Hardware Partitioning for Big Data Analytics

Targeted deployment of hardware accelerators can improve throughput and energy efficiency in large-scale data processing. Data partitioning is critical for manipulating large datasets and is often the limiting factor in database performance. A hardware accelerator for range partitioning and a hardware-software streaming framework provide an order of magnitude improvement in partitioning energy and performance.

Lisa Wu
Raymond J. Barker
Martha A. Kim
Kenneth A. Ross
Columbia University

In the era of big data, diverse fields such as natural language processing, medical science, national security, and business management depend on analyzing massive, multidimensional datasets. These communities rely on computer systems to process data quickly and efficiently. In this article, we discuss specialized hardware to more effectively address this task.

Databases manage large quantities of data, letting users query and update the information that they contain. The database community has been developing algorithms to support fast or even real-time queries over relational databases, and, as data sizes grow, researchers increasingly opt to partition the data for faster subsequent processing. Partitioning enables the resulting partitions to be processed independently and more efficiently (that is, in parallel and with better cache locality). Partitioning is used in virtually all modern database systems including Oracle Database 11g, IBM DB2, and Microsoft SQL Server 2012 to improve performance, manageability, and availability in the face of big data, and the partitioning step itself has become a key determinant of query-processing performance.

In this article, we demonstrate that software implementations of data partitioning have fundamental performance limitations that make it computation-bound, even after parallelization. We describe and evaluate a system that both accelerates data partitioning itself and frees processors for other computations. The system consists of two parts: an area and power-efficient specialized processing element for range partitioning, called the Hardware-Accelerated Range Partitioner (HARP); and a high-bandwidth, hardware-software streaming framework that transfers data to and from HARP and integrates seamlessly with existing hardware and software.

Our approach

As the price of memory drops, modern databases aren't typically disk-I/O-bound,[1,2] with many databases now either fitting into main memory or having a memoryz-resident working set. At Facebook, 800 servers supply over 28 Tbytes of in-memory data to users.[3] Despite the relative scarcity of memory pins, there is ample evidence that these and other large data workloads don't saturate the available bandwidth and are largely

0272-1732/14/$31.00 © 2014 IEEE

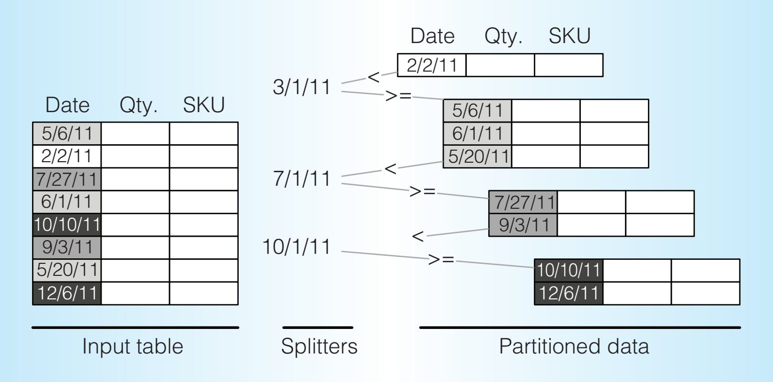

Figure 1. An example table of sales records range partitioned by date into smaller tables. Processing big data one partition at a time makes working sets cache-resident, improving the overall analysis speed.

computation-bound. Servers running Bing, Hotmail, and Cosmos (Microsoft's search, email, and parallel data analysis engines, respectively) show 67 to 97 percent processor use but only 2 to 6 percent memory bandwidth use under stress testing.[4] Google's BigTable and Content Analyzer (large data storage and semantic analysis, respectively) show fewer than 10,000 per millisecond last-level cache misses, which represents just a couple percent of the total available memory bandwidth.[5]

Noting the same imbalances between computing and memory bandwidth, others have opted to save power and scale down memory throughput to better match computing throughput[6,7] or to adjust the resource allocation in server microarchitectures.[8] We propose to resolve the imbalance by deploying specialized hardware to alleviate computing bottlenecks and more fully exploit the available pin bandwidth. This work embodies a unique approach that maximizes memory bandwidth use rather than rebalancing memory and computing together. This is particularly significant in light of the large volumes of data used in modern analyses.

Background and motivation

To begin, we provide some background on partitioning: its role and prevalence in databases, and its software characteristics.

Partitioning background

Partitioning a table splits it into multiple smaller tables called partitions. Each row in the input table is assigned to exactly one partition on the basis of the value of the key field. Figure 1 shows an example table of sales transactions partitioned using the transaction date as the key. This work focuses on a particular partitioning method called range partitioning, which splits the space of keys into contiguous ranges, as illustrated in Figure 1 where sales transactions are partitioned by quarter. The boundary values of these ranges are called splitters.

Partitioning a table allows fine-grained synchronization and data distribution. Moreover, when tables become so large that they or their associated processing metadata can't fit in the cache, partitioning improves the performance of many critical database operations, such as joins, aggregations, and sorts.[9-11] Partitioning is also used in databases for index building, load balancing, and complex query processing.[12] More generally, a partitioner can improve locality for any application that needs to process large datasets in a divide and conquer fashion, such as histogramming, image alignment and recognition, MapReduce-style computations, and cryptoanalysis.

To demonstrate the benefits of partitioning, let's examine joins. A join takes a common key from two tables and creates a new table containing the combined information from both tables. For example, to analyze how weather affects sales, we would join the sales records in `SALES` with the weather records in `WEATHER`, where `SALES.date == WEATHER.date`. If the WEATHER table is too large to fit in the cache, this process will have poor cache locality, as the left side of Figure 2 depicts. On the other hand, if both tables are partitioned by date, each partition can be joined in a pairwise fashion, as the right side of Figure 2 illustrates. When each partition of the `WEATHER` table fits in the cache, the per-partition joins can proceed more rapidly. When the data is large, the time spent partitioning is more than offset by the time saved with the resulting cache-friendly, partition-wise joins.

Join performance is critical because most queries begin with one or more joins to cross-reference tables, and as the most data-intensive and costly operations, their

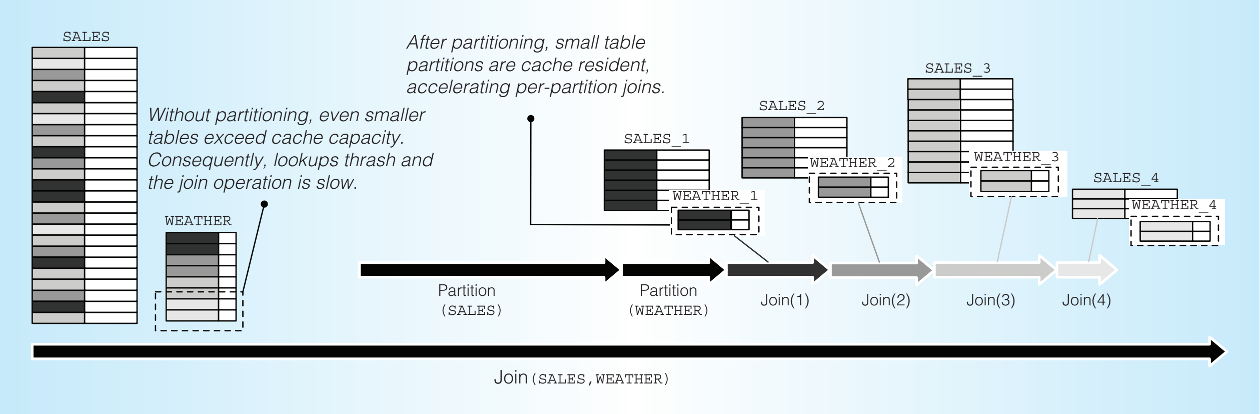
SALES
Without partitioning, even smaller tables exceed cache capacity. Consequently, lookups thrash and the join operation is slow.
WEATHER
After partitioning, small table partitions are cache resident, accelerating per-partition joins.
SALES_1
WEATHER_1
SALES_2
WEATHER_2
SALES_3
WEATHER_3
SALES_4
WEATHER_4
Partition (SALES)
Partition (WEATHER)
Join(1)
Join(2)
Join(3)
Join(4)
Join (SALES,WEATHER)

Figure 2. Joining two large tables exceeds cache capacity. Thus, join implementations partition tables first and then compute partition-wise joins, each of which exhibits substantially improved cache locality.[10,11] Joins are extremely expensive on large datasets, and partitioning represents up to half of the observed join time.[11]

influence on overall performance is large. We measured the fraction of Transaction Processing Performance Council Benchmark H (TPC-H; see http://www.tpc.org/tpch/default.asp) query-execution time attributable to joins using MonetDB (http://www.monetdb.org), an open source database that provides high performance on queries over large datasets. Figure 3 plots the percent of TPC-H runtime spent joining tables. The values shown are the median across the 10 runs of each query. Ranging from 5 to 97 percent, TPC-H spends on average 47 percent of its execution time in a join operation. Current join implementations spend up to half their time in partitioning,[11] thus placing partitioning at approximately 25 percent of TPC-H query-execution time.

In addition to performance, a good partitioner will have several other properties. Ordered partitions, whereby there is an order among output partitions, are useful when a query requires a global data sort. Record order preservation, whereby all records in a partition appear in the same order in which they were found in the input table, is important for some algorithms (for example, radix sorting). Finally, skew tolerance maintains partitioning throughput even when input data is unevenly distributed across partitions. HARP provides all three of these properties as well as high performance and low energy use.

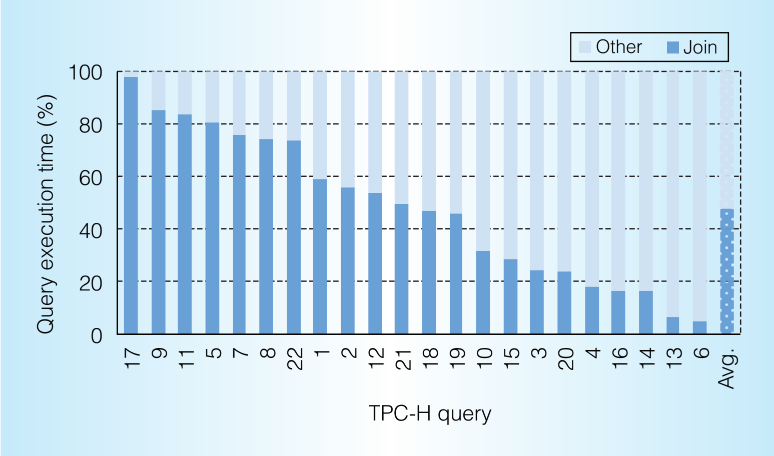

Figure 3. Several key database operations such as join, sort, and aggregation use partitioning to improve performance. Here we see joins consuming 47 percent of the Transaction Processing Performance Council Benchmark H (TPC-H) execution time on MonetDB. With current join algorithms spending roughly half of the join time on partitioning,[11] we estimate that partitioning for joins alone accounts for roughly one quarter of query-execution time.

Software partitioning evaluation

We now characterize the performance and limitations of software partitioning on general-purpose CPUs. Because partitioning scales with additional cores, we analyze both single- and multithreaded performance.[10,11,13]

For these characterizations, we use a microbenchmark that partitions 100 million random records. Although actual partitioning implementations would allocate output space on demand during partitioning, we

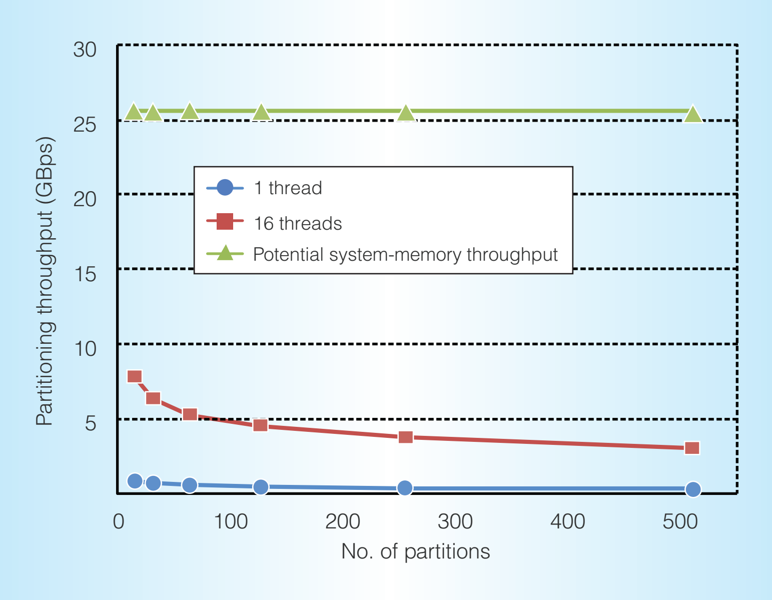

Figure 4. Sixteen threads improve partitioning throughput by 8.5 times, peaking at 2.9 and 2.6 GBps for 128- and 256-way, respectively. However, partitioning remains computation-bound, underusing available memory bandwidth.

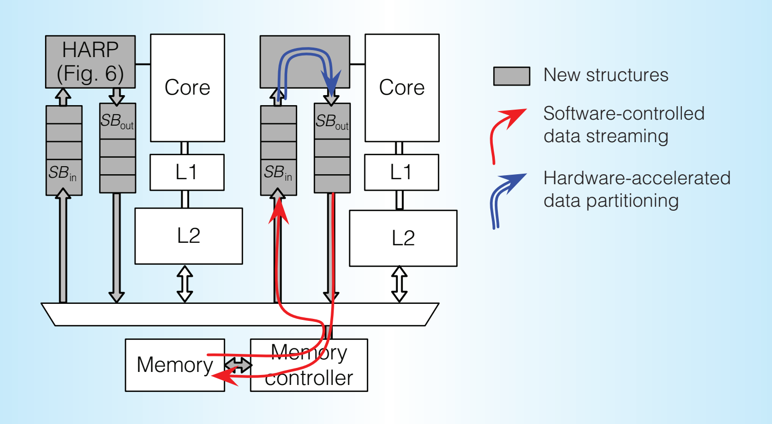

Figure 5. Block diagram of a typical two-core system with Hardware-Accelerated Range Partitioner (HARP) integration. New components (HARP and stream buffers) are shaded.

conservatively preallocate space for the output tables beforehand to streamline the inner loop. The partitioning inner loop runs over an input table reading one record at a time, computing its partition using a partition function, and then writing the record to the destination partition. We implement the partition function using an equality range-partitioning implementation,[14] which performs a binary search of the splitters.

We benchmarked software partitioning throughput on an eight-core Xeon server, and observed from the data in Figure 4 the following: partitioning throughput depends on the number of partitions; partitioning parallelizes reasonably well from one to 16 threads; and while the memory system supports a bandwidth peak of 25.6 Gbytes per second, our optimistic software microbenchmark was able to use only 3 GBps with 16 threads. Even after deploying all computing resources in the server, partitioning remains computing-bound, severely underusing available memory bandwidth. In contrast, we will demonstrate that a single HARP-accelerated thread achieves the throughput of close to 16 software threads using a fraction of the power.

Hardware-Accelerated Range Partitioner

As we saw in the previous discussion, a partitioner's input is a large table, and its output is a set of smaller tables that are easier to process by virtue of their smaller size. Here, we describe the architecture and microarchitecture of a system that incorporates HARP.

Overview

Figure 5 shows a block diagram of the major components in a system with range-partitioning acceleration. Two stream buffers—one running from memory to HARP (SB_{in}) and the other from HARP to memory (SB_{out})—decouple HARP from the rest of the system. The range-partitioning computation is accelerated in hardware (indicated by the double arrow in Figure 5), while inbound and outbound data stream management is left to software (single arrows in Figure 5), maximizing flexibility and simplifying the interface to the accelerator. One set of instructions provides configuration and control for HARP, which freely pulls data from and pushes data to the stream buffers, while a second set of streaming instructions moves data between memory and the stream buffers. Because data moves in a pipeline—that is, streamed in from memory via the streaming framework, partitioned with HARP, and then streamed back out—the lowest-throughput component determines overall system throughput.

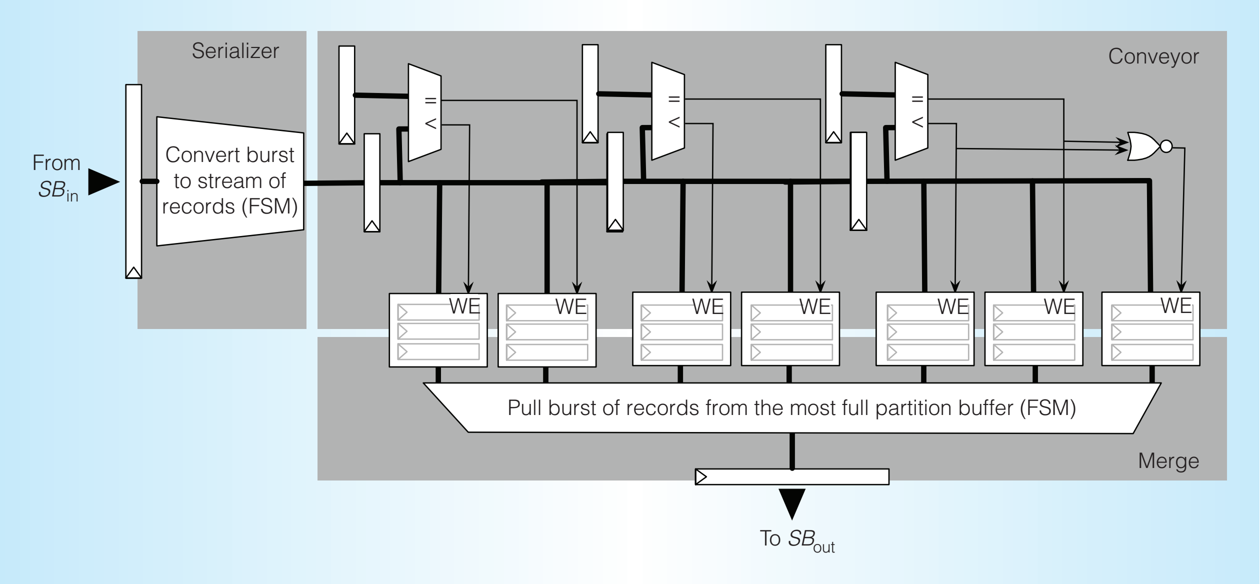
Serializer
Conveyor
From SB_{in}
Convert burst to stream of records (FSM)
=
<
WE
Pull burst of records from the most full partition buffer (FSM)
Merge
To SB_{out}

Figure 6. HARP draws records in bursts, serializing them into a single stream that is fed into a pipeline of comparators. At each stage of the pipeline, the record key is compared with a splitter value, and the record is either filed in a partition buffer (downward) or advanced (to the right) according to the comparison outcome. As records destined for the same partition collect in the buffers, the merge stage identifies and drains the fullest buffer, emitting a burst of records all destined for the same partition. (WE: write enable.)

HARP accelerator

The HARP acceleration is managed via three instructions. The instruction set_splitter is invoked once per splitter to delineate a boundary between partitions; partition_start signals HARP to start pulling data from the SB_{in}; and partition_stop signals HARP to stop pulling data from SB_{in} and drain all in-flight data to SB_{out}. To program a 15-way partitioner, for example, HARP uses seven set_splitter instructions to set values for each splitter value, followed by a partition_start to start partitioning. Because HARP's microarchitectural state is not visible to other parts of the machine, the splitter values are not lost upon interruption.

HARP pulls and pushes records in 64-byte bursts (tuned to match the system vector width and DRAM burst size). The HARP microarchitecture consists of three modules, as Figure 6 depicts, and is tailored to range partition data highly efficiently:

- The serializer pulls bursts of records from SB_{in} and uses a simple finite state machine to pull each record from the burst and feed them, one after another, into the subsequent pipeline. As soon as one burst has been fed into the pipe, the serializer is ready to pull the subsequent burst.
- The conveyor compares record keys against splitters. The conveyor accepts a stream of records from the serializer into a deep pipeline with one stage per splitter. At each stage, the key is compared to the corresponding splitter and routed either to the appropriate partition or to the next pipeline stage. Partition buffers, one per partition, buffer records until a burst of them is ready.
- The merge module monitors the partition buffers as records accumulate. It looks for full bursts of records that it can send to a single partition. When such a burst is ready, merge drains the partitioning buffer, one record per cycle, and sends the burst to SB_{out}.

HARP uses deep pipelining to hide the latency of multiple splitter comparisons. We experimented with a tree topology for the conveyor, analogous to the binary search tree in the software implementation, but found that the linear conveyor architecture was preferable. When the pipeline operates bubble-free, as it does in both cases, it processes one record per cycle, regardless of topology. The only difference in total cycle count between the linear and tree conveyors was the overhead of filling and draining the pipeline at the start and finish. With large record counts, the difference is negligible in time required to fill and drain a *k*-stage pipeline versus a log(*k*)-stage pipeline in the tree version. Although cycle counts were more or less the same between the two approaches, the linear design had a slightly shorter clock period because of the more complex layout and routing requirements in the tree, resulting in slightly better overall throughput.

The integer comparators in HARP can support all SQL data types as partitioning keys. This is because the representations typically lend themselves to integer comparisons. For example, MySQL represents dates and times as integers: dates as 3 bytes, time stamps as 4 bytes, and datetimes as 8 bytes.[15] HARP can also accomplish partitioning ASCII strings alphabetically on the first *N* characters with an *N*-byte integer comparator.

Delivering data to and from HARP

To ensure that HARP can process data at its full throughput, the framework surrounding HARP must stream data to and from memory at or exceeding the rate that HARP can partition. This framework provides software-controlled streams and allows the machine to continue seamless execution after an interrupt, exception, or context switch. We describe a hardware-software streaming framework based on the concept outlined in Jouppi's prefetch stream buffer work.[16]

Software moves data between memory and the stream buffers via four instructions. Instruction `sbload` loads data from memory to SB_{in}, taking as arguments a source address in memory and a destination stream buffer ID. The instruction `sbstore` does the reverse, taking data from the head of the designated outgoing stream buffer and writing it to the specified address. Each `sbload` and `sbstore` moves one vector's worth of data (that is, 128 or 256 bytes) between memory and the stream buffers. A full/empty bit on the stream buffers will block the `sbloads` and `sbstores` until there is space (in SB_{in}) and available data (in SB_{out}). Because the CPU software knows the table size, it knows how many `sbload`/`sbstore` instructions must be executed to partition the entire table.

To ensure seamless execution after an interrupt, exception, or context switch, we make a clean separation of architectural and microarchitectural states. Specifically, only the stream buffers themselves are architecturally visible, with no accelerator state exposed architecturally. This separates the HARP microarchitecture from the context and will help facilitate future extension to other streaming accelerators. Before the machine suspends accelerator execution to service an interrupt or a context switch, the OS will execute an `sbsave` instruction to save the stream buffer contents. Prior to an `sbsave`, HARP must be stopped and allowed to drain its in-flight data to an outgoing stream buffer by executing a `partition_stop` instruction. As a consequence, the stream buffers should be sized to accommodate the maximum amount of in-flight data supported by HARP. After the interrupt has been serviced, before resuming HARP execution, the OS will execute an sbrestore to ensure that the streaming states are identical before and after the interrupt or context switch.

These stream buffer instructions, together with the HARP instructions described in the previous section, allow full software control of all aspects of the partitioning operation, except for the work of partitioning itself, which is handled by HARP.

To implement the streaming instructions, we propose minimal modifications to conventional processor microarchitecture. Figure 7 summarizes the new additions. The `sbload` instructions borrow the existing microarchitectural vector load (for example, Intel's Streaming SIMD Extensions or PowerPC's AltiVec) request path, diverging from vector load behavior when data fills return to the stream buffer instead of the data cache hierarchy. To support this, we add a 1-bit attribute to the existing last-level cache request buffer to differentiate `sbload` requests from conventional vector load requests. This attribute acts as the multiplexer

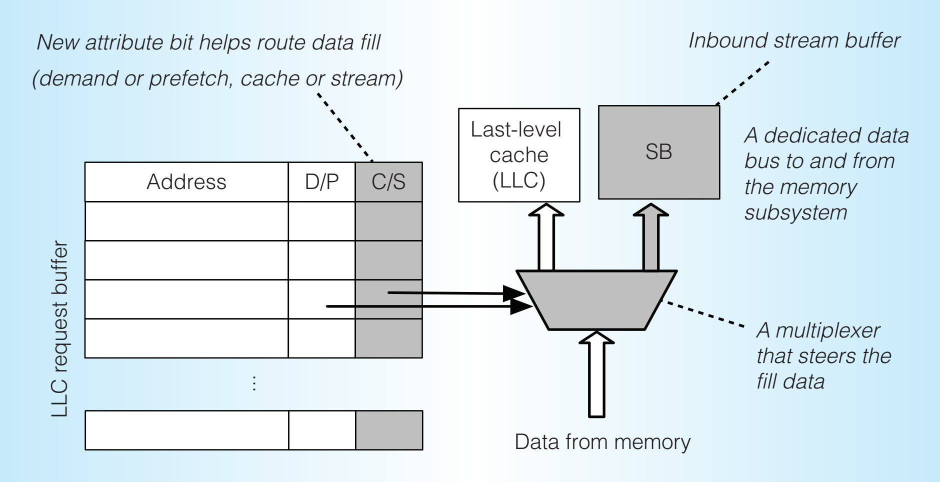
New attribute bit helps route data fill (demand or prefetch, cache or stream)
Inbound stream buffer
Address
D/P
C/S
Last-level cache (LLC)
SB
A dedicated data bus to and from the memory subsystem
LLC request buffer
A multiplexer that steers the fill data
Data from memory

Figure 7. Implementation of streaming instructions into existing datapath of a generic last-level cache request/fill microarchitecture. Required minimal modifications are shaded.

select for the return datapath, as Figure 7 illustrates. Finally, a dedicated bidirectional data bus is added to connect that mux to the stream buffer.

Stream buffers can be made fully coherent to the core caches. The `sbload` instructions already reuse the load request path, so positioning SB_{in} on the fill path such that hits in the cache can be returned to the SB_{in} will ensure that `sbload` requests always produce the most up-to-date values. Figure 7 depicts the scenario when a request misses all levels of the cache hierarchy, and the fill is not cached, as `sbload` requests are noncacheable. On the store side, `sbstore` instructions can copy data from SB_{out} into the existing store buffer, sharing the store datapath and structures such as the write combining and snoop buffers.

Stream loads are most effective when data is prefetched ahead of use, and our experiments indicate that the existing hardware prefetchers are quite effective in bringing streaming data into the processor. Prefetches triggered by stream loads can be handled in one of two ways: fill the prefetched data into the cache hierarchy as occurs in current processors, or fill the prefetched data into the stream buffer. We choose the former because it reduces the additional hardware support needed and incurs minimal cache pollution by marking prefetched data as nontemporal. Because `sbload` requests check the cache and request buffer for outstanding requests before sending the request out to the memory controller, this design allows for coalescing loads and stores and for shorter data return latency when the requests hit in the prefetched data in the cache.

Evaluation

To evaluate the throughput, power, and area efficiency of our design, we implemented HARP in Bluespec System Verilog (www.bluespec.com). The partitioner evaluated here supports 16-byte records with 4-byte keys. Assuming 64-byte DRAM bursts, this works out to four records per burst. We evaluate the overhead of the streaming framework using CACTI.[17] For further details about the methodology, including synthesis settings, please refer to the methodology section of our other work.[18]

We evaluate the proposed HARP system in the following categories:

- throughput comparison with the optimistic software range partitioning from the "Software partitioning evaluation" section,
- area and power comparison with the processor core on which the software experiments were performed, and
- nonperformance partitioner desiderata.

We use the baseline configuration of HARP outlined in the previous paragraph, unless otherwise noted.

HARP throughput

Figure 8 plots the throughput of three range partitioner implementations: single-

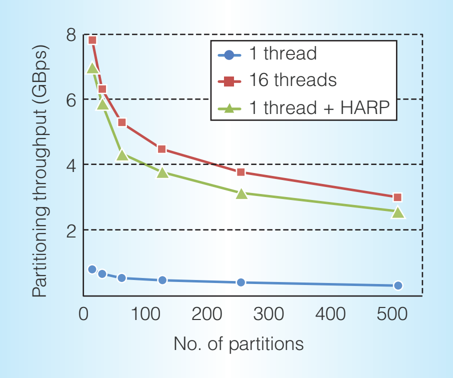

Figure 8. A single HARP unit outperforms single-threaded software from between 7.8 times with 63 or 255 partitions to 8.8 times with 31 partitions, approaching the throughput of 16 threads.

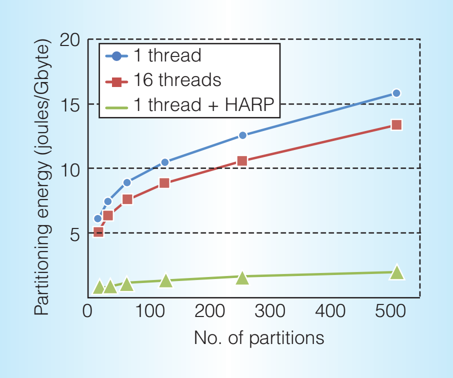

Figure 10. This figure shows the energy consumption in joules per Gbyte of data partitioned using HARP as the number of partitions increases. HARP-augmented cores partition data using 6.3 to 8.7 times less energy than parallel or serial software.

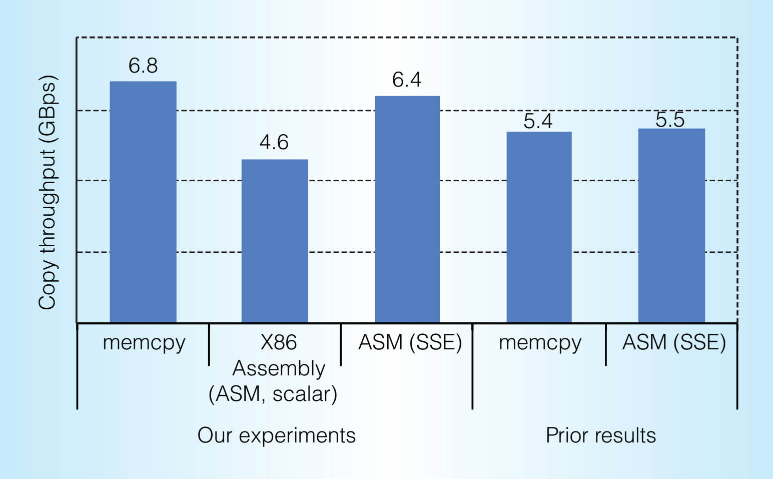

Figure 9. The streaming framework shares much of its implementation with the existing memory system, and as such its throughput will be comparable to the copy throughput of existing systems. (SSE: streaming SIMD extensions.)

threaded software, multithreaded software, and single-threaded software plus HARP. We see that HARP's throughput exceeds a single software thread by 6.5 to 8.8 times, with the difference primarily attributable to the elimination of instruction fetch and control overhead from the splitter comparison and the deep pipeline. In particular, the partitioning operation structure doesn't introduce hazards or bubbles into the pipeline, allowing it to operate in a near-perfect fashion—that is, always full as well as accepting and emitting one record per clock cycle. We confirm this empirically as our measurements indicate average cycles per record ranging from 1.008 (for 15-way partitioning) to 1.041 (for 511-way partitioning). As Figure 8 indicates, 16 threads are required for the software implementation to match the throughput of the hardware implementation. At 3.13 GBps per core with HARP, augmenting all or even half of the eight cores with HARP would provide sufficient computing bandwidth to fully use all DRAM pins.

In terms of absolute numbers, the baseline HARP configuration achieved a 5.06-nanosecond critical path, yielding a design that runs at 198 MHz, delivering partitioning throughput of 3.13 GBps. This is 7.8 times faster than the optimistic single-threaded software range-partitioner described in the "Software Partitioning Evaluation" section.

Streaming throughput

Our results in Figure 9 show that C's standard library memcpy provides similar throughput to hand-optimized vector code, whereas scalar code's throughput is slightly lower. For comparison, we have also included the results of a similar experiment published by IBM Research.[19] Based on these measurements, we will conservatively estimate that the streaming framework can bring in data at 4.6 GBps and write results to memory at 4.6 GBps with a

Table 1. Area and power overheads of HARP units and stream buffers for various partitioning factors.

No. of partitions	HARP unit				Stream buffers			
	Area		Power		Area		Power	
	mm^2	Xeon (%)	Watts	Xeon (%)	mm^2	Xeon (%)	Watts	Xeon (%)
15	0.16	0.4	0.01	0.3	0.07	0.2	0.063	1.3
31	0.31	0.7	0.02	0.4	0.07	0.2	0.079	1.6
63	0.63	1.5	0.04	0.7	1.30	0.2	0.078	1.6
127	1.34	3.1	0.06	1.3	0.11	0.3	0.085	1.7
255	2.83	6.6	0.11	2.3	0.13	0.3	0.100	2.0
511	5.82	13.6	0.21	4.2	0.18	0.4	0.233	4.7

single thread. These data shows that the streaming framework provides more throughput than HARP can take in, but not too much more, resulting in a balanced system.

Area and power efficiency

The addition of the stream buffer and accelerator hardware does increase the area and power of the core. Table 1 quantifies the area and power overheads of the accelerator and stream buffers relative to a single Xeon core. Comparatively, the additional structures are small, with the baseline design point adding just 6.9 percent area and 4.3 percent power for both the HARP and the stream buffers. HARP itself consumes just 2.83 mm^2 and 0.11 W.

Because the stream buffers are sized according to the accelerators they serve, we quantify their area and power overheads for each HARP partitioning factor we consider in Table 1. The proposed streaming framework adds 0.3 mm^2 of area and consumes 10 mW of power for a baseline HARP configuration.

Energy efficiency

From an energy perspective, this slight increase in power is overwhelmed by the improvement in throughput. Figure 10 compares the partitioning energy per gigabyte of data of software (both serial and parallel) against HARP-based alternatives. The data show a 6.3 to 8.7 times improvement in single-threaded partitioning energy with HARP.

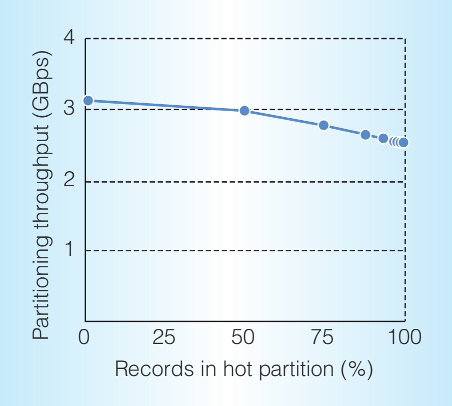

Figure 11. This figure shows the impact of uneven partition distribution on partitioning throughput. As input imbalance increases, throughput drops by at most 19 percent owing to increased occurrence of back-to-back bursts to the same partition.

By design, HARP preserves the record order. All records in a partition appear in the same order that they were found in the input record stream. This is a useful property for other parts of the database system and is a natural consequence of the HARP structure, where only one route exists from input port to each partition and records can't pass one another in-flight.

We evaluate HARPs skew tolerance by measuring the throughput (that is, cycles per record) on synthetically unbalanced record sets. In this experiment, we varied the record distribution from optimal, where records were uniformly distributed across all partitions, to pessimal, where all records were sent to a single partition. Figure 11 shows the gentle degradation in throughput as one partition receives an increasingly large share of records.

This mild degradation is due to the design of the merge module. Recall that this stage identifies which partition has the most records ready and drains them from that partition's buffer to send as a single burst back to memory. Back-to-back drains of the same partition require an additional merge cycle, which rarely happens when records are distributed across partitions. Note that this tolerance is independent of many factors including splitter number, key size, or partitioned table size.

The baseline HARP design supports four records per burst, resulting in a 25 percent degradation in throughput between best- and worst-case skew. This is close to the degradation seen experimentally in Figure 11, where throughput sinks from 3.13 GBps with no skew to 2.53 GBps in the worst case.

This research offers a novel solution to the problem of big data computing efficiency. The database community has spearheaded software innovations to improve performance efficiency of database management systems running on commodity server hardware, and several researchers have proposed running big data analyses on field-programmable gate arrays. However, this is the first work to accelerate partitioning, a generic kernel that is a critical piece of many large data analyses. We have described a specialized database processing element and a streaming framework that provide seamless execution in modern computer systems and exceptional throughput and power efficiency advantages over software. These benefits are necessary to address the ever-increasing demands of big data processing.

Processing data with accelerators such as HARP can alleviate serial performance bottlenecks in the application and free up resources on the server to do other useful work. Because databases and other data-processing systems represent a common, high-value server workload, the impact of improvements in partitioning performance would be widespread.

The design shows how accelerators can be seamlessly integrated into a CPU core. The streaming framework decouples the microarchitecture of the accelerator from the specifics of data layout and management. This allows seamless integration of the accelerator into existing software, as well as a clean mechanism for handling context switches and interrupts by saving and restoring just the contents of the stream buffers.

The research demonstrates the potential of data-oriented specialization. Moving data through the memory subsystem and CPU cache hierarchy consumes more than double the energy of the computation itself.[20] With an application-specific integrated circuit designed to specifically process tables in a streaming fashion, the HARP system delivers an order of magnitude improvement in energy efficiency. The overall system design also makes it easy to introduce other streaming accelerators such as specialized aggregators, joiners, sorters, filters, or compressors to expand both the use and benefits of this approach. MICRO

References

1. A. Ailamaki et al., "DBMSs on a Modern Processor: Where Does Time Go?" *Proc. 25th Int'l Conf. Very Large Data Bases*, 1999, pp. 266-277.
2. G. Graefe and P.-A. Larson, "B-Tree Indexes and CPU Caches," *Proc. 17th Int'l Conf. Data Engineering*, 2001, pp. 349-358.
3. P. Saab, "Scaling Memcached at Facebook," 12 Dec. 2008; https://www.facebook.com/note.php?note_id=39391378919.
4. C. Kozyrakis et al., "Server Engineering Insights for Large-Scale Online Services," *IEEE Micro*, vol. 30, no. 4, 2010, pp. 8-19.
5. L. Tang et al., "The Impact of Memory Subsystem Resource Sharing on Datacenter Applications," *Proc. 38th Ann. Int'l Symp. Computer Architecture*, 2011, pp. 283-294.
6. K.T. Malladi et al., "Towards Energy-Proportional Datacenter Memory with Mobile DRAM," *Proc. 39th Ann. Int'l Symp. Computer Architecture*, 2012, pp. 37-48.
7. Q. Deng et al., "MemScale: Active Low-Power Modes for Main Memory," *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 225-238.

8. N. Hardavellas et al., "Toward Dark Silicon in Servers," *IEEE Micro*, vol. 31, no. 4, 2011, pp. 6-15.
9. Y. Ye, K.A. Ross, and N. Vesdapunt, "Scalable Aggregation on Multicore Processors," *Proc. 7th Int'l Workshop Data Management on New Hardware*, 2011, pp. 1-9.
10. S. Blanas, Y. Li, and J.M. Patel, "Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs," *Proc. ACM Sigmod Int'l Conf. Management of Data*, 2011, pp. 37-48.
11. C. Kim et al., "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs," *Proc. Very Large Data Bases*, vol. 2, no. 2, 2009, pp. 1378-1389.
12. D. Chatziantoniou and K.A. Ross, "Partitioned Optimization of Complex Queries," *Information Systems*, vol. 32, no. 2, 2007, pp. 248-282.
13. J. Cieslewicz and K.A. Ross, "Data Partitioning on Chip Multiprocessors," *Proc. 4th Int'l Workshop Data Management on New Hardware*, 2008, pp. 25-34.
14. K.A. Ross and J. Cieslewicz, "Optimal Splitters for Database Partitioning with Size Bounds," *Proc. 12th Int'l Conf. Database Theory*, 2009, pp. 98-110.
15. MySQL, "Date and Time Data Type Representation," 1997, 2014; http://dev.mysql.com/doc/internals/en/date-and-time-data-type-representation.html.
16. N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, 1990, pp. 364-373.
17. HP Labs, "CACTI," 2008; http://www.hpl.hp.com/research/cacti.
18. L. Wu et al., "Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 249-260.
19. H. Subramoni et al., "Intra-Socket and Inter-Socket Communication in Multi-Core Systems," *IEEE Computer Architecture Letters*, vol. 9, no. 1, 2010, pp. 13-16.
20. W.J. Dally et al., "Efficient Embedded Computing," *Computer*, vol. 41, no. 7, 2008, pp. 27-32.

Lisa Wu is a research staff member at Intel Labs. Her research interests include computer architecture, accelerators, and energy-efficient computing on high-performance computing and big data. Wu has a PhD in computer science from Columbia University, where she performed the work for this article. She is a member of ACM Sigarch.

Raymond J. Barker is a software engineer at Google and an MS student in computer engineering at Columbia University. His research interests include application-specific computer architecture and parallel algorithms. Barker has a BS in computer engineering from Columbia University. He is a member of IEEE.

Martha A. Kim is an assistant professor in the Computer Science Department at Columbia University. Her research interests include computer architecture, parallel hardware and software systems, and energy-efficient computation on big data. Kim has a PhD in computer science and engineering from the University of Washington. She is a member of IEEE and the ACM.

Kenneth A. Ross is a professor of computer science at Columbia University. His research interests include database management systems, particularly their performance on modern multicore machines, GPUs, and other accelerator platforms. Ross has a PhD in computer science from Stanford University. He is a member of the ACM.

Direct questions and comments about this article to Lisa Wu, Intel, Building SC12, 3600 Juliette Ln., M/S 303, Santa Clara, CA 95052; lisa@cs.columbia.edu.