The Case For Heterogeneous HTAP

Raja Appuswamy* Manos Karpathiotakis* Danica Porobic* Anastasia Ailamaki*‡
*École Polytechnique Fédérale de Lausanne ‡RAW Labs SA
firstname.lastname@epfl.ch

ABSTRACT

Modern database engines balance the demanding requirements of mixed, hybrid transactional and analytical processing (HTAP) workloads by relying on i) global shared memory, ii) system-wide cache coherence, and iii) massive parallelism. Thus, database engines are typically deployed on multi-socket multi-cores, which have been the only platform to support all three aspects.

Two recent trends, however, indicate that these hardware assumptions will be invalidated in the near future. First, hardware vendors have started exploring alternate non-cache-coherent shared-memory multi-core designs due to escalating complexity in maintaining coherence across hundreds of cores. Second, as GPGPUs overcome programmability, performance, and interfacing limitations, they are being increasingly adopted by emerging servers to expose heterogeneous parallelism. It is thus necessary to revisit database engine design because current engines can neither deal with the lack of cache coherence nor exploit heterogeneous parallelism.

In this paper, we make the case for Heterogeneous-HTAP (H^2TAP), a new architecture explicitly targeted at emerging hardware. H^2TAP engines store data in shared memory to maximize data freshness, pair workloads with ideal processor types to exploit heterogeneity, and use message passing with explicit processor cache management to circumvent the lack of cache coherence. Using Caldera, a prototype H^2TAP engine, we show that the H^2TAP architecture can be realized in practice and can offer performance competitive with specialized OLTP and OLAP engines.

1. INTRODUCTION

The past few years have witnessed a rise in demand for real-time business intelligence. Organizations increasingly require analytics on fresh operational data to derive timely insights. To meet these requirements, database engines have to efficiently support hybrid transactional and analytical workloads (HTAP) over shared data. Designing a database engine that can serve mixed workloads efficiently is challenging, because OLTP workloads require ACID semantics, high throughput, and performance isolation, while OLAP workloads require interactive response times and data freshness.

Database engines meet these conflicting demands by relying on hardware to support three important functionalities. First, they rely on global shared memory to store a single copy of data that can be accessed by both OLTP and OLAP workloads. Second, they rely on cache coherence to guarantee that two threads running on different cores see a consistent view of data stored in shared memory despite layers of caching. Third, they rely on abundant parallelism to concurrently execute OLTP and OLAP queries. Despite providing massive parallelism, accelerators like GPGPUs have traditionally neither shared memory nor maintained coherence with CPUs. Thus, contemporary database engines are designed to be deployed on high-end multi-socket multi-cores.

Two recent trends, however, necessitate revisiting contemporary database engine design. First, as we move from the multi-core era to the many-core one, maintaining coherence across hundreds of core-private caches has become challenging. Architecture researchers and hardware vendors have started exploring many-core designs that support global shared memory but not system-wide cache coherence [6, 7, 19, 33, 49]. Second, over the past few years, GPGPUs have evolved from memory-limited, niche accelerators into general-purpose processors that support, among other advanced features, globally shared address space and pageable virtual memory. Based on these trends, emerging hardware will likely have three salient properties: i) heterogeneous parallelism, ii) global shared memory, and iii) no system-wide cache coherence. Current database engines are a poor match for emerging hardware because they can neither deal with the lack of cache coherence nor exploit heterogeneous parallelism. As a result, despite underutilizing hardware resources, current engines deployed on emerging hardware will continue to suffer from a "house pattern" [43]: OLTP and OLAP workloads will negatively interfere with each other due to resource contention.

This paper presents Heterogeneous-HTAP (H^2TAP), a new architecture for designing database engines explicitly targeted at emerging hardware. The H^2TAP architecture requires database engines to address all three aspects of emerging hardware explicitly by adhering to two design principles: i) make heterogeneity a first-class design citizen, ii) decouple shared memory from cache coherence. Using these principles, H^2TAP database engines exploit heterogeneity by pairing processors with their ideal workloads, provide

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

GPU	Architecture	Cores	FP32 Power (GFlops)	Mem cap (MB)	Mem b/w (GB/s)	I/f type	I/f b/w (GB/s)
GeForce 8800	Tesla	128	345.6	768	103.7	PCIe 1.0	4
GTX 580	Fermi	512	1581.1	1536	192.3	PCIe 2.0	8
GTX 780 Ti	Kepler	2304	3976.7	3072	288.4	PCIe 3.0	16
GTX 980 Ti	Maxwell	2816	5632	6144	336	PCIe 3.0	16
GTX 1080 Ti	Pascal	3328	10696	10240	400	NVLink	80-200

Table 1: Processing power, memory capacity, and interconnection bandwidth of consumer-grade NVIDIA graphics cards across generations

data freshness for OLAP workloads by storing data in globally shared memory, and use message-passing-based parallelism instead of shared-memory parallelism to scale OLTP workloads even in the absence of cache coherence. We validate the H^2TAP architecture by designing and implementing Caldera, a prototype H^2TAP engine. Our evaluation shows that Caldera can provide transactional throughput comparable to state-of-the-art OLTP engines while providing interactive response time and data freshness for analytical queries using GPGPUs.

2. EMERGING SERVER HARDWARE

This section presents the characteristics of emerging hardware and their mismatch with modern database engines.

2.1 Generalization of GPGPUs

Traditionally, GPGPUs suffered from two major limitations. First, applications that used GPGPUs had to manage host (CPU) and device (GPU) memory separately, thus complicating programmability. Second, GPU device memory capacity was too limited to store all data. Therefore, applications had to manually copy data from system to device memory via the slow PCIe bus before executing a computation on the GPU. As a result, despite work that showed that GPGPUs can provide substantial improvement in performance over CPUs [10, 14, 17, 18, 51], they were not widely used in the industry because analytical queries running on GPGPUs spent most of their time transferring data. As Table 1 shows, however, GPGPUs are evolving from memory-limited accelerators for niche domains to general-purpose processors with radical improvements along the dimensions of performance, interfacing, and programmability[1].

Performance. The latest Pascal GPUs offer 16× higher processing power and 13.3× more memory capacity than their Tesla counterparts. GTX 1080 Ti will have an order of magnitude more cores and 4× higher memory bandwidth than even state-of-the-art multi-core CPUs. Furthermore, GPU cards which are customized for compute acceleration typically pack 2× more memory capacity and processing power over these consumer-grade graphics cards.

Interfacing. PCIe 3.0 already offers 4× higher bandwidth compared to PCIe 1.0, and PCIe 4.0 is expected to double the bandwidth again. In addition, NVIDIA has recently announced NVLink [38], an energy-efficient, high-bandwidth GPU-CPU or GPU-GPU interconnect that will offer at least 5× the bandwidth of the current PCIe 3.0 bus. NVLink is already being used to interconnect IBM Power CPUs and NVIDIA GPUs in Summit and Sierra, two supercomputers commissioned by the U.S DoE [40].

[1]While this work uses NVIDIA terminology, all concepts and contributions presented apply to AMD GPGPUs as well.

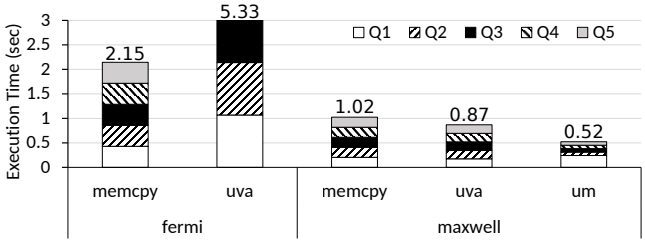

Figure 1: Scan execution time under Fermi/Maxwell GPUs

Programmability. Since CUDA 4.0, NVIDIA Fermi GPUs have supported Unified Virtual Addressing (UVA) [37], which enables GPUs and CPUs to share a single address space. The Kepler architecture added support for Unified Memory (UM) in CUDA 6.0 [37]. UM enables applications to offload memory management entirely to the CUDA runtime, which automatically tracks memory accesses and migrates data to host or device memory depending on the access patterns to improve locality. With CUDA 8.0, the Pascal architecture further extends UM with support for virtual memory-based page faulting in GPU; data allocated on the CPU is automatically faulted in and moved to the GPU one page at a time, only when accessed. Applications can thus oversubscribe GPU memory, i.e., allocate a chunk of memory larger than the GPU device capacity and access it using the same address pointer across CPUs and GPUs.

Figure 1 quantifies the net effect of some of these improvements by showing the results from a microbenchmark that executes five filter queries over a 2GB column of integers using an M2090 Fermi GPU and a GTX 980 Maxwell GPU. Each query launches a *kernel* – a function that all the threads of a GPU device execute in parallel. The three cases in the graph present scenarios where memory is allocated separately on device and host, requiring an explicit copy operation ("memcpy"), or memory is allocated using UVA/UM and requires no copying. In the memcpy case, we report the total time taken to perform the host-to-device input copy, kernel execution, and device-to-host output copy. For UVA and UM, we report the time to execute the kernel.

There are four important observations to be made. First, the memcpy case shows a 2× improvement because of the improvement in bandwidth from 8 GB/s under PCIe 2.0 (Fermi) to 16 GB/s under PCIe 3.0 (Maxwell). Second, while UVA was 2.5× slower than memcpy under Fermi, it is 1.18× faster under Maxwell. This shows that UVA enables efficient CPU–GPU data sharing. Third, under UM, the first query takes 0.24 seconds and is 1.5× slower than under UVA. The remaining queries, however, execute in 0.07 seconds, and are 2.5× faster. After the first query, the CUDA runtime automatically migrates the input array allocated in

UM over to the GPU. Thus, subsequent queries are unaffected by the PCIe bandwidth limitation. This performance improvement required no programming effort and shows the locality benefit of using UM. Finally, comparing Fermi UVA and Maxwell UVA/UM, the execution time gets a 6× reduction with UM, and a 2.46× reduction with UVA. All these speedups are noteworthy because i) they purely stem from improvements in interfaces and programmability, as the kernel does little computation, and ii) they show that efficient CPU–GPU data sharing is possible.

2.2 Specialization of CPUs

In stark contrast to the generalization of GPGPUs is the increasing specialization of commodity multi-socket multi-cores. An aspect of specialization which is particularly relevant to database designers is the design of hardware cache coherence (CC). All widely used multi-cores provide CC-shared memory to ensure that memory store operations performed by one core are visible to load operations performed by another core despite multiple levels of caching. CC also forms the framework for features like atomics on shared memory words and Hardware Transactional Memory. However, as the number of cores increases, the cost and complexity needed to maintain coherence across all core-private caches is also increasing dramatically [33, 50]. Research has also shown that CC presents scalability challenges for latency-sensitive workloads [33, 34].

While designing scalable CC protocols for multi-cores continues to be a challenging topic [32], hardware vendors have started investigating alternative multi-core architectures that vary widely with respect to CC support. The latest Haswell processors support three modes of CC; choosing the right mode impacts the latency and bandwidth of both core-to-core data transfers and memory accesses [34]. SoCs like TI OMAP4, OMAP5, and Samsung Exynos, which are based on the ARM v8 specification, group cores into multiple domains such that coherence is maintained within but not across domains. Intel SCC [19] is a 48-core processor that provides non-CC shared memory with core-to-core message passing capability. IBM Cell Broadband Engine [16] is a single-chip multiprocessor with eight non-CC Synergistic Processor Elements (SPE) optimized for data processing. Given such variation among processors in providing CC, several researchers argue that system-wide CC may no longer be available in the near future [6, 7, 31], and are building software such as operating systems [5, 6, 31, 49], file systems [20], memory management libraries [7], and runtime libraries [11, 29], explicitly targeted at emerging non-CC systems.

2.3 Database engines on emerging hardware

Putting the hardware trends together, we believe that in the near future, the servers that will be used to deploy database engines will have three salient properties: 1) they will support heterogeneous parallelism with CPUs that excel at latency-critical *task-parallel* workloads and GPGPUs that excel at throughput-heavy *data-parallel* workloads, 2) similar to contemporary servers, they will support a global address space that is shared across all processors, and 3) unlike contemporary servers, they will not support system-wide CC. Current engines suffer from three major problems on such hardware.

First, database designs that rely on CC-shared memory for scaling transactional workloads will be incompatible with non-CC hardware. Database engines rely on CC for cross-core data sharing, and more importantly, thread synchronization based on spinlocks, shared-memory atomics, or HTM. In the absence of system-wide CC, the only option today is to scale OLTP workloads using the shared-nothing (SN) design. The SN design, however, is agnostic to the fact that memory is globally shared across all processors, and thus suffers from distributed transaction overheads when running poorly partitionable workloads [42].

Second, while specialized OLAP engines exploit the massive parallelism of GPGPUs [1, 17, 18], all current general-purpose engines ignore them because they traditionally did not share an address space with CPUs, and thus made it difficult to share data across transactional and analytical workloads. Using these contemporary database engines on emerging hardware with GPGPUs that no longer suffer from any such data-sharing limitations would leave abundant heterogeneous parallelism untapped.

Third, even state-of-the-art database engines exhibit a *house pattern* [43]: under mixed workloads, increasing OLAP throughput by scheduling more concurrent analytical queries results in a collapse in transactional throughput due to contention for processing resources. Avoiding the house pattern requires throttling or preempting analytical queries in order to prioritize transaction execution. Such throttling is completely unwarranted in emerging server platforms, especially since the heterogeneous processing resources are underutilized. Given these problems, we believe that it is time to revisit database design for emerging hardware.

3. THE CASE FOR H^{2}TAP

Heterogeneous-HTAP (H^{2}TAP) is a new architecture for building database engines that uses two design principles to exploit all aspects of emerging hardware: 1) make heterogeneity a first class design citizen, 2) decouple shared memory and CC dependencies.

Heterogeneity as an opportunity. The H^{2}TAP architecture exploits heterogeneity based on the observation that the latency-critical nature of OLTP workloads and the bandwidth-intensive nature of OLAP workloads are aligned with the task-parallel nature of CPUs and the data-parallel nature of GPUs respectively. Thus, the H^{2}TAP architecture uses both hardware and workload heterogeneity in a synergistic fashion by introducing the *archipelago* abstraction.

Archipelagos are resource containers defined by a set of processor cores and a target workload. H^{2}TAP uses archipelagos by partitioning cores into a task-parallel archipelago consisting only of CPU cores, and a data-parallel archipelago that can contain both GPUs and CPU cores. Transactions are executed in the task-parallel archipelago while analytical queries are handled by the data-parallel archipelago as shown in Figure 2.

Decoupling shared memory and cache coherence. Despite executing queries in different archipelagos, the H^{2}TAP architecture mandates storing a single copy of data in shared memory that is *globally accessible* across archipelagos. Still, while the H^{2}TAP architecture expects hardware to support shared memory, it does not rely on system-wide CC. Instead, H^{2}TAP engines have to explicitly manage CC in software. The clear separation of workloads across archipelagos simplifies this task to a certain extent – as analytical queries do not update data, H^{2}TAP engines do not have to maintain coherence across archipelagos. However, H^{2}TAP en-

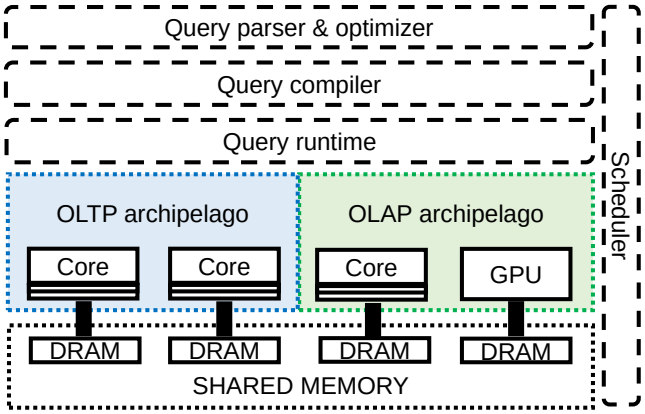

Figure 2: H^{2}TAP deployed over emerging server hardware.

gines should guarantee that transactions running within the task-parallel archipelago obey the ACID properties and analytical queries running in the data-parallel archipelago work on transactionally consistent data despite the lack of CC.

H^{2}TAP blueprint. Figure 2 shows software components that an H^{2}TAP engine would need to implement in order to realize the H^{2}TAP architecture in practice. The *parser and optimizer* form a front end that translates a SQL query into a physical query plan. The *scheduler* is responsible for implementing the archipelago abstraction by managing core–archipelago membership. Using this information, the scheduler can provide run-time elasticity by enabling on-the-fly "migration" of CPU cores between archipelagos. Further, the scheduler also maintains processor and memory utilization statistics within each archipelago. Based on these statistics, it works with the optimizer to determine the target archipelago and cores where each query will be executed. While the H^{2}TAP architecture requires transactional queries to be scheduled on CPUs in the task-parallel archipelago, it enforces no such restrictions on the scheduling of analytical queries in the data-parallel archipelago. Thus, the scheduler can combine dynamic run-time information, such as data locality, with static optimizer cost models to decide if a given analytical query should be executed on CPU or GPU cores in the data-parallel archipelago.

Once the scheduler determines the target execution environment, a *query compiler* produces the query implementation from the physical query plan. Instead of using volcano-style interpretation for executing the query plan, the query compiler generates machine code corresponding to the target processor(s) for the query. Query compilation reduces the interpretation overheads of query execution [26, 27, 36, 44, 46], and masks the effects of (data) heterogeneity [22, 23, 24]; H^{2}TAP extends the concept of heterogeneity to hardware in order to mask the difference in Instruction Set Architectures (x86 or PTX [39]).

Finally, the generated code is passed to the *Query runtime* together with information from the scheduler about the target processor(s) where the query should be executed. The runtime is responsible for both providing a mechanism for sharing data across archipelagos, and shepherding query execution within each archipelago.

H^{2}TAP benefits. The H^{2}TAP architecture provides several benefits. First, archipelagos enable affinitizing workloads to ideal processor types; transactions benefit from task parallelism provided by CPUs and analytical queries benefit from data parallelism provided by GPUs. Second, by enabling CPU cores to change membership between task and data-parallel archipelagos on the fly, the H^{2}TAP architecture improves deployment elasticity because it enables dynamic load balancing. For instance, an H^{2}TAP engine could configure its scheduler to move unused CPU cores from task- to data-parallel archipelago, and use them for running analytical queries under light OLTP workloads. Third, by separating OLTP and OLAP execution, archipelagos eliminate interference and processor resource contention across workloads, and hence the house pattern, by design. Fourth, by decoupling shared memory and CC, the H^{2}TAP architecture enables new database engine designs that can take a middle ground between shared-everything designs, which rely on CC and shared memory, and shared-nothing designs, which are oblivious to both aspects.

H^{2}TAP challenges. Despite the benefits of H^{2}TAP, realizing it in practice also requires answering three questions. First, H^{2}TAP engines have to store data in a layout that is suitable for efficiently running both transactional and analytical workloads. However, research on CPU-based database engines has shown that different workloads benefit from different storage layouts [2, 13]. OLTP workloads benefit from the N-ary Storage Model (NSM) because the whole-record read-write operations performed by transactions can be implemented efficiently using NSM's row-wise layout. OLAP workloads, in contrast, touch only a few attributes, and thus benefit more from the columnar layout of the Decomposition Storage Model (DSM), which minimizes data transfers and utilizes CPU caches better. As H^{2}TAP engines need to support both workloads, the first question to be answered is whether "middle-ground" [2] hybrid layouts [3, 4, 15] work in the H^{2}TAP context as well.

Second, irrespective of the layout used, an H^{2}TAP engine must provide an efficient mechanism to provide analytical queries running in the data-parallel archipelago with access to transactionally-consistent data which is being updated by transactions running on CPUs. Contemporary HTAP engines typically use snapshotting to solve this problem [25]. If we used only CPUs in the data-parallel archipelago, we would be able to use fork-based snapshotting [25] for executing OLAP queries over an immutable database snapshot. Unfortunately, such an approach is not applicable with GPGPUs because CUDA memory allocations cannot be shared across process boundaries due to CUDA runtime limitations. Thus, the second question to be answered is whether alternate software snapshotting techniques [48] can be used to enable cross-archipelago data sharing.

Third, while the H^{2}TAP architecture expects hardware to support globally accessible shared memory, it does not rely on system-wide CC. Thus, an H^{2}TAP engine must be able to scale transactional and analytical workloads despite the lack of CC. Given that analytical queries running in the data-parallel archipelago never update the database due to their read-only nature, H^{2}TAP obviates the need for cross-archipelago CC. However, H^{2}TAP engines must still overcome the lack of coherence within the task-parallel archipelago where concurrent transactions update shared data and metadata. Therefore, the third question to be answered is whether OLTP workloads can be scaled up within task-parallel archipelagos without relying on CC.

4. CALDERA: AN H²TAP QUERY ENGINE

Caldera is a prototype query engine we develop to examine the opportunities offered by the H²TAP architecture and address the challenges it raises. To this end, the Caldera prototype implements only the query runtime and leaves the other components described in Section 3 to future work.

Applying H²TAP. Caldera adheres to the H²TAP architecture by grouping processors into a CPU-only task-parallel archipelago, and a GPU-only data-parallel archipelago. Transactions are executed in the task-parallel archipelago while analytical queries are handled by the data-parallel archipelago.

Caldera stores data in shared memory that is allocated using Unified Virtual Addressing. By using UVA, Caldera exposes a global address space across archipelagos. We use UVA because our current hardware setup uses Maxwell GPUs, which impose strict limits on the maximum Unified Memory allocation size. In the future, we plan to use Unified Memory with Pascal GPUs that have no such limitations.

Data layout. Prior research has focused on building hybrid layouts that can support both transactional and analytical workloads in the traditional HTAP context [2, 3, 4, 15]. For instance, PAX [2] is an alternative storage layout that strikes a balance between the NSM and DSM extremes. Like NSM, PAX organizes data records in pages. Like DSM, PAX groups values of the same attribute together. A page therefore contains *minipages*, each of which only contains values of a single attribute. Due to its organization of data into minipages and pages, PAX enables cache-friendly query execution similar to DSM while providing update cost similar to NSM.

Hybrid layouts like PAX play an even more important role in the new H²TAP scenario because they provide two tangible performance benefits. First, as the GPU memory capacity is limited, data transfer plays a crucial role in determining the overall query execution time due to the limited bandwidth of the PCIe bus. Thus, hybrid layouts will outperform NSM even in the H²TAP scenario due to their ability to reduce the amount of data transferred. Second, GPUs coalesce global memory loads and stores issued by threads into as few memory transactions as possible to both improve performance and reduce memory bandwidth requirements. However, in order for coalescing to work properly, threads should access memory locations sequentially. Thus, a data layout like PAX is a better fit for GPUs than NSM because it enables such coalesced accesses.

Our current prototype supports NSM, DSM, and PAX layouts. Caldera stores data in shared memory as a collection of horizontal *partitions*. Within a partition, records of a table can be stored in any of the three layout types. Figure 3 shows the hierarchical *partition–table–column–page* data organization used by DSM.

OLAP in the data-parallel archipelago. The Caldera prototype uses the kernel-based execution model for executing OLAP queries on the GPU similar to other GPU-based OLAP engines [14, 17, 51]. Each database operator is implemented as a collection of data-parallel primitives, where each primitive is an individual CUDA kernel. OLAP queries are executed by a dedicated CPU thread that executes each database operator by executing the corresponding CUDA kernels one at a time while using UVA to store all input, intermediate, and output data. It is well-known that such kernel-based execution results in sub-optimal use of the GPU due to unwarranted data transfers [41, 51]. In the future, we plan to use a query compilation infrastructure to fuse multiple relational operators in a single kernel.

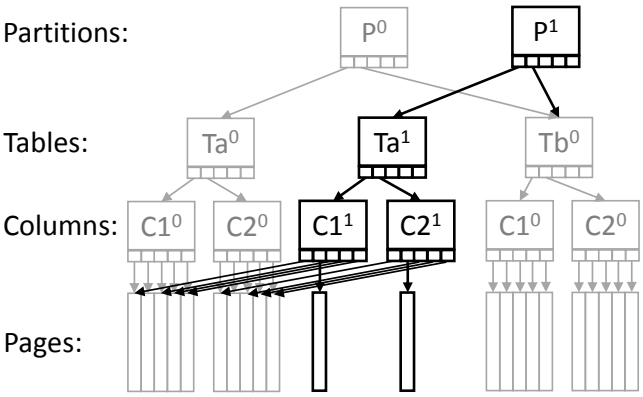

Figure 3: The hierarchical data organization of Caldera for a columnar data layout, and the in-memory state after a transaction has updated table T^a. Superscripts represent epochs.

Caldera always executes OLAP queries on a database snapshot. Thus, users can trade off data freshness for performance by having several OLAP queries share a snapshot, or maximize freshness by taking a snapshot before running each OLAP query. Snapshotting is implemented using a software-based shadow-copying mechanism that works on the hierarchical data organization. We describe it using the layout shown in Figure 3. Each table, column, and page is associated with an epoch number. The query runtime creates a snapshot by performing a shallow copy of the top-level container and incrementing its epoch number. Thus, snapshotting is an instantaneous operation after which the newly created snapshot and the "live" database share all data. After snapshotting, the runtime identifies the columns that are necessary for executing the OLAP query and invokes the GPU kernel, passing in pointers to relevant pages. No data is copied explicitly; the GPU kernel accesses data directly from the UVA-allocated host memory.

Copy-on-write during updates and garbage collection are integrated with transaction management. When a transaction commits, the runtime identifies records to be updated. It uses this information to identify the backing pages for those records, and shadow-copies them by allocating new pages and copying over data from the snapshot. Then, it applies the updates, and marks the pages as "live" by incrementing their epoch number. It repeats this copy-on-write process all the way back to the root, allocating new data structures as required and updating pointers. Similarly, when a snapshot is deleted, the query runtime uses epoch numbers to identify both data and metadata that have been superseded by the copy-on-write process and deletes the old versions to reclaim space.

OLTP in the task-parallel archipelago. Caldera scales OLTP workloads within the task-parallel archipelago by using message passing-based parallelism (that relies on fast core-to-core messaging) rather than shared-memory parallelism (that relies on cache coherence). Caldera schedules one thread per core in the task-parallel archipelago and assigns one data partition to each thread, which then mediates access to partition-local records. Each thread uses two-phase locking (2PL) for concurrency control and a primary-key in-

dex to assist in record lookup. Unlike data, which is shared across archipelagos, the lock tables and indices are private to each thread running in the task-parallel archipelago and do not belong to the snapshot hierarchy depicted in Figure 3. Thus, they refer to logical records whose physical location changes during copy-on-write operations.

An incoming transaction can be scheduled to run on any thread; the chosen thread will act as its host (the *client* thread). The client executes all operations of a transaction using direct function calls to lookup/update records. If the client contains the target record in its partition, it uses its local lock table to decide if the access request can be granted. If so, it grants the lock, performs shadow copying if necessary, and executes the operation.

If the record belongs to a different partition, the client sends a message to the data owner thread (the *server* thread) requesting access to the record, and blocks the transaction. When the server thread receives the message, it tries to acquire the lock. If successful, it grants the lock, performs shadow copying if necessary, and sends a reply message giving the client access to the record. If the acquisition fails, the server thread delays replying back until the lock becomes available. Rather than shipping the whole record in the message, Caldera exploits hardware-supported shared memory to reduce data movement by sending only the record pointer. Upon receiving the reply, the client thread unblocks the transaction and uses the record pointer to directly lookup/update the record.

At transaction commit or abort time, the client thread sends an explicit "release" message for each remote record. Upon receiving a release message, the server thread releases the associated lock and picks a new lock owner. If the new owning transaction is local to the server, it is unblocked and scheduled for execution. Otherwise, the server unblocks it by replying back to the client.

Relying on explicit message passing has several benefits. First, two processors can never simultaneously access a shared memory word because each processor has exclusive access over its partition. Before a thread can access a record, it has to explicitly synchronize with the owning thread by sending it a message. This explicit communication eliminates the need for implicit thread synchronization with latches, atomics, or other CC-dependent hardware features. Thus, all aspects of transaction execution are single-threaded and completely synchronization-free.

Explicit communication also makes maintaining coherence across core-private caches straightforward. In Caldera, two transactions can never concurrently update the same record due to 2PL. Thus, cache management is necessary only to ensure that two transactions running serially on two different cores see the latest version of the record despite the existence of caches. This can be done by adding explicit cache write back and invalidation at two points. When a client thread requests a record from a server thread, the server thread explicitly writes back the dirty data from its local cache before replying back. Similarly, before the client thread sends a release message at commit time, it writes back the data it updated. Doing so guarantees that a thread will always read the latest version of data from the memory instead of an outdated cache. Together, explicit communication and cache management ensure that Caldera can work on non-CC hardware.

Finally, by abstracting away the details of communication using a message passing library, Caldera is portable, as the message-passing layer can be replaced to make it work on CC multicores, non-CC multicores, and even potentially scale-out clusters without any change to the core database logic.

5. EVALUATION

In this section, we present an evaluation of Caldera to show that the H^2TAP architecture can be implemented in practice and can offer performance competitive to that of state-of-the-art OLTP and OLAP engines. As described in Section 4, Caldera uses three features to tackle the challenges posed by the H^2TAP architecture, namely, software snapshotting for cross-archipelago HTAP, message-passing for transaction processing without CC, and PAX as the hybrid layout that enables data sharing across mixed workloads. Thus, in this section, we present the performance and scalability of these three aspects and compare Caldera with Silo [47], a main-memory OLTP engine, MonetDB [8], an open-source column store, and "DBMS-C", a commercial column store.

Experimental setup. All experiments are conducted on a server running RHEL 7.2, equipped with two 12-core Intel Xeon E5-2650L v3 CPUs, 256GB RAM, and a GeForce GTX 980 GPU with 4GB memory. Although the hardware we use supports system-wide CC, Caldera uses it only as the message passing substrate for inter-thread communication.

5.1 HTAP with software snapshotting

We present the OLTP throughput and OLAP response time achieved by Caldera under a mixed workload. For these experiments, we use the TPC-H (SF-300) dataset. We use TPC-H Q6, a selection over the lineitem table, as the OLAP query. In our OLTP workload, each transaction performs ten read-modify-update operations on records randomly chosen from the lineitem table. Thus, the OLTP workload is similar to an update-only YCSB workload [12] with a theta value (zipfian distribution) of zero. We run ten OLAP queries in succession on the GPU. The OLTP workload is executed by the CPU until all OLAP queries terminate. We use the snapshotting flexibility of Caldera to demonstrate the performance-freshness trade off posed by our software shadow copying implementation. Further, it is common in real-world deployments for transactions to access only a "hot" fraction of the dataset [30], whereas OLAP queries scan through all the data. We make the target key range used by the OLTP workload a parameter so that we test sensitivity to skewed OLTP working set sizes.

Figure 4 shows the OLAP query execution time under MonetDB, DBMS-C, and Caldera in the absence of transactions. Both MonetDB and DBMS-C parallelize the query across all 24 cores. MonetDB is 1.27× faster than DBMS-C because it benefits from the use of secondary indexes. Caldera exploits the massive parallelism of the GPU to provide 4.15× and 5.29× speedup over MonetDB and DBMS-C, even though the table is streamed from host memory.

Figure 5 shows the OLTP throughput achieved by Caldera as we vary the working set size from 1% to 100%. The four lines show the throughput as we increase data freshness by varying snapshot frequency from one across all ten OLAP queries to one per OLAP query. Clearly, transactional throughput deteriorates when we increase the working set size or the frequency of snapshots due to software

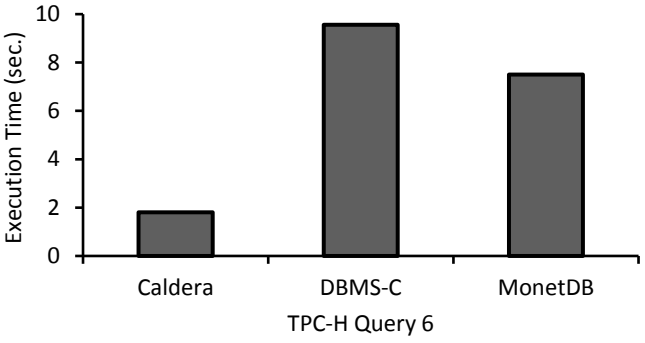

Figure 4: GPU-powered Caldera vs. CPU-powered columnar engines for Q6 of TPC-H. Time for Caldera includes data transfer costs.

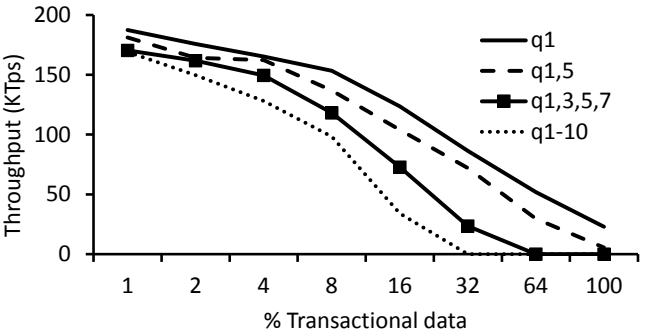

Figure 5: OLTP transaction throughput in the presence of OLAP queries as we vary the OLTP working set and the degree of data freshness.

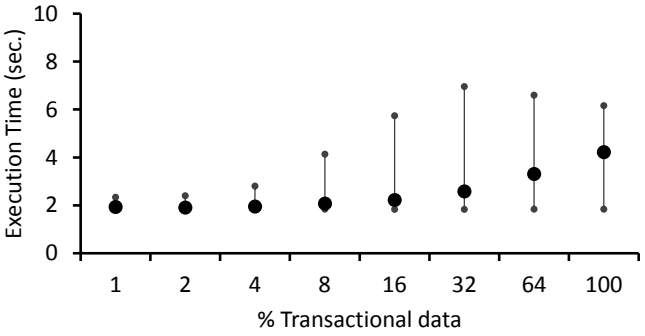

Figure 6: Execution time of OLAP queries in the presence of OLTP queries. All OLAP queries share a single snapshot, but OLTP-triggered copy-on-write stresses memory bandwidth.

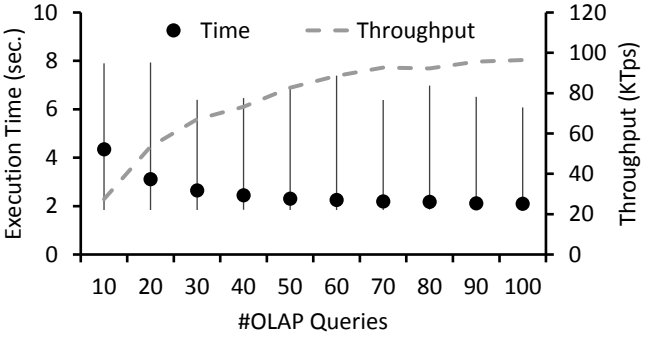

Figure 7: Execution time of OLAP queries and throughput of OLTP queries. We increase the number of queries that share a snapshot from 10 to 100. Increasing snapshot sharing improves performance.

overhead, as Caldera incurs the cost of performing a copy-on-write the first time data is modified after each snapshot.

Snapshotting also affects OLAP response time. Figure 6 shows the average, minimum, and maximum analytical query response times for Caldera when all ten queries share one snapshot as we vary the (OLTP) working set size from 1% to 100%. In the presence of snapshotting, both analytical queries running on the GPU and transactions running on CPU compete for memory bandwidth due to the memory-intensive copy-on-write process. This results in a 2× increase in average response time and a 3× increase in maximum response time. Note that this overhead is not exclusive to the shadow copy implementation of Caldera: Fork-based snapshotting implementations also suffer under update intensive workloads [48]. In addition to such snapshotting-related overheads, current HTAP engines also exhibit the house effect as transaction throughput collapses due to processor resource contention caused by interference between OLAP and OLTP workloads [43]. Under Caldera, in contrast, processor resource contention never occurs due to the strict separation of workloads provided by the archipelago abstraction. Contention for memory bandwidth is purely due to the software overhead of our copy-on-write mechanism and can be reduced using three techniques.

The first optimization is to trade off a degree of data freshness for improved performance by sharing a snapshot across several OLAP queries. Figure 7 shows the throughput and response time for Caldera when we fix the OLTP working set to 100% – the worst case in Figure 6 – and vary the number of queries that share a snapshot from 10 to 100. Initially, almost all transactions perform copy-on-write. Analytical queries that are executed concurrently with these transactions suffer due to shared memory bandwidth. This explains the high worst-case response time for analytical queries. As the copy-on-write process converges, both transactional throughput and analytical response time improve substantially. Comparing Figures 5 and 7, we observe that sharing a snapshot across 100 queries provides nearly a 5× improvement in OLTP throughput even if the working set covers 100% of the data set.

Second, as shown in Figure 5, limiting the OLTP working set to less than 16% of the total data size limits the worst-case deterioration in throughput to only 2× even if we use one snapshot per query. Thus, hybrid data layouts that perform hot–cold data classification [28] will enable Caldera to further reduce the impact on OLTP throughput.

Third, profiling revealed that both memory allocation and memory copying performed during the shadow-copy operation were sources of overhead. Thus, optimizing shadow copying by using alternate snapshotting implementations [45, 48] is another approach for improving OLTP throughput.

5.2 OLTP with message passing

Next, we compare the performance and scalability of Caldera against Silo for OLTP workloads. To avoid confounding performance effects caused by memory allocation, and to keep

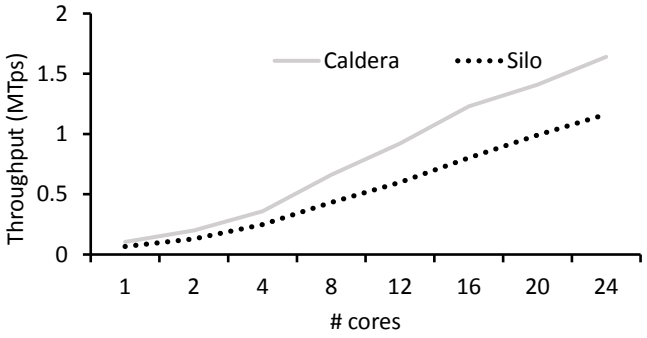

Figure 8: TPC-C scalability as the number of cores increase.

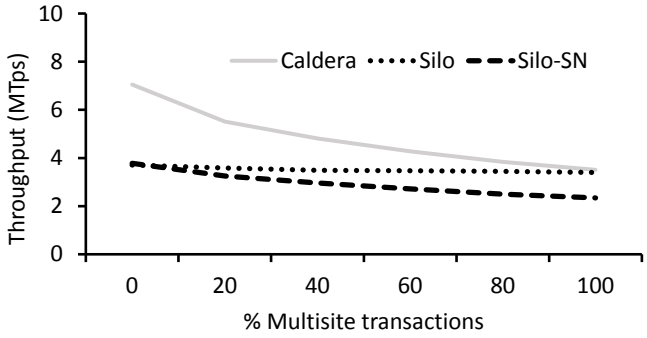

Figure 9: Throughput as the percentage of multi-site transactions increases.

the comparison fair, we use the NSM data layout, and also use malloc as the memory allocator for Caldera.

The first experiment investigates the scalability of both systems for the NewOrder transaction of the TPC-C benchmark. For both systems, we assign a warehouse to a thread and increase the number of threads (and hence the number of warehouses). Figure 8 reports throughput at various thread counts; both systems scale well. Caldera outperforms Silo due to 1) better data locality provided by partitioning, 2) better code locality due to the lack of thread synchronization, and 3) limited message passing overhead because only 10% of NewOrder transactions require remote accesses.

The next experiment investigates throughput sensitivity in the presence of multi-site transactions. We use a read-only microbenchmark in which each transaction reads ten records from a table of 24M records partitioned across 24 cores. Single-site transactions read all ten records from the local partition. Multi-site transactions read two records from a random remote partition and the remaining eight from the local partition. We compare Caldera with two deployments of Silo, namely, *Silo* and *shared-nothing Silo (SN-Silo)*. The default configuration uses a single instance of Silo over all cores. SN-Silo represents how one could use current OLTP engines on emerging non-CC multi-cores; the SN-Silo setup uses one instance of Silo per core and a distributed transaction layer to coordinate multi-site transactions using the two-phase commit (2PC) protocol.

Figure 9 shows the throughput achieved by all three systems as the fraction of multi-site transactions increases. Both Caldera and SN-Silo are affected by multi-site transactions, but for very different reasons; Caldera suffers due to the use of CC as the message passing mechanism while SN-Silo suffers due to the overheads of 2PC. Thus, for emerging hardware, replacing CC with hardware message passing will benefit Caldera, but not SN-Silo. Despite the message passing overhead, Caldera can match Silo's throughput, showing that the message passing-based design used by Caldera provides performance competitive with that of state-of-the-art OLTP engines.

5.3 Data sharing with PAX

The next experiment examines the suitability of PAX for OLAP operations executed on GPUs. For this experiment, we use a main-memory-resident 16 GB table of 270M records. Each record is comprised of 16 integer attributes. We use three different storage layouts for the table: DSM, PAX, and NSM. We set the size of the PAX page to 4KB. Each PAX page contains 16 minipages, and each minipage contains 64 values. We then launch five instances of the following query template:

```
SELECT SUM(col1 + ... + colN) FROM dataset
```

Each instance accesses 1, 2, 4, 8, or 16 attributes, respectively. Figure 10 depicts the response time for each instance.

NSM has the slowest response times because it leads to sub-optimal data access patterns. Specifically, GPUs manage threads in groups. The ideal access pattern in the context of GPUs is one for which all threads in a group perform *coalesced accesses*, i.e., they access a contiguous chunk of memory. When executing a query over NSM data, the values for col1, col2, etc., are not stored contiguously, thus resulting in multiple expensive memory transactions.

PAX and DSM have almost identical response times, with the former being slightly slower. Both the PAX and DSM layouts lead to coalesced memory accesses. In addition, both layouts minimize unnecessary data transfers through the PCIe bus. Specifically, the maximum transfer unit (MTU) through the PCIe bus typically does not exceed 512 bytes. We carefully configure the PAX layout so that the size of each minipage is close to the MTU, and thus maximize the utilization of the PCIe bandwidth.

While our previous experiment showed that NSM suffers due to its inability to perform coalesced accesses, PAX and DSM are able to effectively saturate the PCIe bandwidth. The GPU memory, however, provides an order of magnitude higher bandwidth compared to PCIe. As on-board GPU memory continues to increase in capacity, an important question is whether PAX lags behind DSM if all data were local to the GPU.

To answer this question, we repeated the previous experiment while storing all data in GPU memory. Due to limited memory capacity, we reduced the dataset size from 16GB to 1GB. Figure 11 shows the response time for the three layouts when the query touches only two attributes out of 16. We only report the kernel execution time and not data transfer time for two GPUs belonging to different generations, namely a Fermi GPU (Tesla M2090) and the Maxwell GPU (GTX 980). There are three important observations.

First, comparing the two GPUs, we see that the Maxwell GPU provides a 2.5×, 3.1×, and 3.5× improvement for DSM, PAX, and NSM layouts respectively. These results are encouraging because despite being just a consumer-grade graphics card, the Maxwell GPU (GTX 980) outperforms a previous-generation compute accelerator (Tesla M2090).

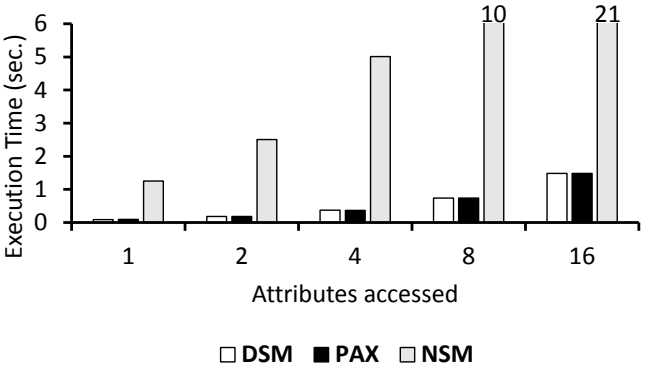

Figure 10: Comparing the efficiency of different data layouts for GPU-based computations.

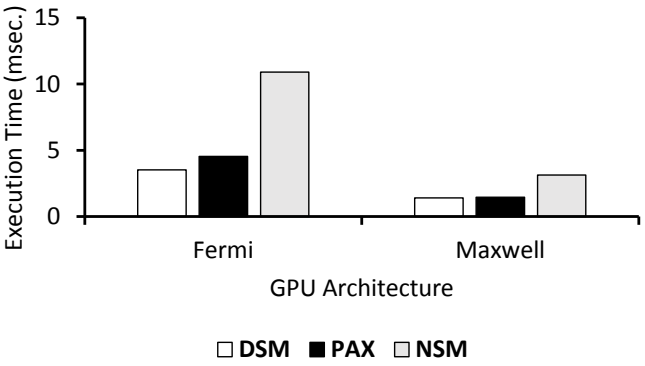

Figure 11: Comparing different data layouts when all data is GPU resident.

Second, comparing relative performance of each layout within a GPU generation, we see that NSM is 3× slower than DSM on Tesla and only 2× slower on Maxwell. Similar, PAX is 1.3× slower on Tesla but matches DSM performance on Maxwell. This result is in sharp contrast with the UVA results we reported in Figure 10, where NSM was 13.74× slower than DSM. This shows that modern GPUs have vastly reduced the performance impact of non-coalesced memory accesses when data fits in GPU memory. Thus, using a PAX-like storage layout that acts as the middle ground between OLTP-oriented NSM and OLAP-oriented DSM is a viable option for H^{2}TAP. A possible next step would be crafting a new data layout dynamically depending on the workload requirements [3, 15], e.g., storing frequently accessed attributes together in a group of columns.

Summary. Overall, the results indicate that it is possible to realize H^{2}TAP in practice and show many of the opportunities and challenges involved in designing H^{2}TAP engines.

6. DISCUSSION

The Caldera prototype is a proof-of-concept implementation that demonstrates the feasibility of the H^{2}TAP architecture. In this section, we discuss several aspects that require further research in order to realize a fully functional H^{2}TAP engine.

Query optimization and scheduling. Caldera separates read-only OLAP queries from read-write transactions and runs each category either on data-parallel (OLAP) or on task-parallel (OLTP) archipelagos. Transactions require synchronization at multiple levels (concurrency control protocols at the logical level, latching at the physical level, atomics at the hardware level). Therefore, Caldera restricts the membership of task-parallel archipelagos to CPUs. However, as OLAP queries can be parallelized well on both CPUs and GPUs, Caldera makes the data-parallel archipelago heterogeneous. Given such heterogeneity, a given OLAP query could potentially be executed on just CPUs, just GPUs, or a mix of both. Thus, an important topic that requires further research is query optimization and scheduling in the heterogeneous OLAP archipelago.

GDB [17] was one of the first prototypes to investigate extensions to analytical cost models in the CPU–GPU query coprocessing scenario for deciding optimal operator placement. CoGaDB [9] is a more recent effort that uses cost models based on observed query execution time that are learned on-the-fly and continuously refined for both picking an optimal query plan and the placement of operators across CPUs and multiple GPUs. We plan to extend Caldera with such heterogeneity-aware query optimizers in the future.

Utilizing other data-parallel hardware. Over the past few years, processor vendors have introduced several new heterogeneous hardware accelerators that compete with GPUs for accelerating data-parallel workloads. For instance, Intel Many Integrated Core processor (also known as Xeon Phi) packs several hyperthreaded, low-frequency in-order cores together with high-bandwidth memory in a single package to provide an order-of-magnitude more hardware contexts than server-grade Xeon processors. Intel HARP platform integrates Field Programmable Gate Arrays (FPGAs) and Xeon processors in a single multi-socket system. Recent research has shown that analytical workloads benefit from such heterogeneous hardware [21, 35].

While this paper focuses on GPUs, the H^{2}TAP architecture is independent of the type of data-parallel hardware used for accelerating OLAP queries. In fact, given that the H^{2}TAP architecture decouples cache coherence from shared memory, it can take advantage of the simpler data-parallel hardware that does not necessarily support system-wide cache coherence. Further, the use of query compilation makes the overall architecture hardware-agnostic; any processor can be integrated into the Caldera framework as long as the query compiler generates specialized code for the target processor.

7. CONCLUSION

Modern database engines are designed to work on multi-socket multi-cores that provide abundant homogeneous parallelism, system-wide CC, and global shared memory. As a result, they are mismatched with emerging server hardware which will make both parallelism and CC support heterogeneous. We introduce H^{2}TAP, a new architecture for building database engines on such hardware. Using Caldera, a prototype H^{2}TAP engine, we show that the H^{2}TAP architecture can be realized in practice and can match the performance of state-of-the-art specialized OLTP and OLAP engines.

Acknowledgments. We would like to thank the anonymous reviewers and the DIAS laboratory members for their constructive feedback. This work is partially funded by the EU FP7 Programme (ERC-2013-CoG) under grant agreement number 617508 (ViDa), the EU FP7 Programme (FP7 Collaborative project) under grant agreement number 317858 (BigFoot), and the Swiss National Science Foundation (Grant No. 200021-146407/1).

8. REFERENCES

[1] MapD. https://www.mapd.com/.

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

[3] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD*, 2014.

[4] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*, 2016.

[5] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *EuroSys*, 2015.

[6] A. Baumann, P. Barham, P. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[7] A. Baumann, C. Hawblitzel, K. Kourtis, T. Harris, and T. Roscoe. Cosh: Clear OS Data Sharing In An Incoherent World. In *TRIOS*, 2014.

[8] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of ACM*, 51(12):77–85, 2008.

[9] S. Breß, H. Funke, and J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *SIGMOD*, pages 1891–1906, 2016.

[10] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 15:1–35, 2014.

[11] J. Cai and A. Shrivastava. Software Coherence Management on Non-coherent Cache Multi-cores. In *VLSID*, 2016.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[13] G. P. Copeland and S. N. Khoshafian. A Decomposition Storage Model. *SIGMOD Record*, 14(4):268–279, 1985.

[14] G. F. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili. Relational algorithms for multi-bulk-synchronous processors. In *PPoPP*, 2013.

[15] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2), 2010.

[16] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26:10–24, 2006.

[17] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *TODS*, 34(4):21:1–21:39, 2009.

[18] 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.

[19] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108–109, 2010.

[20] C. G. III, F. Sironi, M. F. Kaashoek, and N. Zeldovich. Hare: a file system for non-cache-coherent multicores. In *EuroSys*, 2015.

[21] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *PVLDB*, 8(6):642–653, 2015.

[22] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12):972–983, 2016.

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

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

[25] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[26] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014.

[27] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[28] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*, 2016.

[29] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: A Coherent Shared Memory Interface for Cell Be. In *PACT*, 2008.

[30] J. J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, 2013.

[31] F. X. Lin, Z. Wang, and L. Zhong. K2: A Mobile Operating System for Heterogeneous Coherence Domains. *TOCS*, 33(2):4, 2015.

[32] M. Martin, M. Hill, and D. Sorin. Why on-chip cache coherence is here to stay. *CACM*, 55(7):78–89, 2012.

[33] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core Network-on-a-chip Terascale Processor. In *ICS*, 2008.

[34] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *ICPP*, 2015.

[35] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.

[36] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[37] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[38] NVIDIA. NVLink High-Speed Interconnect. http://www.nvidia.com/object/nvlink.html.

[39] NVIDIA. Parallel Thread Execution ISA Version 4.3. http://docs.nvidia.com/cuda/parallel-thread-execution.

[40] NVIDIA. Summit and Sierra Supercomputers: An Inside Look at the U.S. Department of Energy's New Pre-Exascale Systems. Technical report, 11 2014.

[41] J. Paul, J. He, and B. He. GPL: A GPU-based Pipelined Query Processing Engine. In *SIGMOD*, pages 1935–1950, 2016.

[42] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. *PVLDB*, 5(11):1447–1458, 2012.

[43] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K. Sattler. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In *TPCTC*, 2014.

[44] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, 2006.

[45] F. M. Schuhknecht, J. Dittrich, and A. Sharma. RUMA Has It: Rewired User-space Memory Access is Possible! *PVLDB*, 9(10):768–779, 2016.

[46] A. Shaikhha et al. How to Architect a Query Compiler. In *SIGMOD*, 2016.

[47] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[48] D. Šidlauskas, C. S. Jensen, and S. Šaltenis. A Comparison of the Use of Virtual Versus Physical Snapshots for Supporting Update-intensive Workloads. In *DAMON*, 2012.

[49] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *OS Review*, 43(2):76–85, 2009.

[50] Y. Xu, Y. Du, Y. Zhang, and J. Yang. A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs. In *ICS*, 2011.

[51] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *PVLDB*, 6(10):817–828, 2013.