Energy Analysis of Hardware and Software Range Partitioning

LISA WU, ORESTIS POLYCHRONIOU, RAYMOND J. BARKER, MARTHA A. KIM, and KENNETH A. ROSS, Columbia University

8

Data partitioning is a critical operation for manipulating large datasets because it subdivides tasks into pieces that are more amenable to efficient processing. It is often the limiting factor in database performance and represents a significant fraction of the overall runtime of large data queries. This article measures the performance and energy of state-of-the-art software partitioners, and describes and evaluates a hardware range partitioner that further improves efficiency.

The software implementation is broken into two phases, allowing separate analysis of the partition function computation and data shuffling costs. Although range partitioning is commonly thought to be more expensive than simpler strategies such as hash partitioning, our measurements indicate that careful data movement and optimization of the partition function can allow it to approach the throughput and energy consumption of hash or radix partitioning.

For further acceleration, we describe a hardware range partitioner, or HARP, a streaming framework that offers a seamless execution environment for this and other streaming accelerators, and a detailed analysis of a 32nm physical design that matches the throughput of four to eight software threads while consuming just 6.9% of the area and 4.3% of the power of a Xeon core in the same technology generation.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Microprocessor/Microcomputer Applications

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Accelerator, specialized functional unit, streaming data, microarchitecture, data partitioning

ACM Reference Format:
Lisa Wu, Orestis Polychroniou, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2014. Energy analysis of hardware and software range partitioning. ACM Trans. Comput. Syst. 32, 3, Article 8 (August 2014), 24 pages.
DOI: http://dx.doi.org/10.1145/2638550

1. INTRODUCTION

In the era of big data, a diverse set of fields, such as natural language processing, medical science, national security, and business management, depend on sifting through and analyzing massive, multidimensional datasets. These communities rely on computer systems to process vast volumes of data quickly and efficiently. In this article, we

---

This manuscript contains content previously published ISCA '13 [Wu et al. 2013]. This extended article substitutes a state-of-the-art software partitioner [Polychroniou and Ross 2014] for the microbenchmark used in the original paper and includes the new, extensive exploration of software partitioning performance and energy found in Section 3. The research was supported by grants from the National Science Foundation (CCF-1065338 and IIS-0915956) and a gift from Oracle Corporation.
Authors' address: L. Wu, O. Polychroniou, R. J. Barker, M. A. Kim, and K. A. Ross, Computer Science Department, Columbia University, 1214 Amsterdam Avenue, New York, NY 10027; emails: lisa@cs.columbia.edu; orestis@cs.columbia.edu; rjb2150@columbia.edu; martha@cs.columbia.edu; kar@cs.columbia.edu.

DOI: http://dx.doi.org/10.1145/2638550

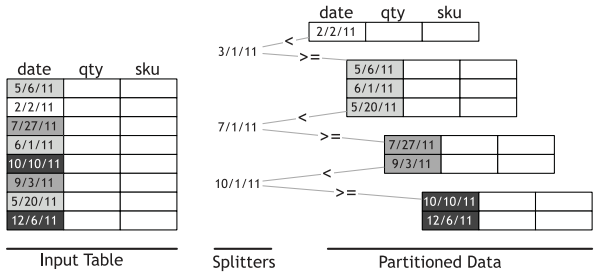

Fig. 1. An example table of sales records range partitioned by date into smaller tables. Processing big data one partition at a time makes working sets cache resident, dramatically improving the overall analysis speed.

deploy specialized hardware and highly optimized software to more effectively address this task.

Databases are designed to manage large quantities of data, allowing users to query and update the information 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, they increasingly opt to *partition* the data for faster subsequent processing. As illustrated in the small example in Figure 1, partitioning assigns each record in a large table to a smaller table based on the value of a particular field in the record, such as the transaction date in the figure. Partitioning enables the resulting partitions to be processed independently and more efficiently (i.e., in parallel and with better cache locality). Partitioning is used in virtually all modern database systems, including Oracle Database 11g [Oracle 2013], IBM DB2 [IBM 2006], and Microsoft SQL Server 2012 [Microsoft 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.

As the price of memory drops, modern databases are not typically disk I/O bound [Ailamaki et al. 1999; Graefe and Larson 2001], with many databases now either fitting into main memory or having a memory-resident working set. At Facebook, 800 servers supply more than 28TB of in-memory data to users [Saab 2008]. Despite the relative scarcity of memory pins, there is ample evidence that these and other large data workloads do not saturate the available bandwidth and are largely compute bound. Servers running Bing, Hotmail, and Cosmos (Microsoft's search, email, and parallel data analysis engines, respectively) show 67% to 97% processor utilization but only 2% to 6% memory bandwidth utilization under stress testing [Kozyrakis et al. 2010]. Google's BigTable and Content Analyzer (large data storage and semantic analysis, respectively) show fewer than 10K/msec last-level cache misses, which represents just a couple of percent of the total available memory bandwidth [Tang et al. 2011].

Noting the same imbalances between compute and memory bandwidth, others have opted to save power and scale down memory throughput to better match compute throughput [Malladi et al. 2012; Deng et al. 2011] or to adjust the resource allocation in server microarchitectures [Hardavellas et al. 2011]. This work explores two avenues to resolve the imbalance. The first evaluates several software implementations of data partitioning, evaluating both their performance and energy characteristics. We identify the relative energy costs of computing output partitions and actually moving data to the proper partition, finding that for well-optimized partition functions the data shuffling incurs most of the energy consumption. In addition, we note that a naive implementation for computing the range partitioning function may be prohibitively

slow, thus special care must be taken to ensure that the range function can be computed in the most efficient way.

The second avenue deploys specialized hardware to alleviate compute bottlenecks and more fully exploit the available pin bandwidth. 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, referred to as the Hardware-Accelerated Range Partitioner (HARP). Synthesized, placed, and routed, a single HARP unit would occupy just 6.6% of the area of a commodity Xeon processor core and can process up to 3.13GB/sec of input three to five times faster times faster than a single software thread and matching the throughput of four to eight threads.
—A high-bandwidth hardware-software streaming framework that transfers data to and from HARP and integrates seamlessly with existing hardware and software. This framework adds 0.3mm^2 area, consumes 10mW power, and provides a minimum of 4.6GB/sec bandwidth to the accelerator without polluting the caches.

Since databases and other data processing systems represent a common, high-value server workload, the impact of improvements in partitioning performance would be widespread.

2. 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 based on 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 (e.g., incoming sales lock and update only the most recent partition) and data distribution (e.g., New York sales records can be stored on the East Coast for faster access). When tables become so large that they or their associated processing metadata cannot fit in the cache, partitioning is used to improve the performance of many critical database operations, such as joins, aggregations, and sorts [Ye et al. 2011; Blanas et al. 2011; Kim et al. 2009]. Partitioning is also used in databases for index building, load balancing, and complex query processing [Chatziantoniou and Ross 2007]. 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 cryptanalysis.

To demonstrate the benefits of partitioning, let us examine joins. A *join* takes a common key from two different tables and creates a new table containing the combined information from both tables. For example, to analyze how weather affects sales, one 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 whole process will have very poor cache locality, as depicted on the left of Figure 2. On the other hand, if both tables are partitioned by date, each partition can be joined in a pairwise fashion as illustrated on the right. When each partition of the `WEATHER` table fits in the cache, the per-partition joins can proceed much 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.

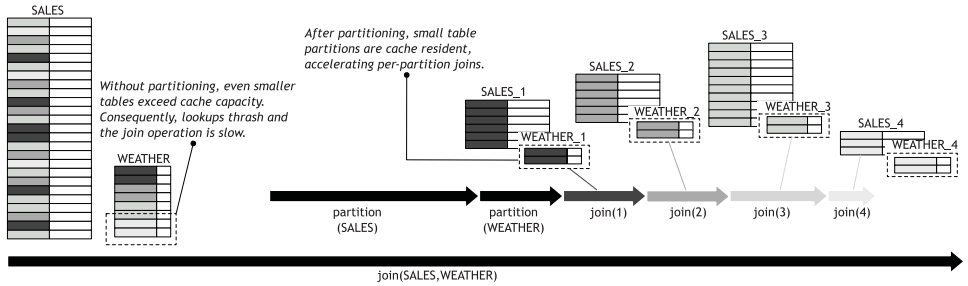

Fig. 2. Joining two large tables easily exceeds cache capacity. Thus, state-of-the-art join implementations partition tables first and then compute partition-wise joins, each of which exhibits substantially improved cache locality [Kim et al. 2009; Blanas et al. 2011]. Joins are extremely expensive on large datasets, and partitioning represents up to half of the observed join time [Kim et al. 2009].

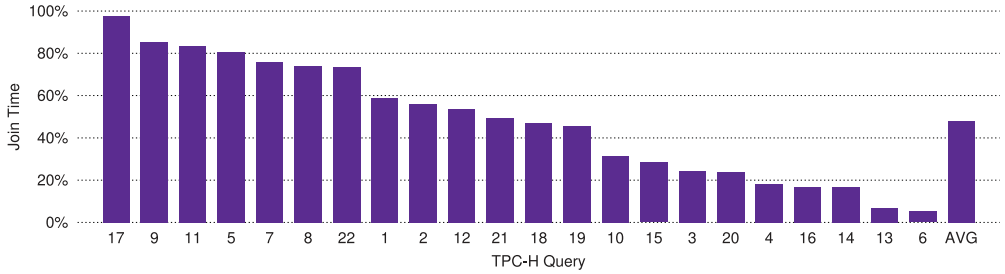

Fig. 3. Several key database operations, such as join, sort, and aggregation, use partitioning to improve their performance. Here we see joins consuming 47% of the TPC-H execution time on MonetDB. With state-of-the-art join algorithms spending roughly half of the join time partitioning [Kim et al. 2009], we estimate that partitioning for joins alone accounts for roughly one quarter of query execution time.

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 influence on overall performance is large. We measured the fraction of TPC-H [Transaction Processing Performance Council 2014] query execution time attributable to joins using MonetDB [Centrum Wiskunde and Informatica 2012], an open source database designed to provide high performance on queries over large datasets.[1] Figure 3 plots the percent TPC-H runtime spent joining tables. The values shown are the median across the 10 runs of each query. Ranging from 97% to 5%, on average TPC-H spends 47% of its execution time in a join operation. State-of-the-art implementations of joins spend up to half their time in partitioning [Kim et al. 2009], thus placing partitioning at approximately 25% 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, is useful when a query requires a global sort of the data. *Record order preservation*, whereby all records in a partition appear in the same order they were found in the input table, is important for some algorithms (e.g., 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.

[1]Data collected using MonetDB 11.11.5 (release configuration, compiled with maximal optimization) on a dual-processor server (Intel Xeon X5660, 6C/12T, 2.8GHz, with 12MB LLC) with 64GB DRAM. MonetDB used up to 24 threads per query, each of which was executed 10 times in random order to minimize the impact of cached results.

3. SOFTWARE PARTITIONING EVALUATION

We characterize the performance, energy, and limitations of software partitioning on general-purpose CPUs. Since partitioning scales with additional cores [Cieslewicz and Ross 2008; Kim et al. 2009; Blanas et al. 2011], we analyze both single- and multi-threaded performance, as well as the impacts of processor frequency scaling and data shuffling policies.

The partitioning operation consists of two logical phases: computing the partition function and shuffling (i.e., moving) the data. Partitioning offers multiple implementation choices for each phase. For example, if we want an implementation that does not require linear auxiliary space, we need to use "in-place" partitioning, which affects the algorithm and its performance significantly. Recent work provides a detailed explanation and exploration of many such variants [Polychroniou and Ross 2014]. We break our analysis of software partitioning down by phase, first examining data shuffling policies in isolation, then later including the computation of the partition function.

3.1. Data Shuffling Discussion

Here, we describe a basic approach to shuffling partitioned data, followed by several optimization strategies. We will apply these techniques in four combinations, listed in Section 3.2, and evaluate them in Section 3.3.

Naive shuffling. The naive approach to shuffling partitioned data is to move one record, or tuple, at a time. For simplicity, let's assume that the partitions will be placed in contiguous space in memory. Keeping a pointer for each partition, incrementing it for each item appended to the partition, each item is written directly to its final output destination and is never considered again. This simplistic approach has a number of weaknesses that dramatically decrease performance when the working set footprint exceeds the size of the cache.

Prior work has identified the TLB thrashing problem [Manegold et al. 2000]. Because output partitions usually exceed page sizes, one can typically expect each partition to write on a different OS page. Since the virtual to physical address translation happens through the TLB, we can estimate that to generate P partitions, we need at least P TLB entries to avoid throttling performance. Current hardware typically has 32 or 64 TLB entries operating at the minimum latency (first-level TLB). If a TLB miss occurs, the page is searched in the next-level TLB, which has more entries but might be shared among multiple cores of the chip and incurs a higher access latency. In the worst case, when the number of partitions exceeds the number of TLB entries, we suffer from a TLB miss for each tuple we move, in addition to cache misses and other stalling effects.

Buffered output data. Others have discovered that moving one tuple at a time is wasteful, because it increases the number of cache conflicts [Satish et al. 2010]. Instead, we can keep a small buffer per partition in a fixed location in memory and, only once the buffer is filled, copy the entire buffer to the memory location of the output partition. Since the buffers, one per output partition, are at a fixed memory address, the conflicts in the cache are minimized. Furthermore, because the buffers typically fit in the cache, they thus do not require many TLB entries, so TLB misses occur only when the buffer is "flushed" to memory and not for every tuple. Thus, the number of partitions (i.e., the partitioning fanout) can be increased.

Researchers further optimized this buffering scheme to take advantage of write-combining and nontemporal stores [Wassenberg and Sanders 2011]. The idea is that each buffer should be at a cache-line granularity to maximize the partition fanout and that wide (can use SIMD registers) nontemporal writes should be used to store the

result to the output to avoid polluting the cache with output data that are not going to be needed again any time soon.

Buffering allows us to shuffle data across more than 1,024 partitions close to the memory bandwidth rate, whereas moving one tuple at a time to the destination will reach 80% of the memory bandwidth with just 64 partitions.

In-place partitioning. The preceding approaches assume the use of auxiliary space to store the output entirely separate from the input. To explore the time-space trade-off, we can also partition in place. Prior work [Polychroniou and Ross 2014] explores a variety of cases for partitioning in place depending on whether we want contiguous segments or not and whether the threads can operate shared-nothing or not.

Here, we will evaluate shared-nothing partitioning to contiguous segments. The naive approach for shared-nothing in-place partitioning to contiguous segments moves one item at a time to its destination and swaps it with the item lying there. Then it moves the new item to its destination, swapping it with the item lying there and so on. The "cycle" closes when we reach the initial item. This algorithm works very well in-cache but suffers from the same problems as the non–in-place naive approach when the working set footprint exceeds the cache size. The solution proposed by Polychroniou and Ross [2014] adapts the buffering and write-combining techniques of Satish et al. [2010] and Wassenberg and Sanders [2011] to accelerate efficient in-place partitioning.

"In blocks" data layout. One more approach for in-place partitioning is to generate the result in blocks [Polychroniou and Ross 2014], where each block is a moderate number of tuples that belong to the same partition. The output is a list of such blocks in random partition order, but each block can be "linked" to all other blocks of the same partition. Many operations, such as joins, could access these lists of blocks amortizing the random access of jumping across blocks. The partitioning technique is almost identical to non–in-place partitioning. Finally, this technique does not require the precomputation of the histogram and thus should also be used for range partitioning, where, as we will see, the cost of computing the partition function is nontrivial.

3.2. Experimental Methodology

In the following section, we compare four data shuffling strategies for partitioned data that apply the techniques described in Section 3.1. All of them lay out data in contiguous arrays—that is, not using the in-blocks layout described previously:

—Nonbuffered, in place
—Buffered, in place
—Buffered, not in place
—Nonbuffered, not in place

Our experiments are run on a server that has four Intel Xeon E5-4620 eight-core CPUs at 2.2GHz based on the Sandy Bridge microarchitecture and a total of 512 GB quad-channel DDR3 ECC RAM at 1,333MHz. Each CPU supports 16 hardware threads through two-way simultaneous multithreading. Each core has a 32KB private L1 data cache and a 256KB L2 cache. Each CPU has an 8MB L3 cache shared across the eight cores. The bandwidth that we measured is 122GB/sec for reading, 60GB/sec for writing, and 37.3GB/sec for copying.

The operating system is Linux 3.2, and the compiler is GCC 4.8 with `-O3` optimization. We use SSE (128-bit SIMD) instructions on AVX registers (256 bit) to use nondestructive three-operand instructions (Intel VEX) that improve pipeline performance. We cannot fully use 256-bit SIMD because our platform (AVX 1) supports 256-bit

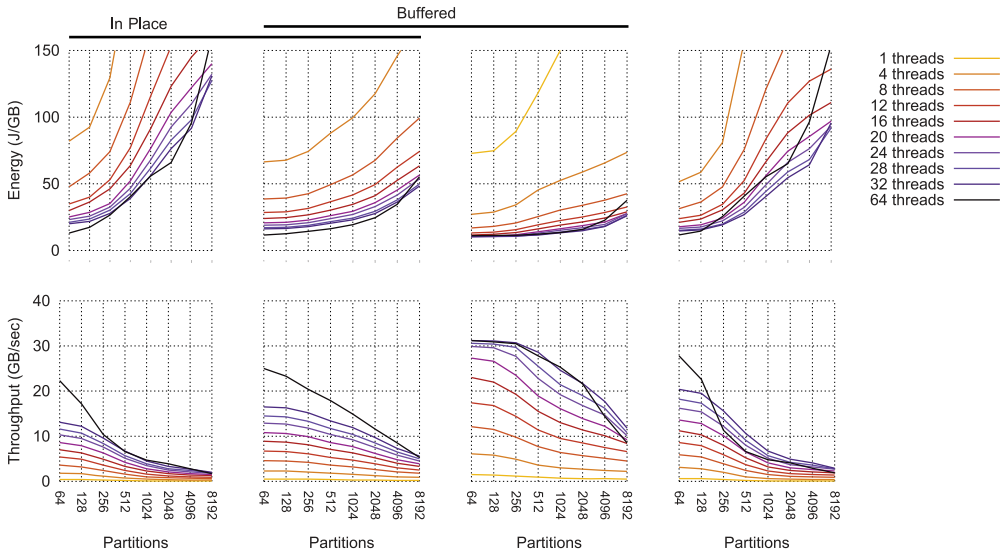

Fig. 4. Four combinations of data shuffling policy. Note that the 64 thread measurements use SMT, whereas the others do not. Cache-line buffering when writing tuples to output partitions boosts partitioning throughput and reduces partitioning energy. When possible, non-in-place partitioning, which requires linear auxiliar space, can also improve throughput and energy, although the effects are not as pronounced as the buffering technique.

operations for floating point operations only. The data are synthetically generated and follow the uniform random distribution.

The input is two separate arrays with 32-bit keys and 32-bit or 96-bit payloads. The buffering approach uses a buffer of two cache lines per partition for 32-bit payloads and four cache lines per partition for 96-bit payloads.

We measure energy using Intel's Running Average Power Limit (RAPL) interface [Intel Corporation 2013]. RAPL uses nonarchitectural, model-specific registers (MSRs) exposing counters that we sample periodically over the course of each experiment.

3.3. Data Shuffling Analysis

To isolate the effects of the data shuffling phase, we use radix partitioning. This is a simple mask operation and can be computed trivially at no cost. Figure 4 plots the throughput and energy of four data movement policies, covering each combination of in-place versus non–in-place, and buffered versus nonbuffered, as described in Section 3.1.

First, we observe that data shuffling parallelizes relatively well, with small thread counts with diminishing benefits as thread counts approach the machine core count.

We also note that energy rises and throughput drops when the number of partitions exceeds the L1 TLB size of the machine (64 entries), especially when using the naive approach that does not buffer in the cache. This is consistent with prior observations of throughput [Manegold et al. 2000].

Comparing buffered to nonbuffered implementations, the charts reveal that buffering is clearly the winner. It improves throughput across all thread counts an average of 1.7 and 2.7 times for in-place and non–in-place shuffling, respectively. With respect to energy, when the partitioning fanouts are small, the power efficiency is relatively equal. However, for larger fanouts, the relative power efficiency improvement matches the relative performance speedup.

Whether the input memory space is reused to write the partitioned outputs (in place) or not has negligible effect on both the partitioning energy and throughput when the data is not buffered (14% average increase in throughput, 7% average decrease in energy). However, when the data is buffered, the effect is more marked, with an average 60% increase in throughput and decrease in energy.

Overall, across all thread counts and partition counts, buffered, non–in-place data shuffling provides the most performance and energy efficiency of the four variants examined. In the following experiment, we will combine this data shuffling policy with nontrivial partition functions to examine the energy and performance characteristics of the full software partitioning operation.

Figure 4 presents measurements with varying number of cores operating at the system maximum of 2.2GHz frequency. Besides the measurement for a single thread, in all other cases we distribute threads evenly across the four CPUs. SMT is enabled only when we measure 64 threads. In all other cases, the threads are bound to a distinct CPU core to avoid resource sharing. We repeated the same experiment with all cores operating at the system minimum of 1.2GHz frequency and found that throughput dropped across the board, the energy impact was negligible, and the preceding observations regarding shuffling strategies all still hold.

3.4. Range Partitioning

Just as radix partitioning is trivial to compute, hash partitions are similarly fast provided the hash function is fast. One of the fastest hash functions is multiplicative hashing, which requires a multiplication and a shift. In contrast, range partitioning is much more expensive but offers several benefits as described in Section 2. The simplest approach is to perform a binary search of the splitter values. To determine the correct partition for a particular record, one traverses a binary tree comparing the key to a splitter value at each node in the tree. The comparisons for a key are sequentially dependent, and the path through the tree is unpredictable. The combination of these properties results, unavoidably, in pipeline bubbles.

Recent work shows that computing the range function becomes the bottleneck of range partitioning, even more so if the data are shuffled using the cache-buffering approach described in Section 3.1 [Polychroniou and Ross 2014]. Cache misses are not the problem here, as the array of splitters is L1 cache resident anyway. Instead, the logarithmic number (to the number of splitters) of cache loads tightly coupled with dependencies exposes their full latency and throttles performance.

A better approach uses a cache-resident range index to compute the range functions [Polychroniou and Ross 2014]. First, the index stores only keys (delimiters) and no pointers. It uses arithmetic to find the next node in the tree. Second, each tree node is especially tuned to be accessed with SIMD instructions. Thus, if the SIMD registers have W cells, the tree node is kW wide and has a $kW + 1$ fanout. Third, since the tree has a small number of levels, the levels are kept in registers and all node accesses can be unrolled across multiple input keys to overlap cache accesses. In more detail, we can have multiple keys accessing the root, then multiple keys accessing the first level, and so on. This relaxes the dependencies and allows parallel cache accesses.

The SIMD-based node-access code can use specific handwritten code to access each tree level. The first level may hold W keys per node, whereas the second level may hold $2W$ per node. Different tree configurations are built to satisfy different partitioning fanouts. For example, with four-wide SIMD, one can build a $8 \times 5 \times 5 = 200$-way range partitioning using two levels with $W = 4$ keys each (the starting $8\times$ refers to the root that is a special case with seven delimiters). Extending the second level to hold $2W = 8$ keys per node gives $8 \times 5 \times 9 = 360$-way partitioning. Adding an additional level gives $8 \times 5 \times 5 \times 5 = 1{,}000$-way partitioning and so on. Having access to wider SIMD, we can

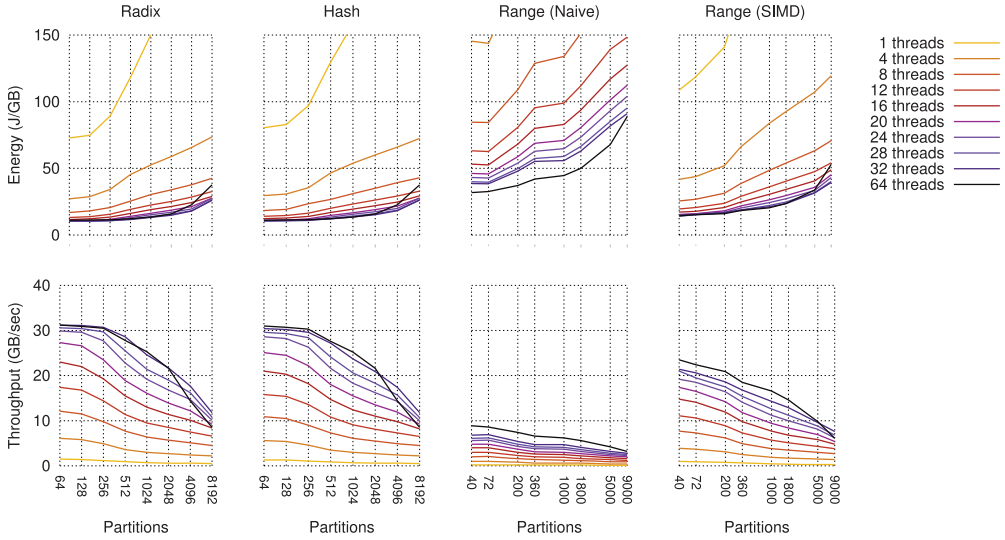

Fig. 5. Comparison of four partitioning techniques. Both range and hash partitioning are highly efficient at low fanouts. Naive range partitioning incurs a substantial overhead, which is largely mitigated by careful exploitation of SIMD instructions when traversing a specially designed tree of splitters.

increase the number of partitions with the same key levels. For example, a $8 \times 5 \times 5$-way tree on four-wide SIMD is as fast as a $8 \times 9 \times 9$-way tree in eight-wide SIMD.

The nonroot node accesses compare one input key with W delimiters from the node. The root access is different, instead comparing W input keys with one delimiter. The root holds 2^n-1 delimiters broadcast in all cells of 2^n-1 SIMD registers. The root access is a binary search in registers, predicating the two directions by using a SIMD blend to combine the following delimiters accordingly for each SIMD lane.

The improvement over binary search is five to six times for nontrivial fanouts, and computing the range functions is about twice as fast as shuffling the data using buffering. It thus decreases performance compared to hash and radix partitioning by less than 50%.

3.5. Full Partitioning Analysis

Figure 5 shows the throughput and energy of four software partitioners: radix, hash, naive range, and the state-of-the-art SIMD-based range partitioner [Polychroniou and Ross 2014]. We first observe that the naive range partitioner incurs a substantial performance and energy overhead relative to hash or range partitioning at small fanouts. SIMD-based range implementation closes the gap and the partition function part is now faster than data shuffling, but not as fast as radix or hash that takes trivial time. For example, at 64 threads, the SIMD-based 1,000-way partitioner runs at 15.9GB/sec and 20.7J/GB, relative to 25.3GB/sec and 13.5J/GB for hash or radix at 1,024 ways.

Figure 6 breaks down the 64-thread partitioning energy down by phase: partition function and data shuffling.[2] Unsurprisingly, the hash and radix partition functions consume relatively little energy, whereas the cost of data shuffling increases with the fanout. The jump in energy consumption for the naive range partitioner that we saw in Figure 5 is due to the cost of the partition function. The SIMD range index optimizes the partition function energy down to roughly one third of the total cost. As with hash

[2]Shuffle [J/GB] $= 0.003 \times \mathit{Fanout} + 9.227$ at 2.2GHz and $0.0028 \times \mathit{Fanout} + 7.7925$ at 1.2GHz.

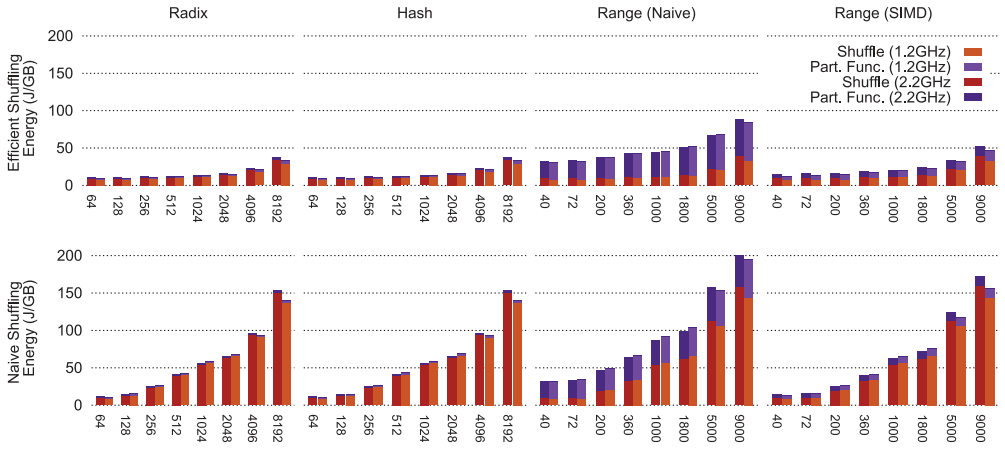

Fig. 6. Energy breakdown of 64 partitioning threads running at 2.2 and 1.2GHz. Hash and radix partitioning energy is dominated by data shuffling, although range partitioning is also affected by the partition function. Processor frequency effects total throughput; however, it has little effect on the energy total or breakdown.

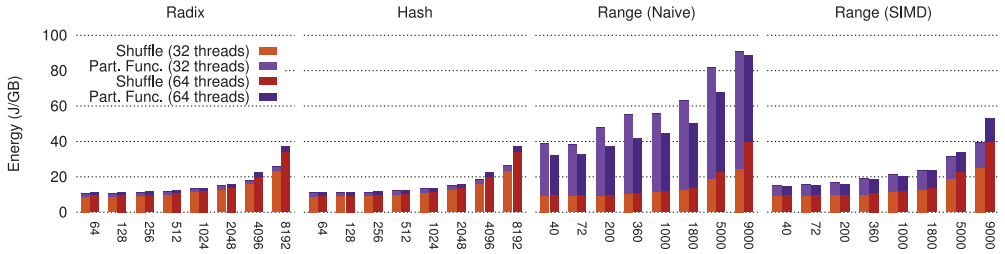

Fig. 7. Energy breakdown of 64 partitioning threads running at 2.2GHz. Data shuffling consumes more power with SMT; however, the range partition function computation is improved.

and radix partitioning, we found that reducing the frequency increased the runtime and reduced the power in rough proportion, creating little change in total energy.

An interesting trade-off is the use of SMT for partitioning. For data shuffling, using SMT halves the available cache per thread, since the threads operate in a shared-nothing fashion. Thus, when operating at the cache capacity (8,192-ways partitioning), the non-SMT approach is better. Radix and hash partitioning exhibit a small performance benefit from using SMT (see Figure 5). However, the non-SMT approach is slightly more power efficient, as shown in Figure 7, which was expected since radix and hash partitioning are memory bound. For range partitioning, SMT is helpful in determining the partition function as the operation is CPU bound. Data shuffling is still less power efficient using SMT, but the partition function is improved since it is CPU bound. These results mirror a key insight behind a recent index traversal accelerator [Kocberber et al. 2013], which balances the hashing and table walking components of that operation by dedicating different numbers of cores to each task.

4. HARDWARE ACCELERATED RANGE PARTITIONING SYSTEM

Here we describe the architecture and microarchitecture of a system that incorporates (HARP).

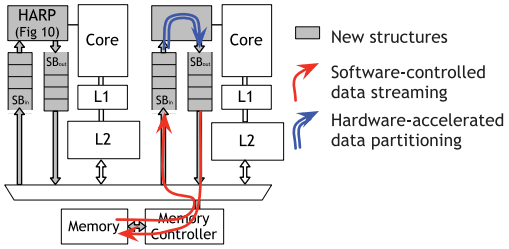

Fig. 8. Block diagram of a typical two-core system with HARP integration. New components (HARP and stream buffers) are shaded. HARP is described in Section 4.2 followed by the software-controlled streaming framework described in Section 4.3.

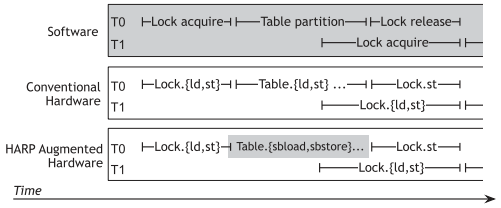

Fig. 9. With or without HARP, correct multithreaded operation relies on proper software-level locking. As illustrated here, the streaming framework works seamlessly with existing synchronization and data layouts.

4.1. Overview

Figure 8 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 8), whereas inbound and outbound data stream management is left to software (single arrows), thereby maximizing flexibility and simplifying the interface to the accelerator. One set of instructions provides configuration and control for the HARP accelerator, 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—streamed in from memory via the streaming framework, partitioned with HARP, and then streamed back out—the overall system throughput will be determined by the lowest-throughput component.

As Figure 9 illustrates, the existing software locking policies in the database provide mutual exclusion during partitioning both in conventional systems and with HARP. As in conventional systems, if software does not use proper synchronization, incorrect and nondeterministic results are possible. Figure 9 shows two threads contending for the same table T; once a thread acquires the lock, it proceeds with partitioning by executing either the conventional software partitioning algorithm on the CPU or streaming loads to feed the table to HARP for hardware partitioning. Existing database software can be ported to HARP with changes exclusively in the partitioning algorithm implementation. All other aspects of table layout and database management are unchanged.

4.2. HARP Accelerator

Instruction set architecture. The HARP accelerator is managed via the three instructions shown in the top of Table I. `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, seven `set_splitter` instructions are used to set values for each of the seven splitter values, followed by a `partition_start` to start HARP's partitioning. Since HARP's microarchitectural state is not visible to other parts of the machine, the splitter values are not lost on interruption.

Microarchitecture. HARP pulls and pushes records in 64-byte bursts (tuned to match system vector width and DRAM burst size). The HARP microarchitecture consists of three modules, as depicted in Figure 10, and is tailored to range partition data highly efficiently:

Table I. Instructions to Control the HARP and the Data Streaming Framework

HARP Instructions
`set_splitter <splitter number> <value>`
Set the value of a particular splitter (splitter number ranges from 0 to 126).
`partition_start`
Signal HARP to start partitioning reading bursts of records from SB_{in}.
`partition_stop`
Signal HARP to stop partitioning and drain all in-flight data to SB_{out}.
Stream Buffer Instructions
`sbload sbid, [mem addr]`
Load burst from memory starting from specified address into designated SB_{in}.
`sbstore [mem addr], sbid`
Store burst from designated SB_{out} to specified address.
`sbsave sbid`
Save the contents of designated stream buffer to memory (to be executed only after accelerators have been drained as described in Section 4.2).
`sbrestore sbid`
Restore contents of indicated stream buffer from memory.

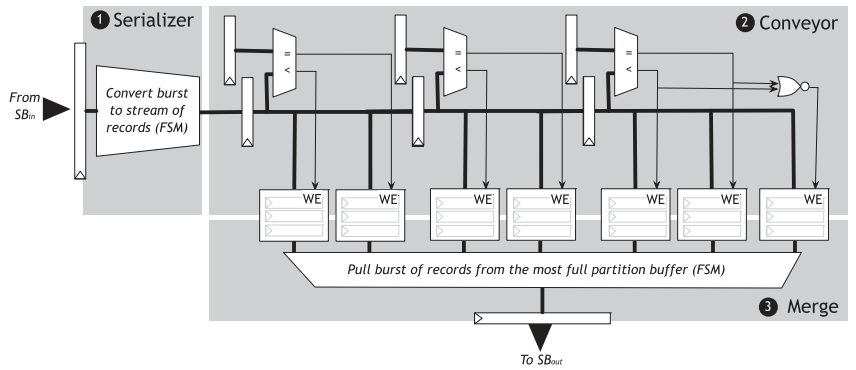

Fig. 10. HARP draws records in bursts, serializing them into a single stream that is fed into a pipeline of comparators. At each stage of the pipeline, the record key is compared with a splitter value, and the record is either filed in a partition buffer (downward) or advanced (to the right) according to the outcome of the comparison. 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.

(1) The *serializer* pulls bursts of records from SB_{in}, and uses a simple finite state machine to pull each individual 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.
(2) The *conveyor* is where the record keys are compared 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.
(3) The *merge* module monitors the partition buffers as records accumulate. It is looking 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, respectively. With large record counts, the difference in time required to fill and drain a k-stage pipeline versus a $log(k)$-stage pipe in the tree version is negligible. Although cycle counts were more or less the same between the two, the linear design had a slightly shorter clock period due to 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, timestamps 4 bytes, and datetimes as 8 bytes [MySQL 2014]. Partitioning ASCII strings alphabetically on the first N characters can also be accomplished with an N-byte integer comparator.

4.3. 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 above 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 [Jouppi 1990].

Instruction set architecture. Software moves data between memory and the stream buffers via the four instructions described at the bottom of Table I. `sbload` loads data from memory to SB_{in}, taking as arguments a source address in memory and a destination stream buffer ID. `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 (i.e., 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 software on the CPU knows how large a table is, it can know how many `sbloads`/`sbstores` 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 microarchitecture of HARP 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 contents of the stream buffers. 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.

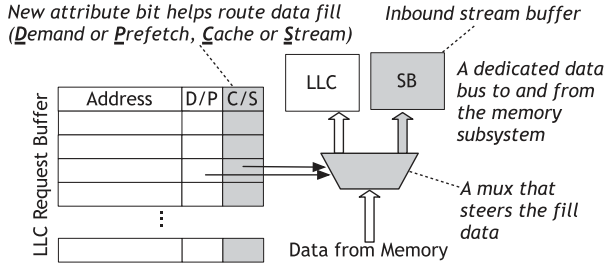

Fig. 11. Implementation of streaming instructions into existing data path of a generic last-level cache request/fill microarchitecture. Minimal modifications required are shaded.

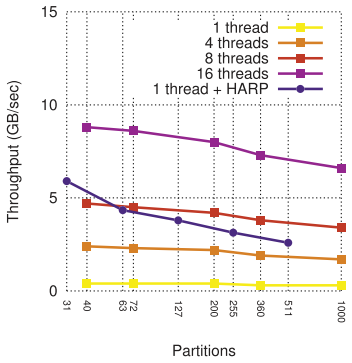

Fig. 12. A single HARP unit outperforms the single-threaded SIMD range partitioner by a factor of three to five, matching the throughput of between four and eight threads.

Microarchitecture. To implement the streaming instructions, we propose minimal modifications to conventional processor microarchitecture. Figure 11 summarizes the new additions. `sbload`'s borrow the existing microarchitectural vector load (e.g., Intel's SSE 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 one-bit attribute to the existing last-level cache request buffer to differentiate `sbload` requests from conventional vector load requests. This attribute acts as the mux select for the return data path, as illustrated in Figure 11. 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. `sbloads` 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 `sbloads` always produce the most up-to-date values. Figure 11 depicts the scenario when a request misses all levels of the cache hierarchy, and the fill is not cached, as `sbloads` are noncacheable. On the store side, `sbstores` can copy data from SB_{out} into the existing store buffer sharing the store data path 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 the following two ways: (1) fill the prefetched data into the cache hierarchy as current processors do or (2) 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 nontemporal. Because `sbloads` 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 shorter data return latency when the requests hit in the prefetched data in the cache.

5. EVALUATION METHODOLOGY

To evaluate the throughput, power, and area efficiency of our design, we implemented HARP in Bluespec System Verilog [Bluespec, Inc. 2012].

Baseline HARP parameters. Each of the design points extends a single baseline HARP configuration with 127 splitters for 255-way partitioning. The baseline supports

16-byte records, with 4-byte keys. Assuming 64-byte DRAM bursts, this works out to four records per burst.

Software range partitioning comparison. We compare HARP's energy and throughput against the state-of-the-art range partitioner presented in Section 3.4. The experimental setup is the same as in Section 3.2, except the record size is increased from the original 8B to 16B to match HARP.

HARP simulation. Using Bluesim, Bluespec's cycle-accurate simulator, we simulate HARP partitioning one million random records. We then convert cycle counts and cycle time into absolute bandwidth (in gigabytes per second).

HARP synthesis and physical design. We synthesized HARP using the Synopsys [Synopsys, Inc. 2013] Design Compiler followed by the Synopsys IC Compiler for physical design. We used Synopsys 32nm Generic Libraries; we chose *HVT* cells to minimize leakage power and normal operating conditions of 0.85V supply voltage at 25°C. The post-place-and-route critical path of each design is reported as logic delay plus clock network delay, adhering to the industry standard of reporting critical paths with a margin.[3] We gave the synthesis tools a target clock cycle of 5 or 2 ns depending on design size and requested medium effort for area optimization.

Xeon area and power comparisons. The per-processor core area and power figures in the analyses that follow are based on Intel's published information and reflect our estimates [Intel Corporation 2012].

Streaming instruction throughput. To estimate the rate at which the streaming instructions can move data into and out of HARP, we measure the rate at which memory can be copied from one location to another (i.e., streamed in and back out again). We benchmark three implementations of memcpy: (1) built-in C library, (2) hand-optimized X86 scalar assembly, and (3) hand-optimized X86 vector assembly. In each experiment, we copy a 1GB table natively on an Intel E5620 server.[4] All code was compiled using `gcc` 4.6.3 with -O3 optimization.

Streaming buffer area and power. We use CACTI [HP Labs 2013] to estimate the area and power of stream buffers. The number of entries in the stream buffers are conservatively estimated assuming that all ways of the partitioner can output in the same cycle. For example, for a 255-way partitioner, we sized SB_{out} to have 255 entries of 64 bytes each.

6. EVALUATION RESULTS

6.1. Area, Power, and Performance

We evaluate the proposed HARP system in the following categories:

(1) Throughput comparison with the software range partitioner from Section 3.4
(2) Area and power comparison with the processor core on which the software experiments were performed
(3) Nonperformance partitioner desiderata

For all evaluations in this section, we use the baseline configuration of HARP outlined in Section 5 unless otherwise noted.

HARP throughput. Figure 12 plots the throughput of three range partitioner implementations: single-threaded software, multithreaded software, and single-threaded

[3]Critical path of the 511-partition design, post-place-and-route, is obtained by scaling the synthesis output, using the Design Compiler to IC Compiler ratio across designs up to 255 partitions.
[4]2.4GHz, 12MB LLC, 32nm lithography with die area of 239 mm^2 [Intel Corporation 2012].

Table II. Area and Power Overheads of HARP Units and Stream Buffers for Various Partitioning Factors

	HARP Unit				Stream Buffers			
Number of	Area		Power		Area		Power	
Partitions	mm^2	% Xeon	W	% Xeon	mm^2	% Xeon	W	% 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%	1.3	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%

*Scaled conservatively from the baseline design using area and power trends seen later in Figures 17 and 18.

software plus HARP. We see that HARP's throughput exceeds a single software thread by three to five times, with the difference primarily attributable to the elimination of instruction fetch and control overhead of the splitter comparison and the deep pipeline. In particular, the structure of the partitioning operation does not introduce hazards or bubbles into the pipeline, allowing it to operate in near-perfect fashion: always full, 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 12 indicates, it requires four to eight threads for the software implementation to match the throughput of the hardware implementation. At 3.13GB/sec per core with HARP, augmenting just a handful of cores with HARP would provide sufficient compute bandwidth to fully utilize all DRAM pins.

As mentioned, Figure 12 uses 16-byte records. Using 8-byte records (as in previous sections) would allow software partitioning to be more efficient. Specifically, the data shuffling part would be up to twice more efficient, given that data shuffling is bound by the RAM bandwidth and for small to middle fanouts (up to 512-way) runs very close to the performance of RAM memcpy. The range partition function would be unaffected.

In terms of absolute numbers, the baseline HARP configuration achieved a 5.06ns critical path, yielding a design that runs at 198MHz, delivering partitioning throughput of 3.13GB/sec. This is three to five times faster than a single thread of the state-of-the-art SIMD range partitioner described in Section 3.4.[5]

Streaming throughput. Our results in Figure 13 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 [Subramoni et al. 2010]. Based on these measurements, we will conservatively estimate that the streaming framework can bring in data at 4.6GB/sec and write results to memory at 4.6GB/sec with a single thread. This 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 do increase the area and power of the core. Table II quantifies the area and power

[5]The ISCA '13 version of this paper [Wu et al. 2013] quoted HARP as 7.8 times faster than a single software thread. Whereas HARP's throughput has not changed, the software comparison point has been updated from a microbenchmark to the complete range partitioner to appear in SIGMOD '14 and described briefly in Section 3.4.

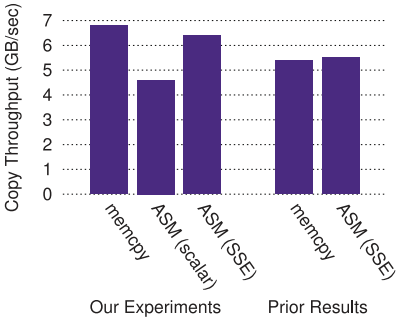

Fig. 13. The streaming framework shares much of its implementation with the existing memory system, and as such its throughput will be comparable to the copy throughput of existing systems.

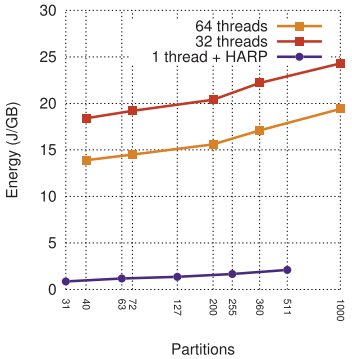

Fig. 14. HARP-augmented cores can partition data using roughly one order of magnitude less energy than 64 parallel threads, which is the most energy-efficient software configuration.

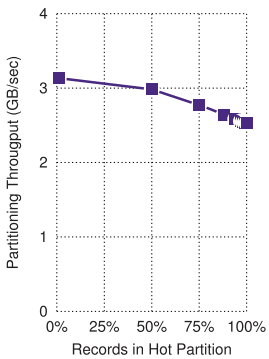

Fig. 15. As input imbalance increases, throughput drops by at most 19% due to more frequent back-to-back bursts to the same partition.

overheads of the accelerator and stream buffers relative to a single Xeon core. Comparatively, the additional structures are very small, with the baseline design point adding just 6.9% area and 4.3% power for both the HARP and the SBs. HARP itself consumes just 2.83mm^2 and 0.11W.

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 II. The proposed streaming framework adds 0.3mm^2 area and consumes 10mW 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 14 compares the partitioning energy per gigabyte of data of software (both serial and parallel) against HARP-based alternatives. The data show roughly an order of magnitude improvement in single-threaded partitioning energy with HARP.[6]

Order preservation. HARP is record order preserving by design. All records in a partition appear in the same order 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 structure of HARP, where there is only one route from input port to each partition, and it is impossible for records to pass one another in flight.

Skew tolerance. We evaluate HARP's skew tolerance by measuring the throughput (i.e., cycles/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 are sent to a single partition. Figure 15 shows the gentle degradation in throughput as one partition receives an increasingly large share of records.

[6]The ISCA '13 version of this article quoted 6.2 to 8.7 times improvement in energy efficiency. This reflects a change in software energy measurement methodology. Whereas before we were estimating core power based on TDP and relative core area on die, we now measure energy for the entire system using Intel's RAPL interface. Because RAPL provides relatively coarse measurement domains, the fairest energy comparison point for software is 64 threads, which fully utilizes all of the available hardware resources.

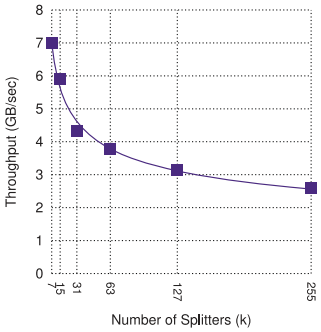

Fig. 16. HARP throughput is most sensitive to the number of partitions, dropping about 38% going from a 15-way to a 63-way partitioner.

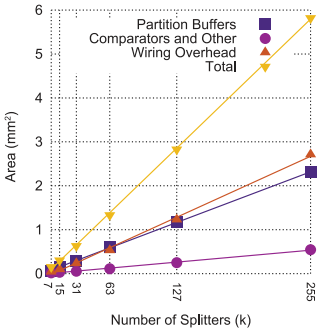

Fig. 17. HARP area scales linearly to the number of partitions because partition buffers dominate area growth and are scaled linearly with the number of partitions.

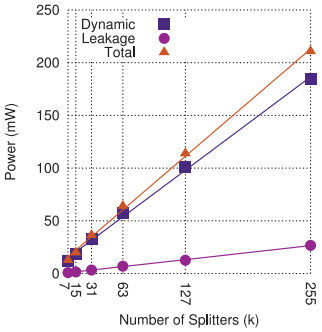

Fig. 18. HARP power consumption also scales linearly with the number of partitions on roughly the same linear scaling as area.

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 cycle in the *merge*, which rarely happens, when records are distributed across partitions. If there are B records per DRAM burst, draining two *different* partition buffers back-to-back takes $2B$ cycles. However, when skew increases, the frequency of back-to-back drains of the *same* partition increases, resulting in an average of $B + 1$ cycles per burst rather than B. Thus, the throughput of the *merge* module varies between $\frac{1}{B}$ cycles/record in the best case to $\frac{1}{B+1}$ in the worst case. Note that this tolerance is independent of many factors, including the number of splitters, the size of the keys, or the size of the table being partitioned.

The baseline HARP design supports four records per burst resulting in a 25% degradation in throughput between best- and worst-case skew. This is very close to the degradation seen experimentally in Figure 15, where throughput sinks from 3.13GB/sec with no skew to 2.53GB/sec in the worst case.

6.2. Design Space Exploration

The number of partitions, key width, and record width present different implementation choices for HARP, each suitable for different workloads. We perform a design space exploration and make the following key observations: (1) HARP's throughput is highly sensitive to the number of splitters when the partitioning factor is smaller than 63; (2) HARP's throughput scales linearly with record width; (3) the overall area and power of HARP grow linearly with the number of splitters; and (4) the smallest and the highest throughput design is not necessarily the best as the streaming framework becomes the system bottleneck, unable to keep HARP fed.

Next, we examine 11 different design points by holding two of the design parameters in Table III constant while varying the third. All reported throughputs are measured using a uniform random distribution of records to partitions. Figures 16 through 18 compare the throughput, area, and power as the number of partitions varies. Figures 18 through 21 show the same comparisons as number of key width and record width vary.

Table III. Parameters for HARP Design Space Exploration with Baseline Configuration Highlighted

HARP Design Space Configurations							
# Splitters	7	15	31	63	127	255	
# *Partitions*	*15*	*31*	*63*	*127*	*255*	*511*	
Key Width (Bytes)					4	8	16
Record Width (Bytes)			4	8	16		

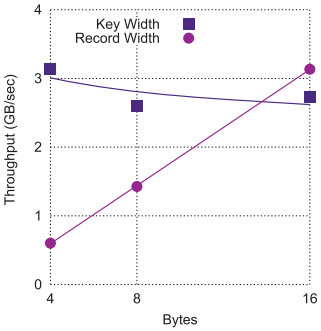

Fig. 19. HARP throughput increases linearly with record width (with keys fixed at 4B) because HARP partitions in record granularity. HARP throughput degrades mildly when key width increases (with records fixed at 16B).

Fig. 20. HARP area is not particularly sensitive to key or record widths. Wiring overhead and partition buffers dominate area at more than 80% of the total partitioner area.

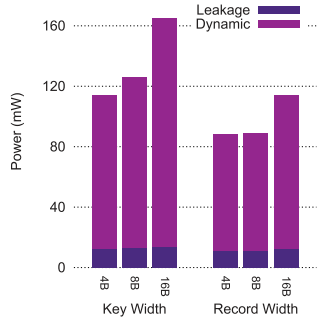

Fig. 21. HARP power consumption is slightly sensitive to key widths because the comparators are doubled in width when the key width doubles.

Throughput analysis. HARP's throughput degrades when the number of splitters or the key width increases. It is sensitive to the number of splitters as evidenced by the 38% drop in throughput from a 63-way to a 15-way partitioner. This is due to an increase in critical path as HARP performs more and wider key comparisons. As the record width increases, the throughput grows linearly, because the time and cycles per record are essentially constant regardless of record width.

Area and power analysis. The area and power of HARP scales linearly in the number of splitters but is otherwise mostly unaffected by key and record size. This is because the partition buffers account for roughly half of the total design area, and they grow linearly with the number of partitions.

Design trade-offs. In these studies, we see that a HARP design supporting a small number of partitions provides the fastest throughput, smallest area, and lowest power consumption. However, it results in larger partitions, making it less likely that the partitioned tables will display the desired improvement in locality. In contrast, a 511-way partitioner will produce smaller partitions but is slightly slower and consumes more area and power. Depending on the workload and the data size to be partitioned, one can make design trade-offs among the parameters that we have explored and choose a design that provides high throughput, low area, and high energy efficiency while maintaining overall system balance.

7. RELATED WORK

Streaming computation. The past decade has seen substantial interest in software-based streaming computation, starting with hardware architectures [Kapasi et al. 2003] and growing to include new parallel languages [Chakraborty and Thiele 2005; Gordon et al. 2006] and middleware support focused on portability and interoperability [Cooper and Schwan 2005; Jain et al. 2006; Neumeyer et al. 2010; Duller et al. 2011].

The hardware support for streaming has been substantially more limited. The vast majority of streaming architectures, such as Cell's SPE [Flachs et al. 2005], RSVP [Ciricescu et al. 2003], or Piperench [Goldstein et al. 1999] are decoupled from the processing core and are highly tailored to media processing. The designs that most closely resemble HARP microarchitecturally are DySER [Govindaraju et al. 2011] and ReMAP [Watkins and Albonesi 2010]. DySER incorporates a dynamically specialized data path into the core. Both DySER and HARP can be viewed as specialized functional units and are sized accordingly (a couple percent of a core area). Although one might be able to program DySER to partition data, its full interconnect between functional units is overkill for partitioning's predictable dataflow. ReMAP [Watkins and Albonesi 2010] has a very different goal, integrating reconfigurable fabric, referred to as specialized programmable logic (SPL), to support fine-grained intercore communication and computation.

Vector ISAs. Nearly all modern processors include vector ISAs, exemplified by X86's MMX and SSE, Visual Instruction Set (VIS) for UltraSPARC, or AltiVec on PowerPC. These ISAs include vector loads and stores, instructions that load 128- or 256-bit datawords into registers for SIMD vector operation. Different opcodes allow the programmer to specify whether the data should or should not be cached (e.g., nontemporal loads).

The SIMD vector extensions outlined previously were universally introduced to target media applications on streaming video and image data. The available operations treat the data as vectors and focus largely on arithmetic and shuffling operations on the vector values. Many programmers have retrofitted and vectorized other types of programs, notably text parsing [Cameron and Lin 2009; Lin et al. 2012] and regular expression matching [Salapura et al. 2012] and database kernels [Zhou and Ross 2002; Govindaraju and Manocha 2005; Krueger et al. 2011]. Our experiments in Section 3 using a state-of-the-art SIMD range partitioning [Polychroniou and Ross 2014] indicate that vector-based traversal improves throughput and energy markedly.

These efforts demonstrate moderate speedups, although at the cost of substantial programmer effort. One recent study of regular expression matching compared different strategies for acceleration [Salapura et al. 2012]. The study concluded that SIMD software was the best option due to the fast data and control transfers between the scalar CPU and the vector unit. The other approaches (including memory bus and network attached accelerators) suffered due to communication overheads. In short, SIMD won not because it was particularly fast computationally, but because it was fast to invoke. This study in part influenced our choice to tightly couple the HARP accelerator with a processing core.

Database machines. Database machines were developed by the database community in the early 1980s as specialized hardware for database workloads. These efforts largely failed, primarily because commodity CPUs were improving so rapidly at the time, and hardware design was slow and expensive [Boral and DeWitt 1983]. Although hardware design remains quite costly, high computing requirements of data-intensive workloads, limited single-threaded performance gains, increases in specialized hardware, aggressive efficiency targets, and the data deluge have spurred us and others to revisit this approach. And although FPGAs have been successfully used to

accelerate a number of data-intensive algorithms [Mohan 2011; Woods et al. 2010; Müller and Teubner 2010], they are power hungry compared to custom logic, and it remains unclear how to approach programming and integrating them.

Memory scheduling. Despite the relative scarcity of memory bandwidth, there is ample evidence both in this article and elsewhere that workloads do not fully utilize the available resource. One recent study suggests that if memory controllers were to operate at their peak throughput, data bus utilization would double, LLC miss penalties would halve, and overall performance would increase by 75% [Ipek et al. 2008]. This observation and others about the performance criticality of memory controller throughput Natarajan et al. [2004] have inspired substantial research in memory scheduling (e.g., Rixner [2004], Rafique et al. [2007], Shao and Davis [2007], Ipek et al. [2008], and Ebrahimi et al. [2011]). Improvements in memory controllers have the advantage of being applicable across all workloads, yet important throughput bound workloads, such as partitioning, are not limited by the memory controller and thus will not see significant benefit from those efforts.

8. CONCLUSIONS

We have studied partitioning, a very common operator for main-memory analytical database processing, from both the software and the hardware design perspective, with regard to performance and power efficiency.

On the software side, we describe the bottlenecks of data shuffling during partitioning due to TLB and cache effects and evaluate known solutions that allows us to reach the memory bandwidth for small to medium partitioning factors. Range partitioning in software is also bound by the cost of computing the range function, which is optimized using a SIMD-based range index. The results indicate that a careful implementation of software partitioning is vital to achieve good performance and power efficiency.

On the hardware side, we presented the design and implementation of HARP. HARP is able to provide a compute bandwidth of three to five times a very efficient software algorithm running on an aggressive Xeon core, with just 6.9% of the area and 4.3% of the power. Processing data with accelerators such as HARP can alleviate serial performance bottlenecks in the application and can free up resources on the server to do other useful work.

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. This proposed framework can be utilized for other database processing accelerators such as specialized aggregators, joiners, sorters, and so on, setting forth a flexible yet modular data-centric acceleration framework.

REFERENCES

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*. 266–277.

Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the International Conference on Management of Data*. 37–48.

Bluespec, Inc. 2012. Bluespec Core Technology. Retrieved July 29, 2014, from http://www.bluespec.com.

Haran Boral and David J. DeWitt. 1983. Database machines: An idea whose time has passed? In *Proceedings of the International Workshop on Database Machines*.

Robert D. Cameron and Dan Lin. 2009. Architectural support for SWAR text processing with parallel bit streams: The inductive doubling principle. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 337–348.

Centrum Wiskunde and Informatica. 2012. An Open-Source Database System. Retrieved July 29, 2014, from http://www.monetdb.org.

Samarjit Chakraborty and Lothar Thiele. 2005. A new task model for streaming applications and its schedulability analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 486–491.

Damianos Chatziantoniou and Kenneth A. Ross. 2007. Partitioned optimization of complex queries. *Information Systems* 32, 2, 248–282.

John Cieslewicz and Kenneth A. Ross. 2008. Data partitioning on chip multiprocessors. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*. 25–34.

Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. 2003. The reconfigurable streaming vector processor (RSVPTM). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 141.

Brian F. Cooper and Karsten Schwan. 2005. Distributed stream management using utility-driven self-adaptive middleware. In *Proceedings of the 2nd International Conference on Automatic Computing*. 3–14.

Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. 2011. MemScale: Active low-power modes for main memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 225–238.

Michael Duller, Jan S. Rellermeyer, Gustavo Alonso, and Nesime Tatbul. 2011. Virtualizing stream processing. In *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*. 269–288.

Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A Joao, Onur Mutlu, and Yale N Patt. 2011. Parallel application memory scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 362–373.

Brian Flachs, Shigehiro Asano, Sang H. Dhong, Peter Hotstee, Gilles Gervais, Roy Kim, Tien Le, Peichun Liu, Jens Leenstra, John Liberty, Brad Michael, Hwa-Joon Oh, Silvia M. Mueller, Osamu Takahashi, A. Hatakeyama, Yukio Watanabe, and Naoka Yano. 2005. A streaming processing unit for a CELL processor. In *Proceedings of the International Solid-State Circuits Conference*. 134–135.

Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. 1999. PipeRench: A co/processor for streaming multimedia acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture*. 28–39.

Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

Naga K. Govindaraju and Dinesh Manocha. 2005. Efficient relational database management using graphics processors. In *Proceedings of the 1st International Workshop on Data Management on New Hardware*. Article No. 1.

Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*. 503–514.

Goetz Graefe and Per-Ake Larson. 2001. B-Tree indexes and CPU caches. In *Proceedings of the 17th International Conference on Data Engineering*. 349–358.

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward dark silicon in servers. *IEEE Micro* 31, 4, 6–15.

HP Labs. 2013. Retrieved July 29, 2014, from http://www.hpl.hp.com/research/cacti/.

IBM. 2006. DB2 Partitioning Features. Retrieved July 29, 2014, from http://www.ibm.com/developerworks/data/library/techarticle/dm-0608mcinerney.

Intel Corporation. 2012. Intel®Xeon®Processor E5620 (12M Cache, 2.40 GHz, 5.86 GT/s Intel®QPI). (2012). Retrieved July 29, 2014, from http://ark.intel.com/products/47925/intel-xeon-processor-e5620-12m-cache-2_40-ghz-5_86-gts-intel-qpi.

Intel Corporation. 2013. Intel 64®and IA-32 Architectures Software Developer's Manual. (2013). Retrieved July 29, 2014, from http://download.intel.com/products/processor/manual/253669.pdf.

Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th International Symposium on Computer Architecture*. 39–50.

Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. 2006. Design, implementation, and evaluation of the linear road benchmark on the

stream processing core. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 431–442.

Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*. 364–373.

Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. 2003. Programmable stream processors. *IEEE Computer* 36, 8, 54–62.

Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2, 1378–1389.

Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 468–479.

Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. 2010. Server engineering insights for large-scale online services. *IEEE Micro* 30, 4, 8–19.

Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast updates on read-optimized databases using multi-core CPUs. *Proceedings of the VLDB Endowment* 5, 1, 61–72.

Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Rob Cameron. 2012. Parabix: Boosting the efficiency of text processing on commodity processors. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture*. 1–12.

Krishna T. Malladi, Benjamin C. Lee, Frank A. Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. 2012. Towards energy-proportional datacenter memory with mobile DRAM. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*. 37–48.

Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. What happens during a join? Dissecting CPU and memory optimization effects. In *Proceedings of the 26th International Conference on Very Large Data Bases*. 339–350.

Microsoft. 2012. Microsoft SQL Server 2012. Retrieved July 30, 2014, from http://technet.microsoft.com/en-us/sqlserver/ff898410.

Mohan C. Mohan. 2011. Impact of recent hardware and software trends on high performance transaction processing and analytics. In *Proceedings of the 2nd TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems*. 85–92.

René Müller and Jens Teubner. 2010. FPGAs: A new point in the database design space. In *Proceedings of the 13th International Conference on Extending Database Technology*. 721–723.

MySQL. 2014. Date and time datatype representation. Retrieved July 30, 2014, from http://dev.mysql.com/doc/internals/en/date-and-time-data-type-representation.html.

Chitra Natarajan, Bruce Christenson, and Fayé Briggs. 2004. A study of performance impact of memory controller features in multi-processor server environment. In *Proceedings of the 3rd Workshop on Memory Performance Issues*. 80–87.

Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed stream computing platform. In *Proceedings of the IEEE International Conference on Data Mining Workshops*. 170–177.

Oracle. 2013. Oracle Database 11g: Partitioning. Retrieved July 30, 2014, from http://www.oracle.com/technetwork/database/options/partitioning/index.html.

Orestis Polychroniou and Kenneth A. Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 755–766.

Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. 2007. Effective management of DRAM bandwidth in multicore processors. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*. 245–258.

Scott Rixner. 2004. Memory controller optimizations for Web servers. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. 355–366.

Paul Saab. 2008. Scaling Memcached at Facebook. Retrieved July 30, 2014, from https://www.facebook.com/note.php?note_id=39391378919.

Valentina Salapura, Tejas Karkhanis, Priya Nagpurkar, and Jose Moreira. 2012. Accelerating business analytics applications. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*. 1–10.

Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 351–362.

Jun Shao and Brian T. Davis. 2007. A burst scheduling access reordering mechanism. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*. 285–294.

Hari Subramoni, Fabrizio Petrini, Virat Agarwal, and Davide Pasetto. 2010. Intra-socket and inter-socket communication in multi-core systems. *IEEE Computer Architecture Letters* 9, 1, 13–16.

Synopsys, Inc. 2013. 32/28nm Generic Library for IC Design, Design Compiler, IC Compiler. Available at http://www.synopsys.com.

L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. 2011. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the International Symposium on Computer Architecture*.

Transaction Processing Performance Council. 2014. TPC-H. Retrieved July 30, 2014, from http://www.tpc.org/tpch/default.asp.

Jan Wassenberg and Peter Sanders. 2011. Engineering a multi-core radix sort. In *Proceedings of the 17th International Conference on Parallel Processing, Volume Part II*. 169–169.

Matthew A. Watkins and David H. Albonesi. 2010. ReMAP: A reconfigurable heterogeneous multicore architecture. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 497–508.

Louis Woods, Jens Teubner, and Gustavo Alonso. 2010. Complex event detection at wire speed with FPGAs. *Proceedings of the VLDB Endowment* 3, 1, 660–669.

Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2013. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 249–260.

Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. 2011. Scalable aggregation on multicore processors. In *Proceedings of the 7th International Workshop on Data Management on New Hardware*. 1–9.

Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 145–156.

Received June 2014; accepted June 2014