INVITED PAPER

Database Optimizations for Modern Hardware

Techniques that make better use of memory and parallelism in evolving computer hardware can increase the performance of databases and other data-intensive applications.

By John Cieslewicz and Kenneth A. Ross

ABSTRACT | Databases are an important workload for modern commodity microarchitectures. Achieving the best performance requires that careful attention be paid to the underlying architecture, including instruction and data cache usage, data layout, branch prediction, and multithreading. Specialized commodity microarchitectures, such as graphics cards and network processors, have also been investigated as effective query coprocessors. This paper presents a survey of recent architecture-sensitive database research. The insights gained from optimizing database performance on modern microarchitectures are also applicable to other domains, particularly those that are similarly data intensive.

KEYWORDS | Computer architecture; computer performance; database management systems; database performance; database query processing

I. INTRODUCTION

Database management systems provide efficient, durable, and structured storage of information. From the advent of these systems, performance has been a core focus of database research because database systems are integral components of business processes, scientific endeavors, and mission-critical infrastructure. In this paper, we present a survey of recent research related to improving database systems on modern commodity architectures.

A. Query Processing

Databases manage structured data in the form of tables (relations) of records (tuples). Database users submit queries to the database system using a query language, such as SQL, that describes the question to be answered. An example of such a query is "SELECT Name FROM People WHERE Age < 50," which asks for the names of people who are under 50 years old. After a user specifies what the database should answer, the database system is left to determine the most effective way to answer a query. The performance of query execution on modern hardware is emphasized in this paper.

The query specified in a high-level language, such as the SQL example above, is translated into a query plan composed of multiple low-level database operators. These operators perform specific tasks on the data stored in the database tables, passing the result of each operation to the next operator in the plan. Some operators filter records based on a predicate, as in the example above. Examples of other operations include: 1) joining together data from multiple tables, 2) aggregating values to derive summaries (e.g., averages or sums), 3) sorting records, and 4) eliminating duplicate records.

Two types of workloads are discussed in this paper: online transaction processing (OLTP) and online analytical processing (OLAP). An OLTP workload consists of numerous transactions that insert, update, or delete data stored in the database. Examples of database transactions in everyday life include bank withdrawals and credit card purchases. Transactions in databases provide certain guarantees to applications that the data are permanently added, updated, or deleted. In the real-world example, this means that if the credit card purchase is approved, the credit card company will not lose the information about your purchase. Transactions typically involve only a small subset of the data in a database, but there may be a large volume of transactions processed concurrently.

OLAP queries, on the other hand, typically perform some sort of analysis to answer questions about the data stored in the database, for instance, summarizing large quantities of information by performing grouping and

Manuscript received March 30, 2007; revised October 28, 2007. This work was supported by the U.S. Department of Homeland Security under a Graduate Research Fellowship and by the National Science Foundation under Grant IIS-0524389.
The authors are with the Department of Computer Science, Columbia University, New York, NY 10027 USA (e-mail: johnc@cs.columbia.edu; kar@cs.columbia.edu).
Digital Object Identifier: 10.1109/JPROC.2008.917744

aggregation. A real-world example is a company analyzing the performance of its stores by aggregating the total sales made by each store over different time periods. In this example, all sales would be grouped by store and time period and then summed within those groups. In contrast to OLTP, OLAP queries often involve processing large amounts of the data stored in the database.

B. Database Performance Challenges

For years, the main performance issue facing databases was I/O cost, that is, writing and reading to and from a disk. The performance of database operators was measured by the amount and type of I/O required. As main memory sizes have rapidly increased along with microprocessor performance [1], it has become possible for many query-processing tasks to be memory resident. Memory-intensive applications, such as databases, have not experienced as much benefit from faster microarchitectures as compute-intensive applications. This is largely due to the fact that processor clock rates have improved much faster than memory latency [1]. When data must be loaded from memory, the processor often stalls. It has been observed that when running a database workload, modern processors spend most cycles waiting for memory accesses [2]. This observation has resulted in recent work focused on optimizing the performance of database operations on modern microprocessors with respect to memory accesses. In Section II, we will present techniques for improving data cache usage, thus reducing the number of accesses to memory. Improvements in data layout are discussed in Section III. Improving instruction cache usage is as important as data cache usage and is presented in Section IV. Long latency memory operations are not the only source of stalls for processors executing database workloads. Conditional branches can also cause processor stalls and degrade performance of database workloads. Section V will describe the problems with conditional branches in database workloads and demonstrate techniques that improve performance.

Increasingly fast clock speeds have recently become problematic because of power consumption and heat dissipation issues as well as diminishing returns associated with extracting more instruction-level parallelism (ILP) from instruction streams. This has resulted in a paradigm shift away from faster uniprocessors towards increased on-chip parallelism in the form of simultaneous multithreading and chip multiprocessors. Future growth in processor performance will depend on increased thread-level parallelism (TLP) rather than ILP [1]. The implications of increased TLP for database operations are discussed in Section VI. Database operations have also been investigated on special-purpose microarchitectures such as graphics cards and network processors. This work is presented in Section VII. We will conclude in Sections VIII and IX with a discussion of future directions for database optimization research on modern hardware.

II. DATA CACHE

Data-intensive applications, such as databases, have not fully utilized the rapid increase in microprocessor performance because memory latency has not improved at the same rate as processor speed. The number of processor cycles per memory access has increased, making a memory reference relatively more expensive on newer processors. Processors running database applications spend a significant amount of time stalled while waiting for data to be loaded from main memory, which can take hundreds of cycles on modern architecture. Ailamaki *et al.* identify data cache misses as a significant component of database execution time, exceeding 50% of the total execution time for some workloads and system configurations [2]. Because of frequent long-latency memory accesses, database operations fail to exhibit the high degree of ILP that modern processors are designed to exploit. Data-intensive applications, such as database workloads, have a lower ILP in part because they exhibit a large number of data dependencies. Having an instruction stream with many data dependencies means that even with aggressive out-of-order execution, the processor will not be able to find an instruction within a reasonably sized instruction window that does not depend on a pending memory request. When this occurs, no instruction can be issued and the pipeline stalls. This new bottleneck in memory performance has resulted in considerable research aimed at improving the ILP of database operations by eliminating some of the off-chip memory accesses, scheduling work to reuse data in the cache, organizing data to occupy the cache more effectively, and prefetching data into the cache ahead of when it is needed to avoid compulsory cache misses.

Before examining techniques used to improve data cache use for database workloads, we first provide a brief description of caches and their importance in modern computer architecture. A cache is a small but fast memory placed on the same die as the processor. The purpose of a cache is to help processors overcome the memory bottleneck described previously. Caches attempt to exploit temporal and spatial locality, making memory references to frequently used or collocated data faster. When an instruction loads data from memory, the processor first checks to see if that data is in the cache. If it is, this is called a cache hit, and the data is loaded from the cache, which is a low-latency operation. If the data is not found in the cache, called a cache miss, the data is loaded from memory, a much longer latency operation. Data is loaded into the cache in chunks called cache lines, which are often tens of bytes in size. By loading an entire cache line, the cache attempts to capture spatial locality. For instance, when processing an array, every entry is examined in sequence; therefore multiple array elements are likely located on the same cache line. Temporal locality is achieved via the cache replacement policy, which usually favors evicting the least recently used (LRU) data to make room for newly requested data. In this way, frequently

accessed data remain cache resident, while rarely accessed data do not. For an in-depth explanation of caching, consult a text devoted to hardware design, such as Hennessy and Patterson [1].

To provide an example of how important it is to take the cache into account when implementing database algorithms, consider the following simplistic example of an operation that scans the same array every time it processes a record. Assume that each record to be processed causes one cache miss when it is read and that there are n input records. Assume further that the array that must be read for each record occupies k cache lines and that the cache can hold $c > 0$ cache lines. If $k \leq c$, then after the first record is processed, the entire array will be found in the cache for all subsequent records. The total number of cache misses will be at most $n + c$. Now consider the case where the array is twice as large as the cache, i.e., $k = 2c$. When we finish reading the entire array, only the second half of the array is in the cache because of the cache's capacity limitation and the typical LRU cache replacement policy. This means that for every input tuple processed, $2c$ cache misses are incurred so the total cache misses for this operation will be $n + 2nc$. If the operation can be changed so that half of the array is used during one pass over the n input records and the other half during a second pass, then the number of misses would be reduced to $2(n + c)$. This example highlights how small changes in data sizes can cause significant changes in cache behavior.

A. Buffering Index Structures

Making effective use of data already in the cache is one part of implementing cache-conscious algorithms. An example of improving data reuse in a database is index structure buffering. Zhou and Ross observe that data structures such as tree-based indexes can exhibit cache thrashing because the entire tree does not fit in the cache [3]. Given that each probe to the tree has an equal chance of choosing any path, it is easy to see that cache reuse could be quite small if the tree is larger than the cache. To mitigate this cache-reuse problem, Zhou and Ross propose buffering accesses to the index structure (see Fig. 1) [3]. Nodes in the index tree are grouped together into pieces that fit within the cache. A buffer is placed at the root of a group. When a buffer of probes fills, all of the probes are then allowed to proceed through the group of nodes. The cache misses required to load this group of index nodes into the cache are amortized over a full buffer of probes. By buffering accesses to index structures, the performance of the index probing is improved by a factor of two to three [3].

B. Cache Conscious Data Structures

Designing data structures to reduce cache misses, which incur long latency cache misses, improves database performance. Rao and Ross propose cache-sensitive search (CSS) trees for read-only searching [4] and cache-sensitive

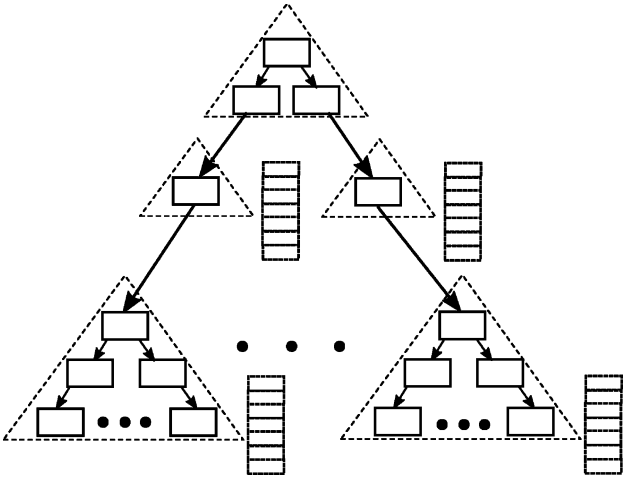

Fig. 1. *An index structure with buffers added.*

B+ (CSB+) trees for index structures requiring cache-conscious updates and reads [5]. In both approaches, cache misses are reduced in part by packing more data into the cache. In CSS and CSB+ trees, index nodes are sized to fit into cache lines and many pointers are eliminated. First, by sizing each node to fit in a cache line, searching within a node to decide which child to visit is accomplished with only one cache miss. Next, pointers are eliminated in CSS trees and CSB+ trees as each node (see Fig. 2) stores only one child pointer. In contrast, a B+ tree node with k keys would have $k + 1$ pointers. The pointers in a CSB+ tree are replaced by allocating all of the child nodes contiguously and then using array indexing to find the required node at an offset from the single child pointer. Because the number of pointers per node is reduced, more keys fit in one node. More keys per node increases the fanout of the tree, which results in a reduced tree height and shorter search path. In turn, the shorter search path coupled with the cache-line-sized nodes results in fewer cache misses per index search [5].

CSS and CSB+ trees are found to outperform traditional, non-cache-conscious index data structures on in-memory processing because they successfully reduce the number of cache misses required to search the tree [4], [5]. CSS trees, for instance, incur 50% fewer cache misses than B+ trees and can be 50% faster than B+ trees for read-only searching [4]. CSB+ trees exceed B+ trees in terms of search performance and allow updates. A properly constructed CSB+ tree is found to perform better than a B+ tree if more than 25% of the index workload consists of searching rather than updates [5].

C. Cache Conscious Joins

Index lookups are not the only operation to benefit from cache-conscious design. Recent work has also investigated improving the cache behavior of relational join operators. Manegold *et al.* provide an extensive

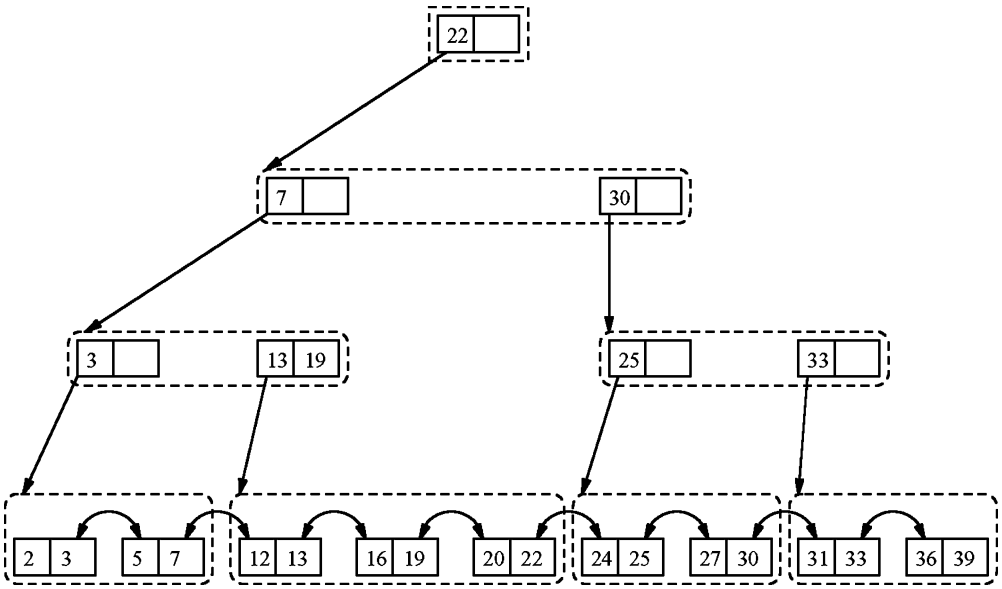

Fig. 2. *A CSB+ tree of order 1. Note that each node has only one child pointer and each node's children are allocated contiguously.*

discussion of optimizing joins on modern hardware [6]. To achieve good cache performance, the two relations to be joined may be partitioned using the same hash function into partitions that fit in the data cache. Tuples from partition i of one relation can only join with tuples in partition i of the other relation, so only one partition from each relation must be examined at a time to determine the join result. Because each partition fits in the cache, cache misses during join processing are low. The challenge, however, is the performance of the partitioning step. If the number of required partitions is very high, then the partitioning step can cause a significant number of TLB and cache misses [6]. The authors propose a radix cluster algorithm to perform the partitioning step in a cache-conscious manner. In the radix cluster, the number of random accesses is low and a large number of clusters can be created by making multiple passes through the data. Experimental results show that as the amount of clustering was increased, the time spent clustering also increased but that the join processing time decreased. Similarly, for smaller numbers of clusters, clustering time was lower but join time was longer. This is because fewer clusters means that clusters are larger and do not fit in the cache, resulting in more cache misses during the join phase. Based on this observation, an optimum number of clusters can be determined. The authors observe a speedup of almost a factor of ten for their most optimized hash join implementation [6].

D. Prefetching

Database performance can be improved by prefetching data into the cache before it is needed. By using prefetching, a memory reference will be a cache hit instead of a long-latency cache miss [6], [7]. Prefetching can be done in hardware or software. In hardware prefetching, the processor observes a pattern of memory references and begins to prefetch the next location in the pattern. An example of a common memory-access pattern is a sequential scan, such as examining every element in an array. Prefetches may also be requested in software by special instructions. These instructions provide hints to the processor about future memory references and the processor makes an attempt to load the requested memory location.

Chen *et al.* study the use of software prefetch instructions to improve tree-based index searching and scanning [8] as well as hash join performance [7]. Software prefetch instructions allow tree indexes to use internal nodes larger than a cache line while overlapping the cost of the additional cache misses to search within a node. Given sufficient hardware capacity for outstanding memory requests, the entire node can be prefetched. Larger node sizes result in shallower trees with shorter search paths, which improves performance. Using prefetching to allow shallower trees results in a speedup of 1.27–1.55 over a B+ tree [8].

Prefetching for a hash join is a challenging task because accessing a hash table bucket based on a hash value results in random memory accesses. Further, looking up values in a hash table involves data dependencies such as following a linked list of overflow cells. The next cell in the linked list to be loaded is not known until the current cell is examined; therefore it is impossible to prefetch more than one item ahead in a linked list. This reduces the amount of prefetching possible within just one probe to the hash table. The solution proposed by Chen *et al.* is to consider the intertuple parallelism among different probe tuples.

A group of probes are processed together in stages, with prefetch instructions used to trigger the memory references that each probe tuple will require in the next stage. The intertuple parallelism can also be pipelined so that, rather than a group of tuples executing the same stage, the tuples are pipelined so that they are in different stages. The pipelined approach is somewhat more successful at hiding cache misses, but the grouped approach is easier to implement, has a lower bookkeeping overhead, and still performs rather well [7]. By using prefetching, the hash join performance is improved by 1.9× to 2.7× compared to the nonprefetched implementation of the GRACE hash join algorithm [7], [9].

Cache-conscious improvements have been made to numerous aspects of database operations. For instance, Ghouting *et al.* use architecture-sensitive optimizations to improve frequent pattern-mining performance by a factor of 3.2×, and up to 4.8×, with a further multithreaded optimization for processors with simultaneous multithreading [10]. Record layout is a key component of data cache optimizations, and, as we will show in the next section, it is important to overall database performance as well.

III. RECORD LAYOUT

Record layout has an impact on database performance by affecting both I/O and cache usage. Most commercial and open-source database systems store records in a row-based or N-ary storage model (NSM). In this model [see Fig. 3(a)], the attributes for one record are stored contiguously. Database operations that require access to the entire record benefit from this locality and can perform operations with one sequential read or write. On the other hand, if only a small fraction of attributes are needed from many records, then a large amount of data is needlessly read. This also has implications for data cache performance because the extraneous data wastes cache space, resulting in more cache misses due to capacity constraints. An alternative to NSM is a column-based or decomposed storage model (DSM). Fig. 3(b) shows a DSM layout, in which the same attribute from multiple records is stored contiguously. Currently, a number of systems employ DSM for main memory and decision support workload optimization [11]–[13].

A. I/O and Cache Performance

DSM has been shown to be advantageous for decision support workloads that typically involve only a subset of attributes. At the I/O level, DSM reads only required data from the disk. It also tightly packs requested data in the cache, not wasting space for unused attributes. On the other hand, if an entire record is required, a column-based storage model must reassemble the record from the constituent columns, incurring overhead that would be avoided by using NSM.

Harizopoulos *et al.* consider performance tradeoffs in read-optimized column-oriented databases [14]. The queries investigated are read-only scan-mostly queries typical of online analytical processing and data warehousing applications. This work concludes that DSM almost always exceeds NSM in terms of I/O performance, assuming that sufficiently large chunks of columns can be read contiguously. In some cases, NSM's CPU performance exceeds that of DSM when records are very small or predicates are not selective (i.e., few tuples are filtered out). Moreover, the authors introduce an analytical model that predicts database performance given disk and processor parameters. Their work suggests that current trends in hardware designs will make DSM storage models an increasingly attractive model for read-only storage [14]. Other work on column storage systems demonstrates the benefits of using compression on columns [15].

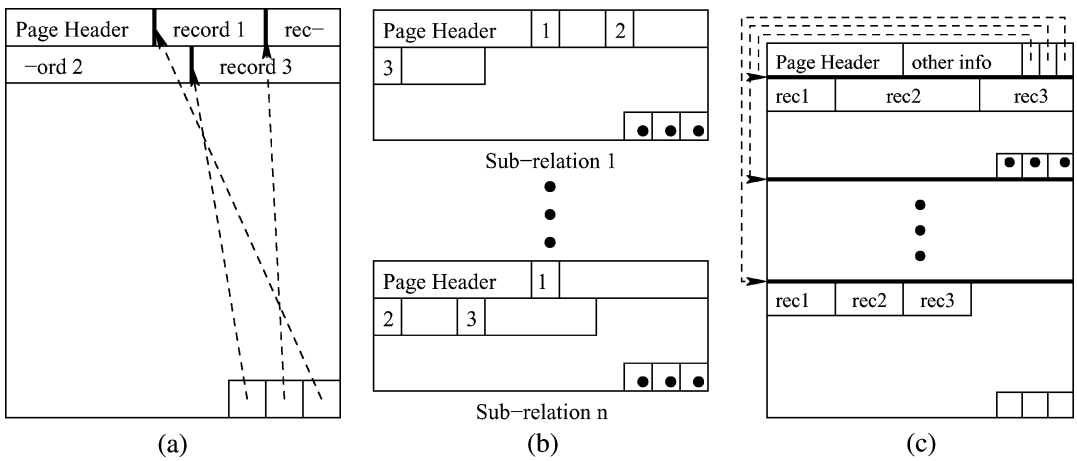

Fig. 3. *Record layout schemes. (a) NSM or row storage, (b) DSM or column storage, and (c) PAX.*

The *partition attributes across* (PAX) storage model shown in Fig. 3(c) is a page layout that provides good cache behavior while not changing the I/O performance of database systems [16]. Using PAX, the records that would be stored together on a page in NSM are still stored on the same page but the on-page storage resembles DSM because the same attribute from all of the records on that page are stored contiguously. Therefore, I/O consists of reading pages containing complete tuples, but only the required attributes are loaded into the cache (as with DSM) thus improving performance. Because all of the constituent parts of a record are contained on the same page, reconstructing a record is not as expensive as with DSM [16]. Cache performance is much better than NSM when only a subset of attributes are required. As with the cache-conscious data structures described in Section II-B, DSM and PAX pack data more tightly in the cache, which results in better cache usage. In an experimental evaluation, PAX greatly improves cache performance while not incurring additional I/O overhead, resulting in a speedup of up to 48% over NSM [16].

B. Flash Memory

New storage technologies also influence data layout. The recent introduction of high-density flash-memory storage devices makes a database engine designer rethink database storage and record layout. The sequential read and write performance of magnetic disk storage is much higher that its random access time because of seek and rotational delays. New flash-memory storage devices offer durable storage with much faster random access times. Besides the differences in random access latency, flash memory also differs from magnetic disks in its support for in-place updates. Flash memory has an erase-before-write limitation, which means that an entire erase unit must be erased before writing the updated value. The erase units are typically larger than a page. For instance, a 128 KB erase unit could hold 16 8-KB pages. Erase operations are also more expensive than reads or writes. To overcome the limitation of flash memory for in-place updates, Lee and Moon propose an in-page logging approach [17]. It should be noted that although an in-place update requires an erase before the write, appending data to an unused portion of an erase unit does not require an erase. Leveraging this fact, Lee and Moon reserve a log region in each erase unit [17]. Updates to the data stored in that erase unit are appended as log entries in this reserved region. During a read, the log must be consulted to calculate current values. When the log region in an erase unit fills, the updates are applied to the corresponding records in that block and the entire unit is rewritten to the flash memory, but with a now empty log region. Experiments comparing disk and flash storage performance were conducted using a commercial database server. Lee and Moon demonstrate significant improvements for read queries involving random access I/O. For instance, when pages are read randomly, the disk query time is reported as 61.07 s compared to 12.05 s for the flash storage. In contrast, during a sequential table scan, the disk takes only 14.04 s compared to 11.02 s for the flash storage [17]. Write performance shows a similar trend; flash storage outperforms magnetic disk storage, especially when the writes are randomly distributed.

Other work that examines data layout includes Zhou and Ross [18], Shao *et al.* [19], Hankins and Patel [20], and Chilimbi *et al.* [21]. Graefe revisits the "five-minute rule," adding flash-based storage to the analysis [22].

IV. INSTRUCTION CACHE

In Section II, we discussed the importance of optimizing database operations to effectively utilize the data cache. In this section, we show that effective use of the instruction cache also greatly affects application performance.

Most modern processors have separate L1 caches for data and instructions. When instructions are not found in the instruction cache, the effects of this miss are more significant than when data is not found in the data cache. Though a data cache miss may cause a stall, modern processors typically allow nonblocking cache misses that can be overlapped with other misses and computation. That is, the processor may continue to make some progress when a data cache miss occurs. (Of course, as we saw in Section II, this overlapping of cache misses with computation has limits, and may not prevent all stalls.) In contrast, when instructions are not found in the cache, the processor must stall until the instructions arrive from a lower cache level or from main memory. The instruction miss cannot be overlapped with other work or misses because the missed instructions define the work to be done. Therefore, avoiding instruction cache misses at the highest (L1) cache level is important in order to avoid compulsory pipeline stalls.

Instruction cache misses can be a source of significant latency in database systems, accounting for an average of 20% of total execution time [2]. One contributor to this problem is that database systems are complex and consist of many complex modules, each with many instructions. These modules, or operators, each perform tasks important for query processing. For instance, some may read tables from disk, while others may apply selection predicates to records, aggregate values into more concise sums or counts, or join records from multiple tables. During query execution, many of the operators are required, and their total instruction footprint often exceeds the available L1 instruction cache. To illustrate this problem, here is a sampling of operator sizes from PostgreSQL on a Pentium architecture whose trace cache holds about the equivalent of a 16 KB instruction cache [23]: merge join (12 KB), index scan (14 KB), sum aggregation (2.7 KB), and a table scan with predicate evaluation (13 KB) [24]. Any combination of even a few of these operators will exceed a 16 KB instruction cache.

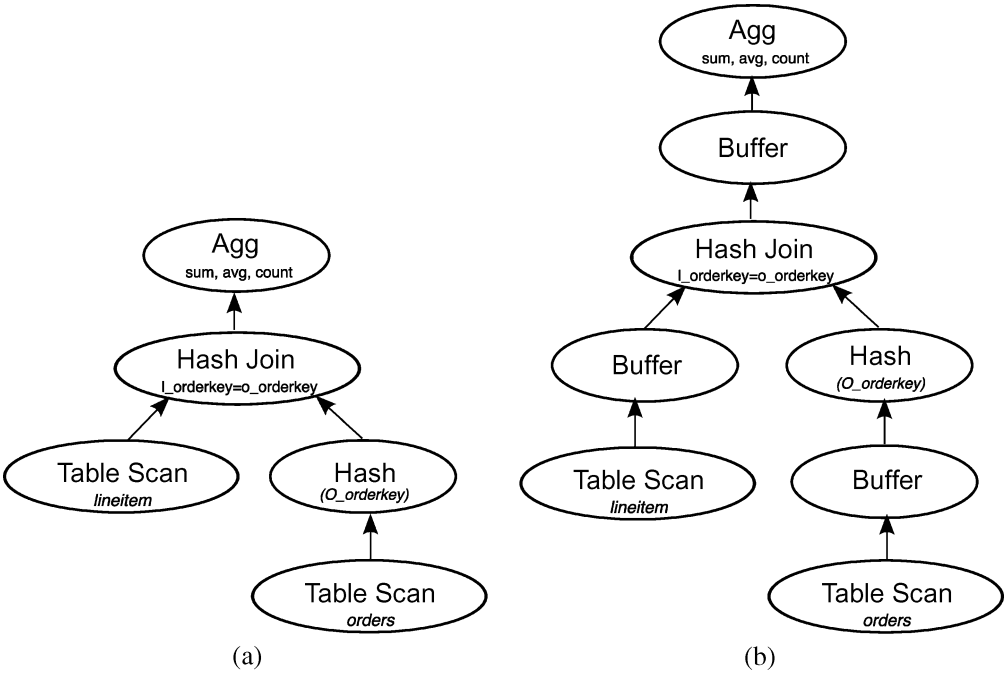

Fig. 4. *A query plan before and after buffer operators are added. This query joins the table "orders" and "lineitem," computing aggregates over the joined result. (a) Original plan and (b) buffered plan.*

Another issue is the manner in which these operators are used to execute a query. The operators are composed in a query plan, which is a tree rooted at the output [see Fig. 4(a)]. Each operator has only one parent (or output) but may have different numbers of children depending on its function. Many commercial and open-source database systems use demand-pull query execution, though some use a block-oriented execution [25]. In demand-pull execution, each operator returns just one tuple to its parent. Execution begins at the output operator, which requests a tuple from its child. Those operators request a tuple from their children in turn. In this manner, a single tuple is pulled up the query to the output, which repeats the process until all output has been generated. This demand-pull model is important because it allows operators to be composed in any order and also allows for pipelining.

Demand-pull query execution pipelines the query operators, reducing the amount of intermediate data that must be stored (only one tuple proceeds up the query plan at a time). Though less intermediate data must be stored, demand-pull execution also results in a long chain of operators, each with its own instructions, which can cause significant instruction cache thrashing. When a child is invoked, its instructions must be brought into the cache via compulsory cache misses. If an operator's descendant operators are sufficiently large or the pipeline sufficiently deep, loading those operators into the instruction cache will evict the original operator's instructions. Demand-pull execution results in a significant number of cache misses even though only one tuple is produced. We will next describe two techniques for improving instruction cache behavior in database systems for both OLAP and OLTP workloads by ordering the computation so that instructions are reused more times before being evicted from the instruction cache.

A. Buffering

A key observation from the description above of instruction cache misses during demand-pull query execution is that processing one tuple through an operator has a high fixed cost that is amortized over only one tuple. An OLAP query will likely need to process many tuples. Processing many tuples reuses the instructions already in the instruction cache, thus amortizing the compulsory instruction cache misses over many tuples. Demand-pull execution minimizes intermediate storage. The converse of this strategy is to run each operator over all input in a bottom-up fashion, materializing all output at each operator. In this case, the instruction cache will be used well but intermediate materialization will cause other performance problems. A middle ground proposed by Zhou and Ross [24] is to buffer a small number of tuples between operators. Filling a buffer allows an operator to process more than one tuple, thus amortizing the costly instruction cache misses. Supplying input from a buffer of tuples is a lightweight operation that does not impact the instruction cache and avoids invoking a subtree of

operators for every request from a parent. If the buffer size is chosen carefully, the intermediate storage issue is not a significant problem.

The addition of buffers within query plans is accomplished with only a small modification to the query infrastructure. A new buffer operator can be inserted between any operators within a query plan. The important question is where and when to buffer in a query plan. The idea is to group consecutive operators into execution groups whose operators fit into the L1 cache. Buffered access to the execution group is accomplished by placing a buffer operator above the execution group in the query plan. By adding buffering to the plan, instruction cache misses are amortized over multiple tuples, and instruction cache thrashing that would otherwise occur is avoided. Zhou and Ross propose an algorithm to determine when and where to perform buffering [24]. Fig. 4 shows a plan with execution groups identified [Fig. 4(a)] and buffer operators inserted [Fig. 4(b)].

The buffering algorithm is evaluated experimentally by adding buffer operators to PostgreSQL. Buffering is found to reduce cache misses by up to 80%, improving query performance by 15% [24].

B. Staged Database Execution

Instruction cache misses account for 40% of execution time during OLTP [26]. Harizopoulos and Ailamaki [26], [27] propose using a staged database design called STEPS to improve the instruction cache performance of OLTP queries. Running an operator over many input tuples, for example, with the buffering technique described in the previous section, is not an option for most OLTP queries because these queries typically involve only a few tuples. Even though the number of tuples involved in one transaction is small, it is typical for many similar transactions to be processed in a very short time period. Therefore, amortizing instruction cache misses across different transactions is desirable.

Because most transactions follow a similar code path, the transaction may be divided into stages that fit in the L1 instruction cache. When one transaction reaches the end of stage, rather than continuing with that transaction, the system makes a context switch to a different thread that needs to execute the same stage that was just completed. This can be repeated for any number of threads/transactions. The first thread incurs all of the instruction cache misses for a stage, and the threads that follow experience only instruction cache hits. Once enough threads/transactions have completed the stage, execution proceeds to the next stage and this process repeats. In practice, the system implemented by Harizopoulos *et al.* does not require the rigid synchronization suggested by the above description.

While executing an OLTP workload, instruction cache misses can account for up to 40% of the execution time. By using the staged design, the authors [26] eliminate two-thirds of instruction cache misses on a TPC-C workload and speed up execution by a factor of 1.4.

V. BRANCHES

Conditional branches cause two kinds of performance hazards on some modern microarchitectures. First, on machines with long pipelines, many cycles may pass before a branch condition is resolved. In the meantime, the pipeline still needs to be filled with instructions even though the outcome of a branch is unknown. Branch prediction hardware tries to use recent history to guess whether the branch will be taken and fill the pipeline with instructions based on that assumption. When the processor guesses correctly, the processor can proceed at full speed. When the branch predictor guesses incorrectly, a delay comparable to the pipeline length is required to flush the pipeline and undo any effects of incorrect instructions. The time that this takes is sometimes referred to as a branch misprediction penalty. Secondly, on machines that rely on instruction-level parallelism, branches represent a control dependency that limits the amount of instruction reordering. All instructions after the branch depend on the instructions before the branch. As a result, neither the compiler nor the hardware is able to fill execution slots with useful work, and the processor suffers dependency stalls.

Not all processors suffer from these problems. For example, machines like the Sun UltraSPARC T1 (Niagara) and the Cray MTA-2 have designs that mitigate the problems of branches. Branch prediction is not necessary on these architectures because threads are interleaved in such a way that by the time a thread is rescheduled, the condition has been resolved and the next instruction to execute is no longer ambiguous. These processors rely less on instruction-level parallelism and more on thread-level parallelism (see Section VI); therefore the dependencies caused by conditional branches rarely impact performance as long as all threads are busy.

A. Addressing the Branch Prediction Cost

Branch mispredictions contribute a measurable latency to commercial database systems, sometimes greater than 15% of overall execution time [2]. Unfortunately, database workloads often have unpredictable branches, meaning that the branch misprediction penalty is paid close to half the time. For example, a branch during a binary search within the node of an index is typically taken half of the time, with no temporal predictability.

One way to manage the cost of branches in a database system is to extend the query optimizer's cost model so that the cost of a mispredicted branch is taken into account [28], [29]. For example, suppose one is scanning a large array of records and would like to return an array of indexes of records than satisfy some simple conditions. If a record represents an employee, then a condition

might be "age > 30 and salary < 50 000." This query could be processed by a variety of query plans having two, one, or even zero branch instructions per record, as shown in the following pseudocode:

```
/* Two branch instructions */
for(i = 0; i < number_of_records; i++) {
    if(r1[i].age > 30 && r1[i].salary < 50 000)
        {answer[j++] = i; }
    }
/* One branch instruction */
for(i = 0; i < number_of_records; i++) {
    if(r1[i].age > 30 & r1[i].salary < 50 000)
        {answer[j++] = i; }
    }
/* No branch instructions */
for(i = 0; i < number_of_records; i++) {
    answer[j] = i;
    j+ = (r1[i].age > 30 & r1[i].salary < 50 000);
    }
```

The "two-branch" example has two conditional branches because the "if" statement will be translated[1] into two nested conditional branches. In contrast, the "one-branch" example uses only one conditional branch instruction, because the outcomes of the conditional tests are combined using the bitwise "&" operator. In this case, both conditions are always evaluated, but in the two-branch example the second condition will only be evaluated if the first is found to be true. Finally, the "no-branch" example transforms the conditional branch into a data dependency, incrementing the answer index by one if the conditions hold and zero if not. Because the number of predicates evaluated varies in these examples, branch prediction rates as well as predicate evaluation cost must be considered to pick the optimal query plan.

Branch misprediction cost is one of several cost factors; therefore simply minimizing the number of conditional branches would be assigning too much weight to branch misprediction. Ross [28], [29] provides query optimization algorithms that minimize overall cost for such queries and validates the model on real processors. Another way to improve the branch misprediction overhead is to organize the data so that branches are more predictable. For example, the underlying records can be ordered so that a conditional sequential scan shows temporal locality in the branching behavior [29].

B. Eliminating Branches

An effective way to eliminate branches is to convert the control dependency of a conditional branch into a data dependency. The no-branch plan of the previous section is an example of such a transformation. Modern instruction sets provide instructions that facilitate these transformations. Examples include the following.

- The CMOV statement on x86 architectures executes a move instruction if the condition code is satisfied and is a NOP (no operation) otherwise.
- Predicated execution on the Itanium architecture, a generalization of the CMOV instruction above that allows conditional execution of arbitrary instructions.
- Single-instruction multiple-data (SIMD) instructions available on many processors. The result of a comparison operation is returned in a mask register than can be manipulated without branches.

Several researchers have exploited these instructions to eliminate branches, improving instruction-level parallelism and reducing branch mispredictions. Zhou and Ross show how SIMD instructions can be used for database operations and demonstrate a dramatic reduction in branch misprediction cost [30]. For example, they show that branch-free sequential scans beat binary search for relatively long arrays of keys, e.g., up to about 60 keys on their experimental platform.

Recent work [30]–[32] has shown how control dependencies can be converted into data dependencies for compression and for hash table lookups, with a noticeable improvement in instruction-level parallelism in addition to fewer branch mispredictions. For hashing [32], [33], it appears that more complex algorithms based on cuckoo hashing [34] turn out to be more efficient than conventional algorithms on some processors [31]. These results are surprising because, at first glance, the cuckoo-based algorithms appear to be doing more work. However, the reduction in branch mispredictions and the improvement in instruction-level parallelism more than compensate for this extra work.

VI. CHIP MULTIPROCESSORS AND MULTITHREADING

In Sections II, IV, and V, we saw how data dependencies and control dependencies can affect database performance. These dependencies impact the performance of a single instruction stream, which until recently was all that mattered because almost all processors supported only one instruction stream at a time. In recent years, this has changed dramatically as many processors now support on-chip thread-level parallelism. Rather than spending transistors to improve the performance of a single thread of computation, processor architects have shifted their focus to providing hardware support for multiple simultaneous instruction streams. Because these instruction streams are independent, a stall due to a data or control dependency in one stream does not stall the entire processor, resulting in higher overall processor utilization. As the trend toward increased on-chip TLP continues, TLP will become increasingly important to database performance. In this

[1]One should verify that the generated assembly code matches one's expectation about how the code should have been compiled.

section, we present some recent work highlighting the opportunities and challenges that on-chip thread-level parallelism presents to database implementation and performance.

Hardware support for thread-level parallelism in commodity processors has thus far been introduced in the form of simultaneous multithreading (SMT) and chip multiprocessors (CMPs). SMT is found in commodity processors such as the Intel Pentium 4 with hyperthreading, the IBM Power5, and the Sun T1. In SMT, multiple hardware threads share one processor core. The processor chooses an instruction (or instructions) from the available hardware threads in a fair manner. The benefit of SMT is that if one hardware thread stalls, the processor is kept busy by executing instructions from other threads, resulting in higher overall processor utilization. A downside is that the instructions from different threads are competing for shared processor execution and cache resources. This competition has two important implications. First, because the pipeline is shared, no thread receives the full power of a dedicated uniprocessor. During compute-intensive tasks, the use of SMT may even slow application execution. Care must also be taken to avoid flooding the processor with useless work, such as spinning while waiting for a lock, which will prevent other threads' instructions from having more access to the processor. Secondly, because the cache resources are often shared between the SMT threads, the various threads can interfere in the cache, causing conflict misses and cache thrashing. Therefore, extracting maximum benefit from SMT processors requires careful design rather than naïvely treating each hardware thread as a separate processor. Later in this section, we will describe techniques for using SMT to effectively overlap useful computation with long-latency memory accesses in database operations.

Recently, all of the major microprocessor manufacturers have moved to chip multiprocessors—a microarchitecture featuring multiple cores on a single die. Major vendors currently have dual- and quad-core CMPs, and Sun Microsystems has an eight-core CMP, the T1 [35]–[37].[2] The major difference between CMP and SMT processors is that a CMP has multiple cores, each with a separate pipeline and execution resources. Designs vary, but many CMPs feature a private L1 cache for each core and an L2 cache that is shared among the cores, though some L2 caches are partitioned. In some cases, the CMP cores also feature SMT, expanding the amount of on-chip thread-level parallelism even further. Next, we will discuss recent work that has explored the benefits and challenges of CMPs for database servers.

A. Improving Database Performance With SMT

Long-latency memory operations that cause the processor pipeline to stall are a well-known cause of latency in databases [2]. The multiple thread contexts available to an SMT processor provide an opportunity to overlap computation with long latency stalls using thread-level parallelism. Zhou *et al.* evaluate different strategies for using multiple hardware thread contexts available on SMT processors [38]. The strategies considered are:

- naïve parallelism—treat each thread context as a separate shared memory processor;
- SMT aware parallelism—design threads to cooperate;
- work-ahead set—explicitly overlap long-latency memory operations with computation.

Naïve parallelism provides a small improvement in performance, but hardware performance counters reveal that the number of cache misses increases significantly. This indicates that the independent workloads cause conflicts in the cache. When the independent workloads are different operations, the performance improvement from SMT is even lower because instructions and data conflict in the cache. Nevertheless, SMT does provide an improvement for database operations even if it is used naïvely. This is good news because many database systems are already designed to take advantage of shared memory parallelism, so no further changes are required to get a small benefit from SMT.

SMT-aware database operator implementation requires designing threads that efficiently share resources, such as the cache. The most obvious way to accomplish this is to have the threads execute the same operation but divide the input among the various threads. Instructions and read-only data structures are shared, which leads to fewer conflicts in the cache. Even though all threads will encounter long-latency memory operations during execution, the hope is that when one thread is blocked, other threads may have instructions that can execute, thus keeping the processor busy.

The "work-ahead set" is an attempt to use TLP to explicitly overlap computation with memory latency. In the work-ahead set model (see Fig. 5), the computation is staged so that a stage ends at each point in the code where a possibly noncache resident memory location is

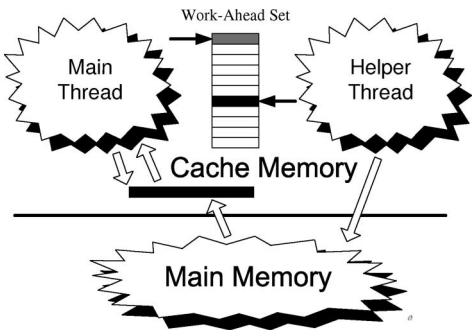

Fig. 5. *The work-ahead set data structure.*

[2]AMD Multi-Core, http://multicore.amd.com.

referenced (e.g., reading input or following a pointer). At this point, the memory location and some state information are pushed into a data structure called the work-ahead set. This operation returns previously enqueued state information and a pointer. The operator uses this returned state to continue processing where it left off when it had encountered an earlier memory reference. In the meantime, a "helper" thread also accesses the work-ahead set data structure and explicitly loads the requested memory locations into the cache. This explicit preloading is different from prefetching in that the load will definitely happen (prefetches can be dropped on some architectures [35], [36]) and the preloading is easier to tune. The helper thread triggers most cache misses and incurs the memory latency. The "main" thread, which performs the actual computation, experiences mostly cache hits and spends most of its time executing instructions rather than blocking. Because the helper thread is lightweight, it does not compete with the main thread for execution resources. Moreover, the helper thread does not affect correctness. If the helper thread does not manage to preload a memory location before the main thread returns to process it, the main thread suffers a cache miss but the result of the algorithm is still correct. In an evaluation of performance, using SMT technology yields a 30% to 70% performance improvement, depending on the operation and the SMT technique employed. SMT-aware parallelism and the work-ahead set outperform naïve parallelism in all cases, with the work-ahead set and SMT-aware techniques performing slightly better or worse than each other depending on the workload [38].

Garcia and Korth also investigate the use of simultaneous multithreading to improve database operations, including sorting and hash join [39], [40].

B. Database Servers on Chip Multiprocessors

Recent work from Hardavellas *et al.* [41] looks at future performance advantages and challenges related to database servers on CMPs. Currently, CMPs fall into two camps—a "fat camp" and a "lean camp." The authors identify the "fat camp" as CMPs with a few wide-issue out-of-order cores with deep pipelines. The "lean camp" CMPs are narrow-issue in-order cores with shallow pipelines. The size of a lean core is much smaller than a fat core, allowing more lean cores to be placed on a die of similar size and resulting in more hardware threads per processor.

Hardavellas *et al.* demonstrate that as L2 cache sizes increase, the L2 hit latency will become a larger contributor to database latency on CMPs, thereby shifting the performance bottleneck from off-chip memory accesses [2] to on-chip L2 hits [41]. Increased L2 cache sizes attempt to capture larger working sets of applications; unfortunately many database operations have a relatively modest primary working set and a very large secondary working set that far exceeds a reasonable L2 cache size [41]. Larger L2 caches require more time to service a hit and do not provide a significant benefit to database operations because smaller L2 sizes already capture the primary working set. The L2 hit time accounts for up to 35% of execution time for the examined workloads, which is seven times greater than the L2 hit time contribution for symmetric multiprocessors with smaller caches on the same workload [41]. An important implication of this trend is that future database performance may depend on good L1 data locality in addition to L2 data locality. In other words, the performance bottleneck is moving up the memory hierarchy.

The authors also conclude that increasing the number of cores on a chip does not lead to a large increase in L2 cache misses, even though the cores share the L2 cache. This is because database operators can share significant amounts of data in the cache, but the increased cache pressure can cause performance degradation even with a lower miss rate [41]. The authors advocate new L2 designs to relieve some of this cache pressure. They also determine that the high number of hardware threads in lean camp processors leads to effective overlapping of memory latency with computation but that while throughput is increased, response time is slower than on fat camp processors [41]. This confirms work by Cieslewicz *et al.* [42] that used the large number of hardware threads available on Cray MTA-2 processors to overlap computation with memory latency during database operations. The Cray MTA-2 achieved processor utilization exceeding 80% during core database operations typically plagued by cache misses such as a hash join and tree-based index lookup [42].

Staged databases, also discussed in Section IV, are advocated as a future database design for CMPs [26], [41]. Dividing query processing into stages and passing packets representing query requests between different stages is a natural way to increase the amount of inter- and intraquery parallelism. It also helps keep hardware threads busy and, as discussed earlier, can improve instruction locality by having one hardware thread execute the same stage over multiple packets.

Johnson *et al.* [43] report an interesting negative result, demonstrating that aggressive work sharing may not always be optimal for database workloads on chip multiprocessors. Many work-sharing techniques have been shown previously to improve database performance. For example, concurrently executing queries might share a scan of a relation that they all require. The system identifies this sharing opportunity and configures the scan operation to send its output to each query. Johnson *et al.* highlight an important tradeoff between work sharing and parallelism: although work sharing may reduce the total amount of work done by the system, it may also reduce the amount of parallelism, thereby degrading performance. In other words, achieving high performance on parallel hardware may actually require doing some redundant work to enable the parallelism, but the benefit from parallelism will outweigh the extra work. For an in-memory workload

on a CMP with a high degree of parallelism, a shared operation can become a bottleneck, forcing all parallel tasks sharing that operation to proceed at the same speed. Using a model to capture the tradeoff between work sharing and parallelism, Johnson *et al.* are able to determine whether or not to share work, resulting in a 20% improvement over a "never-share" scheme and 2.5× over an "always-share" scheme when processing an in-memory workload [43].

Cieslewicz and Ross also highlight the challenges of implementing database operations on CMPs [44]. As CMPs offer higher degrees of on-chip parallelism, some on-chip resources such as a shared cache or off-chip bandwidth are becoming more constrained. Consider the UltraSPARC T1, which supports 32 hardware threads. The T1 has a 3-MB L2 cache shared among all 32 threads, giving each thread just under 100 KB of cache space if the L2 is divided evenly among the threads. This small amount of cache space per thread suggests that, for some database operations, threads should pool their cache resources and cooperatively execute the operation. Interthread cooperation means sharing data between threads, a costly prospect on multiprocessor systems where cache coherency is maintained across an off-chip bus, but thread coordination on a CMP occurs at chip speeds, thus making tighter thread cooperation feasible.

Cache sharing and thread cooperation are investigated by Cieslewicz and Ross in the context of hash-based aggregation on the UltraSPARC T1 [44]. In these experiments, all 32 on-chip threads are applied to the aggregation operation. Based on a lightweight sampling of the input, the expected size of the hash table's working set can be estimated. It can then be determined whether or not the threads should proceed independently, each using its own hash table, or if they should pool their cache resources by sharing one hash table. Making this determination is important because sharing a read–write data structure such as a hash table for aggregation incurs synchronization overhead to ensure that updates are performed atomically, but the shared table allows the cache to hold a larger hash table, thus reducing expensive cache misses. Synchronization overhead is expensive, but not as expensive as a high rate of L2 cache misses. Determining whether or not to share data in the cache based solely on minimizing cache misses does not tell the whole story. If some values in a globally shared table are updated frequently, the synchronization of those updates can become a source of contention among the threads. In the worst case, all threads serialize because of the atomic updates to the frequent value. This contention penalty is very severe and can eliminate any benefit parallelism would provide. By modeling the expected contention, Cieslewicz and Ross propose a new adaptive aggregation algorithm for CMPs. They demonstrate that this adaptive aggregation strategy is effective on a wide range of inputs and that choosing the wrong strategy can result in a performance penalty of more than one order of magnitude [44].

VII. NONTRADITIONAL PROCESSORS

Database operations have also been explored on architectures other than general-purpose processors. A tutorial from the 2006 VLDB conference provides a good overview of many of these query coprocessing techniques [45]. Recent work has included database research using network processors, graphics cards, and the cell broadband engine.

A. Network Processors

Network processors are designed to provide a high degree of parallelism and large amount of bandwidth for network applications. For instance, the network processor studied by Gold *et al.* had eight processor cores, each with eight hardware threads [46]. The network processor has two features that make it significantly different from a general-purpose processor. First, all memory transfers to and from the processor must be explicitly controlled from software. Secondly, threads must explicitly yield control of the processor to other threads, i.e., no preemption occurs. For these two reasons, programming a network processor is more complicated than programming a general-purpose processor [46].

Gold *et al.* map a number of common database operations to the network processor architecture and evaluate the performance of each [46]. The multiple hardware threads available on the network processor allows for a high degree of TLP, which in turn results in more parallel references to memory. The authors compare the performance of a sequential scan, index scan, and hash join on the network processor to the same operations on a 2.8 GHz Pentium 4. They find speedups of 1.9× for a sequential scan to 2.5× for the hash join. An index scan on the network processor exceeds the Pentium performance by a factor of more than 3× on some configurations [46].

B. Graphics Cards

Graphical processing units (GPUs) found on commodity graphics cards are highly specialized hardware with significant processing power. The performance of GPUs is increasing at a rate of 2.5 to 3.0 times per year, which is much faster performance growth than for traditional general-purpose processors [47]. Recent research has investigated ways of harnessing the power of GPUs to perform database operations.

The biggest challenge in using GPUs for database operations is in mapping the database operation to the GPU architecture, which is a demanding programming challenge. GPUs are highly specialized and focused on graphics tasks; therefore only a subset of a fully featured database system's operations can be mapped to a GPU. Nevertheless, the performance gain on some types of database operations makes GPU query coprocessing very worthwhile. What follows is a brief summary of GPU database research; we refer the reader to the cited resources for a full description of GPUs and their applicability to databases.

Govindaraju *et al.* demonstrate techniques for mapping many database operations to GPUs [47]. Operations in their analysis were predicate evaluation, Boolean combinations of predicates, and aggregations including order statistics. Overall they found the GPU to be a good candidate for a database query coprocessor but that the performance benefit depends significantly on the type of operation being accelerated with the GPU [47]. For instance, a range query with a GPU is reported to be 5.5× faster than the same query with a CPU. This speedup includes the cost of copying the data to and from the graphics card. If that copying cost is ignored, the speedup is closer to 40× [47]. In contrast, an aggregate query requiring accumulating a value, such as a sum, is reported to perform 20× slower on a GPU compared to a CPU.

Sorting is an integral component of database workloads. In a 2006 paper, Govindaraju *et al.* introduce a new algorithm called GPUTeraSort that blends CPU and GPU processing to achieve very high sorting performance [48]. Applying GPUTeraSort to the PennySort benchmark, 60 GB can be sorted for a penny, the best reported PennySort result as of the 2006 competition.[3]

Because GPUs are tailored to graphics operations, spatial queries are a natural fit for GPUs. Bandi *et al.* and Sun *et al.* investigate using GPUs as coprocessors to accelerate spatial queries [49], [50].

C. SIMD and Cell

Hardware that can process single instructions on multiple data elements can enhance the parallelism of an application. SIMD has potential for database systems because similar operations are often applied to many independent data elements. Most modern commodity processors have small-scale SIMD capabilities.

Typical SIMD features include wide registers that can be interpreted as a vector of small data types. For example, a 128-bit register could represent sixteen chars, eight shorts, or four 32-bit ints or floats. A rich instruction set is typically provided for these SIMD operands including comparisons, arithmetic, and bitwise operations. Programming with SIMD instructions, however, can be challenging, as there are often strict memory alignment requirements as well as more complex data movement requiring scatter/gather techniques.

As mentioned in Section V-B, Zhou and Ross investigated using SIMD for database operations and were able to obtain speedups for some joins, scans, and index operations [30]. Part of the speedup was due to the enhanced parallelism, while part was due to enhanced branch behavior. SIMD instructions perform comparisons in a data-dependent way, and so the conversion of control dependencies to data dependencies was beneficial, as discussed in Section V. Ross uses SIMD to obtain similar benefits for hashing [32].

[3]http://research.microsoft.com/barc/SortBenchmark/.

Some recent processor architectures, such as the cell broadband engine [51], provide specialized SIMD accelerators. One cell chip contains a power processor, together with eight synergistic processors (SPEs), making it a heterogeneous CMP. While the SPEs have limitations (such as a small local memory), they are extremely efficient at processing SIMD workloads. In some cases, certain database operations could be offloaded to such accelerators [32], [52]. Héman *et al.* [52] demonstrate that the cell is a very effective platform for vectorized database processing, achieving a factor of 20 performance improvement over PowerPC and Itanium2 processors on a TPC-H workload. Gedik *et al.* investigate sorting [53] and windowed stream joins [54] on the cell. Databases with multimedia content might also be able to use such accelerators to process images efficiently.

VIII. FUTURE WORK

Database optimization for modern hardware is an area of research where the work is never finished. Computer hardware continues to evolve at a rapid pace, constantly presenting new challenges and opportunities to researchers who push the envelope of performance. Even without the introduction of new disruptive technologies, the current pace of hardware development suggests a bright future for research in database optimizations on new hardware. Brief examples of some areas of potential future research are presented in this section.

A. Implications of Chip Multiprocessors

As shown in Section VI-B, increased on-chip parallelism presents new challenges for database implementers. If future L2 hit latency increases as predicted by Hardavellas *et al.* [41], then operations may need to be redesigned with a greater emphasis on L1 cache locality. In a related research direction, if per-thread cache space decreases, sharing the limited cache resources between threads may become more important. As described in Section VI-B, sharing the cache resources among threads can improve performance but requires careful attention to issues such as contention for frequently updated values. In short, managing cache sharing and interthread cooperation may become key components of performance optimizations. Many database operations will benefit from a careful analysis of their implementation on CMPs.

The growing presence of CMPs clearly means that database operations need to exploit on-chip parallelism. An open research topic is where to find that parallelism. Databases naturally have parallelism at many different levels. At a high level, independent transactions and queries can be processed in parallel, i.e., interquery parallelism. Many database systems are already designed to take advantage of this type of parallelism, but independent workloads may compete for on-chip resources, such as a shared cache, resulting in less than optimal performance.

This naïve parallelism was discussed in Section VI-A in the context of SMT processors. Saturating a CMP would require the number of active queries or transactions to at least equal the degree of on-chip TLP. For example, a single query would utilize only one-sixteenth of a 16-way CMP if run alone.

Parallelism can also be found within one query plan by running multiple operators concurrently; this is intraquery parallelism. By finding parallelism within one query, the underutilization described above is avoided. These operators could still compete with one another for on-chip resources, and passing data between operators would need to be carefully coordinated. Additionally, the different relative processing rates of various operators must be accounted for so one slow operator does not become a bottleneck for the entire query.

Finally, parallelism can also exists within one operator, that is, intraoperator parallelism. This type of parallelism is used in the CMP aggregation work discussed in Section VI-B [44]. For most database operations, input tuples do not depend on other tuples in the input and can therefore be processed independently in parallel. Intraoperator parallelism allows operators to be designed to avoid degenerate competition between threads for on-chip resources. Such a design may not be appropriate, however, if the amount of input does not justify the use of so many parallel threads of computation. Choosing the right type of parallelism is an open question and may depend on both the workload and the underlying hardware. These types of parallelism can also be mixed, resulting in an even larger space of possible choices.

After a mode of parallelism to exploit is identified, implementing parallel database operations remains a challenge. Parallel programming is known to be more difficult and error prone than sequential programming, and debugging is often more difficult because race conditions and deadlocks may not appear deterministically during testing. Many different threading paradigms exist, but an open question is: Which paradigm is most appropriate to database implementation? Is a new parallel programming paradigm needed for database implementation on CMPs? In a similar vein, how should database researchers address the tradeoff between ease of programming in a higher level paradigm versus achieving better performance using a more complicated lower level paradigm?

B. Larger Memory

This paper focused primarily on optimizations for in-memory query processing. The main memory size on commodity systems continues to increase. Another area of inquiry deals with implications of very large memory sizes, possibly terabytes, on database query performance. Even larger data sets will fit in memory meaning the database will need to use I/O even less. Are existing in-memory query engines up to this task or do we need new solutions as memory sizes scale up?

C. New Storage Technologies

In Section III, we introduced flash memory storage and some of the ways that its introduction is causing a rethinking of database storage. This research is just beginning and deserves to be continued. Database I/O has been heavily optimized to exploit the good sequential read and write capabilities of magnetic disks while avoiding random reads and writes, the Achilles heel of magnetic disk performance. Flash memory supports random I/O very efficiently. Increased use of flash storage means that I/O need not be optimized for sequential access. The implications of this change have yet to be fully explored.

D. Diverse Technologies

The increasingly diverse types of computer hardware also make this a very interesting time to research database optimizations on modern hardware. Just ten years ago, almost all research focused on maximizing performance on single core general-purpose processors. Today there are CMPs with diverging designs, such as the "lean camp" and "fat camp" described in Section VI-B. Additionally, graphics processors and other special-purpose accelerators hold promise for improving the performance of certain database operations. New hybrid hardware, such as the cell broadband engine, also pose new challenges and opportunities. It is interesting to optimize database performance on diverse hardware, but will database implementation and performance become tied to particular types of hardware? For example, techniques that work well on a system with a CPU and a GPU may not be the correct solution for a system with cell processors. And similarly, "lean" and "fat" CMPs may require very different database system designs to achieve optimal performance. Certain database workloads will experience higher performance on some types of hardware than others. Pairing the task with the appropriate hardware and software system is another interesting future challenge.

E. New Metrics

Raw performance metrics, such as query throughput or transaction latency, are no longer the only important metrics for measuring database performance. System cost is an emerging metric. For instance, Google builds large data-analysis systems out of relatively cheap commodity hardware. New benchmarks, such as the PennySort mentioned in Section VII-B, attempt to measure performance per dollar (or in that case, per penny). Energy usage is also becoming an increasingly important benchmark. Large data centers require huge amounts of energy to run the computing and cooling systems. Using less energy and generating less heat are both economically and environmentally sound. A new energy centric benchmark, JouleSort [55], has been proposed as a means of rating and designing systems for maximum performance per joule. The authors of the benchmark find that a system designed to perform well on the JouleSort benchmark is different

than most systems currently found in data centers [55]. Achieving good performance on systems designed for energy efficiency may require new query-processing techniques.

IX. CONCLUSION

In this paper, we have described the performance challenges that modern commodity microarchitectures pose for database workloads. We have also presented optimizations that help databases achieve high performance on modern processors. Looking toward the future, it is evident that on-chip thread parallelism will be an important component of microprocessor performance. An area that will challenge database researchers is how best to utilize multiple hardware threads for database operations. As we have described, new bottlenecks are emerging such as thread communication and coordination associated with reading and writing to shared data structures. Current research is also challenging conventional wisdom about techniques such as work sharing in the context of CMPs, highlighting the importance of carefully considering the new opportunities and challenges that emerging hardware technologies bring to database performance. As the microarchitecture environment continues to change, architecture-sensitive database research continues to adapt to new technologies and challenges. ■

Acknowledgment

The authors thank the anonymous reviewers for their constructive feedback on earlier versions of this paper.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture*, 4th ed. San Mateo, CA: Morgan Kaufmann, 2007.

[2] A. Ailamaki, D. J. DeWitt, M. Hill, and D. A. Wood, "DMBSs on a modern processor: Where does time go?" in *Proc. Int. Conf. Very Large Data Bases*, 1999, pp. 266–277.

[3] J. Zhou and K. A. Ross, "Buffering accesses to memory-resident index structures," in *Proc. Int. Conf. Very Large Data Bases*, 2003, pp. 405–416.

[4] J. Rao and K. A. Ross, "Cache conscious indexing for decision-spport in main memory," in *Proc. Int. Conf. Very Large Data Bases*, 1999, pp. 78–89.

[5] J. Rao and K. A. Ross, "Making B^+-trees cache conscious in main memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 475–486.

[6] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing main-memory join on modern hardware," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 4, pp. 709–730, 2002.

[7] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, "Improving Hash join performance through prefetching," in *Proc. Int. Conf. Data Eng.*, 2004, pp. 116–127.

[8] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving index performance through prefetching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2001, pp. 235–246. [Online]. Available: http://www.doi.acm.org/10.1145/375663.375688, http://www.acm.org/sigs/sigmod/sigmod01/eproceedings/papers/Research-Chen-et-al.pdf

[9] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of Hash to data base machine and its architecture," *New Gen. Comput.*, vol. 1, no. 1, pp. 63–74, 1983.

[10] A. Ghouting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey, "Cache-conscious frequent pattern mining on a modern processor," in *Proc. Int. Conf. Very Large Data Bases*, 2005, pp. 577–588.

[11] P. Boncz, S. Manegold, and M. L. Kersten, "Database architecture for the new bottleneck: Memory access," in *Proc. Int. Conf. Very Large Data Bases*, 1999, pp. 54–65.

[12] M. Stonebreaker, D. J. Abadi, A. Batiki, X. Chen, M. Cherniack, M. Ferreira, D. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column oriented DBMS," in *Proc. Int. Conf. Very Large Data Bases*, 2005, pp. 554–564.

[13] R. MacNicol and B. French, "Sybase IQ multiplex—Designed for analytics," in *Proc. Int. Conf. Very Large Data Bases*, 2004, pp. 1227–1230.

[14] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases," in *Proc. Int. Conf. Very Large Data Bases*, 2006, pp. 487–498.

[15] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2006, pp. 671–682.

[16] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proc. Int. Conf. Very Large Data Bases*, 2001, pp. 169–180.

[17] S.-W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 55–66. [Online]. Available: http://ww.doi.acm.org/10.1145/1247480.1247488

[18] J. Zhou and K. A. Ross, "A multi-resolution block storage model for database design," in *Proc. Int. Database Eng. Applicat. Symp.*, 2003, pp. 22–33.

[19] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger, "Clotho: Decoupling memory page layout from storage organization," in *Proc. Int. Conf. Very Large Data Bases*, 2004, pp. 696–707.

[20] R. A. Hankins and J. M. Patel, "Data morphing: An adaptive, cache-conscious storage technique," in *Proc. Int. Conf. Very Large Data Bases*, 2003, pp. 417–428. [Online]. Available: http://www.vldb.org/conf/2003/papers/S13P03.pdf

[21] T. M. Chilimbi, M. D. Hill, and J. R. Larus. (2000). Making pointer-based data structures cache conscious. *IEEE Comput.* [Online]. *33(12)*, pp. 67–74. Available: http://www.computer.org/computer/co2000/rz067abs.htm

[22] G. Graefe, "The five-minute rule twenty years later, and how flash memory changes the rules," in *Proc. Workshop Data Manage. New Hardware*, 2007.

[23] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processors," *Intel Technol. J.*, vol. 5, no. 1, Feb. 2001.

[24] J. Zhou and K. A. Ross, "Buffering database operations for enhanced instruction cache performance," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 191–202.

[25] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran, "Block oriented processing of relational database operations in modern computer architectures," in *Proc. Int. Conf. Data Eng.*, 2001, pp. 567–574.

[26] S. Harizopoulos and A. Ailamaki, "Improving instruction cache performance in OLTP," *ACM Trans. Database Syst.*, vol. 31, pp. 887–920, 2006.

[27] S. Harizopoulos and A. Ailamaki, "STEPS towards cache-resident transaction processing," in *Proc. Int. Conf. Very Large Data Bases*, 2004, pp. 660–671.

[28] K. A. Ross, "Conjunctive selection conditions in main memory," in *Proc. ACM SIGACT-SIGMOD-DIGART Symp. Principles Database Syst.*, 2002, pp. 109–120.

[29] K. A. Ross, "Selection conditions in main memory," *ACM Trans. Database Syst.*, vol. 29, pp. 132–161, 2004.

[30] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 145–156.

[31] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, "Super-scalar RAM-CPU cache compression," in *Proc. Int. Conf. Data Eng.*, 2006, p. 59.

[32] K. A. Ross, "Efficient hash probes on modern processors," in *Proc. Int. Conf. Data Eng.*, 2007.

[33] M. Zukowski, S. Héman, and P. A. Boncz, "Architecture-conscious hashing," in *Proc. Workshop Data Manage. New Hardware*, 2006.

[34] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Proc. Eur. Symp. Algorithms*, 2001, pp. 121–133.

[35] *OpenSPARC T1 Microarchitecture Specification*, Sun Microsystems, Aug. 2005.

[36] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel, 2006.

[37] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "POWER5 system microarchitecture," *IBM J. Res. Develop.*, vol. 49, no. 4/5, 2005.

[38] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah, "Improving database performance on simultaneous multithreading processors," in *Proc. Int. Conf. Very Large Data Bases*, 2005, pp. 49–60.

[39] P. Garcia and H. F. Korth, "Multithreaded architectures and the sort benchmark," in *Proc. Workshop Data Manage. New Hardware*, 2005.

[40] P. Garcia and H. F. Korth, "Database hash-join algorithms on multithreaded computer architectures," in *Proc. Conf. Comput. Frontiers*, 2006, pp. 241–251.

[41] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi, "Database servers on chip multiprocessors: Limitations and opportunities," in *Proc. Conf. Innov. Data Syst. Res.*, 2007, pp. 79–89.

[42] J. Cieslewicz, J. Berry, B. Hendrickson, and K. A. Ross, "Realizing parallelism in database operations: Insights from a massively multithreaded architecture," in *Proc. Workshop Data Manage. New Hardware*, 2006.

[43] R. Johnson, N. Hardavellas, I. Pandis, N. Mancheril, S. Harizopoulos, K. Sabirli, A. Ailamaki, and B. Falsafi, "To share or not to share?" in *Proc. Int. Conf. Very Large Data Bases*, 2007, pp. 351–362. [Online]. Available: http://www.vldb.org/conf/2007/papers/research/p351-johnson.pdf

[44] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *Proc. Int. Conf. Very Large Data Bases*, 2007, pp. 339–350. [Online]. Available: http://www.vldb.org/conf/2007/papers/research/p339-cieslewicz.pdf

[45] A. Ailamaki, N. K. Govindaraju, S. Harizoupoulos, and D. Manocha, "Query co-processing on commodity processors," in *Proc. Int. Conf. Very Large Data Bases*, 2006, p. 1267.

[46] B. T. Gold, A. Ailamaki, L. Huston, and B. Falsafi, "Accelerating database operations using a network processor," in *Proc. Workshop Data Manage. New Hardware*, 2005.

[47] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 215–226.

[48] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High performance graphics co-processor sorting for large database management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2006, pp. 325–336.

[49] N. Bandi, C. Sun, A. E. Abbadi, and D. Agrawal, "Hardware acceleration in commercial databases: A case study of spatial operations," in *Proc. Int. Conf. Very Large Data Bases*, 2004, pp. 1021–1032.

[50] C. Sun, D. Agrawal, and A. E. Abbadi, "Hardware acceleration for spatial selections and joins," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 455–466.

[51] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell microprocessor," *IBM J. Res. Develop.*, vol. 49, no. 4/5, 2005.

[52] S. Héman, N. Nes, M. Zukowski, and P. Boncz, "Vectorized data processing on the cell broadband engine," in *Proc. Workshop Data Manage. New Hardware*, 2007.

[53] B. Gedik, R. Bordawekar, and P. S. Yu, "CellSort: High performance sorting on the cell processor," in *Proc. VLDB*, 2007, pp. 1286–1207. [Online]. Available: http://www.vldb.org/conf/2007/papers/industrial/p1286-gedik.pdf

[54] B. Gedik, P. S. Yu, and R. Bordawekar, "Executing stream joins on the cell processor," in *Proc. VLDB*, 2007, pp. 363–374. [Online]. Available: http://www.vldb.org/conf/2007/papers/research/p363-gedik.pdf

[55] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "JouleSort: A balanced energy-efficiency benchmark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247522

ABOUT THE AUTHORS

John Cieslewicz received the B.S. degree from Stanford University, Stanford, CA, in 2004 and the M.S. degree from Columbia University, New York, in 2005, both in computer science. He is currently pursuing the Ph.D. degree in computer science at Columbia University.

Mr. Cieslewicz is a U.S. Department of Homeland Security Graduate Research Fellow.

Kenneth A. Ross received the B.Sc. (hons.) degree from the University of Melbourne, Melbourne, Australia, in 1986 and the Ph.D. degree in computer science from Stanford University, Stanford, CA, in 1991.

Since 1991, he has been on the Faculty of the Department of Computer Science, Columbia University, New York, where he is now a Professor.

Prof. Ross received a Packard Foundation Fellowship in 1993, a Sloan Foundation Fellowship in 1994, and a National Science Foundation Young Investigator award in 1994.