_min_a1:=new(oid,int,NSLICES);             #step 3
tot_sum_a2:=new(oid,int,NSLICES);
tot_cnt_a2:=new(oid,int,NSLICES);
bounds@[NCPUS]batloop {                       #step 4
  sl_g1:=rel_g1.slice($h,$t);                 #step 4.b
  sl_g2:=rel_g2.slice($h,$t);
  sl_a1:=rel_a1.slice($h,$t);
  sl_a2:=rel_a2.slice($h,$t);
  grp := group(sl_g1);                        #step 4.c
  grp.derive(sl_g2);
  ext := grp.tunique.mark(oid($h)).reverse;
  tot_ext.insert(ext);
  tot_min_a1.insert( {min}(ext,grp,sl_a1) );  #step 4.d
  tot_sum_a2.insert( {sum}(ext,grp,sl_a2) );
  tot_cnt_a2.insert( {count}(ext,grp,sl_a2) );
}
new_grp := tot_ext.join(rel_g1).group;        #step 5
new_grp.derive(tot_ext.join(rel_g2));
new_ext := new_grp.tunique.mark.reverse;
VAR result_min_a1, result_avg_a2,
    result_g1, result_g2;
{|                                            #step 6 & 7
  result_min_a1 := {min}(new_ext, new_grp, tot_min_a1);
  {
    tmp_sum_a2 := {sum}(new_ext, new_grp, tot_sum_a2);
    tmp_cnt_a2 := {sum}(new_ext, new_grp, tot_cnt_a2);
    result_avg_a2 := [/](tmp_sum_a2, tmp_cnt_a2);
  }
  result_g1 := new_ext.join(rel_g1);
  result_g2 := new_ext.join(rel_g2);
|}
```

Comparing to the sequential code, the parallel version faces the problem of unification of both grouping tables and partial aggregate results. With the assumption of small number of groups and large number of tuples, this cost is marginal. However, the problem with big number of groups, mentioned for sequential execution, is still present.

Benchmarks Results

For evaluation of our algorithm, we have used queries performed over the LINEITEM relation from the TPC-H schema with scaling factor equal to 1 (ca. 6M tuples). Figure 3.4 presents different combinations of the grouping attributes used to see the impact of the number of groups.

Figure 3.5 presents the benchmark results. Looking at the absolute time values, it can be seen that with increasing number of groups the execution slows down. When the number of groups exceeds the cache capacity (i.e. tens of thousands of tuples) the performance degrades dramatically. In such a situation *grouping by sorting* would be an alternative.

The other observation is that the grouping parallelizes better than the selection. There are few reasons for that. First of all, there is little additional data copying. Therefore only for large number of groups the speedup factor is worse. The second reason is that in grouping more CPU cycles are consumed for each tuple than in selection. As presented in Chapter 4, algorithms scale better when they have bigger *instruction-per-byte* ratio. Finally, grouping output is usually small, therefore the influence of slow memory writes is lower.

Used LINEITEM attributes	Cardinalities	Groups in total
linestatus	2	2
linestatus,returnflag	2,3	4
discount	11	11
linestatus,linenumber	2,7	14
linestatus,discount	2,11	22
linestatus,returnflag,linenumber	2,3,7	28
quantity	50	50
discount,tax	11,9	99
quantity,linenumber	50,7	350
discount,tax,linenumber	11,9,7	693
quantity,discount,linestatus	50,11,2	1100
quantity,discount,tax	50,11,9	4950
suppkey	10000	10000
suppkey,linestatus	10000,2	20000
suppkey,returnflag	10000,3	30000

Figure 3.4: Attributes of LINEITEM used for aggregation benchmark

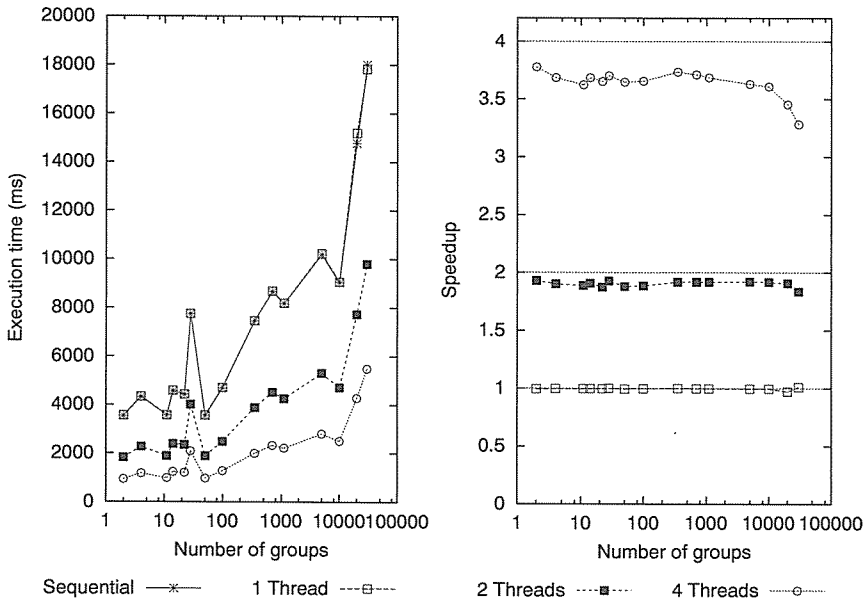

Figure 3.5: Aggregation performance: execution time and speedup on 4-CPU 550MHz Pentium III

3.5. Sorting

In a query represented as a tree of algebraic operators, **sort** is usually used as a root of this tree to order the query result. However, some implementations of other operators use implicit sorting, for example *merge-join* and *grouping by sorting*. In this section we describe a generic solution for sequential and parallel sorting in Monet.

Sequential Version

Sorting in Monet faces similar problem as grouping – usually it is performed over multiple columns. To allow its execution for binary fragmented data, MIL provides two primitives: `sort()` that tail-sorts a given BAT, and `refine()` that performs sub-sorting of a sorted input relation according to the values in another table. They are complementary – `sort()` allows performing single-column sorting, but also is a first step of multi-column sorting, continued by `refine()`.

In general, **sort(attrList,rel)** can be translated into MIL using the following mapping:

```
SORT(rel, a1, t)     => REFINE(rel, rel_a1.sort(), t).mark.reverse;
REFINE(rel, s, a, t) => REFINE(rel, s.refine(rel_a), t )
REFINE(rel, s, nil)  => s
```

The result is a `BAT[void,oid]` with tail values corresponding to the source relations head, sorted according to sorting attribute values.

Parallel Version

For the parallel execution of comparison-based sorting there are usually two strategies:

range-partition and sort First all tuples are divided, according to their value, into multiple non-overlapping ranges. Then each range is sorted locally, and the result is saved in the proper position in the final output (known after partitioning). Since range sizes should be balanced for good parallel execution, finding proper boundary values may be a problem. Since it is equivalent to finding *quantiles* [2], various techniques may be used:

- *sampling* – we take random subset of the input relation and calculate range boundaries. This method gives only probabilistic guarantees.
- some index structures (e.g. balanced trees) allow relatively easy finding quantiles.
- using specialized structures, like histograms or ones described in [30], designed for this purpose.

Since ordering is usually performed as a last stage of a query execution, it can not use any pre-existing data structures. Moreover, finding good range boundaries may be impossible for many datasets (due to *data-skew*). In some cases it can be solved by releasing the constraint about non-overlapping of ranges.

slice, sort and merge In this strategy the data is divided into multiple chunks of the same size (with possibly overlapping values). Then each chunk is sorted locally. In the final step, multiple sorted datasets are merged into a final output. Since merging only works well with limited number of sources, it is sometimes divided into multiple sub-steps.

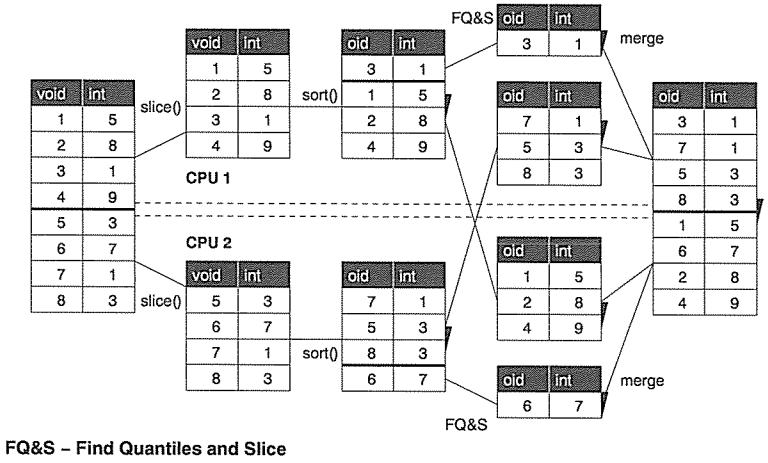

Figure 3.6: Parallel `sort()`

In our solutions for parallelizing sorting in Monet we mix these ideas. We use the merge-solution for the `sort()` operation, since it works fine for any data distribution. For the `refine()` we exploit the fact that the source relation is already sorted to perform fast range-partitioning.

Single-column `sort()` is equivalent to the classical sort operation. The parallel strategy for its execution is presented on Figure 3.6. It is based on the algorithm described in [20], consisting of the following steps:

- create p slices of size $c = \frac{n}{p}$
- let each CPU perform any sequential version of sort on its slice. Monet uses a CPU-optimized version of quick-sort with reasonably cache-friendly access pattern.
- divide all tuples into p approximate ranges using the following strategy. Let each CPU create a sample of its tuples with size s, by choosing each $(k\frac{c}{s})^{th}$ value, for $1 \leq k \leq s$. Now one of the CPUs merges all these samples, and treats result as an *equi-depth histogram* to choose $p-1$ quantiles. The difference between these quantiles and real ones is at most $\frac{n}{s}$ tuples. Even with a small s, e.g. $p * 100$, it guarantees good load balancing.
- each CPU takes values from every slice, that lay between quantiles p and $p+1$ – binary search and slice. Now each CPU has at most p slices, with total size of at most $c + \frac{n}{s}$.
- let each CPU perform p-way merge, saving the output in the known place in the final result.

The operator `refine()` works on two relations `A` and `B`, usually of types `[oid,oid]` and `[oid,any]`. The first one is already sorted on tail which contains non-decreasing identifiers of groups with the same values of already sorted attributes. It produces an `[oid,oid]` relation, where tuples with the same tail in `A` are resorted accordingly to the values in `B`. The result's tail values contains new identifiers of ordered groups.

For parallelization of this operator we propose the following strategy:

- perform range partitioning on the tail of `A`, using binary search (look for non-overlapping ranges).

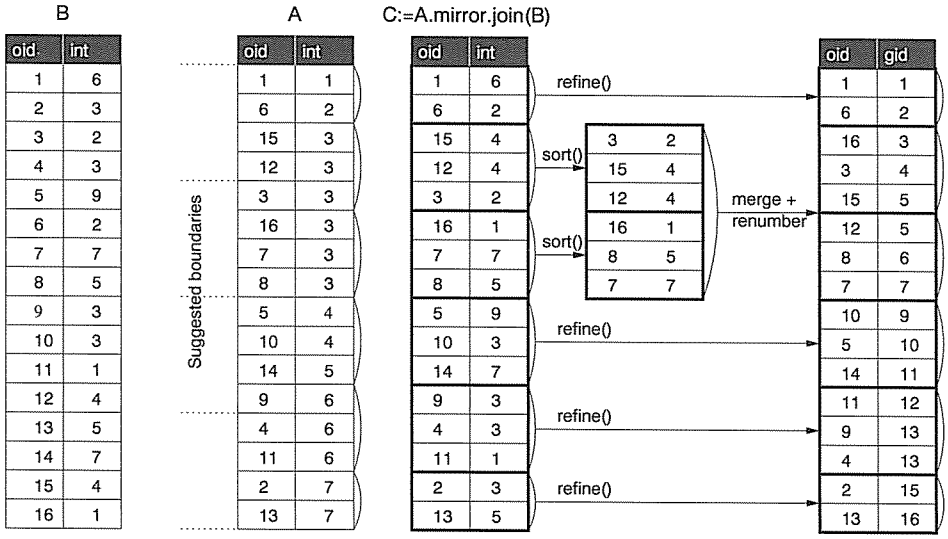

Figure 3.7: Parallel `refine()`

- if the sizes of slices divided according to found boundaries are well balanced, we can simply perform sequential `refine()` on each of them, putting the result in the proper place in the output.

- if the slices are not balanced, it means that there are some border values present in a big number of tuples. For each of such values, we create an additional slice, and perform single-column sorting within it, looking only on values from B's tail (in A they are the same). If the size of the slice is big, the parallel `sort()` presented earlier can be used. For slices with 'problematic' values removed we perform sequential `refine()`.

Figure 3.7 presents this strategy for relations A and B with 16-tuples each, using 4 slices. As we can see in A's tail values, creating balanced slices is impossible, therefore extra ones were created for the tuples containing value 3. Since the total number of them was bigger than the expected size, two slices were created, requiring a merge step (as in `sort()`). For the boundary value 6 an extra slice was also created, but since it was small enough, a simple `refine()` call could be used.

The example shows, that the requested number and size of slices may differ from the ones created. For good load-balancing, the parallel version of `refine()` should create more slices than executing threads, minimizing idle CPU time.

Benchmark Results

Figure 3.8 presents our benchmark results. For the tests we have used again the `LINEITEM` table. Different collections of ordering attributes were used, starting with `QUANTITY`, and then adding `DISCOUNT`, `TAX`, `LINENUMBER` and `SUPPKEY` (one in each pass). Results obtained show that the speedup is far from perfect. There are two reasons for this. The first one is the additional cost of the merging phase, observed by comparing performance of the single-threaded parallel code with traditional sequential one. The other one is again the poor scalability of memory throughput.

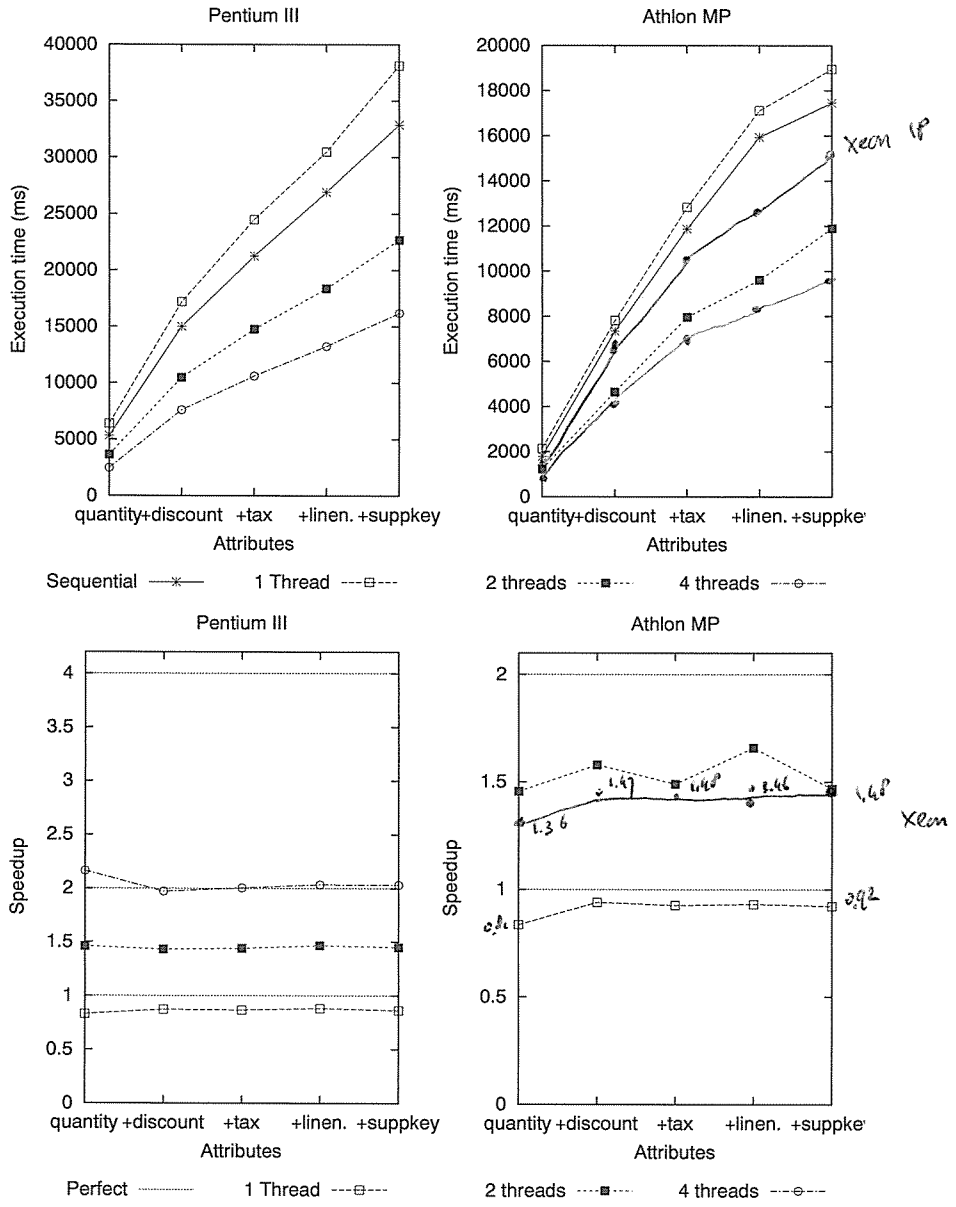

Figure 3.8: Sorting performance: execution time and speedup on 4-CPU 550MHz Pentium III and 2-CPU 1400MHz Athlon MP

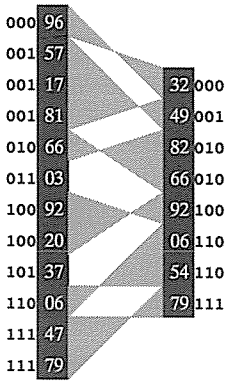

Figure 3.9: Partitioned hash-join algorithm

3.6. Join

Join is the most expensive relational operator. In the general case, its output may be a cartesian product of the input relations, hence its result size may be enormous. Fortunately, most practical joins connect tables with a 1-1 or 1-N relationship. In such a case, the result size is limited by the size of the larger input. Still, for N-M relationships, the result size may be hard to predict.

Because of its complexity, **join** has been extensively researched in the past. Many implementations have been invented, both sequential and parallel [15, 33, 40]. The basic Monet join implementations include *merge-join*, *hash-join* and *theta-join* (using T-tree index). Additionally, for joins where one of the relations is *dense* on the join attribute, *positional-join* may be used [8]. In this thesis, however, we concentrate on the new, cache-conscious join algorithms introduced in Monet [28].

3.6.1. Partitioned Hash-Join

The process of performing **partitioned hash-join** can be divided into two steps. During the first step, both source relations are *partitioned* into H separate *clusters* according to the value of a hash function calculated over the join attribute. In the second, corresponding clusters are joined using any traditional join method – Figure 3.9. This solution proves effective when number of clusters is selected in such a way, that they are small enough to fit into cache memories.

Radix-Cluster

During the clustering phase of partitioned hash-join a following problem may occur. If the number of clusters H is too large, two factors can degrade performance. If H is bigger then number of cache lines, *cache-trashing* can occur. Moreover, a big H can result in *TLB misses*.

The **Radix-Cluster** [27] algorithm solves these problems. It uses an idea known from the *radix-sort* [24, p.168]. For clustering tuples on B bits of hash value into $H = 2^B$ buckets, P sequential passes are performed. Each i-th pass clusters tuples using B_i bits of hash value, dividing already existing groups into $H_i = 2^{B_i}$ new ones. The numbers of bits in each pass give $B = \sum_{i=1}^{P} B_i$, therefore $H = \prod_{i=1}^{P} H_i$. Note that sets of bits used in different passes are disjoint. Moreover, if bits in the consecutive passes are taken from the most-significant to the least-significant, the final clusters are *ordered* on all B bits. The other important aspect is that tuples within each cluster are ordered according to their position in the input.

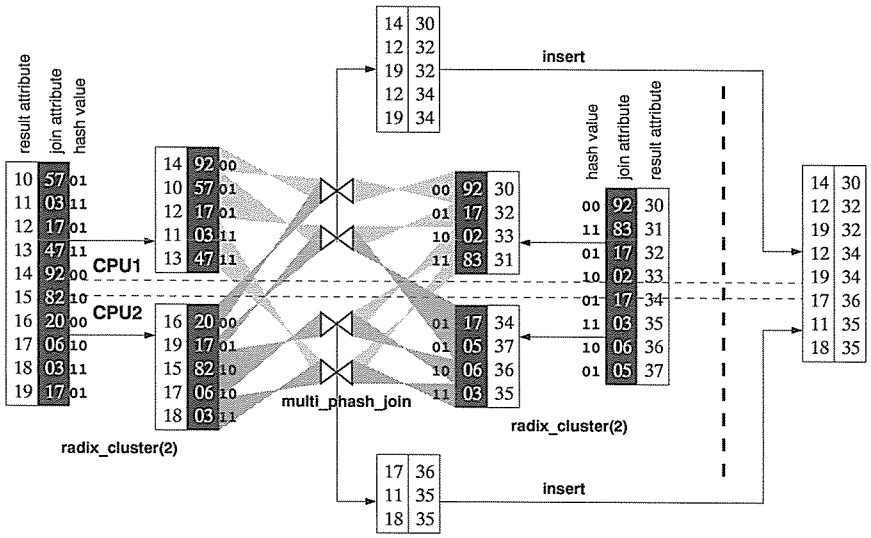

Figure 3.10: Parallel Hash-Join with Radix-Cluster

The strategy presented can use knowledge about cache sizes to adjust the number of bits in each pass, so the number of cache-misses is minimized. Benchmarks [28] show that the cost of extra data scans is compensated by a proper cache exploitation.

In the Monet, an equi-join of two relations using the Radix-Cluster strategy is made with the following rule:

```
JOIN(A,B) => A_clust := a.radix_cluster(BITS);
             B_clust := b.radix_cluster(BITS).reverse;
             result := phash_join(a,b,BITS,hitrate,cutoff);
```

In this rule `BITS` are the numbers of bits in each Radix-Cluster step. The last two parameters are used for optimization: `hitrate` is an estimated number of corresponding tuples in `B` for each tuple from `A`, and `cutoff` is an information that there is at most one corresponding tuple in `B` for each tuple from `A`.

Parallel Version

We concentrate on presenting the parallel version of the partitioned hash-join using the Radix-Cluster strategy.

The first idea for a parallel version of the algorithm was to preserve independence of the clustering and join phases. However, creating a single relation storing the clustered input requires either the specialized implementation of `radix_cluster()` or introducing an extra operator that merges the clustered slices. In both cases it introduces additional overhead.

Figure 3.10 shows an algorithm that solves this problem. Each CPU first performs independent clustering on all inputs. Then, different clusters are assigned to the CPUs, and each of them performs hash-join using `multi_phash_join()`, which is a modified version of `phash_join()` that is able to process data partitioned over multiple sources. The result of the join phase on each CPU is saved in a temporary relation. Next, these relations are copied into the final result. This extra copying phase is necessary for the same reason as with the **select** operator – the size of the output is impossible to predict in advance.

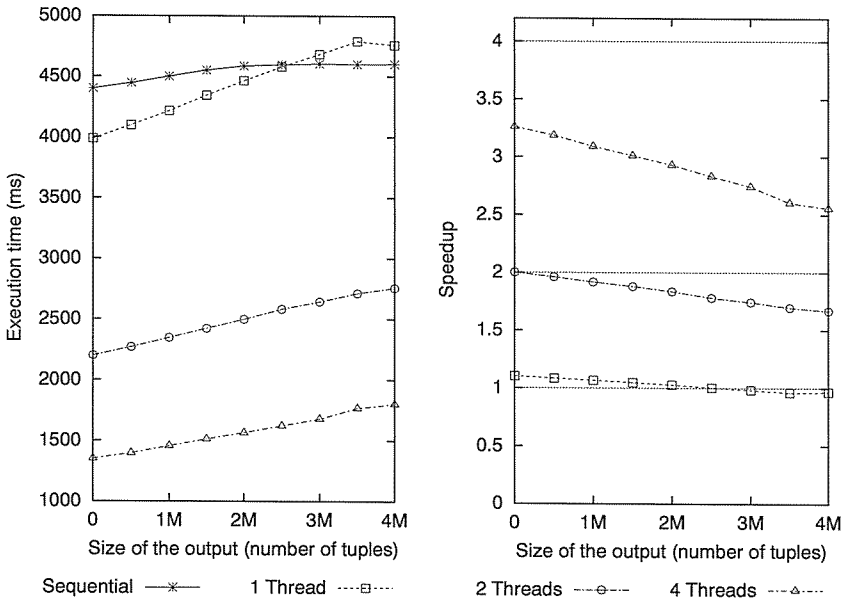

Figure 3.11: Performance and speedup of hash-join with Radix-Cluster on 4-CPU 550MHz Pentium III

Benchmark Results

We performed our tests for join of two relations with 4M tuples with uniform distribution. We used different combinations of the values to influence the size of the output. The results are presented in Figure 3.11

The first diagram presents execution time. The difference between sequential code and the parallel version comes from the optimizations made in `multi_phash_join()`. The second diagram presents speedup with relation to this optimized version. The results are similar to results of the **select** operation (Section 3.3). The speedups are a bit more stable, due to lower impact of the memory writes.

3.7. Attribute Projections

During the execution of an operator tree, for each edge between two operator nodes the *pivot* table is generated. It is a `[void,oid]` BAT storing identifiers of the parent relation in the head and identifiers of the child relation in the tail. Note that the **join** operator creates two such pivots, one for each input relation.

To use attributes that are not directly available at the current execution level, one has to traverse the operator tree downward, until reaching materialized relation. During this traversal, the pivots of visited edges are combined into a *path-pivot* (also `[void,oid]`). Exploiting the `void` column in a pivot head, it can be done using efficient *positional-join*. Having the path-pivot ready, one can use it to fetch attributes from the relational table. Since they are stored in `[void,any]` BATs (Figure 2.8), this can also be done using positional-join.

For example, let us go back to the second tree in Figure 2.3. Assuming that the **select** and **join** operators were executed using translations described in this chapter, to project the `p_city` and `o_value` attributes necessary for the **aggregate** node the following MIL code has to be executed:

```
# ... we have calculated:
```

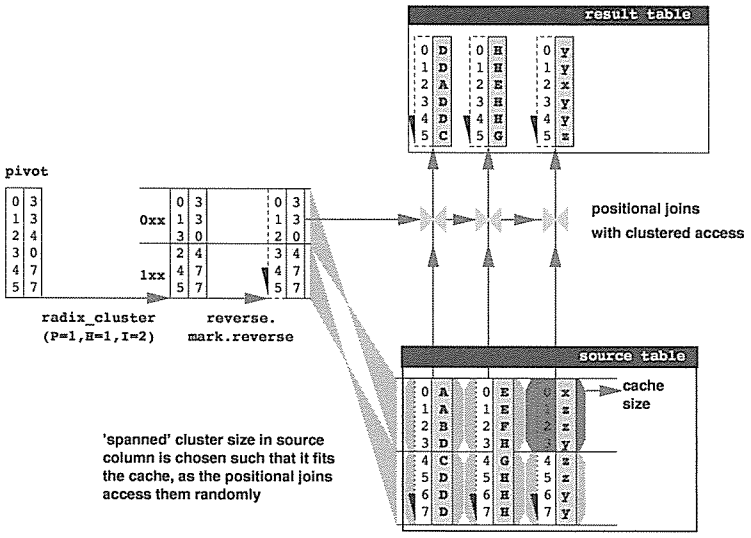

Figure 3.12: Projection using Radix-Cluster (thanks to Peter Boncz)

```
# join_pivotL  [voidJoin,   oidPerson]
# join_pivotR  [voidJoin,   oidSelect]
# select_pivot [voidSelect, oidOrder]

# p_city projection
join_p_city  := join_pivotL.join(person_p_city);   #[voidJoin, p_city]
# o_value projection
path_pivot   := join_pivotR.join(select_pivot);    #[voidJoin, oidOrder]
join_o_value := path_pivot.join(order_o_value);    #[voidJoin, o_value]
```

The only operations performed in this code are the positional-joins, in which for each tuple in the left relation there is *exactly one* tuple in the right relation, hence the result size is equal to the size of the left operand. A parallel version of such algorithm is simple. The left relation is first divided among CPUs and then each CPU performs a positional-join on its chunk of data, putting the result in the proper position (the same as in source) in the final output.

The sequential and parallel executions of this algorithm are characterized by a *random-access* memory pattern. When the size of a relation storing projected attribute exceeds the available cache size, each lookup may generate a cache-miss. In general, for N tuples, we can wait up to $N \times L_{mem}$, where L_{mem} is the main memory access latency.

To improve performance in such situations, Monet introduces a new projection strategy, described in the following subsections.

3.7.1. Projections with Radix-Cluster

Figure 3.12 presents the first idea of making the positional-join more friendly. It uses the **Radix-Cluster** strategy (Section 3.6.1), to cluster the pivot tail before the positional-join phase. With the proper number of clustering bits we can assure that all the projected values for each cluster will fit into the cache. Performing positional-join in such a situation will result in relatively small number of cache-misses. Note that this strategy introduces a new order of tuples, hence it can not be used e.g. for the output of the **sort** operator.

The performance improvement in this strategy comes from the fact, that with the high number of tuples in pivot (relatively to number of tuples in projected attribute relation) we may benefit from cache-reuse. We will present an approximation of when such situation will

occur. Assume that one cache line can store $S = \frac{||L||}{||T||}$ tuples, where $||L||$ and $||T||$ are the sizes of the cache line and the projected attribute value, respectively. We define projection **hit ratio** as an average number of times each tuple in projected relation will be used. If we project `pivot` of size N over attribute relation of the size R this hit ratio is expressed as $\frac{N}{R}$. Described algorithm may not be useful if $\frac{N}{R} < \frac{1}{S}$. In such a situation it is possible that each lookup in positional-join generates a cache-miss, since no cache lines are shared over projected tuples. This would lead to $N \times L_{mem}$ time. Still, for big hit ratios we can expect the cost of this algorithm to be $\frac{R}{S} \times L_{mem}$. Note this cost decreases with increased cache-line size and decreased tuple size.

Parallel Version

The parallel version of the algorithm described is simple. First we perform parallel Radix-Cluster (described in the previous section), then for each cluster obtained we perform sequential positional-join on one of the CPUs.

For implementation of the second phase of this algorithm we used new, more CPU-optimized version of the positional-join, that accepts multiple BATs as an input.

Benchmark Results

Benchmark results are presented in Figure 3.13. The left diagram presents execution time, while the right one shows obtained speedup comparing to a sequential version (empty points) and to a single-threaded parallel version (filled points). We used different numbers of projected attributes, to see what is the influence of both radix-cluster and join steps. The results show, that even on a single CPU this new version is faster thanks to the code optimizations. Moreover, the parallel performance grows with the number of projection columns. This is thanks to efficiency of the projection phase – it is executed for each attribute, while the radix-cluster part is executed only once. However, relative speedup decreases for big number of columns. It shows, that CPU optimized code obtains worse speedup – this trend will be described in more details in Chapter 4.

3.7.2. Radix-Cluster-Decluster

In many situations it is important to preserve the order of tuples during attribute projection. An example is the result of the **sort** operator. In such a situation we can not directly use the Radix-Cluster strategy presented, since it introduces a new data ordering. However, we can exploit the `radix_cluster()` property that tuples inside clusters preserve order from the input.

The **Radix-Cluster-Decluster** algorithm presented in Figure 3.14 works as follows. The source `pivot` table is first radix-clustered on its tail. Two head-aligned relations are derived from the result. The `CLUST_OUTPUTIDS` stores in the tail the position of the projected value in the final output. The `CLUST_INPUTIDS` keeps in the tail the identifiers of tuples in the projected attribute table. Since its tail is radix-clustered we can use the positional projection (as in Section 3.7.1) to obtain `CLUST_VALUES`, with the same `void` head column and projected values in the tail. This relation together with `CLUST_OUTPUTIDS` is an input to **radix_decluster** operator. It uses a strategy similar to the *merge* algorithm – it iterates over clusters from `CLUST_OUTPUTIDS` and from each chooses all tuples with a tail belonging to a current *window* – a continuous area in the result currently ready for writing. For selected tuples the corresponding values from `CLUST_VALUES` are saved at the proper position in the result relation. The window size is adjusted to the size of the cache, so random writes to the memory will minimize number

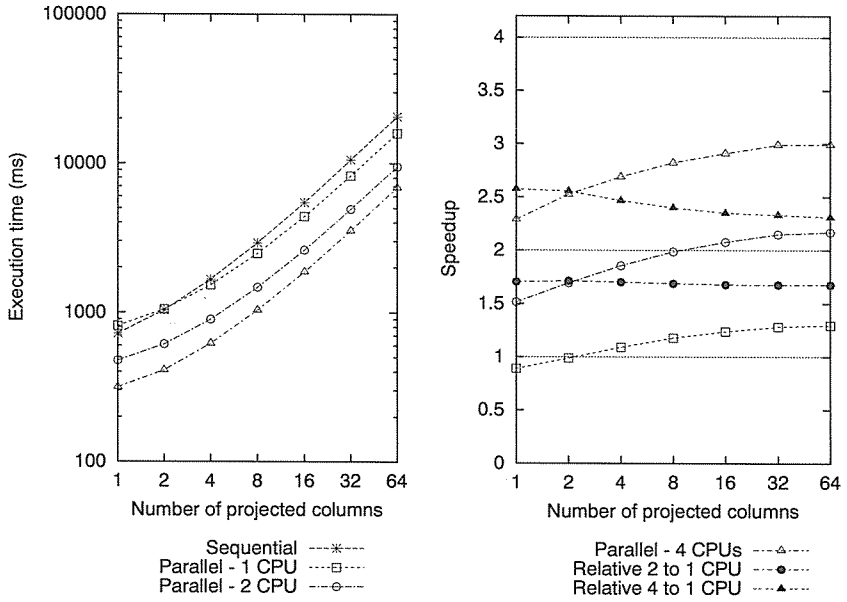

Figure 3.13: Performance of Radix-Cluster projection on 4-CPU 550MHz Pentium III

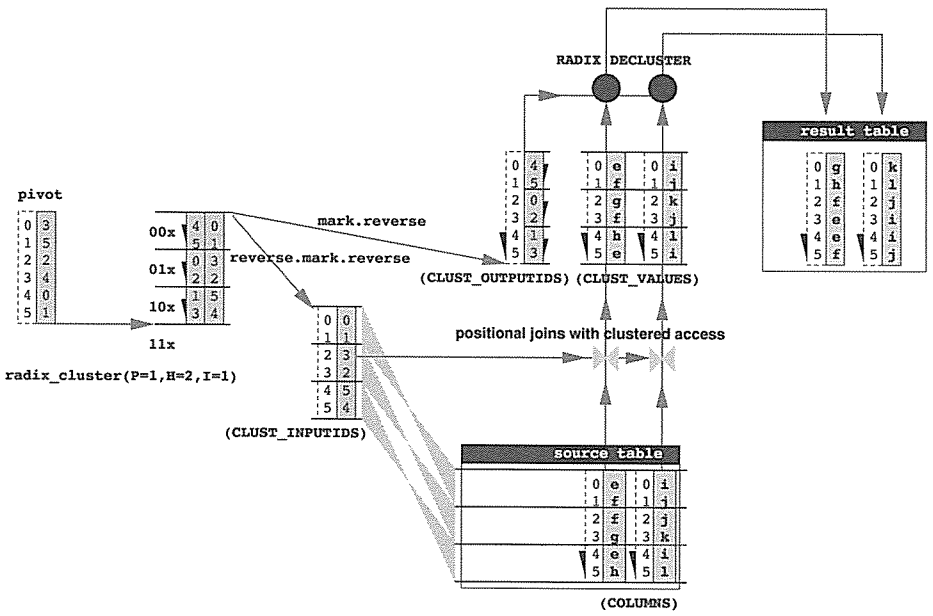

Figure 3.14: Projection using Radix-Cluster-Decluster (thanks to Peter Boncz)

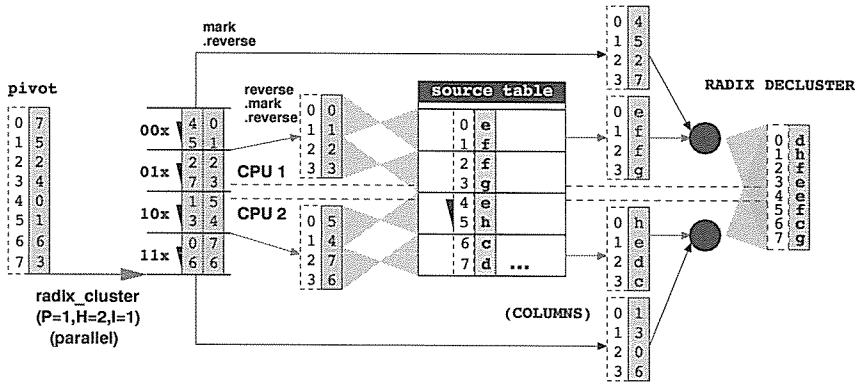

Figure 3.15: Parallel Radix-Cluster-Decluster – Version A: cache coherence problem

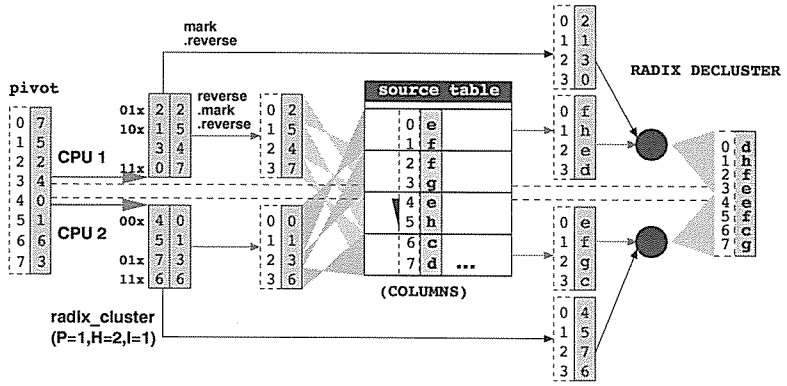

Figure 3.16: Parallel Radix-Cluster-Decluster – Version B: poor cache reuse

of cache-misses. Since we write tuples at positions equivalent to `pivot`'s head column, we know in advance that result relation will be *dense*. Thanks to that, in each scan over all clusters, the entire window is filled, and then it is shifted for the next scan. Assuming that the number of clusters is much lower than number of tuples in the global relation, the execution time of such merging is linear [8].

Parallel Version

The first idea to parallelize Radix-Cluster-Decluster is presented in Figure 3.15. We perform the parallel radix-clustering on the pivot. Then each CPU takes a number of clusters and performs sequential projection and decluster phase. The final **radix_decluster** operators on all CPUs write into a common memory area. This is correct, because each position in the result is written only once. However, multiple CPUs may write into adjacent positions stored in the same cache line. In such a situation the cost of hardware cache-coherency protocol may lead to serious performance degradation.

To avoid the cache-coherency problem, we present another version of the parallel Radix-Cluster-Decluster, explained in Figure 3.16. In this algorithm we perform partitioning of the data *before* radix-cluster phase. Then we proceed with sequential version of the algorithm on each CPU. Since data in each partition comes from the *disjoint* set of the `pivot`'s head, all threads will write into separate memory areas. Hence, the cache-coherence problem will be eliminated.

This solution, however, brings a new problem. As we explained in Section 3.7.1, the Radix-

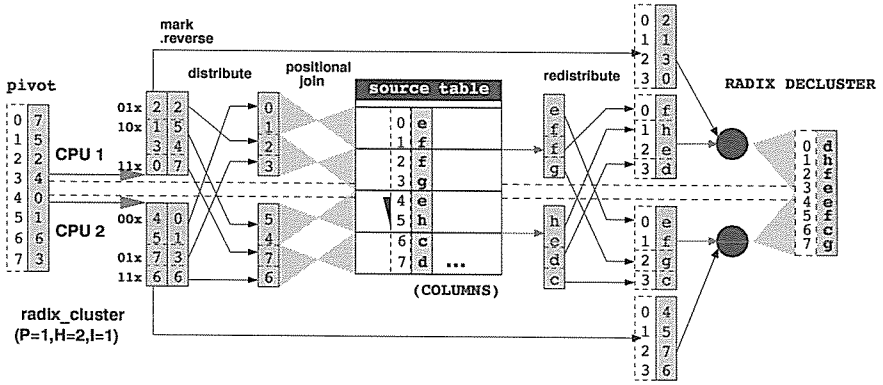

Figure 3.17: Parallel Radix-Cluster-Decluster – Version C

Clustered positional-join is efficient with a high hit ratio in the projected column. With the input of size N fragmented into $NCPUS$ chunks the hit ratio decreases from $\frac{N}{R}$ to $\frac{N}{R*NCPUS}$. Therefore, data sliced for parallel execution results in worse positional-join performance. Additionally, since multiple CPUs may read the same cache line, it may be influenced by cache-coherency protocols as well.

The optimal strategy would be to separate the cache-lines between different CPUs at both positional-join and decluster phase. In such case we would both maximize cache-reuse in the former case, and avoid cache-coherence problems in the latter. For this reason we suggest a new algorithm, presented in Figure 3.17. It can be divided into the following steps:

1. Let each CPU perform Radix-Cluster over part of the `pivot`. We obtain `NCPUS` partial results, each of them consisting of H clusters.

2. Distribute clusters with the same radix-value between CPUs (zero cost operation – only cluster pointers are moved), storing their source position.

3. Perform positional-join for each collection of clusters with the same radix value. Since each cache line in projected attribute is accessed by only one CPU it will result in a good cache re-use.

4. Redistribute clusters to their original position (zero cost operation)

5. Perform radix-cluster for each obtained slice, using the previously saved information from original radix-cluster result. Again, each cache line is used by a single CPU, so no cache-conflicts occur.

For all algorithms described above we have extended appropriate MIL primitives to allow working on data contained in multiple BATs. Moreover we used highly optimized projection and decluster routines to minimize the CPU cost and concentrate on memory-access.

Benchmark Results

We performed benchmarks for all three presented versions of the parallel Radix-Cluster-Decluster. We measured times of projection of 3 attributes, each being a relation of 512K tuples with uniform unique distribution. Note that Radix-Cluster phase in all algorithms is done only once, hence differences in overall results are influenced mostly by the projection and

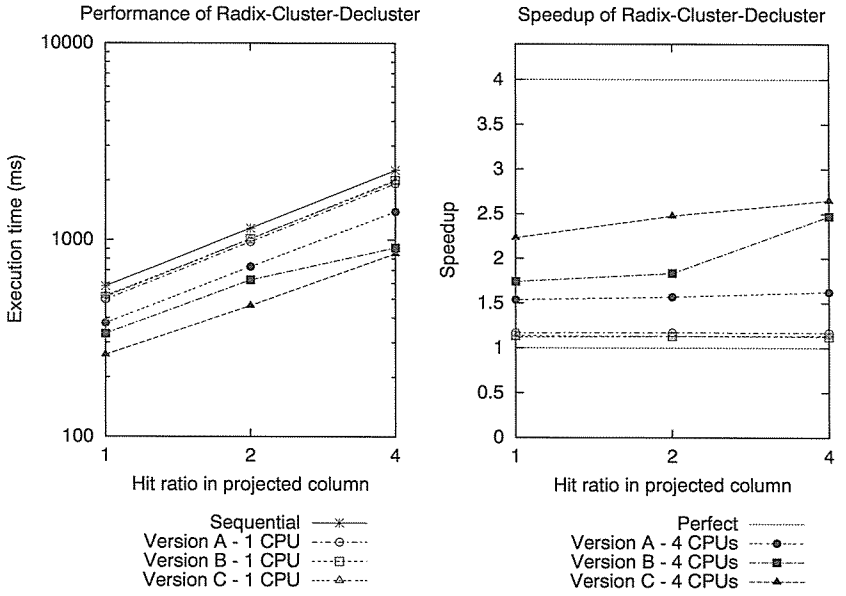

Figure 3.18: Performance of Radix-Cluster-Decluster on 4-CPU 550MHz Pentium III

decluster steps, which are performed for each projected attribute. We performed the tests for various `pivot` relations with different hit ratios.

The results are presented in Figure 3.18. The algorithms behave as expected:

- thanks to highly optimized code all parallel versions run about 10% faster than sequential code on 1 CPU
- Version A of the algorithm obtains poor scalability due to cache conflicts
- Version B shows better performance than Version A. Moreover with higher hit ratio this difference increases further. The reason for that is that the cost of fetching cache-line is amortized by more re-uses of this cache-line.
- Version C is the best in terms of both performance and scalability.

One interesting property of versions A and B is that the first one is a bit faster on a single CPU (thanks to better cache-reuse), while it is much slower with multi-threaded execution (due to cache-coherency problem).

For better analysis of the presented algorithms we performed a detailed benchmark, looking at the performance of various execution phases. We used projection of 2 attributes, with hit ratio equal to 1. The results are presented in Figure 3.19. It contains additional `radix_count()` phase, which is an implementation step of Radix-Cluster, that finds all clusters with the same radix-value in a tail and non-descending heads. The first observation is that Radix-Cluster cost is the same for all algorithms, hence overall time difference comes only from projection and decluster phase. Version A obtains good performance during projection but it is slow during decluster. Version B is good in decluster, but a bit slower during projection phase. Version C is efficient in both phases, and even with more synchronization points between processes it gives the best performance.

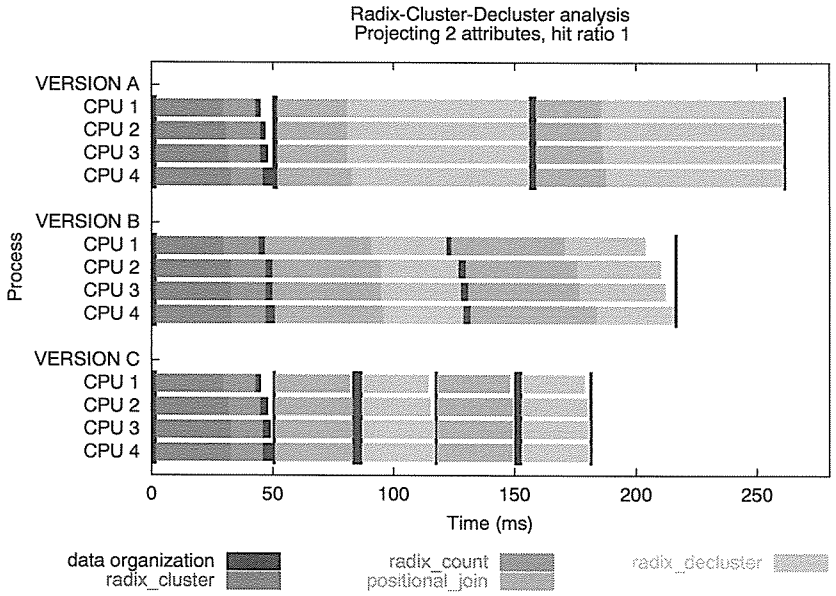

Figure 3.19: Analysis of Radix-Cluster-Decluster performance on 4-CPU 550MHz Pentium III

3.7.3. Projection Strategies for Algebraic Operators

In this section we have presented three strategies for performing projection of data using pivot tables:

- positional-join – effective when the size of the projected attribute does not exceed the cache size or when the pivot tail is ordered. For big relations leads to big number of cache-misses. Preserves pivot ordering.
- Radix-Cluster strategy – thanks to clustering of the pivot tail values it minimizes number of cache-misses, resulting in good performance for big relations. However, it introduces a new pivot ordering.
- Radix-Cluster-Decluster – minimizes number of cache-misses, preserving the pivot ordering. However, due to the declustering phase it is significantly slower than Radix-Cluster.

For different algebraic operators different projection strategies have to be used:

- **expression evaluation** – it does not introduce a new pivot, hence it does not require a projection phase.
- **selection** – since the selection algorithm used in the Monet is a sequential scan, its pivot has tail values ordered according to the position in the source. Thanks to that a positional-join will be effective, because it will not generate any unnecessary cache-misses.
- **aggregation** – aggregation requires projection to return aggregate attributes values. Assuming that presented grouping-by-hashing is used and number of groups is small, a positional-join is efficient.

- **sort** – if the source relation is small enough to fit in the cache positional-join will be efficient. However, when its size is too big, we have to use Radix-Cluster-Decluster, since Radix-Cluster would destroy tuple ordering.
- **join** – if both relations fit in the cache we can use a positional-join for both of them. If one relation is too big, we use the Radix-Cluster strategy for it, while for the other we use positional-join using new pivot ordering introduced by the Radix-Cluster. If both relations exceed cache size we combine Radix-Cluster with the Radix-Cluster-Decluster. We use Radix-Cluster strategy for the relation with projection cost expected higher (bigger size, more attributes), and the Radix-Cluster-Decluster for the other one, preserving new pivot ordering. It can be presented in the following MIL pseudo-code:

```
pivot := join( relA_j, relB_j.reverse );    # bat[oidA, oidB]
r_pivot := radix_cluster(pivot);            # bat[oidA, oidB], tail clustered
pivot_B := r_pivot.reverse.mark.reverse;    # bat[void, oidB], tail clustered
project_radix_cluster(pivot_B, relB);
pivot_A := r_pivot.mark.reverse;            # bat[void, oidA]
radix_cluster_decluster(pivot_A, relA);
```

  Since Radix-Cluster-Decluster preserves head-ordering, the final projections of both relations will share the head `void` column.

3.8. Defragmentation Optimizations

In this section we discuss how to optimize query execution by skipping the data defragmentation phase present in described parallel implementations of the **select** and **join** operators. Our benchmarks show that this step results in a significant overhead. Since it is a simple memory copying, it is memory-bound, making this overhead especially visible during parallel execution.

The necessity of creating an unfragmented pivot comes from the the fact, that the positional-join algorithm works only if data is *densely* stored in a *continuous* memory area, making the value search a simple array lookup. With the fragmented relation, selection of the proper fragment would be necessary before the lookup. This overhead would cost $O(logNSLICES)$ if the slices were created using 'normal' slicing. It can be optimized by slicing the data using a bit-mask of the identifier (most important bits) – then the fragment selection could be performed in constant time. Still, it would decrease the performance significantly. For this reason, Monet does not support fragmented BATs.

Figure 3.20 presents our optimization strategy for the most complex case – projection performed over partitioned attributes basing on partitioned result of the **join** operator. It can be generalized for all other situations. In our algorithm we exploit the fact, that in most cases we replace simple positional-join with Radix-Cluster and Radix-Cluster-Decluster algorithms presented in Section 3.7. In these algorithms the left relation in positional-join is tail-clustered – for each cluster only a part of the right relation will be accessed. The problem occurs if the cluster borders are different than the borders of the fragments of projected attribute relation. In such a case, for the cluster that spans over multiple fragments we have to execute a more complex positional-join algorithm, described above. Assuming that the right relation is fragmented into `NSLICES` fragments, we have to execute such an operation for at most `NSLICES` clusters from the right table. Since the overall number of clusters (usually hundreds or thousands) is much higher than `NSLICES`, the more expensive algorithm will be executed for a small fraction of data, making its cost negligible.

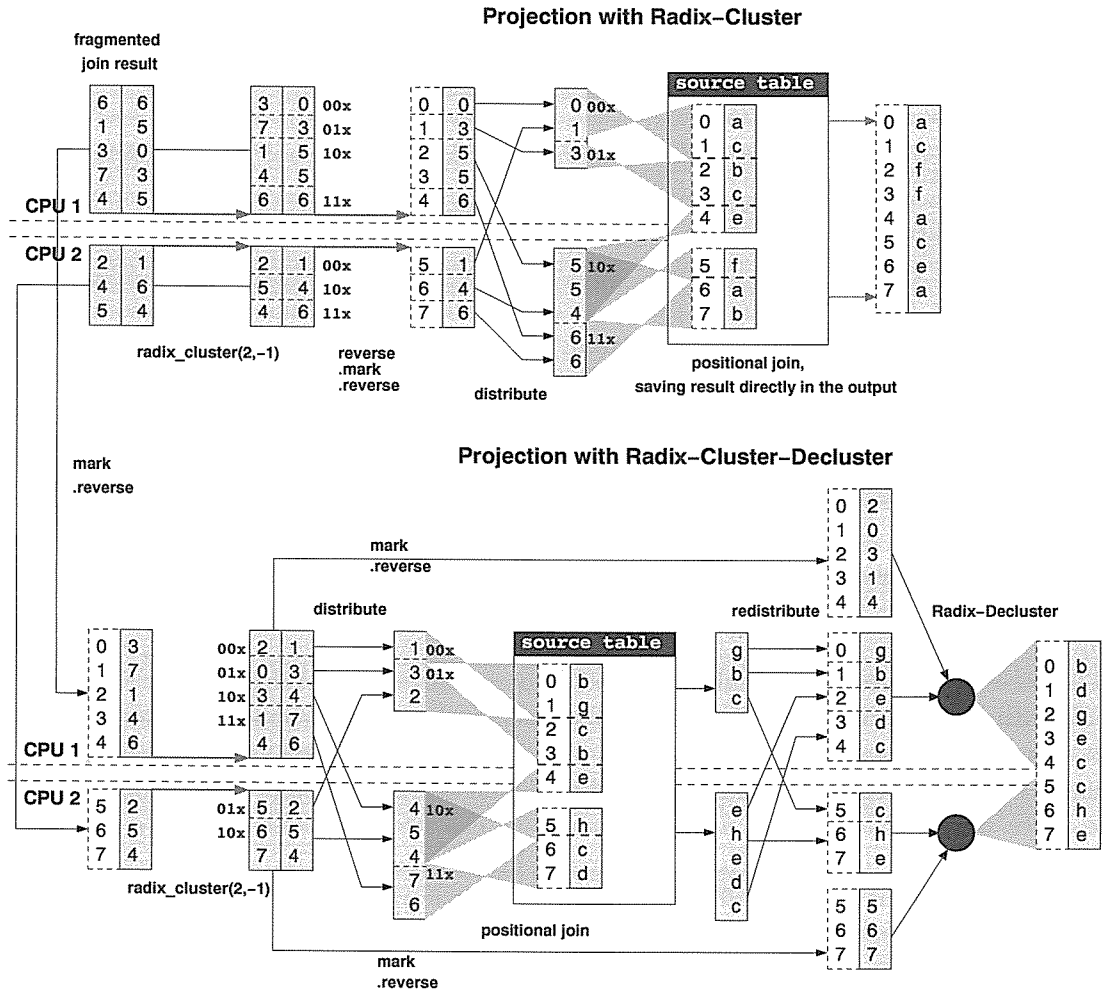

Figure 3.20: Cache-conscious projection of fragmented attribute relations using fragmented join result

During the positional-join in the Radix-Cluster, for each projected cluster we know its position in the result relation. Therefore, we can save the projected value directly into final relation, creating unfragmented result. The strategy for the Radix-Cluster-Decluster step does not differ significantly from the one presented in Figure 3.17. The only difference is special behaviour for some clusters during positional-join phase.

3.9. Summary

In this chapter we have described our work on parallelization of single algebraic operators. As a starting point we took sequential algorithms based on [8]. We have used horizontal parallelism strategy to create parallel versions of these. To allow execution using fragmented relations we have introduced a group of modified MIL primitives able to work on multiple sources. In Section 3.7 we have described the Monet-specific problem of attribute projections and presented three possible algorithms to solve it, providing their parallel versions. We have also discussed a possible strategy for query execution using fragmented operator results, leading to increase in performance.

Benchmarks executed on two hardware platforms show significant performance improvement in SMP environment. However, results are not perfect, mostly due to the memory throughput limitations. This bottleneck will be described in more details in the next chapter.

Chapter 4

Hardware Characteristics

One of the ideas motivating the Monet design is the attempt to adapt to the characteristics of modern hardware. In this section we present a *Calibrator* tool used to obtain detailed specification of the modern machines features. The new contributions are a number of Calibrator extensions developed to measure memory throughput for varying number of CPUs, memory (cache) levels and access patterns.

Using the Calibrator on our reference platforms we analyze limitations imposed by the hardware and discuss their impact on the Monet performance, especially on SMP machines.

4.1. Calibrator

For the purpose of tuning Monet performance to hardware characteristics, a *Calibrator* program is used. It was written by Stefan Manegold in the database research group at CWI. In this section we present a short overview of this tool.

4.1.1. Measurement Methods

The Calibrator runs a collection of micro benchmarks that detect various characteristics in a *generic* and *platform independent* way. For example, to measure the sizes of the cache memories, it runs multiple loops repeatedly accessing the data within the specified memory range. When the range grows, at some point the performance decreases – it means that the algorithm reached the cache-size border. Similar ideas are used for the other cache properties: latency, line size, line number and associativity, as well as for the TLB sizes and miss-latencies. More details can be found in [27].

4.1.2. Results

Figure 4.1 presents the execution of benchmarks checking the cache latencies. It was performed on our test machines: 2-CPU 1400MHz AMD Athlon and 4-CPU 550MHz Intel Pentium III. For the Athlon the cache sizes were detected correctly. However, for the Pentium the L2 cache size in fact is 1024KB – it shows that the results sometimes are not accurate. Still, on most architectures the results are equal to the actual values that may be obtained e.g. using specialized CPU instructions [1].

Looking at the latency times measured in the absolute time, we see that the Athlon has a better performance for the L1 and L2 access and a bit worse for the main memory access. However, if we compare values measured in CPU cycles, we see that the access time to the main memory on Athlon is more than two times longer – the CPU is idle for more than 200 cycles

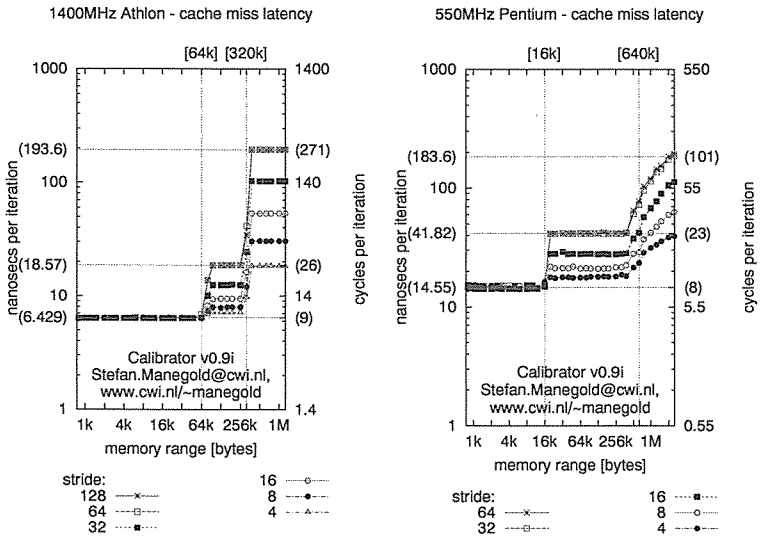

Figure 4.1: Memory latency benchmark in the Calibrator

until the data is fetched. It presents the general trend in the current computer architecture – the memory access time becomes a bottleneck.

This bottleneck greatly influences the design of efficient algorithms. As a simple example, let us compare two basic data structures storing ordered data – sorted arrays and binary trees. In a tree the value lookup, insertion and deletion operations take $O(logn)$, and the data scan $O(n)$. For the array the lookup takes $O(logn)$, for the other operations it is $O(n)$. Before the memory latency became a bottleneck, the trees were often an algorithm of choice. Let us count the number of cache misses that may occur in all situations. In a tree, in the worst case each pointer-jump induces a miss. Therefore, the number of cache misses for the first three operations is $O(logn)$, and for data scan it is $O(n)$. Now let us look at an array, holding data in a continuous memory area. Let S define number of data elements fitting inside one cache-line. The number of cache-misses in an array for the lookup is $O(logn - logS)$, since last $logS$ steps during binary-search will happen inside the same cache line. For the other operations it is $O(\frac{n}{S})$. Additionally, on modern CPUs sequential access does not induce a cache-miss for each cache-line (thanks to memory prefetching), additionally decreasing this value. As we can see, depending on the operations performed on the structure, a simple array may be often a better choice.

4.2. SMP and Memory Throughput

During the work on parallelizing algebraic operators (Chapter 3) the results obtained did not scale perfectly. Preliminary tests related this to limitations of memory subsystems in our SMP platforms, calling for further research in this area. In this section we present more detailed problem analysis.

4.2.1. Extending Calibrator

Our objective was to determine memory throughput in different situations. The reason is that the Monet algebraic operators are highly optimized for the CPU performance, usually resulting in a small instructions-per-byte ratio. Therefore, their access pattern is similar to simple memory accessing functions (e.g. `memcpy()`).

Since we wanted to a create tool that would be portable, extensible and self-adjustable, we decided to use the Calibrator as our base. An additional reason was that Calibrator already delivered information about cache characteristics, useful for us as we want to measure difference in performance at different memory levels.

The Calibrator has been extended with the following features:

- *automatic CPU number detection*: We detect the number of available CPUs in a generic way. For this purpose we run a simple loop with growing number of threads (we only detect powers of 2). When the execution time slows more then the specified threshold, we expect to have crossed the boundary of available processors. The *pthread* library (a POSIX standard) is used for threaded execution.

- *memory throughput calculation*: We created an additional Calibrator module allowing measuring memory throughput with respect to different factors: number of CPUs, memory sizes, access patterns. The module runs specified benchmarking functions (e.g. *read, copy*) on all combinations of memory levels (provided by Calibrator) and number of CPUs (obtained with first extension). The results in the following section are created automatically by this module

- *incorporating into Monet DBMS*: The Calibrator was designed as a standalone program. For our purposes, we have decided to incorporate it into the Monet as a module. It extends MIL language with three functions:

  - `calib_init()` – runs Calibrator tests with given CPU clock frequency and maximum memory range
  - `calib_info()` – gives results of Calibrator tests as a BAT, containing name of parameter in the *head* and corresponding value in the *tail*
  - `calib_free()` – frees Calibrator allocated resources

During the implementation of the Calibrator extensions we discovered that the memory throughput can greatly depend on the *access pattern*. Although we concentrate on *sequential* access, still, the results can vary for different applications. There are three important factors, influencing overall performance[1]:

- The types of operations performed in each iteration – reading and/or writing data, comparisons, "expensive operations" (e.g. function calls, computations)

- *loop unrolling* ratio – in "normal" loop execution the cost of the loop iteration may be significant, especially with a short loop body. Therefore it is often possible to use *loop unrolling* – multiple iterations packed into one. It is presented in Figure 4.2b. Loop unrolling can be obtained manually, or by using optimization option available in many compilers.

- number of cursors – modern CPUs can execute multiple memory load instructions in parallel. This can be exploited by application using multiple *cursors* – abstract *pointers* to traversed data. It is presented in Figure 4.2c.

Additionally architecture-dependent extensions are possible. They include *hardware-prefetching* instructions and single-instruction multiple-data opcodes found in most of modern CPUs. However, such solution is not portable, therefore we decided not to research it in this thesis.

[1] note that the impact of these factors depends on the hardware platform used

(a) normal loop	(b) unrolled loop	(c) multi-cursor

int i;
for(i=0; i<N; i++) {
  dst[i]=src[i];
}
	int i;
	for(i=0; i<N; i+=2) {
	  dst[i]=src[i];
	  dst[i+1]=src[i+1];
	}
		int i, N2=N/2;
		for(i=0; i<N2; i++) {
		  dst[i]=src[i];
		  dst[i+N2]=src[i+N2];
		}

Figure 4.2: Different access patterns for *copy* function

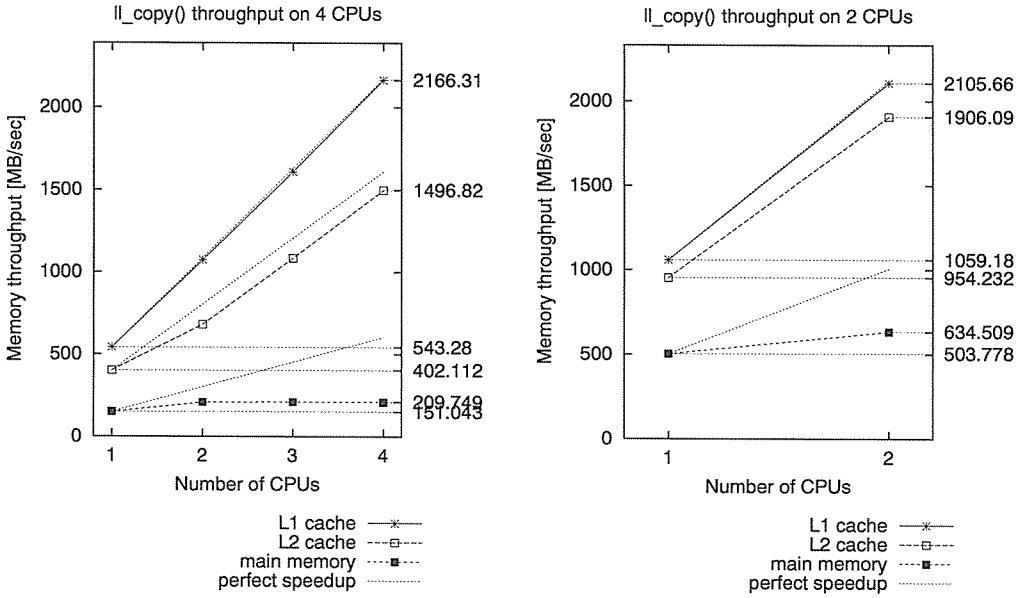

Figure 4.3: Performance of **ll_copy()** on 4-CPU 550MHz Pentium III and 2-CPU 1400MHz Athlon MP

Basing on described aspects, memory throughput module provides a collection of benchmarking functions. The property of the function is specified by its name, following this pattern:

```
<t>_<func>[_lX][_cX]
```

Where:

- `t` is a type of data used, one of: `l-long` (4-bytes), `d-double` or `ll-long long` (both 8-bytes)
- `func` specifies access pattern, possible values are: `copy`, `read`, `write` or `triad` (summing two arrays into the third).
- `_lX` is an option specifying the ratio of loop unrolling
- `_cX` is an option specifying number of cursors used

4.2.2. Results

We performed extensive tests on our two test platforms described in Section 3.1. The results show important limitations of both older (Pentium III) and modern (Athlon MP) CPUs.

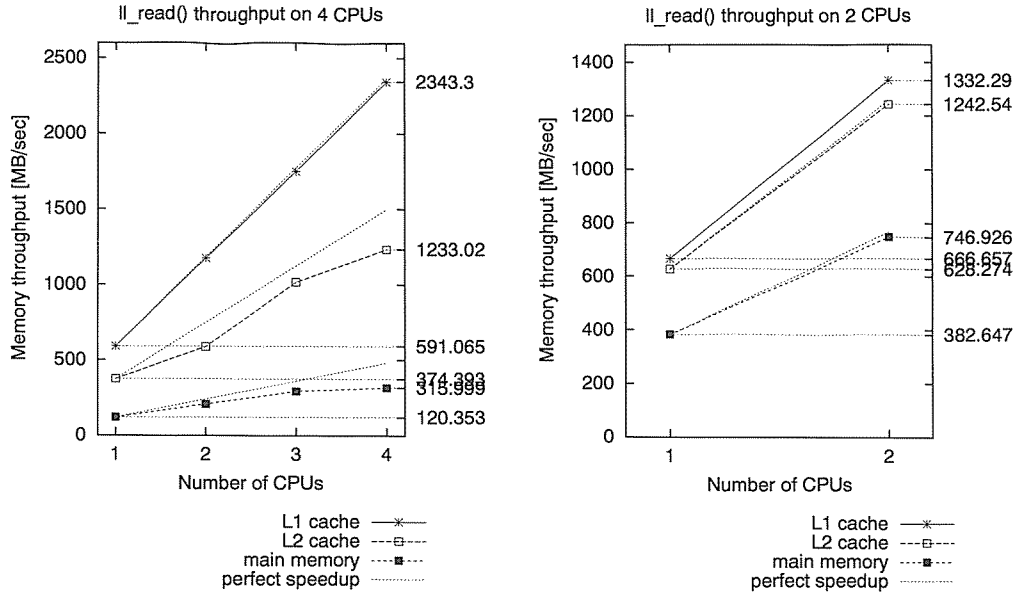

Figure 4.4: Performance of **ll_read()** on 4-CPU 550MHz Pentium III and 2-CPU 1400MHz Athlon MP

Figure 4.3. shows the performance of the **ll_copy()** function on both of our test platforms, for different combinations of the number of CPUs and memory levels. We compare the results with the 'perfect' performance, in which the parallel execution obtains linear improvement. Two important conclusions can be drawn:

1. the performance is different for various cache levels
2. the scalability (with respect to number of CPUs) is different for various cache levels.

We will explain those results in a more detailed way.

The memory throughput difference at various levels does not come as a surprise. It is directly related to the CPU architecture. L1 and L2 are made of much faster **SRAM** memory and are connected with CPU using an efficient bus. Moreover, the fact that L1 is integrated with the CPU core additionally improves its throughput. These results clearly shows, that computation on current computers is often memory-bound – increasing the cache size can dramatically improve overall performance. However, due to high costs of the SRAM chips, cache sizes stay relatively small with respect to the main memory size.

The scalability problems of the main memory come from the fact, that all CPUs share the same memory subsystem. Therefore, although memory throughput is acceptable for 1 CPU, in the SMP environment it leads to a per-CPU performance degradation. For example, on a Pentium machine, there is virtually no difference in the main memory performance using 2 and 4 CPUs. As for the L1 and L2, each CPU has them integrated, therefore their throughput is sufficient. However, private caches may lead to cache-coherence problems.

For the preliminary experiments we have used **ll_copy()** function, which is influenced by both memory reading and writing performance. To obtain more detailed information, we conducted similar tests for **ll_read()** and **ll_write()** functions. The results, presented in Figure 4.4 and Figure 4.5, respectively, show other interesting properties:

- unequal speedup between reading and writing – on both machines **ll_read()** scales much better. For the Pentium, the write throughput stays virtually constant.

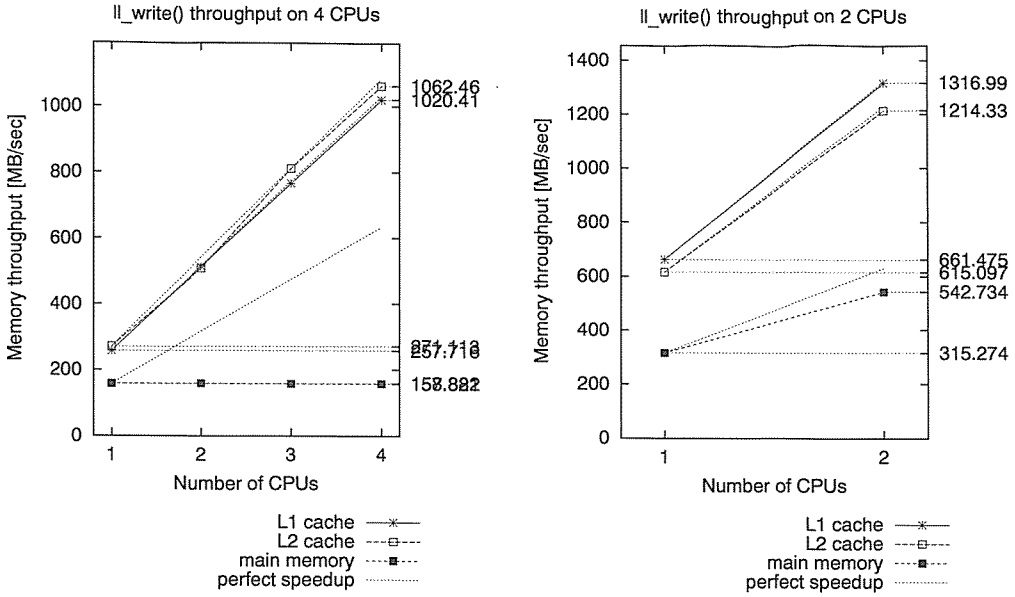

Figure 4.5: Performance of **ll_write()** on 4-CPU 550MHz Pentium III and 2-CPU 1400MHz Athlon MP

- better cache performance scalability on Athlon systems – they are better balanced thanks to differences in architecture: cache-coherence protocol (Athlon's MOESI vs Pentium's MESI), another bus protocol (Alpha EV6 vs GTL), smaller number of CPUs (4 vs 2) and the use of automatic hardware prefetching in Athlon.
- looking at **ll_copy()** performance with relation to **ll_read()** and **ll_write** results, its overall speedup is worse. The reason is the lower loop cost of **ll_copy** – in each iteration there are two memory transfers instead of one. This leads to better performance on a single CPU (where computation is more CPU-bound) and worse speedup (for multi-CPU computations are more memory-bound).

Described performance of current memory chips gives an explanation of the non-optimal results of our algorithms described in Chapter 3.

4.3. Optimization Strategies

The results presented in this chapter lead to an important conclusion. Hierarchical memory system and performance imbalance in various architectural features make creating efficient code hard. For SMP machines it seems to be even more difficult, due to the observed throughput limitations. In this section we provide possible solutions for this problem.

4.3.1. Cache-conscious Algorithms

With the introduction of hierarchical memory systems, the family of *cache-conscious algorithms* becomes more important. They usually concentrate on minimizing the number of *cache-misses*. In general, it can be obtained by using two strategies:

- improving data locality – with the introduction of cache-conscious data structures [11], even traditional algorithms can obtain performance improvement. An example for this

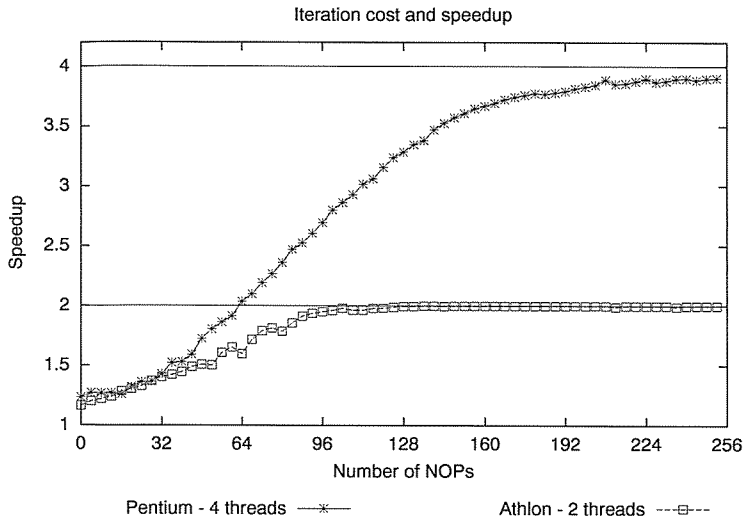

Figure 4.6: The relation between computation cost and speedup on 4-CPU 550MHz Pentium III and 2-CPU 1400MHz Athlon MP

may be a binary tree, where subtrees are *clustered*, i.e. some of connected nodes are located in the same cache line. Such a tree can be traversed using traditional algorithms – the only difference is memory allocation for nodes.

- algorithms with cache-conscious access pattern – in many cases combining both special data structure and reorganizing algorithm's behaviour can lead to performance improvement [27, 20].

4.3.2. Instructions-Per-Byte Ratio

In the Monet CPU-optimized code, the problem of limited throughput on SMP machines comes from the low IPB (*instructions-per-byte*) ratio. Figure 4.6 presents simulated behaviour of functions with the same memory access pattern (simple data copying), but with changing loop cost for each byte. This cost was simulated by inserting extra 'NOP' (no-operation) instructions in each iteration. Results show that speedup increases with bigger IPB ratio. Note that 'NOP' is efficiently processed by the CPU (exploiting inherent parallelism), hence for the 'real' operations the number of instructions needed for good speedup will be lower.

This feature can be exploited in some algorithms. For example, let us look at two programs presented on Figure 4.7, describing two possible approaches of doing some arithmetic expression on all tuples of some relation. The first one simulates Monet behaviour, with its full-materialization strategy. It splits a simple `dst:=2*src+7` statement into two independent `dst2:=2*src` and `dst:=dst2+7` steps. The second program performs the computation directly.

The second version reduces number of memory accesses and additionally spends more CPU cycles per each access. Looking at the results from Figure 4.6 it will not only perform better at single CPU, but also scale better on SMP machines. Of course, for this example the difference will be small, yet for complex operations it may be significant.

Automatic creation of the code similar to the one in Figure 4.7b is not yet possible in Monet. However, one may consider extending Monet with a module translating relational algebra expression into a C code, generating optimal code on-the-fly.

Results presented in Figure 4.6 lead to another question: Is it possible to obtain the same data throughput using more complicated calculations? For SMP machines the answer is often

(a) Monet behaviour	(b) optimized behaviour

int i;
for(i=0; i<N; i++) {
  tmp[i]=2*src[i];
}
for(i=0; i<N; i++) {
  dst[i]=tmp[i]+7;
}
	int i;
	for(i=0; i<N; i+=2) {
	  dst[i]=2*src[i]+7;
	}

Figure 4.7: Difference between Monet behaviour (a) and optimal behaviour (b) for the `dst := [+]( ( [*](src,2) ) , 7);` MIL statement

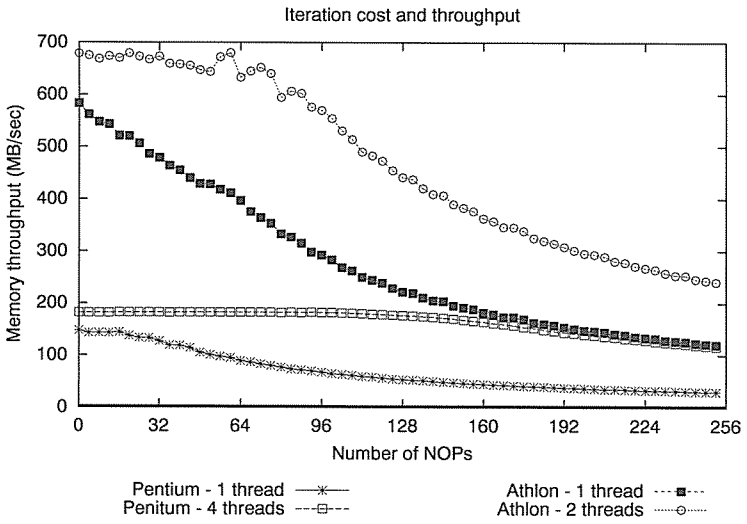

Figure 4.8: The relation between computation cost and memory throughput on 4-CPU 550MHz Pentium III and 2-CPU 1400MHz Athlon MP

"yes". Figure 4.8 presents memory throughput calculated for changing loop cost. The results show that with increasing loop cost, the performance on multiple CPUs stays constant up to some point. It brings the following conclusions:

- on SMP machines code-optimization does not play as important role as in single-CPU system, since execution is memory bound
- performing additional computation in SMP environment does not have to influence overall performance

4.3.3. Partitioned Execution

When performing multiple operations over the same data, there is another possibility of exploiting good cache performance. We divide the data into smaller chunks, and perform all the operations on them. If the chunk is small enough to fit into cache memory, it stays there to be directly reused by the following operators without causing main-memory traffic. Such an approach can be applied on the MIL level. For the problem from Figure 4.7 it would create a code similar to:

```
VAR cnt       := src.count;
```

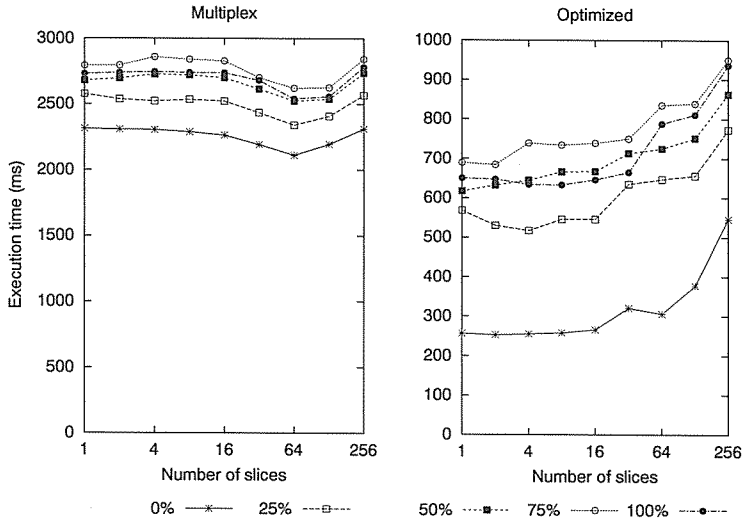

Figure 4.9: Impact of number of slices on SELECT execution on 1 CPU

```
VAR dst_bats  := new(int,BAT);
VAR pos       := 0;
while(cnt>0) {
  size      := min(cnt,CACHE_SIZE);
  src_slice := src.slice(pos,pos+cnt-1); #0-cost operation
  dst2_slice:= [*](src_slice,2);
  dst_slice := [+](dst2_slice,7);
  dst_bats.insert(i,dst_slice);
  pos       := pos + size;
};
```

It will create a BAT consisting of partial results of the computation. With the appropriate operators, it can be later used without additional cost. Since this algorithm for the [+] operator reuses data stored in a cache memory, its performance will greatly increase. Additionally, it creates access pattern that is better scalable on SMP machines.

We applied this strategy to the **select** operator tests used in Section 3.3. Figure 4.9 presents the benchmark results for the code executed on 1 CPU. For the multiplex version we may obtain a performance improvement. The version using optimized range-select operator does not reuse any memory, therefore we only see the overhead on the slices processing. Implementing partitioned execution on the C level (currently it is done in MIL) would minimize this overhead and improve the performance of both versions.

To see the impact of the predicate complexity on the results we have performed additional tests. Figure 4.10 presents the results of executing a a query with 4 predicates similar to previous version. For the multiplex version the results do not change too much. However, for the optimized version we see a good performance improvement. The reason for that is the use of multiple predicates – during combining the outputs the cache is reused. Since range select needs much less CPU work than multiplex version, it is more memory-bound, and obtains much better improvement.

Memory benchmarks described in this section suggest not only better cache performance, but also its better scalability. Therefore we performed additional tests to measure the speedup of the strategy described. Figure 4.11 shows the difference in the speedup of the traditional algorithm (continuous lines) and the partitioned execution (points), using number of slices giving the best results. We see, that the latter obtains better performance, especially for the

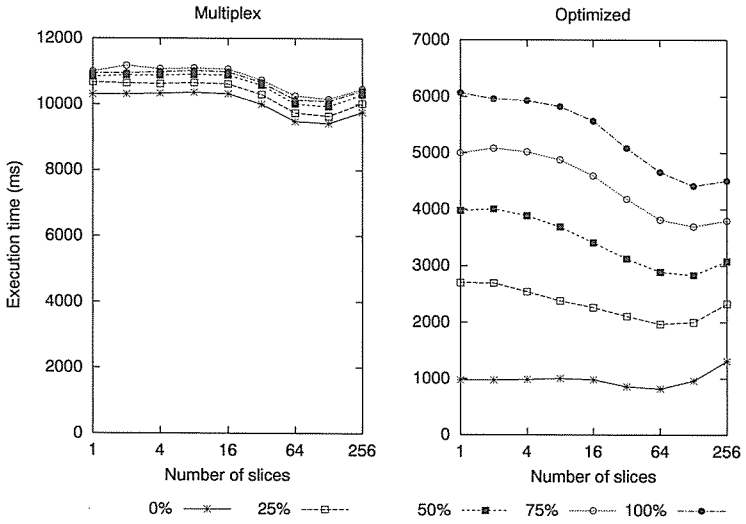

Figure 4.10: Impact of number of slices on multi-predicate SELECT execution on 1 CPU

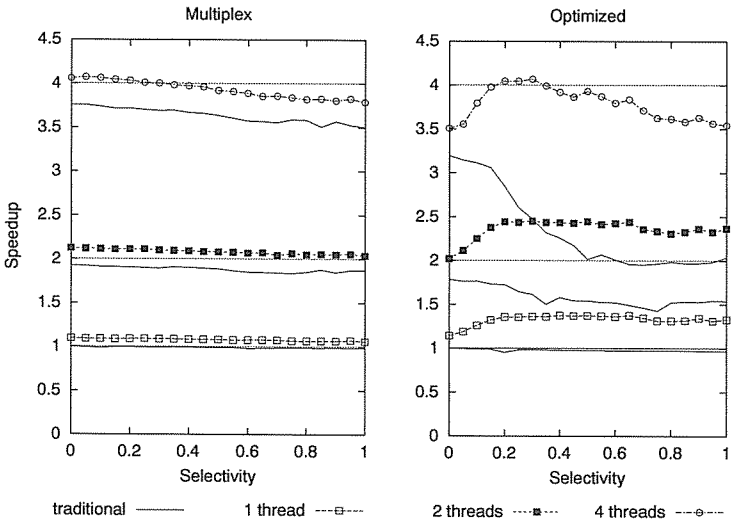

Figure 4.11: Speedup difference between traditional and partitioned multi-predicate SELECT execution

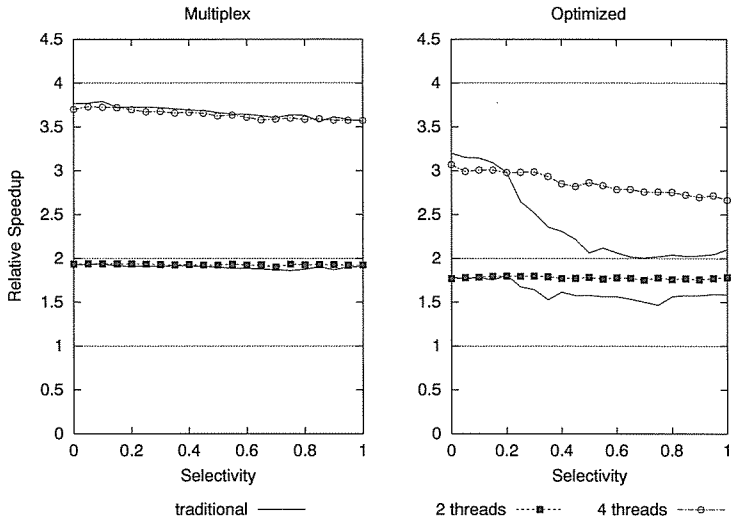

Figure 4.12: Relative speedup difference between traditional and partitioned multi-predicate SELECT execution

optimized range-select version. Figure 4.12 additionally presents a relative speedup of 2 and 4 CPU version comparing to the single-threaded execution. Again, we see that especially for optimized code the improvement is significant.

The results obtained show a big potential of the presented strategy. In the Monet it additionally has one important benefit. The partial results of the operators are proportional to the size of the data. Therefore, using many slices, at the same time less memory has to be allocated.

4.4. Summary

In this chapter we have investigated detailed hardware features influencing our research. In Section 4.1, we started with the description of a Calibrator tool, used for obtaining detailed characteristics of various parts of the hierarchical memory system. We presented results for our test machines and discussed their influence on the programs performance. Section 4.2 presents our extension to the Calibrator measuring the memory throughput at different hierarchy levels, showing the limitations of the current memory chips, especially in SMP systems. This bad memory performance gives an explanation for non-optimal results of benchmarks in Chapter 3. Finally, Section 4.3 discusses possible strategies of improving algorithms performance by adapting to the described hardware features.

Chapter 5

Conclusions

This thesis presents various aspects of parallel query execution in the Monet DBMS. There are three levels of parallelism in DBMS: inter-query, intra-query and intra-operator parallelism. We concentrated on the last option, since the first two were already available in the Monet thanks to its multi-threaded execution and a *MIL Squeezer* tool.

To allow intra-operator parallelism we have described different operators in the relational algebra and presented parallel execution algorithms for each of them. Moreover, we have discussed the attribute projection – an extra step in query-execution in the Monet, necessary due to its decomposed storage model. We have performed performance benchmarks on two SMP platforms: Pentium III and Athlon MP.

The results of the performed benchmarks were often below our expectations. It inspired us to perform detailed analysis into the hardware features of our SMP platforms. We have presented a *Calibrator* tool with new extensions that generically obtains various features of the hierarchical memory system present in modern computer architecture. We have conducted extensive experiments showing bad scalability of memory throughput in our test platforms. We presented various optimization techniques that take this bottleneck into account and allow increasing application performance.

Contributions

The most important contributions of this thesis include:

- *parallel algorithms for algebraic operator execution using the decomposed storage model on shared-everything architecture*
- *efficient parallel versions of cache-conscious Radix-Algorithms*
- *Calibrator extensions allowing detecting various characteristics of the SMP platforms*
- *algorithm optimization strategies for the SMP environment*

Future Work

We conducted experiments on the performance of single algebraic operators. Integration of the presented solutions into the SQL front-end requires more work. Moreover, inter-operator dependencies analysis may result in additional optimizations.

Intra-node parallelism is only a first step towards a fully-parallel DBMS. Execution in the shared-nothing architecture requires more investigation. An interesting area of research would

be hybrid and hierarchical parallel systems, especially with respect to combining cache-efficient execution on SMP machines with high-performance network communication.

Acknowledgments

This thesis was made possible thanks to the joined master's program of my home university in Warsaw and the Vrije Universiteit in Amsterdam. I would like to thank my supervisors at both universities – Henri Bal and Jerzy Tyszkiewicz, as well as people organizing this project – Femke van Raamsdonk and Piotr Rybka.

My adventure at CWI started thanks to Martin Kersten. He offered me a position in his research cluster and was always a source of good ideas and advices, both in research and "real"-life.

My "guardian angel" at CWI was Peter Boncz – although he was busy with his PhD thesis, he taught me a lot about Monet and helped me in improving my writing style. His hacking skills were often an inspiration for me. I would like to thank other people at CWI, especially Stefan Manegold and Niels Nes, for help and friendly working environment.

During my stay in Amsterdam I spent a lot of time with my polish friends: Michał and Gosia. Thanks for a lot of adventures and good luck in your research career!

I would like to thank my family. My parents were my first teachers, showing me the importance of knowledge. My sisters and their love always helped me in hard times.

Finally, thanks to Agnieszka for encouragement, help and patience during this difficult year.

Bibliography

[1] Advanced Micro Devices. *AMD Processor Recognition - Application Note*, January 2002. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/20734.pdf`.

[2] Rakesh Agrawal and Arun Swami. A One-Pass Space-Efficient Algorithm for Finding Quantiles. In *Proc of the Int'l. Conference on Management of Data*, December 1995.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs On A Modern Processor: Where Does Time Go? 1999.

[4] Gene M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, April 1967.

[5] American National Standard for Information Systems. Database language SQL. ANSI X3.135-1992, November 1992.

[6] Peter M.G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA/DB: A Parallel Main Memory Relational DBMS. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):541–554, December 1992.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, May 1995.

[8] Peter A. Boncz. *Monet. A Next-Generation DBMS Kernel For Query Intensive Applications.* PhD Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[9] Peter A. Boncz and Martin L. Kersten. Monet. An Impressionist Sketch of an Advanced Database Systems. 1994.

[10] Peter A. Boncz and Martin L. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2), October 1999.

[11] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[12] Edgar F. Codd. Relational database: a practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, 1982.

[13] George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 268–279, May 1985.

[14] David J. DeWitt. DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Trans. on Computers*, 28(6):395, June 1979.

[15] David J. DeWitt. Multiprocessor Hash-Based Join Algorithms. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, 1985.

[16] Xing Du, Xiaodong Zhang, Yingfei Dong, and Lin Zhang. Architectural Effects of Symmetric Multiprocessors on TPC-C Commercial Workload. *Journal of Parallel and Distributed Computing*, 61:609–640, 2001.

[17] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computing*, C(21):948–960, September 1972.

[18] Ian T. Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Mateo, CA, USA, 1998.

[19] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 152–159, New Orleans, LS, USA, February 1996.

[20] David R. Helman and Joseph JáJá. Sorting on Clusters of SMPs. August 1997.

[21] Jonas S. Karlsson and Martin L. Kersten. Scalable Storage for a DBMS using Transparent Distribution. Technical Report INS-R9710, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, December 1997.

[22] Kimberly Keeton, David A. Patterson, Yong Q. He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proc. of the Int'l Symp. on Computer Architecture*, Barcelona, Spain, July 1998.

[23] Martin L. Kersten, Frans H. Schippers, Carel A. van den Berg, and Peter A. Boncz. Mx documentation tool. 1996.

[24] Donald E. Knuth. *The Art of Computer Programming. Volume 3 - Sorting and Searching.* Addison-Wesley, Reading, MA, USA, 2 edition, 1997.

[25] Jack L. Lo, Luiz Andr Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proc. of the Int'l Symp. on Computer Architecture*, pages 39–50, July 1998.

[26] H. Lu, B. Ooi, and K. Tan. *Query Processing in Parallel Relational Database Systems.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[27] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9(3):231–246, December 2000.

[28] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Trans. on Knowledge and Data Eng.*, 14(3), 2002.

[29] Stefan Manegold, Arjen Pellenkoft, and Martin L. Kersten. A Multi-Query Optimizer for Monet. In *Proc. of the British National Conference on Databases*, Exeter, United Kingdom, July 2000.

[30] Gurmeet S. Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate Medians and other Quantiles in One Pass and with Limited Memory. 1999.

[31] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[32] M.Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, NJ, USA, 2 edition, 1999.

[33] Donovan A. Schneider and David J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, August 1990.

[34] S. Shekhar, S. Ravada, V. Kumar, and D. Chubb. Load-Balancing in High Performance GIS: Declustering Polygonal Maps. *Lecture Notes in Computer Science*, 951:196–206, 1995.

[35] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, Inc., New York, San Francisco, Washington, DC, USA, 4th edition, 2002.

[36] Leonid B. Sokolinsky. Choosing multiprocessor system architecture for parallel database systems. 2000.

[37] Pedro Trancoso, Josep-Lluis Larriba-Pey, Zheng Zhang, and Joseph Torellas. The Memory Performance of DSS Commericial Workloads in Shared-Memory Multiprocessors. In *Int'l. Symp. on High Performance Computer Architecture*, San Antonio, TX, USA, January 1997.

[38] Transaction Processing Performance Council. *TPC Benchmark H version 1.4.0*, 2002. `http://www.tpc.org/tpch/spec/tpch140.pdf`.

[39] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism, June 1995.

[40] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel Evaluation of Multi-Join Queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, May 1995.

[41] Mohammed J. Zaki and Ching-Tien Ho. *Large-Scale Parallel Data Mining*. Springer-Verlag, Berlin, New York, etc., August 2000.