Architecture-Conscious Hashing

Marcin Zukowski Sándor Héman Peter Boncz

CWI
Kruislaan 413
Amsterdam, The Netherlands
{Firstname.Lastname}@cwi.nl

ABSTRACT

Hashing is one of the fundamental techniques used to implement query processing operators such as grouping, aggregation and join. This paper studies the interaction between modern computer architecture and hash-based query processing techniques. First, we focus on extracting maximum hashing performance from super-scalar CPUs. In particular, we discuss fast hash functions, ways to efficiently handle multi-column keys and propose the use of a recently introduced hashing scheme called *Cuckoo Hashing* over the commonly used bucket-chained hashing. In the second part of the paper, we focus on the CPU cache usage, by dynamically partitioning data streams such that the partial hash tables fit in the CPU cache. Conventional partitioning works as a separate preparatory phase, forcing materialization, which may require I/O if the stream does not fit in RAM. We introduce *best-effort partitioning*, a technique that interleaves partitioning with execution of hash-based query processing operators and avoids I/O. In the process, we show how to prevent issues in partitioning with *cache-line alignment*, that can strongly decrease throughput. We also demonstrate overall query processing performance when both CPU-efficient hashing and best-effort partitioning are combined.

1. INTRODUCTION

Hashing is one of the fundamental techniques used to implement query processing operators such as aggregation and join [2, 9]. For a long time, the major optimization for hashing in a DBMS was handling a situation when a hash-table did not fit in RAM. This research has been usually done within the context of join processing and resulted in algorithms like Grace Join [8] and Hybrid Hash Join [6]. Recently, in-memory partitioning [14] and data prefetching [3] techniques were proposed to improve main-memory performance for a hash-table that does not fit in the CPU cache. Especially the former technique can be seen as an application of the I/O-optimized hashing methods to the in-memory scenario. All these techniques focused on optimizing the disk or main memory access time, ignoring the CPU cost of hash-table processing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Proceedings of the Second International Workshop on Data Management on New Hardware (DaMoN 2006) June 25, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-466-9/06/25 ...$5.00.

MonetDB/X100 [1] is a novel DBMS introducing the idea of *vectorized in-cache* processing. The first term, *vectorized*, refers to the fact that all the processing is performed using simple functions working on arrays consisting of single attribute values. Such processing reduces the control-flow overhead and allows the code to be efficiently executed on modern CPUs. The second term, *in-cache*, means that the system concentrates on performing all the computation inside the CPU cache, minimizing the amount of main-memory traffic. These two features combined allow MonetDB/X100 to achieve high-performance in large-volume data processing tasks. However, the novel architecture brings a challenge of developing new data processing algorithms that follow the design principles of the system.

Outline. In this paper we investigate the design and implementation of the hash-based algorithms developed for MonetDB/X100. In particular, we concentrate on lookup-intensive scenarios, typical in an aggregation or a hash-join. First, in Section 2, we focus on efficient utilization of modern CPU resources during hash-table processing. We introduce the ideas of vectorized hash processing, compound key handling and a CPU-friendly Cuckoo Hashing implementation. Then, in Section 3 we show how to scale the high in-cache performance of our hashing routines to a main-memory scenario using the idea of *best-effort partitioning*. Additionally, we analyze the impact of cache associativity on the partitioning performance. Finally, Section 4 concludes the paper and discusses future work.

2. CPU-EFFICIENT HASHING

Accessing the hash table can be divided into three major parts: *(i)* hash-value computation, *(ii)* hash-table position lookup and possibly insertion, and *(iii)* modifying tuples stored in the hash-table. For simplicity, we initially explain our hashing techniques using a query which performs a simple duplicate removal, where step *(iii)* is absent and hash-table keys are unique. We will extend these techniques to more general scenarios in Section 2.6.

2.1 Vectorized Hash-Functions

The MonetDB/X100 database system achieves high CPU efficiency by means of *vectorized processing* [1]. It uses a Volcano-style iterator tree to implement the relational algebra operators. However, each `next()` call, these operator objects yield a *vector* of tuples, instead of a single tuple (which

is usual in most RDBMS implementations). Tuples inside the operator tree are represented as *vectors* – small arrays using vertical decomposition [4]. The vector size is tuned such that all vectors needed by a running query comfortably fit inside the CPU cache (typically 100-10000 tuples). The main benefit of the array layout (vertically decomposed) of tuples inside the query processor is that *primitive functions* need not be aware of the actual record format on disk, allowing them to be vectorized. These primitive functions perform the computational work for the relational operators, such as arithmetic, but also hashing.

Vectorized primitive functions are efficient because: *(i)* they are called only once per vector so function call overhead is amortized, and *(ii)* they express their work as a simple loop over aligned arrays. Modern compilers generate highly efficient code for this, as they can use *loop pipelining*. As a result, the primitives manage to keep multiple CPU processing pipelines busy and achieve high IPC (instructions-per-cycle) efficiency. For example, consider the primitives for hash computation:

```
void hash<TYPE>(unsigned int N, output[N]; <TYPE> input[N]) {
  for(i=0; i<N; i++)
    output[i] = HASH_TYPE(input[i]);
}
void rehash<TYPE>(unsigned int N, output[N]; <TYPE> input[N]) {
  for(i=0; i<N; i++) {
    h = HASH_TYPE(input[i]);
    output[i] ^= (h << 11) ^ (h >> 7);
} }
```

Such specialized hashing primitives are generated for each supported data type. The symbol `HASH_TYPE` is a macro that calculates an integer hash number for a value of a particular type. For simple data types, our hash functions perform a number of simple arithmetic operations (bit-shifts, XOR etc.). Including load/stores they consist of a 8-10 instruction sequence per tuple. On Itanium2 the `hash` and `rehash` primitives run at just 2.5 cycles per tuple (4.3 on Pentium4), thus achieving very high IPC.

As we must be able to support hashing on multi-column keys, the `rehash()` primitive combines computation of a hash number with mixing it with a previous hash number. Thus, to calculate a vector of hash numbers for a `<int,float>` key, one needs to first call `hash_int()` and then `rehash_float()`.

Note, that our primitives are not optimized for the hash-function quality, like *perfect hash-functions* [5]. Instead, they are designed for high computational efficiency. The rationale for this is optimizing the algorithms for the most-common case, treating the possibly increased number of *hash-function collisions* as special cases. This is in line with our previous research on compression [18], where most values were decompressed using optimized routines and a *patching* technique was used to fix the problematic values.

2.2 Bucket-Chained Hash Tables

Having hash-values computed, a hash-table lookup can be performed. The most commonly used hash data structure is the *bucket-chained hash table*, considered the simplest and fastest hash data structure for main-memory applications [12]. The top part of Figure 1 presents an example implementation, where a sparse `buckets` array stores pointers to the dense "data" part keeping all so-far seen keys. To insert a tuple, its hash value is first converted into a bucket number. Usually, tables with a size of a power of 2 are used, to allow for fast modulo computation using the

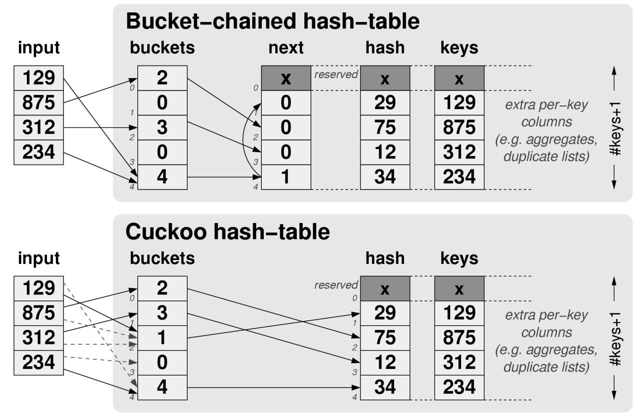

Figure 1: Data organization in bucket-chained and Cuckoo hash-tables. Hash function used is (`value mod 100`). For indexing we use (`hash mod 5`) (for both) and (`(hash div 10) mod 5`) (for Cuckoo only).

bitwise-AND operation. When different values map onto the same bucket, a *hash-table collision* occurs. The amount of collisions is strongly related to the *load factor* of the hash table, which is the amount of values stored in the hash table divided by the amount of buckets, following a binomial distribution. Even with a load factor of 1, the expected collision rate is $1/e$ (36%), and with smaller amounts of buckets it quickly rises.

In bucket-chained hashing all colliding tuples are stored in a linked list connected to each bucket (values 129 and 234 in our example). Hash lookup thus consists of computing a bucket, and traversing the collision list behind it until the found value matches the search key. The following routine takes a vector of hash values as input and produces a vector of group ids (indices into the `keys` array), with a zero-value for missing keys:

```
void lookup(unsigned int N, output[N], input[N];
            // the hash table
            unsigned int NBITS, NKEYS;
            unsigned int buckets[1 << NBITS];
            unsigned int next[NKEYS], hash[NKEYS])
{
  unsigned int buck, group_id;
  unsigned int mask = (1 << NBITS) - 1;
  for(int i=0; i<N; i++) {
     buck = input[i] & mask;
     group_id = buckets[buck];
     while(group_id != 0 && hash[group_id] != input[i])
       group_id = next[group_id]; /* follow linked list */
     output[i] = group_id;
} }
```

Note, that the input hash values are also used for comparison against values stored in the hash table. This is further explained in Section 2.4.

Regrettably, the code pattern of traversing a linked list is known to limit IPC, as it introduces both control and data dependencies. Perfect hash functions could help in avoiding collisions, but their computational cost makes them inapplicable in our system. In the next section we present another way of avoiding the linked list traversal during hash lookup.

2.3 Cuckoo Hashing

Our proposed solution is based on a recently introduced dynamic hash-table algorithm called *Cuckoo Hashing* [15]. It

does not require in-advance knowledge of the data distribution, and a hash table can be constructed incrementally, at a relatively low cost. The idea is to have two hash functions to get two possible buckets for each tuple. When inserting a new tuple, one of the positions is chosen. If that bucket is occupied, the old tuple is "kicked out" and put in its other position, possibly kicking out another tuple. The process continues until the insertion succeeds. In the example in the bottom part of Figure 1 the value 234 causes a conflict at offset 4 with 129, making it move to its second offset, 2. Since it is already occupied by 312, that value is again moved to offset 1, which is free. Note, that the data part of the hash-table is the same as with bucket-chained hashing, making the further processing steps identical.

It may happen that Cuckoo Hash gets into a loop and cannot insert a certain value. It was shown that in practical situations with a load factor of ≤ 0.5, this is highly unlikely [15]. Still, the chain search process can take a very long time (again, with a small probability). To overcome this problem, our insertion routine limits the depth of this search, and store problematic values in a separate linked list, that needs to be checked for all misses. As in practice this linked list is empty or contains only a few tuples, the extra overhead is negligible.

The Cuckoo Hash lookup function needs to check two positions as indicated by its two hash functions. Since the size of the hash table is usually significantly smaller than the range of the hash function, we take two disjunct parts of the same hash value as our bucket positions. The following is a straightforward implementation of Cuckoo Hashing:

```
void cuckoo_lookup( ..same as before.. ) {
  for(int i=0; i<N; i++) {
    // find the possible locations
    buck1 = input[i] & mask ;            // use different parts
    buck2 = (input[i] >> NBITS) & mask ; // of the hash number
    idx1 = buckets[buck1]; // 0 for empty buckets,
    idx2 = buckets[buck2]; // 1+ for non empty
    // check which one matches
    if (idx1 && hash[idx1] == input[i])
      group_id = idx1; // first position matches
    else if (idx2 && hash[idx2] == input[i])
      group_id = idx2; // second position matches
    else
      group_id = 0;    // nothing matches, mark as a miss
    output[i] = group_id;
} }
```

Since the searched key can be either in the first, in the second or in neither of the locations, the `if-then-else` branches in the second part of the algorithm are hard to predict by the *branch-predictor*, thereby resulting in suboptimal performance. To overcome this problem, we replace these branches with some integer arithmetic, similar as in [16]. This converts the *control dependency* into a *data dependency*, which is less limiting on CPU instruction throughput:

```
    ...
    // check which one matches
    mask1 = -(input[i] == hash[idx1]); // 0xFF..FF for a match,
    mask2 = -(input[i] == hash[idx2]); // 0 otherwise
    group_id = mask1 & idx1 | mask2 & idx2; // at most 1 matches
    output[i] = group_id;
    ...
```

Thanks to this optimization, the Cuckoo Hash lookup can be fully *vectorized*, as each iteration of the loop *(i)* is independent of the previous iterations, *(ii)* contains no function calls, *(iii)* no if-then-else, and *(iv)* no nested loops. Such loops exhibit maximal instruction independence, can be loop-pipelined by compilers, and obtain high IPC on modern CPUs.

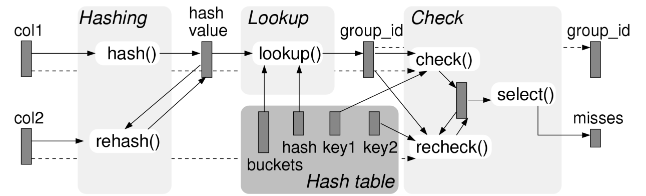

Figure 2: Hash table lookup for compound keys

2.4 Compound Keys and Miss Detection

During the search in both hash-table implementations, the values stored in the hash table need to be compared against the input values. In case of compound (multi-attribute) keys, it is impossible to generate a specialized primitive function for each combination of possible data types. One solution is to pass an array of type-specific comparison functions and in each search phase perform multiple comparisons. However, the cost of an extra nested loop and function calls would dramatically reduce the performance.

Our approach to that problem is based on the observation that while *hash-table collisions* are quite likely, the probability of a *hash-function collision* is orders of magnitude lower. As Figure 2 shows, during the hash-table lookup process we compare the hash values instead of the actual keys. However, this solution can result (again, with a low probability) in a *false hit*, if two key values have the same hash value. To overcome it, a special validation step is required, which compares actual key values from the input and a hash table. It uses a series of type-specific routines, one for each key attribute: [1]

```
void check<TYPE>(unsigned int N, NKEYS, group_id[N];
                 bool output[N];
                 <TYPE> input[N], keys[NKEYS]) {
  for(int i=0; i<N; i++)
    output[i] = (input[i] != keys[group_id[i]]);
}
void recheck<TYPE>(..same parameters..) {
  for(int i=0; i<N; i++)
    output[i] |= (input[i] != keys[group_id[i]]);
}
```

A list of hard cases (misses or hash-function collisions) is finally produced with the `select()` primitive:

```
unsigned int select(unsigned int N, output[N]; bool input[N]) {
  for(int nsel=i=0; i<N; i++)
    if (input[i]) output[nsel++] = i;
  return nsel;
}
```

The final output is a `group_id` vector with the offsets for each key, and a vector of selected problematic input values. The ones with group-ID 0 are search misses, and have to be inserted in the hash table. The other cases are hash-function collisions, which require a slower but precise search function call. In the end, the `group_id` vector will contain hash table indices for all the input tuples. They can later be used for additional processing, as discussed in Section 2.6.

A final observation to be made is that hard cases are expected to be rare. Thus, the `select()` will in general only

[1] The hash table slot 0 (used to mark search misses) represents "impossible" values. Thus, `check()` will identify these misses as a hard case.

Table 1: **Hash Lookup Performance for selecting distinct single-column keys (20M tuples, 512 buckets).**

primitive	**Pentium4** (cycles/tuple)	**Itanium2** (cycles/tuple)
hash (per attribute)	4.3	2.5
cuckoo lookup	25.2	6.7
bucket-chained lookup		
load-factor 0.50	23.2	13.2
load-factor 0.75	34.1	16.3
load-factor 1.00	38.7	17.8
	(only for multi-attribute keys)	
check (per attribute)	6.7	5.4

select a low percentage of tuples (and often will select nothing). This can be exploited by a special version of the selection primitive that treats the vector of byte-sized boolean values as a vector of 64-bits integers with only one eighth of the entries, making a sub-selection. Only for that sub-selection a byte-by-byte-check is done. This implementation of `select()` reduces the cost of selection by roughly a factor 8, such that its cost drops below 1 cycle per tuple. We thus ignore the cost of selecting the hard cases in the remainder.

2.5 Micro-Benchmarks

Table 1 shows lookup performance on our 3GHz Pentium4 Xeon (16KB L1, 1MB L2) and 1.3GHz Itanium2 (16KB L1, 256KB L2. 3MB L3) test platforms, both with 4GB RAM. The performance of bucket-chained hashing degrades when the load factor rises, as the average length of the collision list increases. Since Cuckoo Hashing does not use the linked list, its performance does not depend on the load factor. Moreover, since the `next` array is obsolete, it requires less memory. Therefore, with respect to memory consumption, a load-factor of 1.00 in bucket-chaining corresponds with a load factor of 0.50 in Cuckoo Hash (its maximum – otherwise its failure rate increases).

On Pentium4, Cuckoo beats bucket-chained hashing above 50% fill-ratio. On Itanium2, however, it beats bucket-chained hashing in all situations, by up to a factor 3. The large difference between Itanium2 and Pentium4 occurs because the former architecture can achieve high IPC (we get above 5 here) when abundant instruction independence is present in the code, whereas the Pentium4 is notorious for staying below 2 despite this. Thus, Cuckoo Hashing clearly favors wide-issue CPU architectures (such as Itanium2, but also the new Intel Core CPU generation).

2.6 Extensions

Simple Keys. The presented algorithms can be easily optimized for single-attribute keys. Since all the functions are type-specialized, it is possible to express the whole vectorized hash lookup in one primitive. The main reason to do so is to eliminate the cost of accessing intermediate results between the primitives. Additionally, a single-attribute comparison can be directly used, making the `hash` part of the hash-table and the check phase obsolete.

Aggregation. The result of a hash lookup is a `group_id` vector with offsets in the data part of the hash table. For aggregation, extra columns can be added there for storing the current results of aggregate functions. Having `group_id` together with the input data, these can be easily updated.

Hash-join. To use the described data structures for a hash-join, one simply needs to add an extra column in the data part for the non-key attributes. If the join-key is not unique, an extra data structure is necessary to store lists of values with the same key.

3. CACHE-OPTIMIZED HASHING

The CPU-efficient hashing algorithms described in the previous section can only achieve high performance when they operate on data stored in the CPU cache, as the main-memory access cost is an order of magnitude higher than the entire per-tuple processing time. Two main techniques were proposed to improve the hash-table performance in a main-memory scenario.

The first technique, proposed by Chen et al., uses explicit *memory prefetching* instructions inside the hash lookup routine [3]. This transforms hash-lookup throughput from a memory latency-limited into a memory bandwidth-limited workload, which can strongly improve overall hash-join performance. Our CPU-optimized hashing, however, has become too fast for memory bandwidth. Given that a hash lookup takes 7 cycles and touches at least two cache lines, on a 1.3GHz CPU this implies bandwidth usage of 24GB/s, which exceeds RAM bandwidth available in current hardware (4-6GB/s). For that reason, we employ the second technique, based on *hash-table partitioning*. This idea was originally introduced for I/O based hashing in Grace Join [8] and Hybrid Hash Join [6] algorithms. More recently, with Radix-Cluster [14], this work has been extended to hash-partitioning into the CPU cache.

The problem with these partitioned hashing techniques is that all the data needs to be first fully partitioned, and only then processed [9]. This works fine in the disk-based scenario, as the temporary space for the partitions is usually considered unlimited. Main memory capacity, however, can not be assumed to be unlimited, meaning that if the data does not fit in RAM during partitioning, it has to be saved to disk. Since using the disk when optimizing for in-cache processing is reasonable only in extreme scenarios, we propose a new hash partitioning algorithm that, while providing in-cache processing, prevents spilling data to disk.

3.1 Best-Effort Partitioning

Best-effort partitioning (BEP) is a technique that interleaves partitioning with execution of hash-based query processing operators without using I/O. The key idea is that if the available partition memory is filled, data from one of the partitions is passed on to the processing operator (aggregation, join), freeing space for more input tuples. In contrast to conventional partitioning, BEP is a pipelinable operator that merely reorders the tuples in a stream so that many consecutive tuples come from the same partition. Operators that use BEP, like Partitioned Hash Join and Partitioned Hash Aggregation, create a separate hash table per partition, and detect which hash table should be used at a given moment looking at the input tuples. When one of the hash tables is active, the operations on it are performed for many consecutive tuples, hence the cost of loading the hash-table into the cache is amortized among them.

Algorithm 1 presents an implementation where each partition consists of multiple *buffers*. When no more buffers are available, we choose the biggest partition to be processed, for two reasons. Firstly, it frees most space for the incoming

Algorithm 1 Best-Effort Partitioning (BEP)

```
InitBuffers(numBuffers)
while tuple = GetNextTuple(child) do
  p ← Hash(tuple) mod numPartitions
  if MemoryExhausted(p) then
    if NoMoreBuffers() then
      maxp ← ChooseLargestPartition()
      ProcessPartition(maxp)
      FreeBuffers(maxp)
    end if
    AddBuffer(p)
  end if
  Insert(p,tuple)
end while
for p in 0..numPartitions − 1 do
  ProcessPartition(p)
  FreeBuffers(p)
end for
```

tuples. Secondly, with more tuples passed for processing, the time of loading the hash-table is better amortized due to increased cache-reuse

3.2 Partitioning and Cache Associativity

The main-memory performance of data partitioning algorithms, with respect to the number of partitions, number of attributes, sizes of the CPU cache and TLB. has been studied in [14] and [13]. However, to our knowledge, one other important property of modern cache memories has been ignored so far: cache associativity.

A typical cache memory is organized as a collection of cache-lines. For example, a simple 64 KB cache with 64-byte cache-lines consists of 1024 cache-lines. If the cache is *fully-associative*, the global LRU policy guarantees that the most recently used lines will be kept in the cache. However, even on this small scale, the LRU policy turns out to be too expensive. Instead, the CPU manufactures either use *N-way associative* or *directly mapped* (1-way) caches. For each memory address there is a set of N cache lines that this address can go to, and a LRU replacement policy is used within this set. As Figure 3 presents, the cache-line offset is usually determined by using the lowest bits of the memory address. Due to associativity, the meaningful set of bits of the memory address is decreased, by 2 in our example case.

Partitioning Primitive. Like all MonetDB/X100 query processing operators, the functionality of best-effort partitioning is provided by a primitive function, that partitions a vector of tuples in the following inner loop:

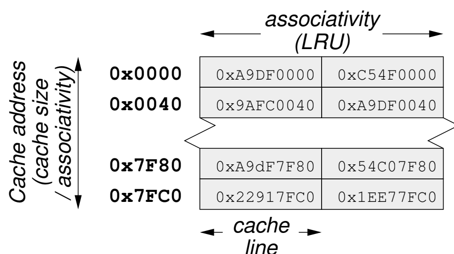

Figure 3: Organization of a 64 kilobyte 2-way associative cache memory with 64-byte cache-lines

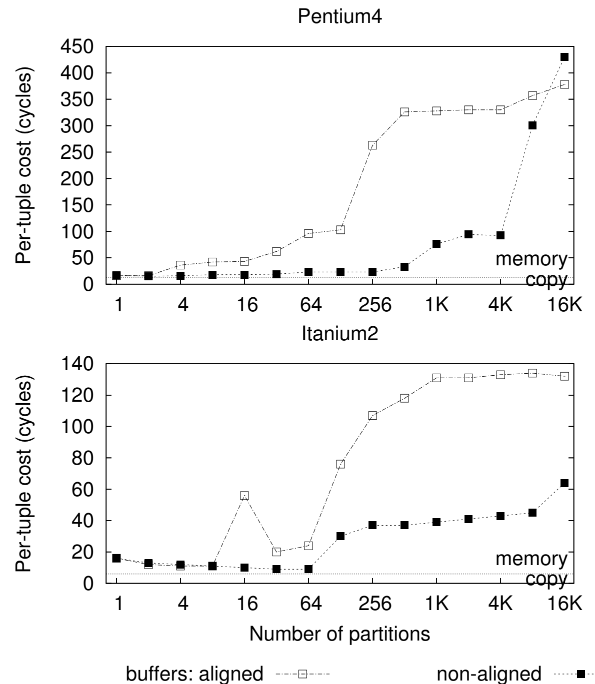

Figure 4: Impact of number of partitions and buffer allocation method on partitioning performance on various hardware architectures

```
for(i=0; i<n; i++) {
  partno = HASH_TYPE(src[i]) & PARTITION_MASK;
  dst[partno][counts[partno]++] = src[i];
}
```

It is a common situation that the addresses of these `dst[p]` buffers are aligned to the page size. As a result, using the cache from Figure 3 and a page size of 8KB, all these addresses will map onto only 4 separate cache addresses, holding 2 cache-lines each. That means, that if we partition into more than 8 buffers, there is a high probability that, when we refer to a buffer that has been recently used, the cache-line with its data has already been replaced, possibly causing a cache-miss. Since the partitioning phase is usually performed using hash-values, data is roughly uniformly distributed among partitions. As a result, this *cache associativity thrashing* may continue during the entire execution of this primitive. Since the previous experiments with Radix-Cluster [14] were primarily performed on a computer architecture where high fan-out partitioning deteriorated due to slow (software) TLB miss handling, these issues had previously not been detected. A simple solution for this problem is to shift each buffer address with a different multiple of a cache line size, such that all map to different cache offsets.

Figure 4 presents the performance of the partitioning phase with both aligned and non-aligned buffers on Pentium Xeon and Itanium2 CPUs As the number of partitions grows, the performance of aligned buffers goes down, quickly approaching the cost of random-memory access per each tuple. The non-aligned case, on the other hand, manages to achieve speed comparable to simple memory-copying even for 256 partitions. When more partitions are needed, it is possible to use a multi-pass partitioning algorithm [14]. BEP can be easily extended to handle such a situation.

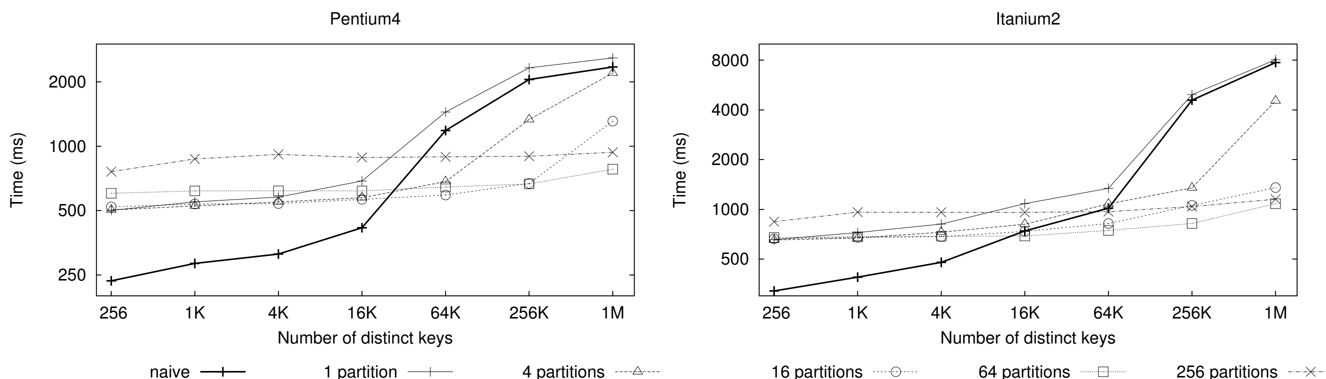

Figure 5: Aggregation Performance with varying number of partitions and distinct keys (20M tuples)

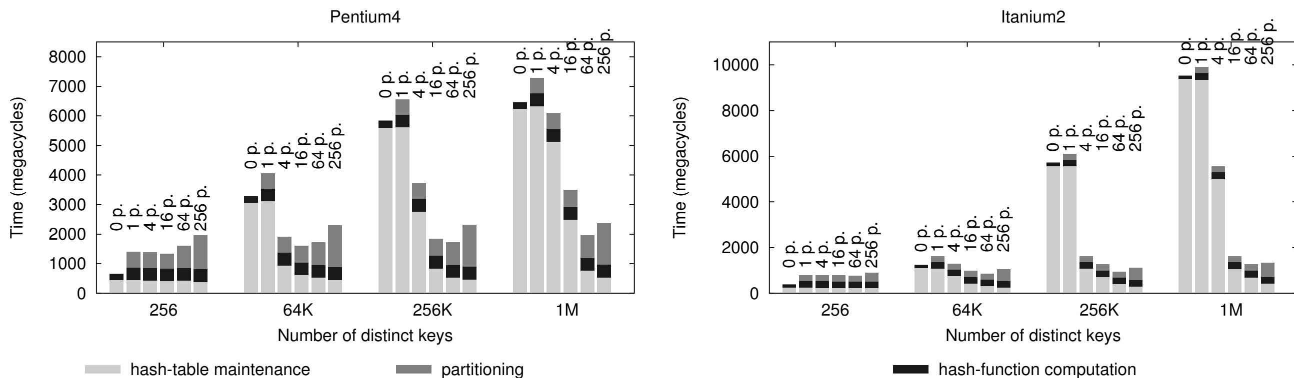

Figure 6: Execution profiling with varying number of partitions and distinct keys (20M tuples)

3.3 BEP Performance

Performance of hash processing with best-effort partitioning is influenced by a number of factors presented in Table 2. The first group, data and query properties define the number of tuples stored in a hash table and their width, determining a size of the hash table. The second group, partitioning settings, determine the size of per-partition hash tables. Finally, the hardware factors influence the recommended size of the small hash tables, hence the partitioning fan-out. Moreover, cache and memory latencies influence the desirable cache-reuse factor, which determines the amortized cost of data access.

We now discuss in detail one particular scenario of using BEP for partitioned hash aggregation. This setting is later used in experiments on our Itanium2 machine. The relevant hardware and algorithm parameters are listed in Table 2, which in its rightmost column also contains the specific hardware characteristics of Itanium2. Note that Itanium2 has a large and fast L3 cache, which is the optimization target (in case of Pentium4, it is best to optimize for L2).

Example Scenario. Assume we need to find 1M unique values in a 20M single-attribute, 4-byte long tuples using 50MB of RAM on our Itanium2 machine with a 3MB L3 cache with 128-byte cache-lines. A hash table with a load factor of 0.5 occupies 20MB using optimized single-column Cuckoo Hashing: 16MB for the bucket array and 4MB for the values. Using 16 partitions will divide it into 1.25MB (cache-resident) hash-tables. There will be 30MB of RAM left for partitions, and assuming uniform tuple distribution (which is actually the worst case scenario for our algorithm), the largest partition during overflow occupies 1.875MB, holding 480K 4-byte tuples. Thus, when this partition is processed, 480K keys are looked-up in a hash-table, using 4 random memory accesses per-tuple, resulting in 1875K accesses. Since the hash table consists of 10240 128-byte cache lines, each of them will be accessed 188 times. With main-memory and (L3) cache latencies of 201 and 14 cycles, respectively, this results in an average access cost of 15 cycles.

Experiments. Figure 5 compares in a micro-benchmark naive (non-partitioned) and best-effort partitioning hash aggregation, in a `"SELECT DISTINCT key FROM table"` query on a 20M 4-byte wide tuples table, with a varying number of distinct keys. When this number is small, the hash table fits in the CPU cache, hence the partitioning only slows down execution. When the number of keys grows, the hash table exceeds the cache size, and best-effort partitioned execution quickly becomes fastest. Figure 6 shows a performance break-down into partitioning cost, hash table maintenance (lookup and inserts) and hash function computation. With more partitions, the data locality improves, making the hash table maintenance faster. On the other hand, more partitions result in a slower partitioning phase. Finally, we see that with partitioned execution the cost of the hash-function is two times higher, as it is computed both in partitioning and lookup phases. Depending on the cost of computing this function (especially when it is computed over multiple

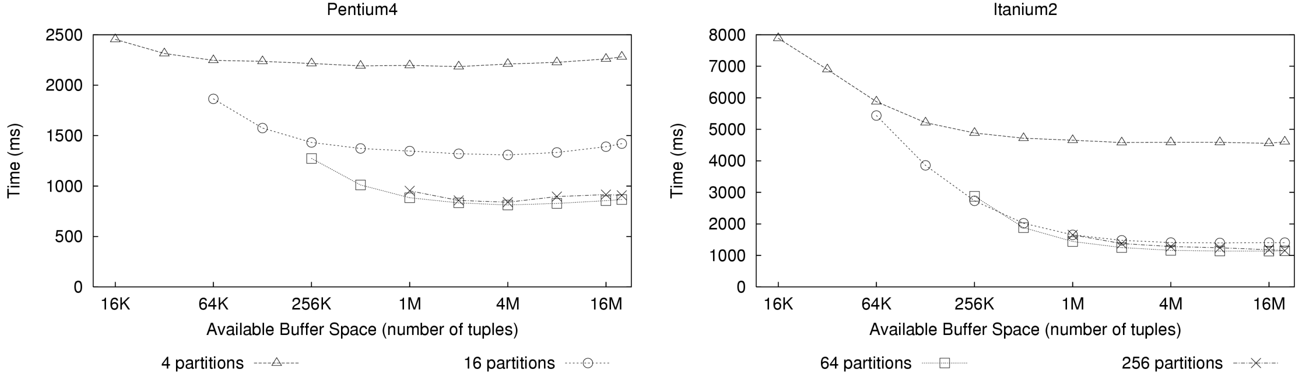

Figure 7: Impact of Available Buffer Space (20M tuples, 1M unique values)

Table 2: Best-Effort Partitioning parameters

Description	Symbol	Example
Query properties		
Number of distinct values	D	1 M
Number of tuples	T	20 M
Input width	$\widehat{i}$	4 B
Hash-table: data width	$\widehat{h_d}$	4 B
Hash-table: buckets width	$\widehat{h_b}$	8 B
Hash-table: per-key memory = $\widehat{h_d} + 2 \cdot \widehat{h_b}$ (Cuckoo, 50% fill ratio)	$\widehat{h_w}$	20 B
Hash-table: size = $D \cdot \widehat{h_w}$	$\|H\|$	20 MB
BEP settings		
Available buffer memory	$\|M\|$	30 MB
Number of partitions	P	16
Partition: size = $\frac{\|M\|}{P}$	$\|M_p\|$	1.875 MB
Partition: tuples buffered = $\frac{\|M_p\|}{\widehat{i}}$	T_p	480 K
Partition: hash-table size = $\frac{\|H\|}{P}$	$\|H_p\|$	1.25 MB
Number of per-lookup random accesses (Cuckoo)	a	4
Hardware properties (Example = Itanium2)		
Cache size	$\|C\|$	3 MB
Cache line width	$\widehat{C}$	128 B
Cache latency	l_C	14 cycles
Main-memory latency	l_M	201 cycles

attributes), it can be more beneficial to store it during partitioning and reuse it during lookup.

The performance of partitioned execution depends highly on the cache-reuse ratio during one processing phase, which in turn depends on the amount of buffer space. As Figure 7 shows, with an increasing number of buffered tuples, performance improves since more tuples hit the same cache line. If the number of partitions is big enough to make the hash table fit in the cache, adding more partitions does not change performance given the same buffer space. Finally, we see that the performance curve quickly flattens, showing that the performance can be close to optimal with significantly lower memory consumption. In this case, processing time with a buffer space of only 2M tuples is the same as with 20M tuples (which is equivalent to full partitioning). We see this reduced RAM requirement as the main advantage of best-effort partitioning.

Cost Model. We now formulate a cost model to answer the question what is the amount of buffer memory that should be given to BEP to achieve (near) optimal performance.

The cost of the amortized average data access cost during hash-table lookup depends on the cache-reuse factor:

$$access_cost = l_C + \frac{l_M}{reuse_factor}$$

The cache-reuse factor is the expected amount of times a cache line is read while looking up in the hash table all tuples from a partition. It can be computed looking at the query, partitioning and hardware properties from Table 2:

$$reuse_factor = \frac{T_p \cdot a \cdot \widehat{C}}{|H_p|} = \frac{|M| \cdot a \cdot \widehat{C}}{\widehat{i} \cdot D \cdot \widehat{h_w}}$$

A good target for the cache-reuse factor is to aim for an amortized RAM latency close to the cache performance, for example 25% higher:

$$\frac{l_M}{reuse_factor} = \frac{l_C}{4}$$

This, in turn, allows us to compute the required amount of memory BEP needs:

$$|M| = \frac{l_M \cdot 4 \cdot \widehat{i} \cdot D \cdot \widehat{h_w}}{l_C \cdot a \cdot \widehat{C}}$$

In the case of our Itanium2 experiments we arrive at:

$$|M| = \frac{201 \cdot 4 \cdot 4 \cdot 1M \cdot 20}{14 \cdot 4 \cdot 128} = 9{,}409{,}096 \text{ B} = 2{,}352{,}274 \text{ tuples}$$

and in case of the Pentium 4:

$$|M| = \frac{370 \cdot 4 \cdot 4 \cdot 1M \cdot 20}{24 \cdot 4 \cdot 128} = 10{,}103{,}464 \text{ B} = 2{,}525{,}866 \text{ tuples}$$

This prediction is confirmed in Figure 7, where a buffer of 2M tuples results in the optimal performance.

As a final observation, it is striking that the amount of partitions does not play a role in the formula. The cost model does assume, though, that the hash table fits in the CPU cache. This once again is confirmed in Figure 7, which shows that once partitions are small enough for them to fit in the CPU cache, performance does not change. Note that on Pentium4, the 16 partition line is in the middle, because at that setting the hash-tables (20MB/16 = 1.25MB) are just a bit too large to fit L2, but average latency has gone down with respect to pure random access.

3.4 BEP Discussion

Best-effort partitioning can be easily applied to all relational operations. In aggregation, the ProcessPartition() function simply incrementally updates the current aggregate results. In joins and set-operations, the partitioning can first be used for the build relation, and then for the probe relation, using the same small hash tables. This allows, for example, cache-friendly joining of two relations if only one of them fits in main memory. This can be further extended to multi-way joins using *hash teams* [10].

The flexibility of BEP memory requirements is useful in a scenario where the memory available for the operator changes during its execution. If the buffer manager provides BEP with extra memory, it can be simply utilized as additional buffer space. If, on the other hand, available memory is reduced, BEP only needs to pass some of the partitions to the processing operator and free the buffers they occupied.

The ideas behind BEP can be applied in a scenario with more storage levels. For example, it is possible that the hash-table does not fit in main memory, and the partitioned data is too large to fit on disk, calling for the use of tertiary storage [17], e.g. a magnetic tape. In this situation, BEP can be used to buffer the data on disk and periodically process memory-size hash tables, again using BEP to make it cache-friendly. This scenario raises the question whether it is possible to build a *cache-oblivious* data structure [7] with properties similar to those of BEP.

BEP is related to a few other processing techniques besides vanilla data partitioning. *Early aggregation* [11] allows computing aggregated results for part of the data and later join combine them. In parallel *local-global aggregation* [9], tuples can be distributed using hash-partitioning among multiple nodes. If the combined memory of these nodes is enough to keep the whole hash table, I/O-based partitioning is not necessary. In *hybrid hashing* [6], the effort is made to keep as much data in memory as possible, spilling only some of the partitions to disk. While there are clearly similarities between BEP and these techniques, BEP provides a unique combination of features: *(i)* it allows efficient processing if the data does not fit in the first-level storage (cache), *(ii)* it optimizes data partitioning for a limited second-level storage (main memory), *(iii)* it allows a non-blocking partitioning phase, and, finally, *(iv)* it can be easily combined with dynamic memory adjustments.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we have taken an in-depth look at the interaction between computer architecture and hashing in database operators. This work was done in the context of the MonetDB/X100 system that uses a vectorized in-cache processing model. We first explained how a bucket-chained hash table can be employed in such an engine and why its linked-list traversal violates the spirit of vectorized processing. Then we presented a vectorized version of the recently introduced Cuckoo Hash algorithm, that improved the lookup performance by a factor 3 on Itanium2 systems.

In the second part of the paper we turned our attention to hash partitioning, which is required for scalable hashing algorithms. We proposed a new hash partitioning variant, called *best-effort partitioning*, that can be used in a pipelined fashion and provides the same performance as traditional full partitioning, with significantly lower memory requirements. On the way, we exposed CPU cache associativity thrashing as a threat to partitioning performance, and proposed a simple solution for this.

In the future we plan to study the performance of presented algorithms in large-scale scenarios, e.g. on a TPC-H benchmark. Furthermore, we will investigate novel hashing schemes and their properties when applied in database scenarios, both during hash-table based operator execution and value lookup during data (de) compression.

5. REFERENCES

[1] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[2] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. VLDB*, Singapore, 1984.

[3] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proc. ICDE*, Boston, MA, USA, 2004.

[4] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, Austin, TX, USA, 1985.

[5] Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.

[6] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proc. SIGMOD*, Boston, MA, USA, 1984.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. FOCS*, New York, NY, USA, 1999.

[8] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *Proc. VLDB*, Kyoto, Japan, Aug. 1986.

[9] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[10] G. Graefe, R. Bunker, and S. Cooper. Hash joins and join teams in Microsoft SQL Server. In *Proc. VLDB*, New York, USA, 1998.

[11] P.-A. Larson. Grouping and duplicate elimination: Benefits of early aggregation. Technical Report MSR-TR-97-36, Microsoft, December 1997.

[12] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proc. VLDB*, Kyoto, Japan, 1986.

[13] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-Conscious Radix-Decluster Projections. In *Proc. VLDB*, Toronto, Canada, 2004.

[14] 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.

[15] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[16] K. A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS*, Washington, DC, USA, 2002.

[17] S. Sarawagi. Query processing in tertiary memory databases. In *Proc. VLDB*, Zurich, Switzerland, 1995.

[18] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE*, Atlanta, GA, USA, 2006.