Databases and Hardware: The Beginning and Sequel of a Beautiful Friendship

Anastasia Ailamaki
EPFL and RAW Labs
natassa@epfl.ch

ABSTRACT

Fast query and transaction processing is the goal of 40 years of database research and the reason of existence for many new database system architectures. In data management, system performance means acceptable response time and throughput on critical-path operations, ideally with scalability guarantees. Performance is improved with top-of-the line research on data processing algorithms; efficiency, however, is contingent on seamless collaboration between the database software and hardware and storage devices. In 1980, the goal was to minimize disk accesses; in 2000, memory replaced disks in terms of access costs. Nowadays performance is synonymous to scalability; scalability, in turn, translates into sustainable and predictable use of hardware resources in the face of embarrassing parallelism and deep storage hierarchies while minimizing energy needs - a challenging goal in multiple dimensions.

We discuss work done in the past four decades to tighten the interaction between the database software and underlying hardware and show that, as application and microarchitecture roadmaps evolve, the effort of maintaining smooth collaboration blossoms into a multitude of interesting research avenues with direct technological impact.

1. INTRODUCTION

Ever since the first steps of computer science, data management has been an active research and industrial field. Data management systems have been an enabler for numerous influential applications, from the ubiquitous transaction processing, to big data analytics (e.g., scientific exploration, sensor networks, business intelligence) and high-performance online services (e.g., social networks like Twitter and Facebook, or realtime complex event processing financial applications). These ever-evolving applications have been the driving force for many innovations in the database and architecture communities for several decades: systems have had to evolve along with the applications to support them efficiently.

Emerging application requirements in recent years have led to a widespread belief that "one size does not fit all" and that specific applications require specialized system designs. This led to a multitude of system designs, each geared towards a different application. Practitioners use streaming engines for data arriving in high-velocity streams, column-oriented engines for data analytics, main-memory-based systems for transactions, etc. The design space contains trade-offs along many dimensions, involving choice of i) data layout (row oriented, column oriented, hierarchical or hybrid), ii) storage device (disk, SSD or main memory), etc. Most data management systems "pick a side" along each of these dimensions.

The specialization trend, along with the increasing complexity of data management applications and the introduction of ever more sophisticated hardware, inspire customized algorithms, many of them using hardware accelerators such as GPUs and FPGAs. These algorithms, however, often make many assumptions about data formats, distributions, workload types and hardware properties. It is therefore almost impossible to integrate these algorithms in any other system; instead of re-using efficient, established technology, system developers must "reinvent the wheel". Instead of having to choose between sub-optimal general purpose solutions and a multitude of specialized solutions that don't quite fit the specific combination of hardware and application, the next generation of applications calls for customized (or even *customizable*) systems that choose the optimal hardware-conscious data storage and processing algorithm dynamically.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

2. MAJOR HARDWARE TRENDS

Recent hardware trends in the areas of processing and storage hardware deeply affect data management applications. This section identifies their most influential characteristics.

2.1 Abundant Heterogeneous Parallelism

Processor designs are rapidly becoming heterogeneous, increasing in complexity and following non-uniform architectures. In step with Moore's Law, hardware provides increasing opportunities for parallelism rather than faster processors since 2005. Therefore, instead of increasing frequency, we observe an increase in the number of cores on a processor. Today's low power multicores have dozens of cores on the same chip. Exploiting parallelism is crucial for utilizing the available architectural resources and enabling faster software. Designing scalable systems that can take advantage of the underlying core parallelism, however, remains a challenge [1]. In traditional high-performance transaction processing, the inherent communication leads to scalability bottlenecks on today's multicore and multisocket hardware. Even systems that scale on one generation of multicores often fail to scale up on the next generation. On the other hand, in traditional online analytical processing, the database operators that were designed for single-core processors fail to exploit the abundant parallelism offered by modern hardware. The continuously changing hardware is forcing data management systems to evolve too.

As the number of processing cores increases with ever more complex topologies, the low latency interconnects minimize distances between servers. Remote Direct Memory Access (RDMA) technology allows applications to access memory on a remote machine directly, without involving the operating system or the processor. By using RDMA over fast network fabrics, such as Infiniband and converged Ethernet, the latency of remote machine accesses is comparable to the latency of accessing remote memory in a traditional big-memory machine. In the near future, rack-scale systems, akin to "cluster-in-a-box" [5], will use highly customized system-on-a-chip (SoC) nodes, containing only processing cores, memory, and I/O interfaces and communicating via low-latency interconnect fabrics. Software designers eventually will tune their applications to rack-scale systems [7], just as they did when large shared-memory machines and supercomputers started being used for enterprise and scientific applications respectively; applications were rewritten to emphasize on locality, a requirement for good performance.

Decreasing energy budgets is another important trend leading to dark silicon [6]. As a consequence, heterogeneous chips containing specialized data management accelerators are becoming more appealing. GPUs are a popular target for data management operators for over a decade. Another approach is to use reconfigurable chips, such as field programmable gate arrays (FPGAs), to offload selected data processing paths. However, further demand for energy efficiency has sparked a number of proposals ranging from custom chips that accelerate individual operations such as partitioning and hashing, to query task accelerators, to designs that specialize chips to complete analytical queries [19].

2.2 Deep Storage Hierarchies

Traditionally, data management systems have been designed with disks as the primary storage, the main memory acting as the caching layer, and tape as the archival storage. Nowadays, however, as costs of DRAM keep decreasing, the storage hierarchy has shifted upwards with the main memory serving as the primary data storage and processor's caches keeping track of the data processing state. Given large main memories that can hold working sets of many applications, we have witnessed a flurry of main-memory-centric OLTP and OLAP system designs. Yet, in many cases, the large number of concurrent user requests and the increasingly complex queries they issue force storing data on disk. Hence, modern data management systems need to be able to use different levels of the storage hierarchy interchangeably [3].

A number of memory technologies are making the storage hierarchy more complex. Flash-based solid state drives (SSD) are a maturing technology that offers persistent storage with faster access times compared to traditional spinning disks. Even though this makes them an attractive drop-in replacement for disks, it is not very efficient as it exacerbates poor durability and can cause unpredictable performance that stems from wear leveling. Despite inspiring many flash-optimized data management operators as well as systems that use flash as a caching layer between main memory and the spinning disk, the jury is still out on the optimal way to integrate SSDs into data management systems [15].

Non-volatile RAM is a family of emerging memory technologies that promise to offer persistence of flash at the speed and density of DRAM. Although competing technology proposals offer different performance and durability characteristics, they all have in common persistence as well as the access latencies between DRAM and flash, and higher durability than flash. Also, they exhibit asymmetry in read and write latencies common to disks, while offering byte addressability of the main memory. These properties make them a compelling part of the storage hierarchy that offers many interesting research opportunities [18].

Overall, the hardware platforms are offering software designers more and more parallelism in many shapes and forms. Therefore, data management algorithms need to be as general and modular as possible in order to take advantage of the available parallelism and adapt to the storage hierarchy for every specific application.

3. OLTP ON MODERN HARDWARE

Making a sophisticated software system hardware-conscious is an elaborate task. This section illustrates the challenges posed by the evolving hardware to data management by focusing on a single application class (transaction processing) and a single hardware element (the processor). We discuss how well modern disk- and main-memory-oriented transaction processing systems utilize the cache hierarchies and the abundant non-uniform parallelism of modern multi-socket multi-core servers.

3.1 Improving Micro-architectural Behavior

Transaction processing systems have been known to show sub-optimal micro-architectural behavior. Up to 80% of the execution cycles go to memory stalls; L1 instruction cache and/or long latency data misses from the last-level cache account for the majority of the overall stall time. As a result, the instructions-per-cycle (IPC) fraction's value barely reaches one on machines that can retire up to four instructions in a cycle [1]. This section overviews several methods improving utilization of hardware resources for transaction processing systems, and presents micro-architectural analysis of main-memory optimized transaction processing systems.

Instruction reuse. The key observation in transaction processing is that transactions exhibit significant overlap in their instruction footprint due to the common database operations they execute such as index probe, index scan, update tuple, etc. This offers an opportunity to achieve better L1 instruction cache locality by scheduling transactions in a way that would enable instruction reuse across transactions based on their common actions [17].

STEPS, a software technique, and STREX, a hardware technique, time-multiplex a batch of similar transactions by context-switching on a single core so that the instructions fetched by one transaction in the batch are reused by all the other transactions of the batch on the same core. Going one step further, SLICC, a hardware technique, and ADDICT, a software-guided hardware technique, spread the computation by dynamically migrating the transactions to multiple cores to allow multiple transactions to reuse the cached instructions on multiple cores.

While STEPS and STREX are single-core algorithms, SLICC and ADDICT need to have a number of cores large enough for the aggregate capacity of L1 instruction caches to fit the instruction footprints of all the concurrently running transactions. Being software-level/software-guided methods, STEPS and ADDICT are able to use statistical information from the software side to determine the context-switch/migration points at which the transactions should be context-switched or migrated to different cores. STREX and SLICC, however, are completely software-oblivious, missing the opportunity of using software hints to decide on the context-switch/ migration points. Instead, they use heuristics based on cache miss rates to find the context-switch and migration points.

Dictating which transactions should run on which cores is harder and less efficient at the software-level than at the hardware-level, because software-level scheduling is platform-dependent and requires drastic modifications to the operating system software. On the other hand, software-level hints can substantially improve the

prediction of better context-switch/migration points. Being a software guided hardware mechanism, ADDICT exploits both the cheap hardware level transaction scheduling mechanism and the software level hints to find the better migration points. Moreover, ADDICT exploits the aggregate L1 instruction cache capacity of the multi-core architectures. As a result, among the four proposed solutions, ADDICT reduces the instruction misses most drastically (by 85%) due to its careful hardware-software co-design [17].

Main-memory optimized systems. Recent years have witnessed the rise of main-memory optimized transaction processing systems. Unlike the traditional disk-based systems, main-memory optimized systems process all the data in main memory, and therefore can omit the buffer manager component. In addition, they usually adopt more lightweight concurrency control mechanisms, cache-conscious index structures, code generation, and a cleaner code base since they are usually designed from scratch. Hence, one would expect better micro-architectural behavior from main-memory systems.

Despite all the design differences and system-specific optimizations, main-memory OLTP systems exhibit very similar behavior to the disk-based ones: more than half of the execution time goes to memory stalls where L1 instruction cache and/or the long-latency data misses from the last-level cache are the dominant factors in the overall stall time. Main-memory systems adopting aggressive transaction compilation techniques can reduce the instruction misses to almost zero; however, the reduction in instruction stalls amplifies the impact of last-level cache data misses. As a result, the IPC value barely reaches one on machines that are able to retire up to four for both traditional disk-based and new generation main-memory transaction processing systems. Therefore, pure software-level optimizations are not enough to fully exploit micro-architectural features. As in ADDICT, one needs to optimize the hardware and software together to improve the utilization of the micro-architectural resources.

3.2 Non-uniform core topologies

Transaction processing applications traditionally run on high-end servers. Up until recently, such servers had uniform core-to-core communication latencies. With the introduction of multi-socket multi-cores, however, for the first time we have *Islands*, i.e., groups of cores that communicate fast among themselves and slower with other groups. In current mainstream servers, each chip is an Island; as the number of cores on a chip increases, however, soon we will identify Islands within a single chip.

Given all the effort to make transaction processing systems scale on multi-cores, one would expect that they perform well on multi-sockets, too. However, this is not the case. To be fair, multi-sockets are not an ideal platform for distributed transaction processing deployments either. In fact, no single optimal configuration exists: the ideal configuration depends on the hardware topology and the workload. For example, a shared-nothing configuration is twice as fast as a shared-everything one for perfectly partitionable workloads. By contrast, the situation is completely the opposite for non-partitionable workloads and workloads that exhibit heavy skew. Island-sized shared-nothing configurations fall between the two extremes, so none of the previous alternatives can be applied directly.

One way to address this challenge is by adapting a scalable logically partitioned shared-everything system to Islands using automatic partitioning of the system state and dynamically assigning worker threads to specific partitions [16]. In this way, one can remove all intersocket accesses from the critical path of transaction execution for perfectly partitionable workloads. For other workloads, we rely on finding a good partitioning and placement scheme that balances the load across partitions and minimizes the synchronization overheads across Islands. Finally, to ensure robust performance in the presence of shifting workload patterns, we use a lightweight monitoring mechanism to detect and quick repartitioning mechanism to adapt to any change.

4. NIMBLE DATA MANAGEMENT

A defining characteristic of modern data management applications is their diverse requirements. Despite the increasing processing capabilities of modern hardware platforms, having to deal with multiple, heterogeneous applications renders many brute force / general-purpose approaches ineffective, and requires a tighter co-design of data management systems for specific hardware and applications. The key is to make dynamic behavior the primary design goal of a modern data management system, as shown in the next two cases of rack-scale OLTP and just-in-time data management.

4.1 Rack-scale OLTP

Software designers often choose to augment existing systems when new requirements arise as we exemplified over the long history of general purpose DBMS which ended up being superseded by an array of specialized systems. We illustrate challenges of this approach on the example of transaction processing for future highly parallel rack-scale platforms.

OLTP systems are typically designed to either scale out or scale up, yet neither is optimal for rack-scale hardware. Scale-out systems mostly utilize shared-nothing designs that run on clusters of machines and offer very high performance for easily partitionable workloads due to explicit data partitioning and reduced synchronization. However, they completely ignore the opportunities for optimizing communication between nodes located on the same physical machine thus lowering per machine throughput. Scale-up systems typically use main memory optimized shared-everything designs that rely on the precise synchronization among threads to achieve robust high performance. However, when deployed in a distributed way, they are sensitive to the delays introduced by the network communication.

Inadequacies of state-of-the-art concurrency control and coordination protocols stem from scale-up and scale-out requirements respectively. On the one hand, concurrency control protocols for main-memory-optimized, scale-up designs need to minimize the duration of any critical sections so as not to introduce any scalability bottlenecks. This makes them sensitive to delays introduced in the critical path of transaction execution. On the other hand, coordination protocols aim at minimizing the number of messages between nodes in the distributed system as communication latencies dominate all other delays in the system. This allows them, however, to add significant local processing overhead, that is prohibitive for lean main memory optimized systems.

Neither scale up nor scale out designs can be easily augmented to fit the rack-scale hardware platforms. To that end, concurrency control protocols need to become resilient to the communication delays, while the coordination protocols need to become lighter to capture the best of both worlds. Judicious use of semantic information from the application with the focus on locality on every node is a promising way toward efficient rack-scale OLTP designs.

4.2 Just-in-time Data Management

Traditional data management systems have multi-layer architectures, each layer being responsible for enriching the previous one's functionality, and operating on a different level of abstraction (one goal being logical and physical data independence). Every layer of abstraction, however, bring indirection along, which needs to be handled in the code base. As a result, the query engines of

many mature systems become increasingly general-purpose and hardware-agnostic, losing optimization opportunities associated with specific use cases; the abstractions they use are "too high".

One work-around traditional engines employ is to duplicate, hard-code, or pre-compile certain "frequent code paths", hoping to increase performance. This approach, however, leads to bloated code bases and does not scale, as it is impractical to pre-specify the code needed for any ad-hoc query. "Pre-cooked" query operators introduce significant interpretation overhead during query evaluation. The overarching problem of these operators is that gain in performance is countered by loss of generality so while some applications enjoy faster response times, the performance of others suffers.

An alternative approach to using "pre-cooked" query operators is generating the code to answer database queries *just-in-time*. By using just-in-time code generation, one gets the best of both worlds; engines can still reason in high-level abstractions, while at the same time producing specialized code for each incoming query, and coalescing all the system layers at runtime. Depending on the system's code generation capabilities, the resulting code can target specific data types / formats, underlying hardware features, etc.

Years after first being applied in System R, runtime code generation recently re-gains popularity, the modern hardware capabilities and characteristics being a contributing factor. HIQUE is a notable effort, dynamically instantiating code templates to generate hardware-specific (e.g., cache-conscious) code. HyPer [12], Impala [14], and Tupleware [4] employ more sophisticated code generation techniques, using the LLVM compiler infrastructure. Hyper specializes to the underlying hardware by advocating "register-based processing"; it tries to avoid CPU register contents being flushed to memory. The query engine of HyPer is, therefore, push-based to simplify the control flow in the generated code and to exploit data locality via pipelining. On the other side of low-level code generation, LegoBase [13] advocates "abstraction without regret" and staged compilation. By using this staged approach, LegoBase can exploit and combine opportunities from the software and hardware layers. The query engine and the optimization rules of LegoBase are both written in the high-level language Scala. Different optimizations can be applied in every query translation step from the original Scala representation to the C code that is eventually generated. All the aforementioned systems are essentially self-compiling DBMS, shedding away unnecessary abstractions to become hardware-conscious.

ViDa [10] is a new database engine which uses *data virtualization* to produce custom database systems just-in-time, while minimizing the performance hit. ViDa virtualizes data by abstracting it out of its original form and manipulating it regardless of the way it is stored or structured. Vida accesses heterogeneous datasets efficiently, regardless of their location, format, and of the queries that are to be executed over them. ViDa handles the underlying model heterogeneity by using an expressive internal query language able to handle relations, hierarchies, and arrays, as well as transformations between these collection types. Existing languages can also be translated to ViDa's internal language, so that users have the power to choose the language best suited for an analysis. ViDa also employs low-level code generation techniques to adapt its entire query engine just-in-time. By enhancing techniques described in H2O [2] and RAW [11], the ViDa engine i) dynamically adapts its data storage layout based on the incoming query workload, and ii) generates its access paths just-in-time to adapt to the underlying data files and to the incoming queries.

ViDa is the first *just-in-time* database system: it is generated instantaneously and then it customizes itself fully and dynamically to the user's hardware, data, and queries. All the software and hardware heterogeneity is masked away from the user, who can launch queries on top of her "virtual", custom database without data preparation. "Self-designing" data systems adopt a similar philosophy: a database is designed automatically to fit specific data, queries and hardware platform [9]. One promising step is the Ocelot project, which demonstrated how CPU and GPU operators can be combined to efficiently run queries over the MonetDB column store [8]. By combining query-driven, hardware-aware, code-generated operator pipelines optimized for the current system resource availability across processing and storage hierarchies, one can efficiently process any query on any hardware platform.

5. SUMMARY

In this paper, we first discuss major hardware trends with lasting impact on the data management systems. Then we illustrate the complexity of hardware/data management co-design through challenges and promising solutions in the interaction of processors and transaction processing systems. Finally, we argue in favor of just-in-time design of data management systems that dynamically adapt to hardware and application requirements.

Acknowledgements: Danica Porobic, Utku Sirin, and Manos Karpathiotakis (EPFL).

6. REFERENCES

[1] A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, and I. Psaroudakis. How to stop under-utilization and love multicores. In *SIGMOD*, pages 189–192, 2014.

[2] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *SIGMOD*, 2014.

[3] R. Barber et al. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *ICDE*, pages 1246–1252, 2015.

[4] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*, 2015.

[5] A. Dhodapkar, G. Lauterbach, S. Li, D. Mallick, J. Bauman, S. Kanthadai, T. Kuzuhara, G. Xu, and C. Zhang. SeaMicro SM10000-64 Server: Building Datacenter Servers Using Cell Phone Chips. In *HotChips*, 2011.

[6] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, 2011.

[7] T. Harris. Hardware Trends: Challenges and Opportunities in Distributed Computing. *ACM SIGACT News*, 46(2):89–95, 2015.

[8] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[9] S. Idreos. DASlab: The Data Systems Laboratory at Harvard SEAS. *SIGMOD Record*, 44(1):46–51, 2015.

[10] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*, 2015.

[11] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7(12):1119–1130, 2014.

[12] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the Hybrid OLTP & OLAP Main-Memory Database System Hyper. *IEEE DEBull*, 36(2):41–47, 2013.

[13] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[14] M. Kornacker et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[15] I. Petrov, R. Gottstein, and S. Hardock. DBMS on modern storage hardware. In *ICDE*, pages 1545–1548, 2015.

[16] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. Atrapos: Adaptive transaction processing on hardware islands. In *ICDE*, pages 688–699, 2014.

[17] P. Tözün, I. Atta, A. Ailamaki, and A. Moshovos. ADDICT: advanced instruction chasing for transactions. *PVLDB*, 7(14):1893–1904, 2014.

[18] S. D. Viglas. Data Management in Non-Volatile Memory. In *SIGMOD*, 2015.

[19] L. Wu, O. Polychroniou, R. J. Barker, M. A. Kim, and K. A. Ross. Energy Analysis of Hardware and Software Range Partitioning. *ACM TOCS*, 32(3), 2014.