MonetDB/X100: Hyper-Pipelining Query Execution

Peter Boncz, Marcin Zukowski, Niels Nes

CWI
Kruislaan 413
Amsterdam, The Netherlands
{P.Boncz,M.Zukowski,N.Nes}@cwi.nl

Abstract

Database systems tend to achieve only low IPC (instructions-per-cycle) efficiency on modern CPUs in compute-intensive application areas like decision support, OLAP and multimedia retrieval. This paper starts with an in-depth investigation to the reason why this happens, focusing on the TPC-H benchmark. Our analysis of various relational systems and MonetDB leads us to a new set of guidelines for designing a query processor.

The second part of the paper describes the architecture of our new X100 query engine for the MonetDB system that follows these guidelines. On the surface, it resembles a classical Volcano-style engine, but the crucial difference to base all execution on the concept of *vector processing* makes it highly CPU efficient. We evaluate the power of MonetDB/X100 on the 100GB version of TPC-H, showing its raw execution power to be between one and two orders of magnitude higher than previous technology.

1 Introduction

Modern CPUs can perform enormous amounts of calculations per second, but only if they can find enough independent work to exploit their parallel execution capabilities. Hardware developments during the past decade have significantly increased the speed difference between a CPU running at full throughput and minimal throughput, which can now easily be an order of magnitude.

One would expect that query-intensive database workloads such as decision support, OLAP, datamining, but also multimedia retrieval, all of which require many independent calculations, should provide modern CPUs the opportunity to get near optimal IPC (instructions-per-cycle) efficiencies.

However, research has shown that database systems tend to achieve low IPC efficiency on modern CPUs in these application areas [6, 3]. We question whether it should really be that way. Going beyond the (important) topic of cache-conscious query processing, we investigate in detail how relational database systems interact with modern super-scalar CPUs in query-intensive workloads, in particular the TPC-H decision support benchmark.

The main conclusion we draw from this investigation is that the architecture employed by most DBMSs inhibits compilers from using their most performance-critical optimization techniques, resulting in low CPU efficiencies. Particularly, the common way to implement the popular Volcano [10] iterator model for pipelined processing, leads to tuple-at-a-time execution, which causes both high interpretation overhead, and hides opportunities for CPU parallelism from the compiler.

We also analyze the performance of the main memory database system MonetDB[1], developed in our group, and its MIL query language [4]. MonetDB/MIL uses a column-at-a-time execution model, and therefore does not suffer from problems generated by tuple-at-a-time interpretation. However, its policy of full column materialization causes it to generate large data streams during query execution. On our decision support workload, we found MonetDB/MIL to become heavily constrained by memory bandwidth, causing its CPU efficiency to drop sharply.

Therefore, we argue for combining the column-wise execution of MonetDB with the incremental materialization offered by Volcano-style pipelining.

We designed and implemented from scratch a new query engine for the MonetDB system, called X100,

Proceedings of the 2005 CIDR Conference

[1] MonetDB is now in open-source, see `monetdb.cwi.nl`

that employs a *vectorized* query processing model. Apart from achieving high CPU efficiency, MonetDB/X100 is intended to scale up towards non main-memory (disk-based) datasets. The second part of this paper is dedicated to describing the architecture of MonetDB/X100 and evaluating its performance on the full TPC-H benchmark of size 100GB.

1.1 Outline

This paper is organized as follows. Section 2 provides an introduction to modern super-scalar (or *hyper-pipelined*) CPUs, covering the issues most relevant for query evaluation performance. In Section 3, we study TPC-H Query 1 as a micro-benchmark of CPU efficiency, first for standard relational database systems, then in MonetDB, and finally we descend into a standalone hand-coded implementation of this query to get a baseline of maximum achievable raw performance.

Section 4 describes the architecture of our new X100 query processor for MonetDB, focusing on query execution, but also sketching topics like data layout, indexing and updates.

In Section 5, we present a performance comparison of MIL and X100 inside the Monet system on the TPC-H benchmark. We discuss related work in Section 6, before concluding in Section 7.

2 How CPUs Work

Figure 1 displays for each year in the past decade the fastest CPU available in terms of MHz, as well as highest performance (one thing does not necessarily equate the other), as well as the most advanced chip manufacturing technology in production that year.

The root cause for CPU MHz improvements is progress in chip manufacturing process scales, that typically shrink by a factor 1.4 every 18 months (a.k.a. Moore's law [13]). Every smaller manufacturing scale means twice (the square of 1.4) as many, and twice smaller transistors, as well as 1.4 times smaller wire distances and signal latencies. Thus one would expect CPU MHz to increase with inverted signal latencies, but Figure 1 shows that clock speed has increased even further. This is mainly done by *pipelining*: dividing the work of a CPU instruction in ever more stages. Less work per stage means that the CPU frequency can be increased. While the 1988 Intel 80386 CPU executed one instruction in one (or more) cycles, the 1993 Pentium already had a 5-stage pipeline, to be increased in the 1999 PentiumIII to 14 while the 2004 Pentium4 has 31 pipeline stages.

Pipelines introduce two dangers: *(i)* if one instruction needs the result of a previous instruction, it cannot be pushed into the pipeline right after it, but must wait until the first instruction has passed through the pipeline (or a significant fraction thereof), and *(ii)* in case of IF-*a*-THEN-*b*-ELSE-*c* branches, the CPU must

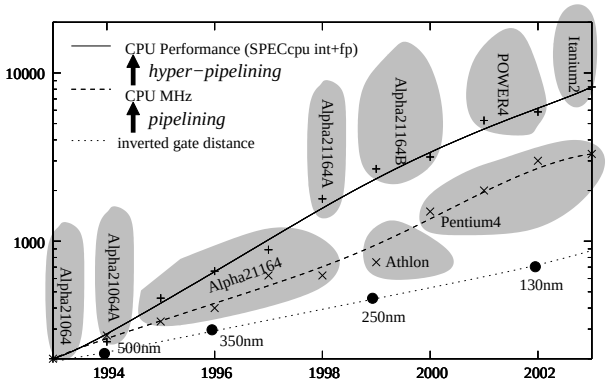

Figure 1: A Decade of CPU Performance

predict whether *a* will evaluate to true or false. It might guess the latter and put *c* into the pipeline, just after *a*. Many stages further, when the evaluation of *a* finishes, it may determine that it guessed wrongly (i.e. *mispredicted* the branch), and then must *flush* the pipeline (discard all instructions in it) and start over with *b*. Obviously, the longer the pipeline, the more instructions are flushed away and the higher the performance penalty. Translated to database systems, branches that are data-dependent, such as those found in a selection operator on data with a selectivity that is neither very high nor very low, are impossible to predict and can significantly slow down query execution [17].

In addition, *super-scalar* CPUs[2] offer the possibility to take multiple instructions into execution in parallel if they are independent. That is, the CPU has not one, but multiple pipelines. Each cycle, a new instruction can be pushed into each pipeline, provided again they are independent of all instructions already in execution. A super-scalar CPU can get to an IPC (Instructions Per Cycle) of > 1. Figure 1 shows that this has allowed real-world CPU performance to increase faster than CPU frequency.

Modern CPUs are balanced in different ways. The Intel Itanium2 processor is a VLIW (Very Large Instruction Word) processor with many parallel pipelines (it can execute up to 6 instructions per cycle) with only few (7) stages, and therefore a relatively low clock speed of 1.5GHz. In contrast, the Pentium4 has its very long 31-stage pipeline allowing for a 3.6GHz clock speed, but can only execute 3 instructions per cycle. Either way, to get to its theoretical maximum throughput, an Itanium2 needs 7x6 = 42 independent instructions *at any time*, while the Pentium4 needs 31x3 = 93. Such parallelism cannot always be found, and therefore many programs use the resources of the Itanium2 much better than the Pentium4, which explains why in benchmarks the performance of both CPUs is similar, despite the big clock speed difference.

[2]Intel introduced the term *hyper-pipelined* as a synonym for "super-scalar", to market its Pentium4 CPU.

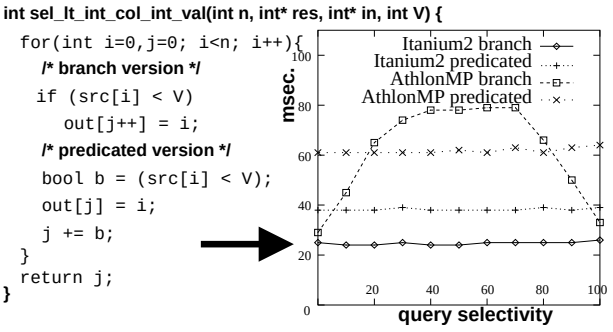

Figure 2: Itanium Hardware Predication Eliminates Branch Mispredictions

Most programming languages do not require programmers to explicitly specify in their programs which instructions (or expressions) are independent Therefore, *compiler optimizations* have become critical to achieving good CPU utilization. The most important technique is *loop pipelining*, in which an operation consisting of multiple dependent operations `F(), G()` on all `n` independent elements of an array `A` is transformed from:

```
F(A[0]),G(A[0]), F(A[1]),G(A[1]),..  F(A[n]),G(A[n])
```

into:

```
F(A[0]),F(A[1]),F(A[2]), G(A[0]),G(A[1]),G(A[2]), F(A[3]),..
```

Supposing the pipeline dependency latency of `F()` is 2 cycles, when `G(A[0])` is taken into execution, the result of `F(A[0])` has just become available.

In the case of the Itanium2 processor, the importance of the compiler is even stronger, as it is the compiler which has to find instructions that can go into different pipelines (other CPUs do that at runtime, using *out-of-order* execution). As the Itanium2 chip does not need any complex logic dedicated to finding out-of-order execution opportunities, it can contain more pipelines that do real work. The Itanium2 also has a feature called branch *predication* for eliminating branch mispredictions, by allowing to execute *both* the THEN and ELSE blocks in parallel and discard one of the results as soon as the result of the condition becomes known. It is also the task of the compiler to detect opportunities for branch predication.

Figure 2 shows a micro-benchmark of the selection query `SELECT oid FROM table WHERE col < X`, where `X` is uniformly and randomly distributed over `[0:100]` and we vary the selectivity `X` between 0 and 100. Normal CPUs like the AthlonMP show worst-case behavior around 50%, due to branch mispredictions. As suggested in [17], by rewriting the code cleverly, we can transform the branch into a boolean calculation (the "predicated" variant). Performance of this rewritten variant is independent of the selectivity, but incurs a higher average cost. Interestingly, the "branch" variant on Itanium2 is highly efficient and independent of selectivity as well, because the compiler transforms the branch into hardware-predicated code.

Finally, we should mention the importance of on-chip caches to CPU throughput. About 30% of all instructions executed by a CPU are memory loads and stores, that access data on DRAM chips, located inches away from the CPU on a motherboard. This imposes a physical lower bound on memory latency of around 50 ns. This (ideal) minimum latency of 50ns already translates into 180 wait cycles for a 3.6GHz CPU. Thus, only if the overwhelming majority of the memory accessed by a program can be found in an on-chip *cache*, a modern CPU has a chance to operate at its maximum throughput. Recent database research has shown that DBMS performance is strongly impaired by memory access cost ("cache misses") [3], and can significantly improve if *cache-conscious* data structures are used, such as cache-aligned B-trees [16, 7] or column-wise data layouts such as PAX [2] and DSM [8] (as in MonetDB). Also, query processing algorithms that restrict their random memory access patterns to regions that fit a CPU cache, such as radix-partitioned hash-join [18, 11], strongly improve performance.

All in all, CPUs have become highly complex devices, where the instruction throughput of a processor can vary by orders of magnitude (!) depending on the cache hit-ratio of the memory loads and stores, the number of branches and whether they can be predicted/predicated, as well as the amount of independent instructions a compiler and the CPU can detect on average. It has been shown that query execution in commercial DBMS systems get an IPC of only 0.7 [6], thus executing *less* than one instruction per cycle. In contrast, scientific computation (e.g. matrix multiplication) or multimedia processing does extract average IPCs of up to 2 out of modern CPUs. We argue that database systems do not need to perform so badly, especially not on large-scale analysis tasks, where millions of tuples need to be examined and expressions to be calculated. This abundance of work contains plenty of independence that should be able to fill all the pipelines a CPU can offer. Hence, our quest is to adapt database architecture to expose this to the compiler and CPU where possible, and thus significantly improve query processing throughput.

3 Microbenchmark: TPC-H Query 1

While we target CPU efficiency of query processing in general, we first focus on *expression calculation*, discarding more complex relational operations (like join) to simplify our analysis. We choose Query 1 of the TPC-H benchmark, shown in Figure 3, this query is CPU-bound because on all RDBMSs we tested. Also, this query requires virtually no optimization or fancy join implementations as its plan is so simple. Thus, all database systems operate on a level playing field and mainly expose their expression evaluation efficiency.

The TPC-H benchmark operates on a data ware-

house of 1GB, the size of which can be increased with a Scaling Factor (SF). Query 1 is a scan on the `lineitem` table of SF*6M tuples, that selects almost all tuples (SF*5.9M), and computes a number of fixed-point decimal expressions: two column-to-constant subtractions, one column-to-constant addition, three column-to-column multiplications, and eight aggregates (four SUM()s, three AVG()s and a COUNT()). The aggregate grouping is on two single-character columns, and yields only 4 unique combinations, such that it can be done efficiently with a small hash-table, requiring no additional I/O nor even CPU cache misses (for accessing the hash-table).

In the following, we analyze the performance of Query 1 first on relational database systems, then on MonetDB/MIL and finally in a hand-coded program.

TPC-H Query 1 Experiments				
DBMS "X"	28.1	1	1 AthlonMP 1533MHz,	609/547
MySQL 4.1	26.6	1	1 AthlonMP 1533MHz,	609/547
MonetDB/MIL	3.7	1	1 AthlonMP 1533MHz,	609/547
MonetDB/MIL	3.4	1	1 Itanium2 1.3GHz,	1132/1891
hand-coded	0.22	1	1 AthlonMP 1533MHz,	609/547
hand-coded	0.14	1	1 Itanium2 1.3GHz,	1132/1891
MonetDB/X100	0.50	1	1 AthlonMP 1533MHz,	609/547
MonetDB/X100	0.31	1	1 Itanium2 1.3GHz,	1132/1891
MonetDB/X100	0.30	100	1 Itanium2 1.3GHz,	1132/1891
(sec*#CPU)/SF	**SF**	**#CPU,**		**SPECcpu int/fp**
Oracle10g	18.1	100	16 Itanium2 1.3GHz,	1132/1891
Oracle10g	13.2	1000	64 Itanium2 1.5GHz,	1408/2161
SQLserver2000	18.0	100	2 Xeon P4 3.0GHz,	1294/1208
SQLserver2000	21.8	1000	8 Xeon P4 2.8GHz,	1270/1094
DB2 UDB 8.1	9.0	100	4 Itanium2 1.5GHz,	1408/2161
DB2 UDB 8.1	7.4	100	2 Opteron 2.0GHz,	1409/1514
Sybase IQ 12.5	15.6	100	2 USIII 1.28GHz,	704/1054
Sybase IQ 12.5	15.8	1000	2 USIII 1.28GHz,	704/1054
TPC-H Query 1 Reference Results (www.tpc.org)				

Table 1: TPC-H Query 1 Performance

3.1 Query 1 on Relational Database Systems

Since the early days of RDBMSs, query execution functionality is provided by implementing a physical relational algebra, typically following the Volcano [10] model of pipelined processing. Relational algebra, however, has a high degree of freedom in its parameters. For instance, even a simple ScanSelect(R, b, P) only at query-time receives full knowledge of the format of the input relation R (number of columns, their types, and record offsets), the boolean selection expression b (which may be of any form), and a list of projection expressions P (each of arbitrary complexity) that define the output relation. In order to deal with all possible R, b, and P, DBMS implementors must in fact implement an *expression interpreter* that can handle expressions of arbitrary complexity.

```
SELECT   l_returnflag, l_linestatus,
         sum(l_quantity) AS sum_qty,
         sum(l_extendedprice) AS sum_base_price,
         sum(l_extendedprice * (1 - l_discount))
           AS sum_disc_price,
         sum(l_extendedprice * (1 - l_discount) *
             (1 + l_tax)) AS sum_charge,
         avg(l_quantity) AS avg_qty,
         avg(l_extendedprice) AS avg_price,
         avg(l_discount) AS avg_disc,
         count(*) AS count_order

FROM     lineitem
WHERE    l_shipdate <= date '1998-09-02'
GROUP BY l_returnflag, l_linestatus
```

Figure 3: TPC-H Query 1

One of the dangers of such an interpreter, especially if the granularity of interpretation is a tuple, is that the cost of the "real work" (i.e. executing the expressions found in the query) is only a tiny fraction of total query execution cost. We can see this happening in Table 2 that shows a `gprof` trace of a MySQL 4.1 of TPC-H Query 1 on a database of SF=1. The second column shows the percentage of total execution time spent in the routine, excluding time spent in routines it called (excl.). The first column is a cumulative sum of the second (cum.). The third column lists how many times the routine was called, while the fourth and fifth columns show the average number of instructions executed on each call, as well as the IPC achieved.

The first observation to make is that the five operations that do all the "work" (displayed in boldface), correspond to only 10% of total execution time. Closer inspection shows that 28% of execution time is taken up by creation and lookup in the hash-table used for aggregation. The remaining 62% of execution time is spread over functions like `rec_get_nth_field`, that navigate through MySQL's record representation and copy data in and out of it. Other factors, such as locking overhead (`pthread_mutex_unlock`, `mutex_test_and_set`) or buffer page allocation (`buf_frame_align`) seem to play only a minor role in this decision support query.

The second observation is the cost of the `Item` operations that correspond to the computational "work" of the query. For example, `Item_func_plus::val` has a cost of 38 instructions per addition. This performance trace was made on an SGI machine with MIPS R12000 CPU[3], which can execute three integer or floating-point instructions and one load/store per cycle, with an average operation latency of about 5 cycles. A simple arithmetic operation `+(double src1, double src2) : double` in RISC instructions would look like:

```
LOAD src1,reg1
LOAD src2,reg2
ADD reg1,reg2,reg3
STOR dst,reg3
```

The limiting factor in this code are the three

[3]On our Linux test platforms, no multi-threaded profiling tools seem to be available.

cum.	excl.	calls	ins.	IPC	function
11.9	11.9	846M	6	0.64	ut_fold_ulint_pair
20.4	8.5	0.15M	27K	0.71	ut_fold_binary
26.2	5.8	77M	37	0.85	memcpy
29.3	**3.1**	**23M**	**64**	**0.88**	**Item_sum_sum::update_field**
32.3	3.0	6M	247	0.83	row_search_for_mysql
35.2	**2.9**	**17M**	**79**	**0.70**	**Item_sum_avg::update_field**
37.8	2.6	108M	11	0.60	rec_get_bit_field_1
40.3	2.5	6M	213	0.61	row_sel_store_mysql_rec
42.7	2.4	48M	25	0.52	rec_get_nth_field
45.1	2.4	60	19M	0.69	ha_print_info
47.5	2.4	5.9M	195	1.08	end_update
49.6	2.1	11M	89	0.98	field_conv
51.6	2.0	5.9M	16	0.77	Field_float::val_real
53.4	1.8	5.9M	14	1.07	Item_field::val
54.9	1.5	42M	17	0.51	row_sel_field_store_in_mysql..
56.3	1.4	36M	18	0.76	buf_frame_align
57.6	**1.3**	**17M**	**38**	**0.80**	**Item_func_mul::val**
59.0	1.4	25M	25	0.62	pthread_mutex_unlock
60.2	1.2	206M	2	0.75	hash_get_nth_cell
61.4	1.2	25M	21	0.65	mutex_test_and_set
62.4	1.0	102M	4	0.62	rec_get_1byte_offs_flag
63.4	1.0	53M	9	0.58	rec_1_get_field_start_offs
64.3	0.9	42M	11	0.65	rec_get_nth_field_extern_bit
65.3	**1.0**	**11M**	**38**	**0.80**	**Item_func_minus::val**
65.8	**0.5**	**5.9M**	**38**	**0.80**	**Item_func_plus::val**

Table 2: MySQL gprof trace of TPC-H Q1: `+,-,*,SUM,AVG` takes <10%, low IPC of 0.7

load/store instructions, thus a MIPS processor can do one `*(double,double)` per 3 cycles. This is in sharp contrast to the MySQL cost of #ins/Instruction-Per-Cycle (IPC) = 38/0.8 = 49 cycles! One explanation for this high cost is the absence of *loop pipelining*. As the routine called by MySQL only computes one addition per call, instead of an array of additions, the compiler cannot perform loop pipelining. Thus, the addition consists of four *dependent* instructions that have to wait for each other. With a mean instruction latency of 5 cycles, this explains a cost of about 20 cycles. The rest of the 49 cycles are spent on jumping into the routine, and pushing and popping the stack.

The consequence of the MySQL policy to execute expressions tuple-at-a-time, is twofold:

- `Item_func_plus::val` only performs one addition, preventing the compiler from creating a pipelined loop. As the instructions for one operation are highly dependent, empty pipeline slots must be generated (stalls) to wait for the instruction latencies, such that the cost of the loop becomes 20 instead of 3 cycles.
- the cost of the routine call (in the ballpark of 20 cycles) must be amortized over only one operation, which effectively doubles the operation cost.

We also tested the same query on a well-known commercial RDBMS (see the first row of Table 1). As we obviously lack the source code of this product, we cannot produce a `gprof` trace. However, the query evaluation cost on this DBMS is very similar to MySQL.

The lower part of Table 1 includes some official TPC-H Query 1 results taken from the TPC website.

SF=1		SF=0.001		tot	res	(BW = MB/s)
ms	BW	us	BW	MB	size	**MIL statement**
127	352	150	305	45	5.9M	s0 := select(l_shipdate).mark
134	505	113	608	68	5.9M	s1 := join(s0,l_returnflag)
134	506	113	608	68	5.9M	s2 := join(s0,l_linestatus)
235	483	129	887	114	5.9M	s3 := join(s0,l_extprice)
233	488	130	881	114	5.9M	s4 := join(s0,l_discount)
232	489	127	901	114	5.9M	s5 := join(s0,l_tax)
134	507	104	660	68	5.9M	s6 := join(s0,l_quantity)
290	155	324	141	45	5.9M	s7 := group(s1)
329	136	368	124	45	5.9M	s8 := group(s7,s2)
0	0	0	0	0	4	s9 := unique(s8.mirror)
206	440	60	1527	91	5.9M	r0 := [+](1.0,s5)
210	432	51	1796	91	5.9M	r1 := [-](1.0,s4)
274	498	83	1655	137	5.9M	r2 := [*](s3,r1)
274	499	84	1653	137	5.9M	r3 := [*](s12,r0)
165	271	121	378	45	4	r4 := {sum}(r3,s8,s9)
165	271	125	366	45	4	r5 := {sum}(r2,s8,s9)
163	275	128	357	45	4	r6 := {sum}(s3,s8,s9)
163	275	128	357	45	4	r7 := {sum}(s4,s8,s9)
144	151	107	214	22	4	r8 := {sum}(s6,s8,s9)
112	196	145	157	22	4	r9 := {count}(s7,s8,s9)
3724		2327		**TOTAL**		

Table 3: MonetDB/MIL trace of TPC-H Query 1

Query 1 is dominated by computations in a full scan and this scales linearly with table size. The query is also "embarrasingly parallel" using horizontal parallelism, such that TPC-H results on parallel systems most likely achieved linear speedup. Thus, we can compare throughput for different systems, by normalizing all times towards SF=1 and a single CPU. We also provide the SPECcpu int/float scores of the various hardware platforms used. We mainly do this in order to check that the relational DBMS results we obtained are roughly in the same ballpark as what is published by TPC. This leads us to believe that what we see in the MySQL trace is likely representative of what happens in commercial RDBMS implementations.

3.2 Query 1 on MonetDB/MIL

The MonetDB system [4] developed by our group, is mostly known for its use of vertical fragmentation, storing tables column-wise, each column in a Binary Association Table (BAT) that contains `[oid,value]` combinations. A BAT is a 2-column table where the left column is called *head* and the right column *tail*. The algebraic query language of MonetDB is a column-algebra called MIL [5].

In contrast to the relational algebra, the MIL algebra does not have any degree of freedom. Its algebraic operators have a fixed number of parameters of a fixed format (all two-column tables or constants). The expression calculated by an operator is fixed, as well as the shape of the result. For example, the MIL `join(BAT[`t_l, t_e`] A, BAT[`t_e, t_r`] B) : BAT[`t_l, t_r`]` is an equi-join between the tail column of `A` and head column of `B`, that for each matching combination of tuples returns the head value from `A` and tail value from `B`. The mechanism in MIL to join on the other column (i.e. the head, instead of the tail) of `A`, is

to use the MIL `reverse(A)` operator that returns a view on `A` with its columns swapped: `BAT`$[t_e, t_l]$. This `reverse` is a zero-cost operation in MonetDB that just swaps some pointers in the internal representation of a BAT. Complex expressions must be executed using multiple statements in MIL. For example, `extprice * (1 - tax)` becomes `tmp1 := [-](1,tax); tmp2 := [*](extprice,tmp1)`, where `[*]()` and `[-]()` are *multiplex* operators that "map" a function onto an entire BAT (column). MIL executes in column-wise fashion in the sense that its operators always consume a number of materialized input BATs and materialize a single output BAT.

We used the MonetDB/MIL SQL front-end to translate TPC-H Query 1 into MIL and run it. Table 3 shows all 20 MIL invocations that together span more than 99% of elapsed query time. On TPC-H Query 1, MonetDB/MIL is clearly faster than MySQL and the commercial DBMS on the same machine, and is also competitive with the published TPC-H scores (see Table 1). However, closer inspection of Table 3 shows that almost all MIL operators are memory-bound instead of CPU-bound! This was established by running the same query plan on the TPC-H dataset with SF=0.001, such that all used columns of the `lineitem` table as well as all intermediate results fit inside the CPU cache, eliminating any memory traffic. MonetDB/MIL then becomes almost twice as fast. Columns 2 and 4 list the bandwidth (BW) in MB/s achieved by the individual MIL operations, counting both the size of the input BATs and the produced output BAT. On SF=1, MonetDB gets stuck at 500MB/s, which is the maximum bandwidth sustainable on this hardware [1]. When running purely in the CPU cache at SF=0.001, bandwidths can get above 1.5GB/s. For the multiplexed multiplication `[*]()`, a bandwidth of only 500MB/s means 20M tuples per second (16 bytes in, 8 bytes out), thus 75 cycles per multiplication on our 1533MHz CPU, which is even worse than MySQL.

Thus, the column-at-a-time policy in MIL turns out to be a two-edged sword. To its advantage is the fact that MonetDB is not prone to the MySQL problem of spending 90% of its query execution time in tuple-at-a-time interpretation "overhead". As the multiplex operations that perform expression calculations work on entire BATs (basically arrays of which the layout is known at compile-time), the compiler is able to employ loop-pipelining such that these operators achieve high CPU efficiencies, embodied by the SF=0.001 results.

However, we identify the following problems with full materialization. First, queries that contain complex calculation expressions over many tuples will materialize an entire result column for each function in the expression. Often, such function results are not required in the query result, but just serve as inputs to other functions in the expression. For instance, if an aggregation is the top-most operator in the query plan, the eventual result size might even be negligible (such as in Query 1). In such cases, MIL materializes much more data than strictly necessary, causing its high bandwidth consumption.

Also, Query 1 starts with a 98% selection of the 6M tuple table, and performs the aggregations on the remaining 5.9M million tuples. Again, MonetDB materializes the relevant result columns of the `select()` using six positional `join()`s. These joins are not required in a Volcano-like pipelined execution model. It can do the selection, computations and aggregation all in a single pass, not materializing any data.

While in this paper we concentrate on CPU efficiency in main-memory scenarios, we point out that the "artificially" high bandwidths generated by MonetDB/MIL make it harder to scale the system to disk-based problems efficiently, simply because memory bandwidth tends to be much larger (and cheaper) than I/O bandwidth. Sustaining a data transfer of e.g. 1.5GB/s would require a truly high-end RAID system with a very large number of disks.

```
static void tpch_query1(int n, int hi_date,
  unsigned char*__restrict__ p_returnflag,
  unsigned char*__restrict__ p_linestatus,
  double*__restrict__        p_quantity,
  double*__restrict__        p_extendedprice,
  double*__restrict__        p_discount,
  double*__restrict__        p_tax,
  int*__restrict__           p_shipdate,
  aggr_t1*__restrict__       hashtab)
{
 for(int i=0; i<n; i++) {
  if (p_shipdate[i] <= hi_date) {
    aggr_t1 *entry = hashtab +
      (p_returnflag[i]<<8) + p_linestatus[i];
    double discount = p_discount[i];
    double extprice = p_extendedprice[i];
    entry->count++;
    entry->sum_qty += p_quantity[i];
    entry->sum_disc += discount;
    entry->sum_base_price += extprice;
    entry->sum_disc_price += (extprice *= (1-discount));
    entry->sum_charge += extprice*(1-p_tax[i]);
}}}
```

Figure 4: Hard-Coded UDF for Query 1 in C

3.3 Query 1: Baseline Performance

To get a baseline of what modern hardware can do on a problem like Query 1, we implemented it as a single User Defined Function (UDF) in MonetDB, as shown in Figure 4. The UDF gets passed in only those columns touched by the query. In MonetDB, these columns are stored as arrays in `BAT[void,T]`s. That is, the `oid` values in the head column are densely ascending from 0 upwards. In such cases, MonetDB uses `void`s ("virtual-oids") that are not stored. The BAT then takes the form of an array. We pass these arrays as *__restrict__* pointers, such that the C compiler knows that they are non-overlapping. Only then can it apply

loop-pipelining!

This implementation exploits the fact that a GROUP BY on two single-byte characters can never yield more than 65536 combinations, such that their combined bit-representation can be used directly as an array index to the table with aggregation results. Like in MonetDB/MIL, we performed some common subexpression elimination such that one minus and three AVG aggregates can be omitted.

Table 1 shows that this UDF implementation (labeled "hand-coded") reduces query evaluation cost to a stunning 0.22 seconds. From the same table, you will notice that our new X100 query processor, that is the topic of the remainder of this paper, is able to get within a factor 2 of this hand-coded implementation.

4 X100: A Vectorized Query Processor

The goal of X100 is to *(i)* execute high-volume queries at high CPU efficiency, *(ii)* be extensible to other application domains like data mining and multi-media retrieval, and achieve those same high efficiencies on extensibility code, and *(iii)* scale with the size of the lowest storage hierarchy (disk).

In order to achieve our goals, X100 must fight bottlenecks throughout the entire computer architecture:

Disk the ColumnBM I/O subsystem of X100 is geared towards efficient sequential data access. To reduce bandwidth requirements, it uses a vertically fragmented data layout, that in some cases is enhanced with lightweight data compression.

RAM like I/O, RAM access is carried out through explicit memory-to-cache and cache-to-memory routines (which contain platform-specific optimizations, sometimes including e.g. SSE prefetching and data movement assembly instructions). The same vertically partitioned and even compressed disk data layout is used in RAM to save space and bandwidth.

Cache we use a Volcano-like execution pipeline based on a *vectorized* processing model. Small (e.g. 1000 values) vertical chunks of cache-resident data items, called "vectors" are the unit of operation for X100 execution primitives. The CPU cache is the only place where bandwidth does not matter, and therefore (de)compression happens on the boundary between RAM and cache. The X100 query processing operators should be cache-conscious and fragment huge datasets efficiently into cache-chunks and perform random data access only there.

CPU vectorized primitives expose to the compiler that processing a tuple is independent of the previous and next tuples. Vectorized primitives for projections (expression calculation) do this easily, but we try to achieve the same for other query processing operators as well (e.g. aggregation). This allows compilers to produce efficient loop-pipelined code. To improve the CPU throughput further (mainly by reducing the number of load/stores in the instruction mix), X100 contains facilities to compile vectorized primitives for whole expression sub-trees rather than single functions. Currently, this compilation is statically steered, but it may eventually become a run-time activity mandated by an optimizer.

To maintain focus in this paper, we only summarily describe disk storage issues, also because the ColumnBM buffer manager is still under development. In all our experiments, X100 uses MonetDB as its storage manager (as shown in Figure 5), where it operates on in-memory BATs.

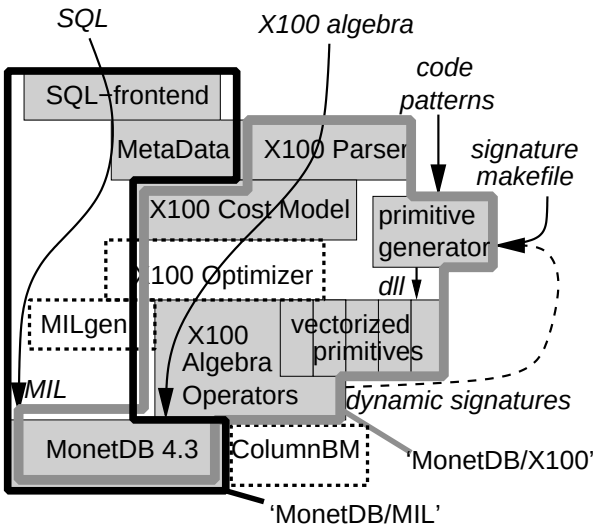

Figure 5: X100 Software Architecture

4.1 Query Language

X100 uses a rather standard relational algebra as query language. We departed from the column-at-a-time MIL language so that the relational operators can process (vectors of) multiple columns at the same time, allowing to use a vector produced by one expression as the input to another, while the data is in the CPU cache.

4.1.1 Example

To demonstrate the behavior of MonetDB/X100, Figure 6 presents the execution of a simplified version of a TPC-H Query 1, with the following X100 relational algebra syntax:

```
Aggr(
  Project(
    Select(
```

```
    Table(lineitem),
    < (shipdate, date('1998-09-03'))),
  [ discountprice = *( -( flt('1.0'), discount),
                       extendedprice) ]),
[ returnflag ],
[ sum_disc_price = sum(discountprice) ])
```

Execution proceeds using Volcano-like pipelining, on the granularity of a vector (e.g. 1000 values). The `Scan` operator retrieves data vector-at-a-time from Monet BATs. Note that only attributes relevant to the query are actually scanned.

A second step is the `Select` operator, which creates a *selection-vector*, filled with positions of tuples that match our predicate. Then the `Project` operator is executed to calculate expressions needed for the final aggregation. Note that "discount" and "extendedprice" columns are not modified during selection. Instead, the selection-vector is taken into account by `map`-primitives to perform calculations only for relevant tuples, writing results at the same positions in the output vector as they were in the input one. This behavior requires propagating of the selection-vector to the final `Aggr`. There, for each tuple its position in the hash table is calculated, and then, using this data, aggregate results are updated. Additionally, for the new elements in the hash table, values of the grouping attribute are saved. The contents of the hash-table becomes available as the query result as soon as the underlying operators become exhausted and cannot produce more vectors.

4.1.2 X100 Algebra

Figure 7 lists the currently supported X100 algebra operators. In X100 algebra, a `Table` is a materialized relation, whereas a `Dataflow` just consists of tuples flowing through a pipeline.

`Order`, `TopN` and `Select` return a `Dataflow` with the same shape as its input. The other operators define a `Dataflow` with a new shape. Some peculiarities of this algebra are that `Project` is just used for expression calculation; it does not eliminate duplicates. Duplicate elimination can be performed using an `Aggr` with only group-by columns. The `Array` operator generates a `Dataflow` representing a N-dimensional array as an N-ary relation containing all valid array index coordinates in column-major dimension order. It is used by the RAM array manipulation front-end for the MonetDB system [9].

Aggregation is supported by three physical operators: *(i)* direct aggregation, *(ii)* hash aggregation, and *(iii)* ordered aggregation. The latter is chosen if all group-members will arrive right after each other in the source `Dataflow`. Direct aggregation can be used for small datatypes where the bit-representation is limited to a known (small) domain, similar to the way aggregation was handled in the "hand-coded" solution (Section 3.3). In all other cases, hash-aggregation is used.

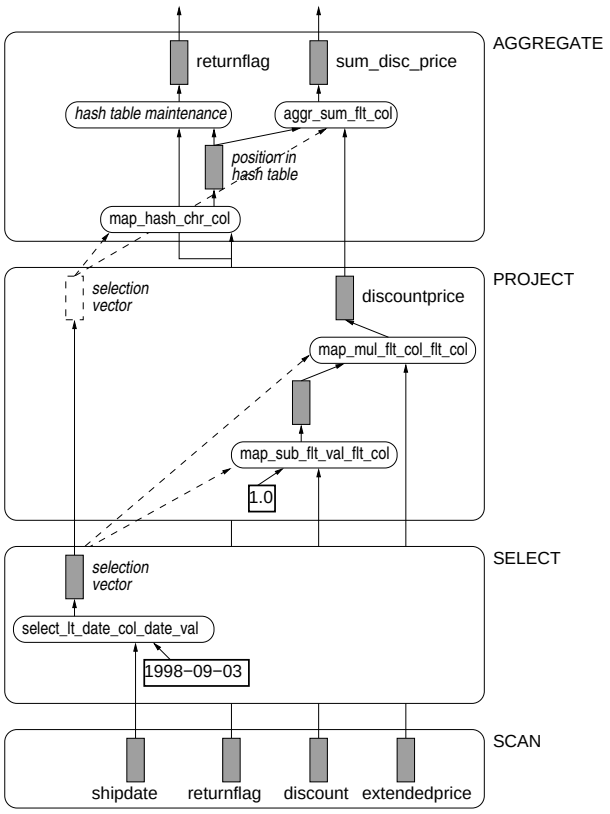

Figure 6: Execution scheme of a simplified TPC-H Query 1 in MonetDB/X100

X100 currently only supports left-deep joins. The default physical implementation is a `CartProd` operator with a `Select` on top (i.e. nested-loop join). If X100 detects a foreign-key condition in a join condition, and a join-index is available, it exploits it with a `Fetch1Join` or `FetchNJoin`.

The inclusion of these fetch-joins in X100 is no coincidence. In MIL, the "positional-join" of an `oid` into a `void` column has proven valuable on vertically fragmented data stored in dense columns. Positional joins allow dealing with the "extra" joins needed for vertical fragmentation in a highly efficient way [4]. Just like the `void` type in MonetDB, X100 gives each table a virtual `#rowId` column, which is just a densely ascending number from 0. The `Fetch1Join` allows to positionally fetch column values by `#rowId`.

4.2 Vectorized Primitives

The primary reason for using the column-wise vector layout is **not** to optimize memory layout in the cache (X100 is supposed to operate on cached data anyway). Rather, vectorized execution primitives have the advantage of a low *degree of freedom* (as discussed in Section 3.2). In a vertically fragmented data model, the execution primitives only know about the columns they operate on without having to know about the overall table layout (e.g. record offsets). When compil-

Table(ID) : Table
Scan(Table) : Dataflow
Array(List<Exp<int>>) : Dataflow
Select(Dataflow, Exp<bool>) : Dataflow
Join(Dataflow, Table, Exp<bool>, List<Column>) : Dataflow CartProd(Dataflow, Table, List<Column>) Fetch1Join(Dataflow, Table, Exp<int>, List<Column>) FetchNJoin(Dataflow, Table, Exp<int>, Exp<int>, Column, List<Column>)
Project(Dataflow, List<Exp<*>>) : Dataflow
Aggr(Dataflow, List<Exp<*>>, List<AggrExp>) : Dataflow OrdAggr(Dataflow, List<Exp<*>>, List<AggrExp>) DirectAggr(Dataflow, List<Exp<*>>, List<AggrExp>) HashAggr(Dataflow, List<Exp<*>>, List<AggrExp>])
TopN(Dataflow, List<OrdExp>, List<Exp<*>>, int):Dataflow
Order(Table, List<OrdExp>, List<AggrExp>) : Table

Figure 7: X100 Query Algebra

ing X100, the C compiler sees that the X100 *vectorized primitives* operate on restricted (independent) arrays of fixed shape. This allows it to apply aggressive loop pipelining, critical for modern CPU performance (see Section 2). As an example, we show the (generated) code for vectorized floating-point addition:

```
map_plus_double_col_double_col(int n,
  double*__restrict__ res,
  double*__restrict__ col1, double*__restrict__ col2,
  int*__restrict__ sel)
{
  if (sel) {
    for(int j=0;j<n; j++) {
      int i = sel[j];
      res[i] = col1[i] + col2[i];
    }
  } else {
    for(int i=0;i<n; i++)
      res[i] = col1[i] + col2[i];
} }
```

The `sel` parameter may be NULL or point to an array of `n` selected array positions (i.e. the "selection-vector" from Figure 6). All X100 vectorized primitives allow passing such selection vectors. The rationale is that after a selection, leaving the vectors delivered by the child operator intact is often quicker than copying all selected data into new (contiguous) vectors.

X100 contains hundreds of vectorized primitives. These are not written (and maintained) by hand, but are *generated* from *primitive patterns*. The primitive pattern for addition is:

```
any::1 +(any::1 x,any::1 y) plus = x + y
```

This pattern states that an addition of two values of the same type (but without any type restriction) is implemented in C by the infix operator `+`. It produces a result of the same type, and the name identifier should be `plus`. Type-specific patterns later in the specification file may override this pattern (e.g. `str +(str x,str y) concat = str_concat(x,y)`).

The other part of primitive generation is a file with *map signature requests*:

```
+(double*, double*)
+(double, double*)
+(double*, double)
+(double, double)
```

This requests to generate all possible combinations of addition between single values and columns (the latter identified with an extra `*`). Other extensible RDBMSs often only allow UDFs with single-value parameters [19]. This inhibits loop pipelining, reducing performance (see Section 3.1). [4]

We can also request *compound primitive* signatures:

```
/(square(-(double*, double*)), double*)
```

The above signature is the Mahanalobis distance, a performance-critical operation for some multi-media retrieval tasks [9]. We found that the compound primitives often perform twice as fast as the single-function vectorized primitives. Note that this factor 2 is similar to the difference between MonetDB/X100 and the hand-coded implementation of TPC-H Query in Table 1. The reason why compound primitives are more efficient is a better instruction mix. Like in the example with addition on the MIPS processor in Section 3.1, vectorized execution often becomes load/store bound, because for simple 2-ary calculations, each vectorized instruction requires loading two parameters and storing one result (1 work instruction, 3 memory instructions). Modern CPUs can typically only perform 1 or 2 load/store operations per cycle. In compound primitives, the results from one calculation are passed via a CPU register to the next calculation, with load/stores only occurring at the edges of the expression graph.

Currently, the *primitive generator* is not much more than a macro expansion script in the make sequence of the X100 system. However, we intend to implement dynamic compilation of compound primitives as mandated by an optimizer.

A slight variation on the `map` primitives are the `select_*` primitives (see also Figure 2). These only exist for code patterns that return a boolean. Instead of producing a full result vector of booleans (as the `map` does), the `select` primitives fill a result array of selected vector positions (integers), and return the total number of selected tuples.

Similarly, there are the `aggr_*` primitives that calculate aggregates like `count`, `sum`, `min`, and `max`. For each, an initialization, an update, and an epilogue pattern need to be specified. The primitive generator then generates the relevant routines for the various implementations of aggregation in X100.

The X100 mechanism of allowing database extension developers to provide (source-)code patterns in-

[4] If X100 is used in resource-restricted environments, the size of the X100 binary (less than a MB now) could be further reduced by omitting the column-versions of (certain) execution primitives. X100 will still be able to process those primitives although more slowly, with a vector size of 1.

stead of compiled code, allows all ADTs to get first-class-citizen treatment during query execution. This was also a weak point of MIL (and most extensible DBMSs [19]), as its main algebraic operators were only optimized for the built-in types.

4.3 Data Storage

MonetDB/X100 stores all tables in vertically fragmented form. The storage scheme is the same whether the new ColumnBM buffer manager is used, or MonetDB `BAT[void,T]` storage. While MonetDB stores each BAT in a single contiguous file, ColumnBM partitions those files in large (>1MB) chunks.

A disadvantage of vertical storage is an increased update cost: a single row update or delete must perform one I/O for each column. MonetDB/X100 circumvents this by treating the vertical fragments as immutable objects. Updates go to delta structures instead. Figure 8 shows that deletes are handled by adding the tuple ID to a deletion list, and that inserts lead to appends in separate *delta columns*. ColumnBM actually stores all delta columns together in a chunk, which equates PAX [2]. Thus, both operations incur only one I/O. Updates are simply a deletion followed by an insertion. Updates make the delta columns grow, such that whenever their size exceeds a (small) percentile of the total table size, data storage should be reorganized, such that the vertical storage is up-to-date again and the delta columns are empty.

An advantage of vertical storage is that queries that access many tuples but not all columns save bandwidth (this holds both for RAM bandwidth and I/O bandwidth). We further reduce bandwidth requirements using *lightweight compression*. MonetDB/X100 supports *enumeration types*, which effectively store a column as a single-byte or two-byte integer. This integer refers to `#rowId` of a mapping table. MonetDB/X100 automatically adds a `Fetch1Join` operation to retrieve the uncompressed value using the small integer when such columns are used in a query. Notice that since the vertical fragments are immutable, updates just go to the delta columns (which are never compressed) and do not complicate the compression scheme.

MonetDB/X100 also supports simple "summary" indices, similar to [12], which are used if a column is clustered (almost sorted). These summary indices contain a `#rowId`, the running maximum value of the column until that point in the base table, and a reversely running minimum at a very coarse granularity (the default size is 1000 entries, with `#rowid`s taken with fixed intervals from the base table). These summary indices can be used to quickly derive `#rowId` bounds for range predicates. Notice again, due to the property that vertical fragments are immutable, indices on them effectively require no maintenance. The delta columns, which are supposed to be small and in-memory, are not indexed and must always be accessed.

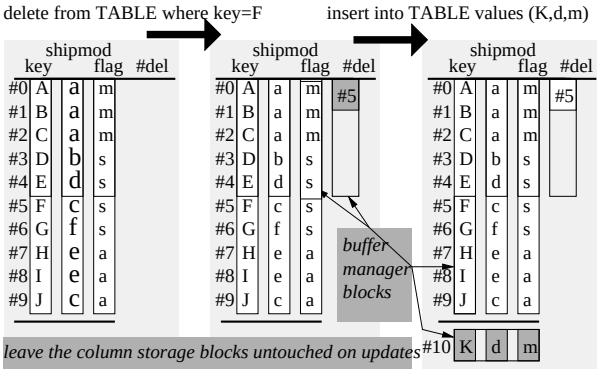

Figure 8: Vertical Storage and Updates

5 TPC-H Experiments

Table 4 shows the results of executing all TPC-H queries on both MonetDB/MIL and MonetDB/X100. We ran the SQL benchmark queries on an out-of-the-box MonetDB/MIL system with its SQL-frontend on our AthlonMP platform (1533MHz, 1GB RAM, Linux2.4) at SF=1. We also hand-translated all TPC-H queries to X100 algebra, and ran them on MonetDB/X100. The comparison between the first two result columns clearly shows that MonetDB/X100 outperforms MonetDB/MIL.

Both MonetDB/MIL and MonetDB/X100 use join indices over all foreign key paths. For MonetDB/X100 we sorted the `orders` table on date, and kept `lineitem` clustered with it. We use summary indices (see Section 4.3) on all date columns of both tables. We also sorted both suppliers and customers on (region,country). In all, total disk storage for MonetDB/MIL was about 1GB, and around 0.8GB for MonetDB/X100 (SF=1). The reduction was achieved by using enumeration types, where possible.

We also ran TPC-H both at SF=1 and SF=100 on our Itanium2 1.3GHz (3MB cache) server with 12GB RAM running Linux2.4. The last column of Table 4 lists official TPC-H results for the MAXDATA Platinum 9000-4R, a server machine with four 1.5GHz (6MB cache) Itanium2 processors and 32GB RAM running DB2 8.1 UDB[5].

We should clarify that all MonetDB TPC-H numbers are in-memory results; no I/O occurs. This should be taken into account especially when comparing with the DB2 results. It also shows that even at SF=100, MonetDB/X100 needs less than our 12GB RAM for each individual query. If we would have had 32GB of RAM like the DB2 platform, the hot-set for all TPC-H queries would have fit in memory.

While the DB2 TPC-H numbers obviously do include I/O, its impact may not be that strong as its test platform uses 112 SCSI disks. This suggests that disks were added until DB2 became CPU-bound. In

[5]These results are from stream 0, also known as the "power test", in which there are no concurrent queries nor updates.

MonetDB/MIL	MonetDB/X100, 1CPU		DB2, 4CPU	
Q	SF=1	SF=1	SF=1	SF=100	SF=100
1	3.72	0.50	0.31	30.25	229
2	0.46	0.01	0.01	0.81	19
3	2.52	0.04	0.02	3.77	16
4	1.56	0.05	0.02	1.15	14
5	2.72	0.08	0.04	11.02	72
6	2.24	0.09	0.02	1.44	12
7	3.26	0.22	0.22	29.47	81
8	2.23	0.06	0.03	2.78	65
9	6.78	0.44	0.44	71.24	274
10	4.40	0.22	0.19	30.73	47
11	0.43	0.03	0.02	1.66	20
12	3.73	0.09	0.04	3.68	19
13	11.42	1.26	1.04	148.22	343
14	1.03	0.02	0.02	2.64	14
15	1.39	0.09	0.04	14.36	30
16	2.25	0.21	0.14	15.77	64
17	2.30	0.02	0.02	1.75	77
18	5.20	0.15	0.11	10.37	600
19	12.46	0.05	0.05	4.47	81
20	2.75	0.08	0.05	2.45	35
21	8.85	0.29	0.17	17.61	428
22	3.07	0.07	0.04	2.30	93
	AthlonMP	Itanium2			

Table 4: TPC-H Performance (seconds)

any case, and taking into account that CPU-wise the DB2 hardware is more than four times stronger, MonetDB/X100 performance looks very solid.

5.1 Query 1 performance

As we did for MySQL and MonetDB/MIL, we now also study the performance of MonetDB/X100 on TPC-H Query 1 in detail. Figure 9 shows its translation into X100 Algebra. X100 implements detailed tracing and profiling support using low-level CPU counters, to help analyze query performance. Table 5 shows the tracing output generated by running TPC-H Query 1 on our Itanium2 at SF=1. The top part of the trace provides statistics on the level of the vectorized primitives, while the bottom part contains information on the (coarser) level of X100 algebra operators.

A first observation is that X100 manages to run all primitives at a very low number of CPU cycles per tuple - even relatively complex primitives like aggregation run in 6 cycles per tuple. Notice that a multiplication (`map_mul_*`) is handled in 2.2 cycles per tuple, which is way better than the 49 cycles per tuple achieved by MySQL (see Section 3.1).

A second observation is that since a large part of data that is being processed by primitives comes from vectors in the CPU cache, X100 is able to sustain a very high bandwidth. Where multiplication in MonetDB/MIL was constrained by the RAM bandwidth of 500MB/s, MonetDB/X100 exceeds 7.5GB/s on the same operator [6].

[6] On the AthlonMP it is around 5GB/s

```
Order(
  Project(
    Aggr(
      Select(
        Table(lineitem)
        < ( l_shipdate, date('1998-09-03'))),
      [ l_returnflag, l_linestatus ],
      [ sum_qty = sum(l_quantity),
        sum_base_price = sum(l_extendedprice),
        sum_disc_price = sum(
          discountprice = *( -(flt('1.0'), l_discount),
                             l_extendedprice ) ),
        sum_charge = sum(*( +( flt('1.0'), l_tax),
                            discountprice ) ),
        sum_disc = sum(l_discount),
        count_order = count() ]),
    [ l_returnflag, l_linestatus, sum_qty,
      sum_base_price, sum_disc_price, sum_charge,
      avg_qty = /( sum_qty, cnt=dbl(count_order)),
      avg_price = /( sum_base_price, cnt),
      avg_disc = /( sum_disc, cnt), count_order ]),
  [ l_returnflag ASC, l_linestatus ASC])
```

Figure 9: Query 1 in X100 Algebra

Finally, Table 5 shows that Query 1 uses three columns that are stored in enumerated types (i.e. l_discount, l_tax and l_quantity). X100 automatically adds three `Fetch1Join`s to retrieve the original values from the respective enumeration tables. We can see that these fetch-joins are truly efficient, as they cost less than 2 cycles per tuple.

5.1.1 Vector Size Impact

We now investigate the influence of vector size on performance. X100 uses a default vector size of 1024, but users can override it. Preferably, all vectors together should comfortably fit the CPU cache size, hence they should not be too big. However, with really small vector sizes, the possibility of exploiting CPU parallelism disappears. Also, in that case, the impact of interpretation overhead in the X100 Algebra next() methods will grow.

Figure 10 presents results of the experiment, in which we execute TPC-H Query 1 on both the Itanium2 and AthlonMP with varying vector sizes. Just like MySQL, interpretation overhead also hits MonetDB/X100 strongly if it uses tuple-at-a-time processing (i.e. a vector size of 1). With increasing vector size, the execution time quickly improves. For this query and these platforms, the optimal vector size seems to be 1000, but all values between 128 and 8K actually work well. Performance starts to deteriorate when intermediate results do not fit in the cache anymore. The total width of all vectors used in Query 1 is just over 40 bytes. Thus, when we start using vectors larger than 8K, the cache memory requirements start to exceed the 320KB combined L1 and L2 cache of the AthlonMP, and performance starts to degrade. For Itanium 2 (16KB L1, 256KB L2, and 3MB L3), the performance degradation starts a bit earlier, and then

input count	total MB	time (us)	BW MB/s	avg. cycles	**X100 primitive**
6M	30	8518	3521	1.9	map_fetch_uchr_col_flt_col
6M	30	8360	3588	1.9	map_fetch_uchr_col_flt_col
6M	30	8145	3683	1.9	map_fetch_uchr_col_flt_col
6M	35.5	13307	2667	3.0	select_lt_usht_col_usht_val
5.9M	47	10039	4681	2.3	map_sub_flt_val_flt_col
5.9M	71	9385	7565	2.2	map_mul_flt_col_flt_col
5.9M	71	9248	7677	2.1	map_mul_flt_col_flt_col
5.9M	47	10254	4583	2.4	map_add_flt_val_flt_col
5.9M	35.5	13052	2719	3.0	map_uidx_uchr_col
5.9M	53	14712	3602	3.4	map_directgrp_uidx_col_uchr_col
5.9M	71	28058	2530	6.5	aggr_sum_flt_col_uidx_col
5.9M	71	28598	2482	6.6	aggr_sum_flt_col_uidx_col
5.9M	71	27243	2606	6.3	aggr_sum_flt_col_uidx_col
5.9M	71	26603	2668	6.1	aggr_sum_flt_col_uidx_col
5.9M	71	27404	2590	6.3	aggr_sum_flt_col_uidx_col
5.9M	47	18738	2508	4.3	aggr_count_uidx_col
					X100 operator
0		3978			Scan
6M		10970			Fetch1Join(ENUM)
6M		10712			Fetch1Join(ENUM)
6M		10656			Fetch1Join(ENUM)
6M		15302			Select
5.9M		236443			Aggr(DIRECT)

Table 5: MonetDB/X100 performance trace of TPC-H Query 1 (Itanium2, SF=1)

decreases continuously until data does not fit even in L3 (after 64K x 40 bytes).

When the vectors do not fit in any cache anymore, we are materializing all intermediate results in main memory. Therefore, at the extreme vector size of 4M tuples, MonetDB/X100 behaves very similar to MonetDB/MIL. Still, X100 performance is better since it does not have to perform the extra `join` steps present in MIL, required to project selected tuples (see Section 3.2).

6 Related Work

This research builds a bridge between the classical Volcano iterator model [10] and the column-wise query processing model of MonetDB [4].

Apart from formalizing the iterator model of query processing, Volcano also generalized various forms of parallel query processing [20], where one of the possible strategies is to dedicate a seperate process to each query operator. This differs from our approach in the sense that we reduce overhead by making a process spend *significant* time in each operator for each query processing iteration (i.e. processsing a vector of tuples rather than a single one).

The work closest to our paper is [14], where a *blocked* execution path in DB2 is presented. Unlike MonetDB/X100, which is designed from the ground up for vectorized execution, the authors only use their approach to enhance aggregation and projection operations. In DB2, the tuple layout remains NSM, al-

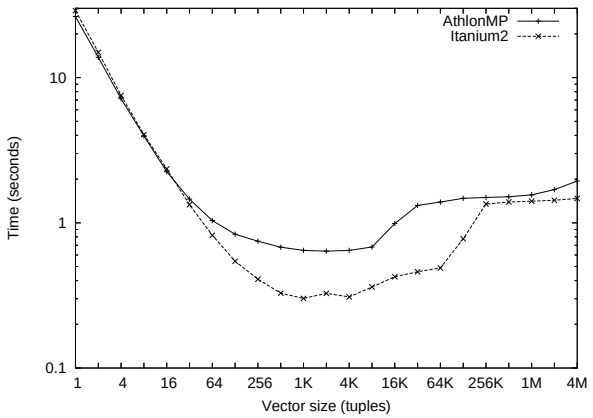

Figure 10: Query 1 performance w.r.t. vector-size

though the authors discuss the possibility to dynamically remap NSM chunks into vertical chunks dynamically.

Also closely related is [22], which also suggests block-at-a-time processing, again focusing on NSM tuple layouts. The authors propose to insert "Buffer" operators into the operator pipeline, which call their child N times after each other, buffering the results. This helps in situations where the code-footprint for all operators that occur in a query tree together exceed the instruction cache. Then, when the instructions of one operator are "hot" it makes sense to call it multiple times. Thus, this paper proposes to do block-wise processing, but without modifying the query operators to make them work on blocks. We argue that if our approach is adopted, we get the instruction cache benefit discussed in [22] for free. We had already noticed in the past, that MonetDB/MIL due to its column-wise execution spends so much time in each operator that instruction cache misses are not a problem.

A similar proposal for block-at-a-time query processing is [21], this time regarding lookup in B-trees. Again the goals of the authors are different, mainly better use of the data caches, while the main goal of MonetDB/X100 is to increase the CPU efficiency of query processing by loop pipelining.

As far as data storage is concerned, the update scheme of MonetDB/X100 combines the decomposed storage model (DSM) [8], with PAX [2] for tuples that are updated. This idea is close to the suggestion in [15] to combine DSM and NSM for more flexible data mirroring, and use of inverted lists to handle updates efficiently. In fact, a PAX block can be seen as a collection of vertical vectors, such that X100 could run right on top of this representation, without conversion overhead.

7 Conclusion and Future Work

In this paper, we investigate why relational database systems achieve low CPU efficiencies on modern CPUs. It turns out, that the Volcano-like tuple-at-a-time execution architecture of relational systems introduces interpretation overhead, and inhibits compilers from using their most performance-critical optimization techniques, such as loop pipelining.

We also analyzed the CPU efficiency of the main memory database system MonetDB, which does not suffer from problems generated by tuple-at-a-time interpretation, but instead employs a column-at-a-time materialization policy, which makes it memory bandwidth bound.

Therefore, we propose to strike a balance between the Volcano and MonetDB execution models. This leads to pipelined operators that pass to each other small, cache-resident, vertical data fragments called vectors. Following this principle, we present the architecture of a brand new query engine for MonetDB called X100. It uses *vectorized primitives*, to perform the bulk of query processing work in a very efficient way. We evaluated our system on the TPC-H decision support benchmark with size 100GB, showing that MonetDB/X100 can be up to two orders of magnitude faster than existing DBMS technology.

In the future, we will continue to add to MonetDB/X100 more vectorized query processing operators. We also plan to port the MonetDB/MIL SQL-frontend to it, and fit it with a histogram-based query optimizer. We intend to deploy MonetDB/X100 in data-mining, XML processing and multimedia and information retrieval projects ongoing in our group.

We will also continue our work on the ColumnDB buffer manager. This work embodies our goal to make MonetDB/X100 scale out of main memory, and preferably achieve the same high CPU efficiencies if data is sequentially streamed in from disk instead of from RAM. Therefore, we plan to investigate lightweight compression and multi-query optimization of disk access to reduce I/O bandwidth requirements.

Finally, we are considering the use of X100 as an energy-efficient query processing system for low-power (embedded,mobile) environments, because it has a small footprint, and its property to perform as much work in as few CPU cycles as possible, translates to improved battery life in such environments.

References

[1] *The STREAM Benchmark: Computer Memory Bandwidth*. `http://www.streambench.org`.

[2] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, Rome, Italy, 2001.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. VLDB*, Edinburgh, 1999.

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

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

[6] Q. Cao, J. Torrellas, P. Trancoso, J.-L. Larriba-Pey, B. Knighten, and Y. Won. Detailed characterization of a quad pentium pro server running tpc-d. In *Proc. ICCD*, Austin, USA, 1999.

[7] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proc. SIGMOD*, Santa Barbara, USA, 2001.

[8] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, Austin, USA, 1985.

[9] R. Cornacchia, A. van Ballegooij, and A. P. de Vries. A case study on array query optimisation. In *Proc. CVDB*, 2004.

[10] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

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

[12] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proc. VLDB*, New York, USA, 1998.

[13] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), Apr. 1965.

[14] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proc. ICDE*, Heidelberg, Germany, 2001.

[15] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *Proc. VLDB*, Hong Kong, 2002.

[16] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proc. SIGMOD*, Madison, USA, 2000.

[17] K. A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS*, Madison, USA, 2002.

[18] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proc. VLDB*, Santiago, 1994.

[19] M. Stonebraker, J. Anton, and M. Hirohama. Extendability in POSTGRES. *IEEE Data Eng. Bull.*, 10(2):16–23, 1987.

[20] A. Wilschut, J. Flokstra, and P. Apers. Parallel Evaluation of Multi-Join Queries. In *San Jose, USA*, San Jose, CA, USA, May 1995.

[21] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proc. VLDB*, Toronto, Canada, 2003.

[22] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proc. SIGMOD*, Paris, France, 2004.