Stream Query Processing on Emerging Memory Architectures

Chelsea Mafrica, John Johnson, Santiago Bock, Thao N. Pham,
Bruce R. Childers, Panos K. Chrysanthis, Alexandros Labrinidis
University of Pittsburgh, Department of Computer Science
Email: {cem37, jdj20, sab104, thao, childers, panos, labrinid}@cs.pitt.edu

***Abstract*—Stream query processing is becoming increasingly important as more time-oriented data is produced and analyzed online. Stream processing is typically memory-resident for the fastest processing of ephemeral data. With workload consolidation, processing separate data streams on the same processor may lead to harmful contention between query workloads. This contention may become particularly problematic as new main memory technologies are adopted, such as phase-change memory, that have asymmetric read and write latency. This work presents a preliminary study of performance implications of consolidation and emerging memory on stream query processing. We show that contention in the memory subsystem worsens with a phase-change main memory, suggesting that new stream optimization and hardware approaches will be required to achieve quality of service and quality of data guarantees in future computer servers.**

I. Introduction

Three important trends, spanning workload consolidation, stream processing, and computer architecture, are materializing that have potentially important consequences on one another. There is increasing consolidation of workloads to run on the same servers for increased utilization (e.g., in the cloud). Stream processing is also becoming prevalent as more and more ephemeral data is produced that must be analyzed online in real time. Finally, main memory organizations are moving toward designs incorporating alternative bit cell technologies due to DRAM scaling obstacles.

The challenge is the way these trends interact. For consolidation, multiple, independent instances of stream processing may be placed on the same system. Data management for the stream is often memory-resident for fast processing to meet quality of service (QoS) and quality of data (QoD) guarantees. The data is also often short-lived, entering and exiting quickly. By consolidating memory-resident processing workloads, adverse contention can arise in the memory subsystem. For instance, two workloads may compete for the same memory resources, such as the row buffers that cache memory pages internally to the DRAM, the queues that hold pending memory operations, and the banks that support parallel access. When contention is severe, QoS and/or QoD may be harmed, possibly conflicting with scheduling and optimization.

At the same time that memory-resident stream processing has taken hold and consolidation has become standard, the memory subsystem itself is undergoing radical redesign. DRAM is at the point where scaling to smaller sizes has become problematic. According to the International Technology Roadmap for Semiconductors, scaling DRAM much further than current node sizes faces major obstacles for reliability and power leakage. This has set off a race to find new memory that can continue to rapidly increase capacity (i.e., memory chip density) to keep pace with larger and larger data sets, including those in stream processing.

978-1-4673-6688-5/15/$31.00 ©2015 IEEE

While there are several potential memory techniques, such as tiered memory and 3D stacked memory, that might help alleviate the challenge with DRAM, novel memory technologies will likely play a critical role. Phase-change memory (PCM), spin-torque transfer memory (STT) and domain-wall memory (DWM), among other more exotic technologies, are all candidates for replacing the traditional charge-based (capacitor) design used by DRAM. Although structure and operation of these new technologies differ, the basic approach is similar—avoid using charge to store a bit. Typically, resistance is used.

PCM is a leading candidate: Prototypes have demonstrated exceptional scalability and read performance on par with DRAM. However, PCM uses a heating and cooling process to write a bit into a chalcogenide glass, leading to long write latency, high write energy and wear out. Although many architectural solutions have been proposed to manage writes in PCM (e.g., wear leveling), it is likely that the asymmetric latency of reads and writes will be exposed to software. For memory-resident stream query processing under consolidation, PCM's long write latency may cause especially adverse contention. To mitigate this competition, the query processing and system layers may need to work directly and cooperatively with the hardware to achieve QoS and QoD goals.

Given these three trends, there is a need to understand to what degree they impact one another, and whether action needs to be taken to mitigate the impact. For instance, if memory contention worsens due to PCM's long latency writes, then the stream query scheduler may need to be memory-aware to best schedule and optimize queries. Alternatively, the memory architecture may need to be aware of query processing to partition memory resources, under the control of software, for predictable behavior and priority enforcement.

This paper describes a first preliminary look at this problem. The goal is simple: Determine whether new memory technologies, i.e., PCM, worsen contention when multiple, separate stream query tasks are co-located. The answer will guide the data management, systems and computer architecture communities on whether this problem is worth further investigation, and the development of new solutions.

To answer the question, we use a state-of-the-art stream query processing system, AQSIOS [4], and a detailed, accurate memory simulator, HMMSim [2]. We examine a hybrid memory architecture that has both DRAM and PCM. In one case, we allocate the continuous query (CQ) data entirely to DRAM, and in another case, to PCM. By examining these two choices, we cover a spectrum of architectures; e.g., a hybrid main memory that uses some DRAM and some PCM for the CQ data will likely have behavior that falls between the DRAM-only and the PCM-only cases. The cases are useful for finding evidence that memory contention will worsen, and ultimately must be collectively managed at the system, stream query processing, and hardware layers.

II. In-Memory Query Processing

A. Stream Processing

Today the ubiquity of sensing devices as well as mobile and web applications continuously generate a huge amount of data which takes the form of streams. These data streams are typically high-volume, often high-velocity (speed) and high-variability (bursty). In order to meet the near-real-time requirements of the monitoring applications and of the emerging "Big Data" applications [7], data streams need to be continuously processed and analyzed. Data stream management systems (DSMSs) become the popular solutions to handle data streams by efficiently supporting continuous queries (CQs). CQs are stored queries that execute continuously, looking for interesting events over data streams as data arrives, *on the fly*.

Most DSMS architectures, including our AQSIOS DSMS prototype, provide a CQ processing engine, together with a query optimizer, a scheduler, and a load manager/shedder. Each submitted CQ is compiled and optimized into a query plan consisting of multiple relational operators (i.e., select, project, join, or aggregates), one or more source operators, and an output operator. Each operator has one or more input queues depending on its type. Tuples produced by an operator will be placed in the input queues of the next operators downstream. Since CQs exist in the DSMS for a long time, their plans are optimized together, forming a query network, in which a query can share with others some of its operators. In such a case, the intermediate tuples produced by the shared operator will be placed in a shared input queue for the two operators downstream. The output of each CQ is continuously stored or streamed to applications.

As opposed to traditional database management systems, join and aggregate operators are defined over a window specified in terms of two intervals: *range (r)* and *slide (s)*. For example, an aggregate CQ may compute the average stock price over the last hour (i.e., $r = 1$ hr) and update it every 30 minutes (i.e., $s = 30$ min). The range and slide intervals could be defined either based on the number of tuples or time-based. AQSIOS considers the more general time-based definition for both the range and slide.

During execution, the scheduler is responsible for assigning each operator a time slot to run. In order to reduce context switching overhead, the schedulers allow *batch processing* by letting each operator process up to a predefined number of tuples in its input queue during each invocation. Our AQSIOS prototype besides the standard *round-robin* (RR) fair scheduling policy, implements the priority-based scheduler *Highest Rate (HR)* [13], which optimizes average response time of CQs and the two-level class, class-based scheduler *CQC* [10]. In the event that a DMSM becomes overloaded, it sheds the excess load, typically by dropping tuples at the source operators [11].

B. Hybrid Memory Architecture

A typical architecture for a hybrid memory is shown in Figure 1. This architecture has one or more CPUs (a.k.a. cores) with private instruction and data L1 caches. Requests from CPUs are queued in the L1 queues. An L2 cache is shared by all CPUs. A single L2 queue handles requests from all CPUs. We assume there is enough bandwidth to serve all request currently in the L1 and L2 queues with the same delay. However, if more requests are issued than a queue can hold, the previous level is stalled and does not issue requests until a queue slot becomes free.

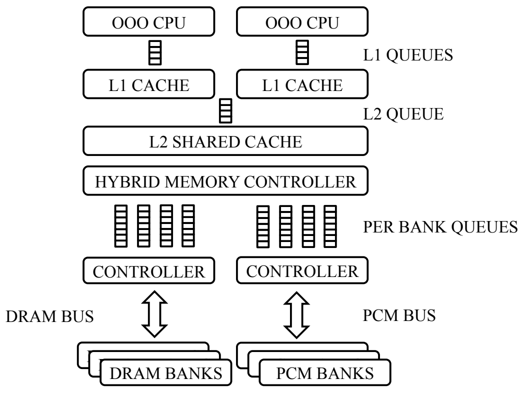

Fig. 1. Hybrid main memory architecture

A *hybrid-memory controller* forwards L2 miss requests to the memory (DRAM or PCM). Memory requests from the hybrid memory controller go to the DRAM or PCM controllers. Each controller has several queues (one queue per bank). The controller records the state of each bank independently, and issues commands to the banks based on their state and pending requests in the queues. The controller also schedules the bus that is shared by all banks of each memory type.

The memory queues operate differently than the cache queues in that there is no assumption that there is enough bandwidth to serve all requests in the queue. As a result, requests in each memory queue are serviced one at a time based on bank and bus availability.

With PCM, the performance bottleneck is often associated with *writes* due to severely limited write bandwidth [5]. For example, Choi et al. describe a state-of-the-art prototype 20*nm* 8Gb PCM chip that has 800MB/s read bandwidth, but a paltry 40MB/s write bandwidth [3]. The bandwidth is limited for two reasons. First, programming a PCM cell takes much longer than a DRAM write because a phase-change material has to be heated and then quenched to change between crystalline (low resistance) and amorphous (high resistance) states. Second, heating uses a joule heater attached to PCM. To program a cell requires large electrical current, and consequently, only a small number of cells can be programmed simultaneously [6].

The extent to which restricted write bandwidth harms stream query performance is dependent, of course, on the write patterns of the associated data stream (e.g., event rates) and query operations. A workload that has more writes is likely to be more harmed by limited write bandwidth. However, even a CQ workload that is *not* write-intensive can be adversely affected when run with other query workloads. There is aggregation of writes across all workloads, and any writes will have the potential to occupy memory banks for a long time period, which locks up those banks (busy with writes) from servicing other requests, including reads on which further query processing depends. Thus, memory operations, whether reads or writes, may sit in queues waiting for service in a

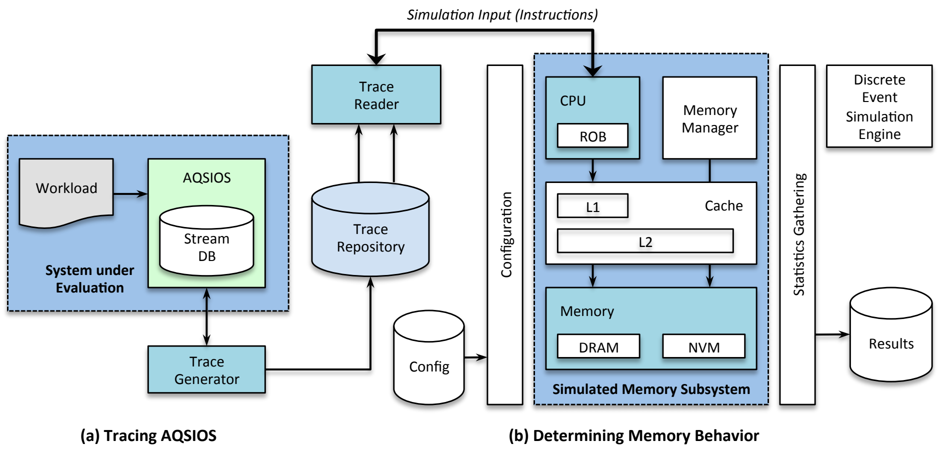

Fig. 2. Tracing and simulation flow for analyzing memory behavior.

hybrid memory. This increased wait time percolates up through the hardware into the software stack, leading to performance degradation and missed QoS and QoD guarantees.

III. Analyzing Memory Behavior

To understand how server consolidation, stream processing and hybrid memory architecture interact, we developed an analysis framework that incorporates stream query processing and a hybrid main memory simulator (Figure 2).

The analysis framework is used in two phases. First, a CQ system is traced. Second, the trace drives a hybrid memory simulator. We describe these phases next.

A. Tracing Query Workloads

The first phase creates, initializes and traces stream query workloads using AQSIOS, which is the "system under evaluation" shown in Figure 2(a). AQSIOS is instrumented to trace the query system at the instruction level using a custom Pintool [9]. The trace captures all instructions executed by the application, including instruction and data memory address references. This trace can be related to important events during CQ processing, such as the production of tuples from one operator to the next.

Since instruction-level tracing is inherently slow (slowdowns during tracing may be 100× or more!) and consumes much disk storage, the trace phase is conducted only for a small, representative snapshot of execution. In our analysis framework, stream processing is warmed-up before the snapshot is taken. The framework determines the warm-up, which includes DSMS initialization and query setup. We also allow the first range of queries to be processed before the snapshot is taken. In this way, all the major data structures are populated and warmed prior to tracing.

The trace snapshot is taken over a period long enough to process several tuples through the system. For our experiments (described in Section IV), the snapshot is 1 billion instructions long. The trace produced for this period is around 1 GB. Several workloads are run through AQSIOS, each one producing an instruction-level trace that is stored in the trace repository. Traces in the repository are used by the second phase.

Queries We use two query networks as described below:

- **QN-A:** A query network that consists of three classes of queries, whose priorities are 6, 3 and 1 with delay targets 300ms, 400ms and 500ms, respectively. All the three classes have the same set of 11 queries, consisting of five aggregates, two window joins, and four selects.
- **QN-B:** The same as QN-A except that we triple the size of the first class so that, when using the real input trace for the first class, the resulting workload is heavy enough to create some load impact in the system.

Inputs We use two streams of synthetic data, denoted $SD_{constant}$ and SD_{step_pareto}, and one of real data SD_{real}. We generated the input tuples for each source beforehand and stored them in a file. Each tuple has a timestamp, which indicates the time the tuple arrives during execution (relative to the experiment's start time) and reflects the input rate.

- $\mathbf{SD_{constant}}$: All the input streams coming to the three classes have a constant input rate of [800-1500] tuples/s.
- $\mathbf{SD_{step\_pareto}}$: The input rate (per control period) of classes 2 and 3 follows a Pareto distribution in the range of [800-1500] and [300-800], respectively, with skew equal to 1. These input rates are expected to overload the classes if they are limited to their originally assigned capacity portions. For class 1, which is the class of highest priority, we change the range for its input rate distribution after every 50-second period in order to vary the amount of excess capacity it can share with the other classes (except for the input of the query segment that can be shared with class 3, which has the same input rate as class 3, so that we can keep the entire workload of class 3 to be at the same level during the experiment).
- $\mathbf{SD_{real}}$: The same input rate patterns as in SD_p are used for class 2 and 3, while the input rate of class 1 is a trace of TCP packets between the Lawrence Berkeley Laboratory and the rest of the world[1].

B. Simulating Hybrid Memory

In the second phase of the analysis framework, traces are extracted from the repository and used to dispatch memory operations into the hybrid memory simulator. The simulator has five main components as shown in Figure 2(b).

The *trace reader* loads traces, decompresses them and sends them to the CPUs. The trace reader can load multiple separate traces to create a trace for multiple simulated CPUs. During simulation, each trace is "pinned" to a core, i.e.,

[1] Dataset LBL-PKT-4/lbl-pkt-n.tcp is publicly available at the following URL: http://ita.ee.lbl.gov/html/contrib/LBL-PKT.html.

TABLE I. WORKLOAD MIXES

Name	Description
W1	const_1000_noshared+const_1220_identical
W2	const_1000_noshared+const_1300_identical
W3	const_1000_noshared+const_1500_identical
W4	const_1000_noshared+real_noshared
W5	const_1000_noshared+step_pareto_noshared
W6	const_1000_noshared+step_pareto_shared
W7	const_1200_identical+const_1300_identical
W8	const_1200_identical+const_1500_identical
W9	const_1200_identical+step_pareto_noshared
W10	const_1220_identical+real_noshared
W11	const_1300_identical+const_1500_identical
W12	const_1300_identical+step_pareto_noshared
W13	const_1500+const_1300_identical
W14	const_1500+real_noshared
W15	const_1500+step_pareto_noshared
W16	real_noshared+const_1500_identical
W17	real_shared+step_pareto_noshared
W18	step_pareto_noshared+const_1500_identical
W19	step_pareto_shared+const_1500_identical

there is no process migration between cores. The trace loader dispatches the traces independently to each core. We use two cores, which execute separate traces, corresponding to different stream processing system instances. The traces are created from the query workloads described above; we run 19 pairwise mixes of these workloads (Table I, described in Section IV).

The *memory manager* translates a virtual address to a physical address used by the CPU and memory hierarchy. Because memory mapping depends on how the operating system allocates pages, the simulator supports different models of allocation behavior. For instance, memory allocation models for random, round-robin and other mappings can be used. For our experiments, we use round-robin mapping of pages to memory banks to interleave physical addresses of the traces.

The *CPU module* receives trace entries from the trace reader and recreates the instructions executed during trace collection. Each instruction consists of an instruction memory access, and zero or more data memory accesses, which are sent to the L1 I or D cache. Data accesses proceed once the instruction returns from the memory hierarchy. The CPU tracks in-flight instructions (reorder buffer) and retires them after all read data has come back from the caches.

The *cache module* models a multi-level cache with private L1 instruction and data caches and shared L2 cache. The caches receive memory requests from the CPU. The caches either send requested data to the previous level (CPU or L1), or forward the request to the next level (L2 or memory). Each cache has a queue, which limits the number of requests at or below the cache level. If a queue is full, requests from previous levels are stalled until a queue slot is freed. The queues, however, do not constrain how many requests can be serviced at a level. Thus, the caches have enough bandwidth to service all incoming requests.

The *memory module* is a detailed model of DDR4 DRAM. It allows modeling PCM that can be accessed through the DDR interface. The model includes multiple banks, row buffers, per-bank or global queues, a bus and a scheduler. Bandwidth of memory devices is limited by the number of requests that can be serviced by all banks and by the bus bandwidth (configurable parameters). The memory module also implements a *hybrid memory controller* that redirects requests to either DRAM or PCM, based on physical address.

The simulator is highly configurable with several parameters, including cache block and page size, CPU issue width, CPU reorder buffer size, cache size, cache associativity, cache

TABLE II. ARCHITECTURAL PARAMETERS

Parameter	**Value**
4GHz processor	4-issue wide, out-of-order core, 128-entry reorder buffer 2 cores used in experiments
L1 I/D private cache	1KB per core, 4-way, LRU, 3 cycle hit, 16-entry queue
L2 unified shared cache	1KB, 16-way, LRU 32 cycle hit, 32-entry queue
4GB DRAM memory @ 1000MHz	64 banks, 32-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-12-12 (ns)
4GB PCM memory @ 400MHz	64 banks, 8-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-55-150 (ns)
PCM/DRAM bus	64-bit single-channel

access latency, cache queue size, number of memory banks, memory bus speed, and row buffer open, access and close latency. The simulator collects statistics for each simulation object, such as CPUs, caches, memory banks and queues. It also records the time spent by requests as they go through the memory components (queues, cache tag arrays, memory banks, buses). These times are aggregated at the CPU module when requests come back from the hierarchy.

IV. IMPACT OF MEMORY SUBSYSTEM

For the experiments, we traced several CQ workloads in AQSIOS. We measured how long AQSIOS takes to start-up and process the first window of tuples. This takes 14 billion instructions, which are skipped prior to tracing instructions. One billion instructions are captured in the trace, along with all instruction and data memory addresses. The traces are mixed by the trace reader to emulate behavior of consolidating separate CQ stream processing instances on the same computer.

We use the query workloads in Table I for HR and RR with and without priority. The workloads have real input rate patterns (i.e., SD_{real}) and query networks of real operators (select, window aggregate and join, etc.), which appear in typical monitoring continuous queries, such as the Linear Road Benchmark [1]. We use single size input tuples that corresponds to the size of TCP packets plus timestamp fields. Intermediate and output tuples could grow to twice the size of the input tuples as a result of join operations. AQSIOS supports multiple query classes with different priorities. In the table, "shared" means the optimizer allows common subexpression computations to be shared across query classes. "No shared" means sharing happens in queries only within the same (priority) class. "Identical" means all queries are the same, i.e., perform the same operations. The other terms refer to the synthetic data described in Section III-A. Our future work will examine sensitivity to workload parameters and other workloads like TPC-H.

We consider hybrid main memory that has DRAM and PCM that is addressable by the CPUs. We simulate CQ data either in DRAM or PCM, i.e., the ends of the spectrum between a conventional DRAM main memory and a PCM main memory. A hybrid memory of both DRAM and PCM would likely have performance between these two designs.

The major architecture parameters for the hybrid memory system are given in Table II. We use standard DRAM and PCM access times and energy from the literature [12], [8]. We use a 64-byte cache block, and PCM latencies from Qureshi et al. [12] are adjusted to account for this block size. For PCM, t_{RP} is 0 for clean row buffers (due to non-volatility) and 150ns for each dirty block in the buffer (due to power constraints of PCM). The simulated main memory is configured with 8

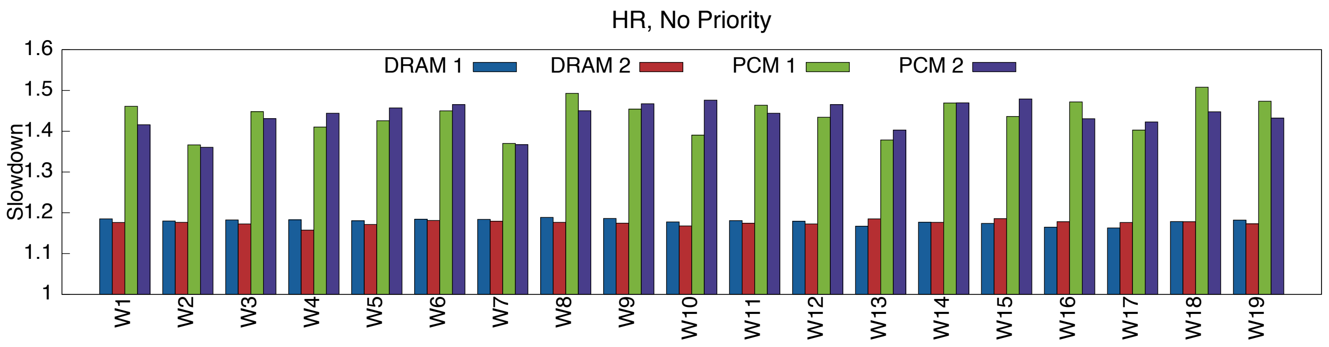

Fig. 3. Slowdown on workloads with HR, no priority.

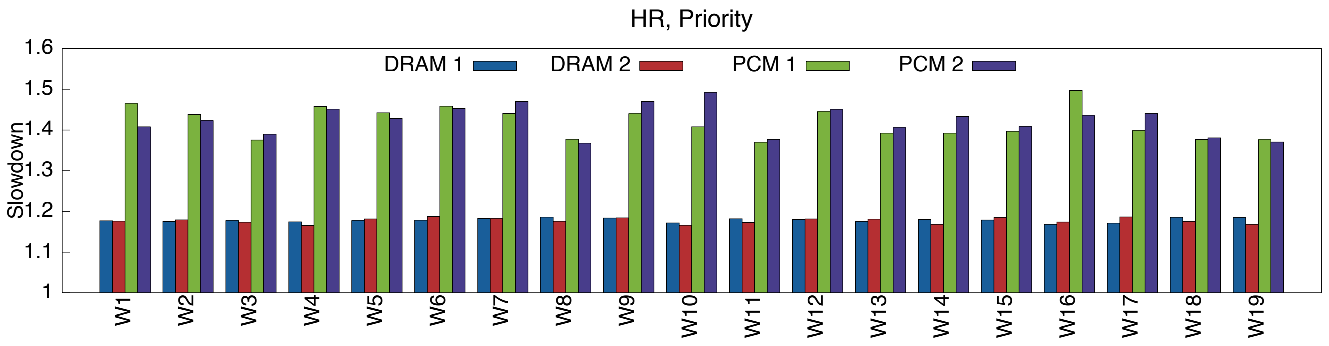

Fig. 4. Slowdown on workloads with HR, priority.

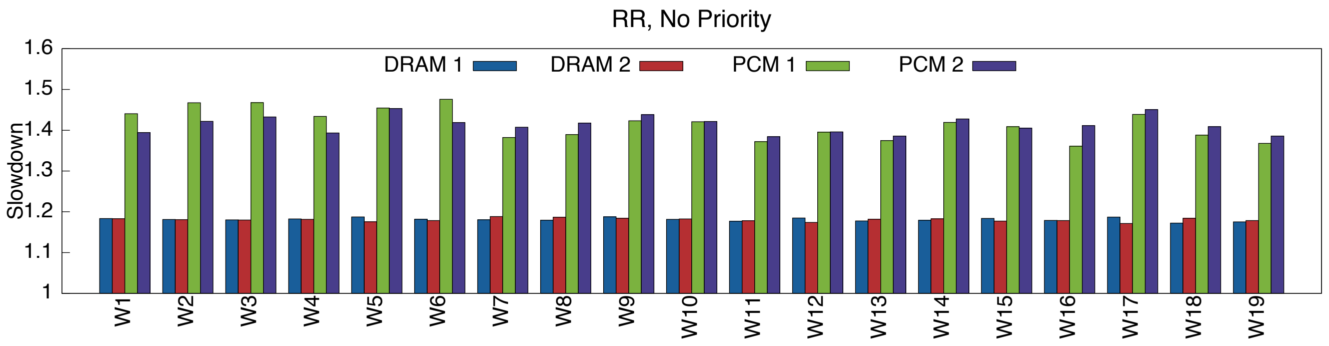

Fig. 5. Slowdown on workloads with RR, no priority.

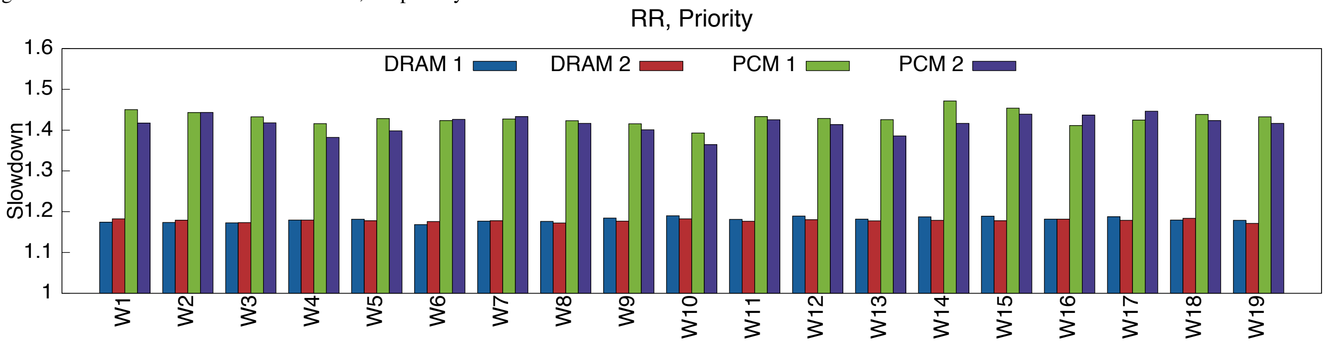

Fig. 6. Slowdown on workloads with RR, priority.

memory banks per rank, 2 ranks per DIMM, and 4 DIMMs. In total, there are 64 banks for both PCM and DRAM, which corresponds to a server main memory.

Because we trace only a short snapshot of total execution, we found that the instruction traces did not sufficiently exercise the main memory when configured with a large, deep cache hierarchy. This situation happens because tracing drastically slows down the execution time of AQSIOS, which influences how many tuples are processed (since processing is time dependent). We found that this reduction in number of tuples caused conventional large caches to exhibit good locality, even for data streaming (which should actually have moderate to poor cache locality). Rather than scaling up the number of tuples processed and the size of the instruction trace snapshot, which would increase tracing and simulation time, we scaled down the cache sizes (see the table) to approximate expected behavior during actual processing. We considered eliminating the caches altogether, but our simulator does not support this functionality. Thus, we configured the L1 and L2 caches to be the smallest possible that the simulator supports (1KB). For our initial study, we believe this is sufficient to understand how DRAM and PCM affect the performance of stream processing.

Figures 3 to 6 show slowdowns for DRAM and PCM of the workloads under the AQSIOS configurations. The graphs have four bars per workload (pair of traces). The first two bars are the slowdown of each trace in a pair when the CQ data is in DRAM. The second two bars are the slowdown of the pair when the CQ data is placed in PCM. Slowdown is execution time of a trace run as part of a pair (with contention) divided by the execution time of the same trace run by itself (no contention). Execution time is the number of cycles reported by HMMSim. The amount of slowdown reflects severity of contention on the fixed work captured in the traces.

The results have a few important trends. The first trend is that *stream processing can indeed suffer from memory contention*; the processing of tuples by the operators generates a significant number of memory operations. For the traces, typically more than 40% of the instructions reference data memory. When two instances of the query processing system share the same memory, they compete for access to memory read and write queues, the row buffers, and the memory bus.

Across the four AQSIOS configurations, there is about a 1.18 slowdown when the CQ data is located in DRAM. It is interesting that the slowdown varies only by a small amount for all pairs in all configurations. We believe this consistent behavior is due to the nature of the traces, exhibiting similar behavior. It is also a property of the memory — the DRAM has enough bandwidth to serve requests from a pair of traces, which does not expose differences in trace sensitivity to contention. We also note the configuration of AQSIOS has only a small performance influence, particularly for DRAM. We believe this happens because the query processing is not taxed enough by the workloads to change the way queries are optimized and scheduled. Furthermore, RR is oblivious to priorities. As opposed to HR, RR keeps data longer in buffers/queues in memory with very little opportunity for an already cached data to be used by consecutively executing operators. In the case of HR, a tuple produced by one operator will be used/consumed with high probability by the immediately following operator.

The second trend is *competition for memory resources is much worse in the PCM case*. The graphs depict slowdowns 1.36 to 1.47 for the four configurations. The occupancy of the memory queues is higher than in the DRAM case since memory banks are more likely to be locked-up servicing writes. This result indicates that CQ workloads have enough writes to put pressure on servicing critical reads (i.e., read data dependencies can impede subsequent instructions). Our memory controller applies typical memory scheduling optimizations, such as giving priority to reads over writes. However, once a write has been issued to a memory bank, subsequent reads to that same bank must wait. Reads to different banks, which are not busy with writes, are scheduled in parallel.

The final trend is that *the memory competition for PCM causes more variability in execution*, even though the traces all have similar work. The issue is even small differences in sensitivity to memory behavior of a trace will be magnified by much longer write latency and lower write bandwidth of PCM. That is, the increased competition for the memory resources exposes minor differences in sensitivity in a pair of traces. For instance, workload W10 in HR, No Priority has this behavior. In the DRAM case, the pair of traces have slowdown of 1.17 and 1.16. In the PCM case, the relative slowdown difference is much greater, with slowdowns of 1.39 and 1.47. This observation also holds with priority.

Overall, from these preliminary results, we conclude that asymmetric read and write latency and bandwidth of a hybrid memory can magnify memory contention. It exposes more sensitivity of a query workload to competition, leading to increased execution time and variability. These results are only initial evidence; a more thorough study under a broader range of workloads, query processing, and memory architectures is needed, and is planned for the future.

V. Conclusion

This paper examines how performance of stream query processing is influenced by consolidation and future memory architectures. Using several query workloads with AQSIOS and the HMMSim hybrid memory simulator, we undertook a preliminary study to determine how much query processing might be affected. We found that memory contention is in fact worse in phase-change main memory systems, dramatically restricting query processing. Our results suggest that memory-resident processing will face significant performance challenges in the future. Our future work will improve this preliminary study with analysis on a greater variety of query processing systems and workloads with consolidation. We will also consider more memory architectures, including hardware and software-managed hybrids of DRAM and PCM.

VI. Acknowledgments

This material is based upon work supported by NSF grants CCF-1422331, CNS-1012070 and CBET-1250171.

References

[1] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *VLDB*, 2004.

[2] S. Bock, B. R. Childers, R. Melhem, and D. Mosse. Understanding the Limiting Factors of Page Migration in Hybrid Main Memory. In *CF'15*, 2015.

[3] Y. Choi et al. A 20nm 1.8V 8Gb PRAM with 40MB/s Program Bandwidth. In *ISSCC*, Feb 2012.

[4] P. K. Chrysanthis. AQSIOS - Next Generation Data Stream Management System. *CONET Newsletter*, June 2010.

[5] Y. Du, M. Zhou, B. Childers, R. Melhem, and D. Mossé. Delta-compressed Caching for Overcoming the Write Bandwidth Limitation of Hybrid Main Memory. *ACM TACO*, Jan. 2013.

[6] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem. Bit Mapping for Balanced PCM Cell Programming. In *ISCA*, 2013.

[7] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big Data and its Technical Challenges. *CACM*, Jul 2014.

[8] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory As a Scalable Dram Alternative. In *ISCA*, 2009.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[10] L. A. Moakar, A. Labrinidis, and P. K. Chrysanthis. Adaptive Class-Based Scheduling of Continuous Queries. In *SMDB*, 2012.

[11] T. N. Pham, P. K. Chrysanthis, and A. Labrinidis. Self-managing load shedding for data stream management systems. In *SMDB*, 2013.

[12] M. Qureshi, M. Franceschini, and L. Lastras-Montano. Improving read performance of Phase Change Memories via Write Cancellation and Write Pausing. In *HPCA*, 2010.

[13] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and Metrics for Processing Multiple Heterogeneous Continuous Queries. *ACM TODS*, Mar 2008.