Main Memory Performance for Database Systems

Kenneth A. Ross
Columbia University
kar@cs.columbia.edu

1 Introduction

As random access memory gets cheaper, computers with large main memories become increasingly affordable. Thus for both main memory databases and disk-resident databases, the efficient use of a large main memory has become an important performance goal. For disk-based databases, CPU and main-memory related performance has begun to dominate I/O performance as the critical bottleneck [GL01, ADHW99].

It is thus imperative that database architects switch from optimizing their systems for I/O to optimizing them for memory performance. It is not even clear that monolithic commercial systems can be adequately tuned for in-memory performance, given the design decisions embedded in their implementations. Systems such as Monet that are designed for in-memory performance yield orders of magnitude improvements in query performance over commercial systems for some kinds of workload [BRK98].

In this abstract, we attempt to highlight the critical architectural issues that affect in-memory database performance, and to outline a research program for designing database systems to fit these architectural parameters.

2 Architectural Parameters

2.1 Cache Behavior

Cache memories are small, fast static RAM memories that improve program performance by holding recently referenced data [Smi82]. Memory references satisfied by the cache, called hits, proceed at processor speed; those unsatisfied, called misses, incur a cache miss penalty and have to fetch the corresponding cache block from the main memory. Modern architectures typically have two levels of cache (L1 and L2) between the CPU and main memory. While the L1 cache can perform at CPU speed, the L2 cache and main memory accesses normally introduce latencies in the order of 10 and 100 cycles respectively. Cache memories can reduce the memory latency only when the requested data is found in the cache. The effectiveness of a cache mainly depends on the memory access pattern of the application.

It has been observed by many that while CPU speeds have been improving dramatically over time, memory latency has been improving only marginally [CLH98]. As a result, the relative cost of a cache miss has been steadily growing, and is likely to continue to grow in the near future. Thus, unless special care is taken, memory latency will become an increasing performance bottleneck, preventing applications from fully exploiting the power of modern hardware.

We cannot simply apply well-known techniques for buffering disk-based data in RAM to the next level of the memory hierarchy. The cache replacement policies are coded in hardware, meaning that we cannot tune them to perform well for database workloads. Instead, we need to tune the database algorithms themsleves to work well with conventional cache replacement hardware.

2.2 Pipelining and Branch Prediction

Conditional branch instructions present a significant problem for modern pipelined CPUs because the CPU does not know in advance which of the two possible outcomes will happen. CPUs try to *predict* the outcome of branches, and have special hardware for maintaining the branching history of many branch instructions. A mispredicted branch incurs a substantial delay because many instructions in the pipeline have to be flushed, and some operations undone. [ADHW99] calculates the branch misprediction penalty for a Pentium II processor as 17 cycles.

2.3 SIMD Technology

SIMD instructions come in various flavors, including "MMX," "SSE," and "SSE2" on Intel machines, "VIS" on UltraSparc machines, and "3DNow!" on AMD machines. Additional vendors with SIMD architectures include Hewlett-Packard, MIPS, DEC (Compaq), Cyrix, and Motorola. SIMD instructions were designed to accelerate the performance of applications such as motion video and graphics. Such applications typically use compute-intensive algorithms to perform repetitive operations on large arrays of numbers.

There are many kinds of SIMD instruction besides comparison, including various arithmetic and logical operations. Within a single architecture, there are various granularities available for SIMD operation. For example, there are two additional packed-greater-than instructions in MMX: one compares eight 8-bit values, and another compares two 32-bit values.

No current compilers for high-level programming languages support SIMD in a way that automatically identifies parallelizable sections of code [SS00]. Despite the availability of some shared libraries and macro packages, the most effective way of programming with SIMD extensions is by using assembly language [SS00]. Despite being tedious and error-prone, such an approach allows the greatest flexibility and precision when coding. For tight inner loops requiring just tens of lines of assembly code, it is practical to write assembly language code even if one has to rewrite the code for each platform.

2.4 Compilers

Compilers for languages such as `C` are available across a multitude of platforms. However, in terms of performance, compilers vary substantially. In order to design algorithms for main-memory performance, an understanding of the operation of the compiler is often necessary. Further, general purpose compilers such as `gcc` do not take advantage of recently introduced special features or instructions available on specific architectures. On the other hand, some compilers produced by hardware vendors such as Intel take advantage of such features, but only on Intel architectures.

2.5 Avoiding Architecture Dependence

Modern CPU chips have broad similarities compared to one another. They all have a pipelined architecture, in which several instructions are active in the pipeline at the same time. They all have a memory hierarchy involving several levels of cache between the registers and RAM, with similar cache replacement policies coded in hardware.

On the other hand, modern CPU chips also have numerous specific differences. For example, each has its own instruction set, and its own degree of internal parallelism. Chip manufacturers face trade-offs in their designs; improving on one aspect of a chip may lead to compromises in another aspect of the chip. Different manufacturers choose different trade-offs.

Thus, when developing software that needs to run "close to the metal," we are faced with the prospect of making architecture-dependent decisions. One can imagine three levels of architecture dependence: (a) independence, in which the same code can be run across many platforms; (b) semi-dependence, in which the same code can be run on various platforms if recompiled with a change in the values of a few system-specific parameter constants; (c) dependence, in which significant amounts of new code must be written for each candidate platform.

We should aim for semi-dependence. Code should be written in (a standardized fragment of) a language like `C` that can be compiled on many platforms. When we develop a cost model, the choice of values for the various parameters will depend on the underlying architecture, and possibly also on the compiler. (This is comparable to conventional disk-based databases, where the optimal choices for parameters such as the disk block size depend to some degree on the disk device and the operating system.) A model should be broad enough to include any architecture with pipelines and a memory hierarchy. Some parameters can be measured at run-time [MBK00].

One may still be concerned that architectural differences not included in the model may influence the performance being measured. This is a classic tradeoff between the generality of a model, and its accuracy when applied to particular real systems. For example, in disk-based systems it is still common to use a model for a disk that has a single average random I/O cost and a single average sequential I/O cost. Disk device enhancements, such as track buffering, reordering of requests, and faster data flow on the outer tracks of the disk, are commonly omitted from the model even though they may affect real disk performance. Including such detailed behavior would make the model unwieldy.

We face a similar dilemma here. Accounting for pipelined processors and a memory hierarchy is a first step. We believe that these are the primary influences on performance measurements. Yet there are additional secondary influences that should be examined in future research.

3 Strategies for Enhancing Main-memory Performance

3.1 Prefetching

The idea of prefetching is to overlap the latency of retrieving data from RAM with useful computation. This approach is effective if one has advance knowledge of what data will be used in the near future. See [VL00] for a survey. An obvious example of such behavior is a sequential scan of an array. One can prefetch the soon-to-be-processed elements of the array while performing computations on the current elements of the array.

3.2 Utilizing the Cache

Another approach involves making sure that the cache is used effectively by clustering contemporaneously accessed elements within a cache line. [CHL99, CDL99] describe two general-purpose transformations that can be applied to `C` programs. One such transformation restructures a `struct` element so that attributes that are accessed together are adjacent, and within a cache line. A second transformation modifies the memory allocation method so that referenced objects are placed in the same cache line as a popular referencing object.

One can do more sophisticated transformations if one is interested in optimizing a particular algorithm or data structure. For example, [RR99, RR00] show how to define index structures that fit more keys into a cache line than other proposed methods. As a result, fewer cache lines need to be accessed, and the method has better cache behavior.

3.3 Avoiding Branch Mispredictions

To avoid branch mispredictions, one approach is to try to avoid conditional branches. For example, to evaluate the condition `f1(r) && f2(r) && ... && fk(r)`, a traditional compiler would generate code with `k` conditional branches. One could alternatively write the code as `f1(r) & f2(r) & ... & fk(r)` which has just one conditional branch. Which of the two methods (or combinations of the methods) is best depends on the selectivity of the conditions and the cost of the functions, as well as the cost of a branch misprediction. In submitted work, we have looked at this problem in the context of applying multiple selection conditions to an array of records. On different parameter ranges, different methods win, often by a factor of two or more for common examples.

3.4 SIMD

SIMD operations allow low-level operations to be performed on multiple data items in parallel. There seems to be much opportunity for using such methods in database systems, since many database operations involve the repetitive processing of data elements.

In submitted work, we have applied SIMD techniques to the problem of indexing. We use the SIMD instructions to compare the search key with two data elements at the same time, rather than one. Apart from the improved parallelism, we can additionally avoid branch instructions because the results of the tests are available as data values that can be manipulated arithmetically.

3.5 Blocked Algorithms

Suppose one wishes to join two relations that are both much larger than the cache. Then one can avoid a substantial number of cache misses by dividing the relations into blocks that are smaller than the cache, and perform sub-joins block by block [SKN94]. This technique makes use of temporal data reference locality.

3.6 Miscellaneous Techniques

Other important techniques include code specialization, limited use of nonbasic data types, columnwise storage, and data alignment. We omit discussion of these techniques here.

4 Putting it all Together

Even if we optimize each piece of the system to perform well in isolation, we may observe degraded performance when combined. For example, suppose we have implemented a main-memory database system in which there are many concurrently running processes, each doing some useful work. A process executes for a time-slice, then is switched out by the operating system until its next time-slice arrives.

In the interim, much computation may have taken place. That computation may have polluted the cache, leading to poor cache behavior. This effect would be particularly important for methods relying on temporal locality, such as in Section 3.5. Increased time-slices are possible up to a point, but time-slices cannot be increased without limit.

High-level design decisions are necessary. These decisions affect most of how the database system will be implemented. For example, one could try to design the memory management subsystem to allocate memory in a way that reduces expected data contention. Another example might be the use of a convention that database algorithms avoid polluting the cache; when possible they read data directly to the registers from RAM (using special move instructions provided by the hardware). Such a convention will be effective only if employed consistently by all algorithms in the system.

5 Conclusion

There are many issues to be explored for main-memory performance of database systems. These issues are important, and will be of increasing importance as CPU-level issues such as the RAM latency come to dominate the performance.

References

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the International Conference on Very Large Data Bases*, pages 266–277, September 1999.

[BRK98] P. A. Boncz, T. Ruhl, and F. Kwakkel. The drill down benchmark. In *Proceedings of the VLDB Conference*, pages 628–632, 1998.

[CDL99] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[CLH98] Trishul M. Chilimbi, James R. Larus, and Mark D. Hill. Improving pointer-based codes through cache-conscious data placement. Technical Report 98, University of Wisconsin-Madison, Computer Science Department, University of Wisconsin-Madison Madison, Wisconsin 53706, 1998.

[GL01] Goetz Graefe and Per-ake Larson. B-tree indexes and cpu caches. In *Proceedings of the 20th ICDE Conference*, 2001.

[MBK00] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? - Dissecting CPU and Memory Optimization Effects. In *Proceedings of the International Conference on Very Large Data Bases*, September 2000.

[RR99] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th VLDB Conference*, pages 78–89, 1999.

[RR00] Jun Rao and Kenneth A. Ross. Making B$^+$-trees cache conscious in main memory. In *Proceedings ACM SIGMOD Conference*, pages 475–486, 2000.

[SKN94] A. Shatdal, C. Kant, and J.F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the International Conference on Very Large Databases*, pages 510–521, 1994.

[Smi82] Alan J. Smith. Cache memories. *ACM Computing Surverys*, 14(3):473–530, 1982.

[SS00] Nathan Slingerland and Alan Jay Smith. Multimedia extensions for general purpose microprocessors: A survey. Technical report CSD-00-1124, University of California at Berkeley, 2000.

[VL00] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.