# Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning

Lisa Wu lisa@cs.columbia.edu
Raymond J. Barker rjb2150@columbia.edu
Martha A. Kim martha@cs.columbia.edu
Kenneth A. Ross kar@cs.columbia.edu

Department of Computer Science
Columbia University
New York, New York

## ABSTRACT

The global pool of data is growing at 2.5 quintillion bytes per day, with 90% of it produced in the last two years alone [24]. There is no doubt the era of big data has arrived. This paper explores targeted deployment of hardware accelerators to improve the throughput and energy efficiency of large-scale data processing. In particular, data partitioning is a critical operation for manipulating large data sets. It is often the limiting factor in database performance and represents a significant fraction of the overall runtime of large data queries.

To accelerate partitioning, this paper describes a hardware accelerator for range partitioning, or HARP, and a hardware-software data streaming framework. The streaming framework offers a seamless execution environment for streaming accelerators such as HARP. Together, HARP and the streaming framework provide an order of magnitude improvement in partitioning performance and energy. A detailed analysis of a $32nm$ physical design shows 7.8 times the throughput of a highly optimized and optimistic software implementation, while consuming just 6.9% of the area and 4.3% of the power of a single 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

## Keywords

Accelerator, Specialized functional unit, Streaming data, Microarchitecture, Data partitioning

This research was supported by grants from the National Science Foundation (CCF-1065338 and IIS-0915956).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ISCA* '13 Tel Aviv, Israel
Copyright 2013 ACM 978-1-4503-2079-5/13/06 ...$15.00.

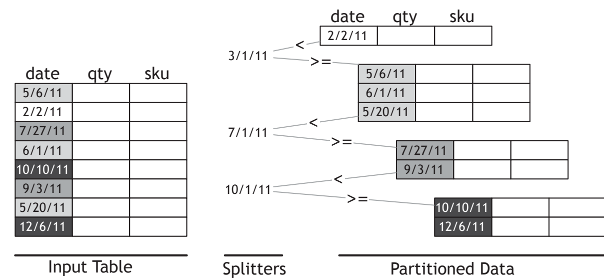



**Figure 1: An example table of sales records *range partitioned* by date, into smaller tables. Processing big data one partition at a time makes working sets cache-resident, dramatically improving the overall analysis speed.**

## 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, multi-dimensional data sets. These communities rely on computer systems to process vast volumes of data quickly and efficiently. In this paper we deploy specialized hardware 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 Figure 1. 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 [40], IBM DB2 [23], and Microsoft SQL Server 2012 [34] 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 [1, 20], with many databases now either fitting into main memory or having a memory-resident

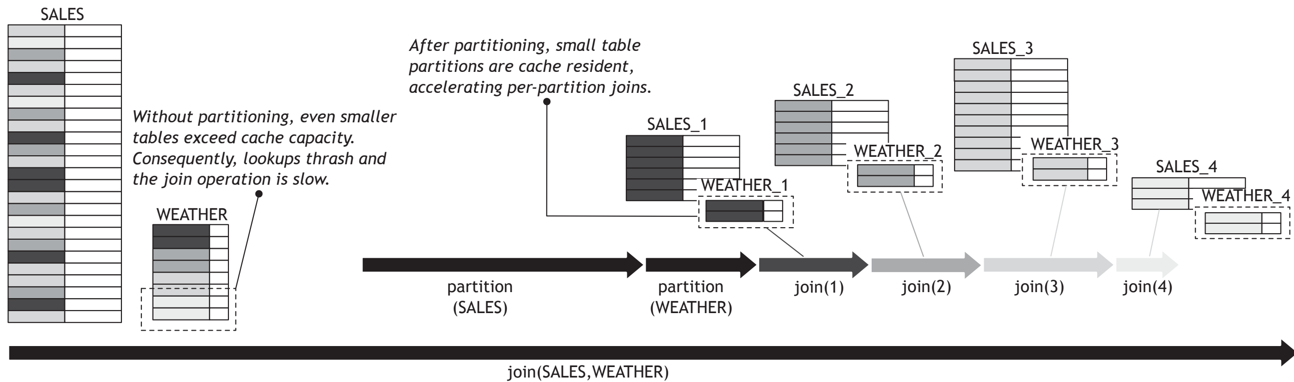



**Figure 2: Joining two large tables easily exceeds cache capacity. Thus, state of the art join implementations partition tables first and then compute partition-wise joins, each of which exhibits substantially improved cache locality [29, 2]. Joins are extremely expensive on large datasets, and partitioning represents up to half of the observed join time [29].**

working set. At Facebook, 800 servers supply over 28 $TB$ of in-memory data to users [44]. 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 $67-97\%$ processor utilization but only $2-6\%$ memory bandwidth utilization under stress testing [30]. Google's BigTable and Content Analyzer (large data storage and semantic analysis, respectively) show fewer than 10 $K/msec$ last level cache misses, which represents just a couple of percent of the total available memory bandwidth [50].

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 [33, 12] or to adjust the resource allocation in server microarchitectures [21]. We propose to resolve the imbalance by deploying specialized hardware to alleviate compute bottlenecks and more fully exploit the available pin bandwidth. In this paper we demonstrate that software implementations of data partitioning have fundamental performance limitations that make it compute-bound, even after parallelization. We then 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, or 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 3.13 $GB/sec$ of input, 7.8 times faster than a single software thread and matching the throughput of 16 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 0.3 $mm^2$ area, consumes 10 $mW$ power, and provides a minimum of 4.6 $GB/sec$ 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. BACKGROUND AND MOTIVATION

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

### 2.1 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 cache, partitioning is used to improve the performance of many critical database operations, such as joins, aggregations, and sorts [54, 2, 29]. Partitioning is also used in databases for index building, load balancing, and complex query processing [8]. 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 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

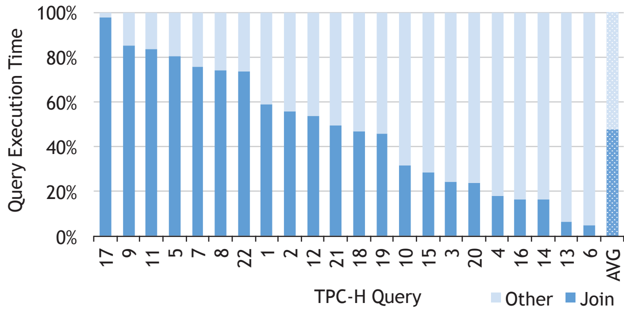



**Figure 3: Several key database operations such as join, sort, and aggregation use partitioning to improve their performance. Here we see joins consuming 47% of the TPC-H execution time on MonetDB. With state of the art join algorithms spending roughly half of the join time partitioning [29], we estimate that partitioning for joins alone accounts for roughly one quarter of query execution time.**

```
NumRecs ← 10^8                                  ▷ Alloc. and init. input
in ← malloc(NumRecs · RecSize)
for r = 0..(NumRecs − 1) do
    in[r] ← RandomRec()
end for
for p = 0..(NumParts − 1) do                    ▷ Alloc. output
    out[p] ← malloc(NumRecs · RecSize)
end for
for i = 0..NumRecs do                           ▷ Partitioning inner loop
    r ← in[i]
    p ← PartitionFunction(r)
    ∗(out[p]) ← r
    out[p] ← out[p] + RecSize
end for
```

**Figure 4: After initializing an input table and pre-allocating space for the output tables, the partitioning microbenchmark iterates over the input records, computes the output partition using** $PartitionFunction()$**, and writes it to that partition.**

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.

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 [51] query execution time attributable to joins using MonetDB [6], 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 ten runs of each

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

```
inline unsigned int
PartitionFunction(register parttype key) {
  register unsigned int low = 0;
  register unsigned int hi = N+1;
  register unsigned int mid = hi >> 1;
  for( int i = 0; i < D; i++ ) {
    __asm__ volatile("CMP %4, %2\n"
                     "CMOVG %3, %0\n"
                     "CMOVL %3, %1\n"
                     : "=a"(low), "=b"(hi)
                     : "r"(key), "r"(mid), "r"(R[mid]),
                       "a"(low), "b"(hi)
                     );
      mid = (hi + low) >> 1;
  }
  return (mid << 1) - (key == R[mid]);
}
```

**Figure 5: The implementation of** $PartitionFunction()$ **for** ***range*** **partitioning. For each record, the range partitioner traverses an array of `N` splitters. This optimized code performs a binary search up to `D = log2(N)` levels deep.**

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 [29], 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 amongst 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.

## 2.2 Software Partitioning Evaluation

We now characterize the performance and limitations of software partitioning on general purpose CPUs. Since partitioning scales with additional cores [9, 29, 2], we analyze both single- and multi-threaded performance.

For these characterizations, we use a microbenchmark whose pseudocode is shown in Figure 4. First, it initializes an input table with a hundred million random records. While actual partitioning implementations would allocate output space on demand *during* partitioning, we 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 compare three partitioning methods which are differentiated by the implementations of the partition function:

- **Hash**: A multiplicative hash of each record's key determines its destination partition.
- **Direct**: Like hash partitioning, but eliminates hashing cost by treating the key itself as the hash value.
- **Range**: Equality range partitioning using the state of the art implementation [43], which performs a binary

| System Configuration | |
|---|---|
| Chip | 2X Intel E5620<br>4C/8T, 2.4 $GHz$, 12 $MB$ LLC |
| Memory | 24 $GB$ per chip, 3 Channels, DDR3 |
| Max Memory BW | 25.6 $GB/sec$ per chip |
| Max TDP | 80 $Watts$ per chip |
| Lithography | 32 $nm$ |
| Die area | 239 $mm^2$ per chip |

**Table 1: Hardware platform used in software partitioning and streaming experiments (Sections 2.2 and 3.3 respectively). Source: Intel [25].**

search of the splitters. We show the exact code in Figure 5 as this is the software against which we will evaluate HARP.

The software partitioners were compiled with `gcc` 4.4.3 with -O3 optimization and executed on the hardware platform described in Table 1. Each reported result is the median of 10 runs, partitioning $10^8$ records per run. We experimented with 8 byte records as in [29] and 16 byte records as in prior work [9, 2], but show the latter results here as they provide higher throughput and are most directly comparable to HARP. These software measurements are optimistic. The input keys are evenly distributed across partitions, while this is not typically the case in real-world data. Moreover, the microbenchmark pre-allocates exactly the right amount of memory and performs no bounds checking during partitioning, whereas, in the real world, it is impossible to know exactly how many records will land in each partition, making it impossible to pre-allocate perfectly.[2]

Figure 6 shows the throughput of the *hash*, *direct*, and *range* partitioners for 128-way and 256-way partitioning (i.e., 128 and 256 output partitions). Examining single-threaded performance, we see that the hash function computation incurs negligible cost relative to the direct method. Our per-record hash partitioning times match prior studies [29], as does the drop in throughput between 128- and 256-way single-threaded partitioning which is consistent with earlier observations that 128-way partitioning is the largest partitioning factor that does not incur excessive L1 TLB thrashing.

Range partitioning's throughput is lower than direct or hash partitioning because it must traverse the splitter array to determine the destination partition for each record, despite the heavily optimized implementation shown in Figure 5. It is possible to improve the traversal even further by using SIMD instructions as described by Schlegel et al. [46] and we found that a SIMD-enhanced binary search improves the throughput of range partitioning up to 40%. However, the overall throughputs, 0.29 $GB/sec$ without SIMD, and 0.4 $GB/sec$ with, represent a tiny fraction of the 25.6 $GB/sec$ maximum throughput potential of the machine. There are inherent bottlenecks in software range partitioning. In particular, to determine the correct partition for a particular record, the best-known software algorithm, used here, 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.

[2]To pre-allocate partitions, Kim et al. [29] make an additional pass through the input to calculate partition sizes so that partitions are free of fragmentation, arguing that since the partitioning process is compute-bound, the extra pass through the data has only a small performance impact. An alternate approach is simply to allocate large chunks of memory on demand as the partitioning operation runs.

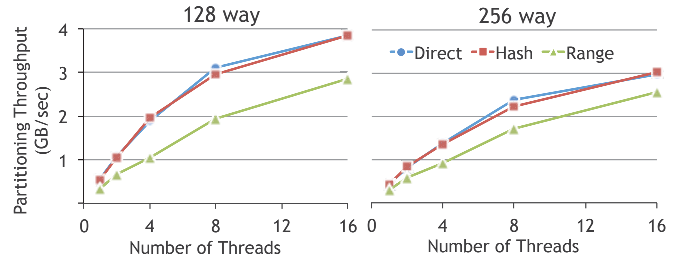



**Figure 6: Range partitioning is the most costly for both 128- and 256-way partitioning. As parallel threads are added, throughput improves.**

Because partitioning scales with multiple threads, we also consider the performance of multithreaded software implementations. As the data in Figure 6 indicate, 16 threads improve range partitioning throughput by 8.5$X$ peaking at 2.9 and 2.6 $GB/sec$ for 128- and 256-way partitioning respectively. Even after deploying all compute resources in the server, partitioning remains compute-bound, severely underutilizing the available memory bandwidth. In contrast, we will demonstrate that a single HARP-accelerated thread is able to achieve the throughput of close to 16 software threads, but at a fraction of the power.

# 3. HARDWARE ACCELERATED RANGE PARTITIONING SYSTEM

As we saw in Section 2, 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 a hardware accelerated range partitioner or HARP.

## 3.1 Overview

Figure 7 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 7), while 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 throughput of this system will be determined by the lowest-throughput component.

As Figure 8 illustrates, the existing software locking policies in the database provide mutual exclusion during partitioning both in conventional systems and with HARP. As in

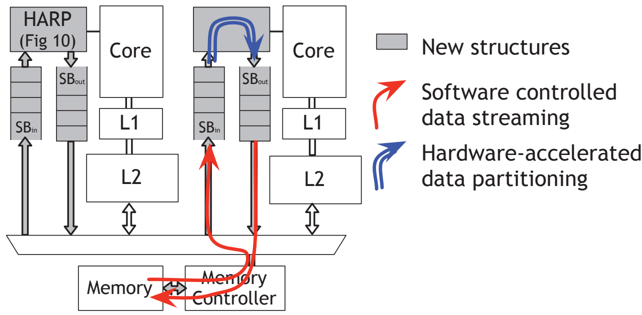



**Figure 7: Block diagram of a typical 2-core system with HARP integration. New components (HARP and stream buffers) are shaded. HARP is described in Section 3.2 followed by the software controlled streaming framework described in Section 3.3.**

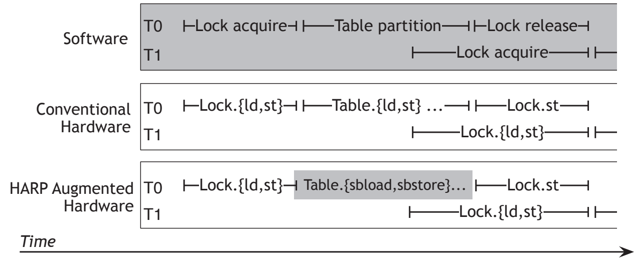



**Figure 8: With or without HARP, correct multi-threaded operation relies on proper software-level locking. As illustrated here, the streaming framework works seamlessly with existing synchronization and data layouts.**

conventional systems, if software does not use proper synchronization, incorrect and nondeterministic results are possible. Figure 8 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.

## 3.2 HARP Accelerator

**Instruction Set Architecture** The HARP accelerator is managed via the three instructions shown in the top of Table 2. `set_splitter` is invoked once per splitter to delineate a boundary between partitions; `partition_start` signals HARP to start pulling data from the $SB_{in}$; `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, 7 `set_splitter` instructions are used to set values for each of the 7 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 upon 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 9 and is tailored to range partition data highly efficiently.

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 Instructions** |
|---|
| `set_splitter <splitter number> <value>`<br>Set the value of a particular splitter (splitter number ranges from 0 to 126). |
| `partition_start`<br>Signal HARP to start partitioning reading bursts of records from $SB_{in}$. |
| `partition_stop`<br>Signal HARP to stop partitioning and drain all in-flight data to $SB_{out}$. |
| **Stream Buffer Instructions** |
| `sbload sbid, [mem addr]`<br>Load burst from memory starting from specified address into designated $SB_{in}$. |
| `sbstore [mem addr], sbid`<br>Store burst from designated $SB_{out}$ to specified address. |
| `sbsave sbid`<br>Save the contents of designated stream buffer to memory. (To be executed only after accelerators have been drained as described in Section 3.2). |
| `sbrestore sbid`<br>Restore contents of indicated stream buffer from memory. |

**Table 2: Instructions to control the Hardware Accelerated Range Partitioner (HARP) and the data streaming framework.**

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

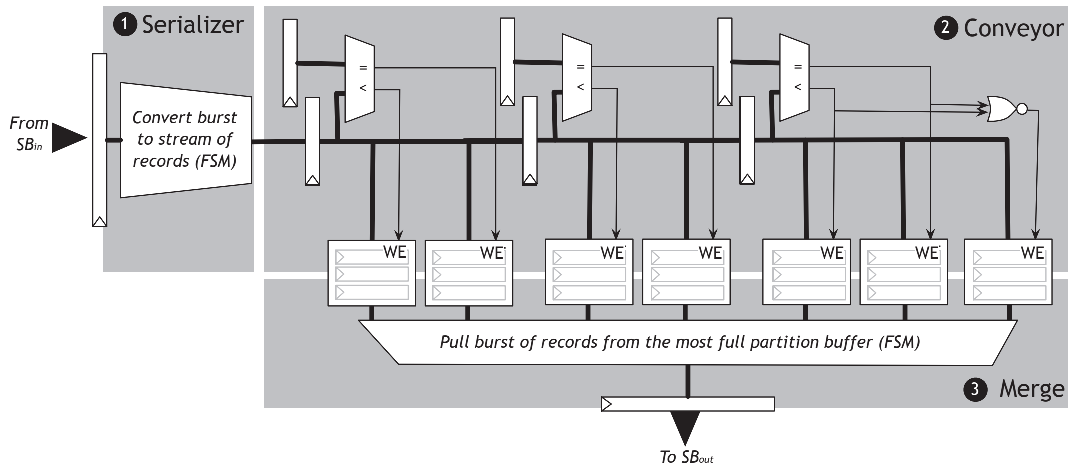



**Figure 9: HARP draws records in bursts, serializing them into a single stream which is fed into a pipeline of comparators. At each stage of the pipeline, the record key is compared with a splitter value, and the record is either filed in a partition buffer (downwards) or advanced (to the right) according 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.**

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. While 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 [37]. Partitioning ASCII strings alphabetically on the first N characters can also be accomplished with an N-byte integer comparator.

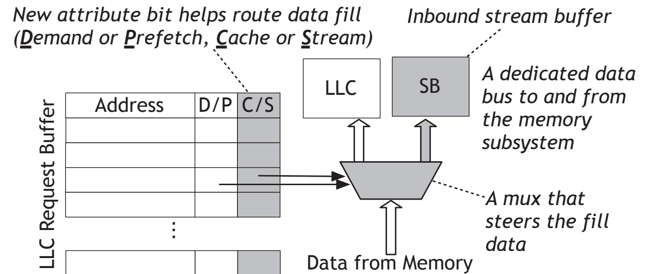



**Figure 10: Implementation of streaming instructions into existing data path of a generic last level cache request/fill microarchitecture. Minimal modifications required are shaded.**

## 3.3 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 [28].

**Instruction Set Architecture** Software moves data between memory and the stream buffers via the four instructions described at the bottom of Table 2. `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 `sbload`s and `sbstore`s 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`/`sbstore`s 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 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.

**Microarchitecture** To implement the streaming instructions, we propose minimal modifications to conventional processor microarchitecture. Figure 10 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 10. Finally, a dedicated bi-directional data bus is added to connect that mux to the stream buffer.

Stream buffers can be made fully coherent to the core caches. `sbload`s 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`s always produce the most up-to-date values. Figure 10 depicts the scenario when a request misses all levels of the cache hierarchy, and the fill is not cached, as `sbload`s are non-cacheable. On the store side, `sbstore`s 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 non-temporal. Because `sbload`s 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.

# 4. EVALUATION METHODOLOGY

To evaluate the throughput, power, and area efficiency of our design, we implemented HARP in Bluespec System Verilog [3].

**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 4 records per burst.

**HARP Simulation** Using Bluesim, Bluespec's cycle-accurate simulator, we simulate HARP partitioning 1 million random records. We then convert cycle counts and cycle time into absolute bandwidth (in $GB/sec$).

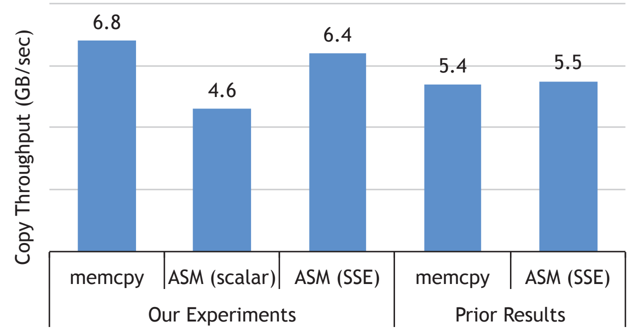



**Figure 11: The streaming framework shares much of its implementation with the existing memory system, and as such its throughput will be comparable to the copy throughput of existing systems.**

| | HARP Unit | | | | Stream Buffers | | | |
|---|---|---|---|---|---|---|---|---|
| Num. | Area | | Power | | Area | | Power | |
| Parts. | $mm^2$ | % 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[4] | 13.6% | 0.21[4] | 4.2% | 0.18 | 0.4% | 0.233 | 4 .7% |

**Table 3: Area and power overheads of HARP units and stream buffers for various partitioning factors.**

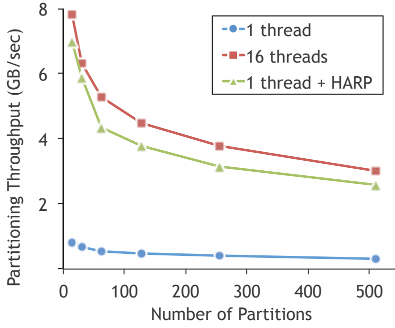
**HARP Synthesis and Physical Design** We synthesized HARP using the Synopsys [49] Design Compiler followed by the Synopsys IC Compiler for physical design. We used Synopsys 32 $nm$ Generic Libraries; we chose $HVT$ cells to minimize leakage power and normal operating conditions of 0.85 $V$ supply voltage 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.

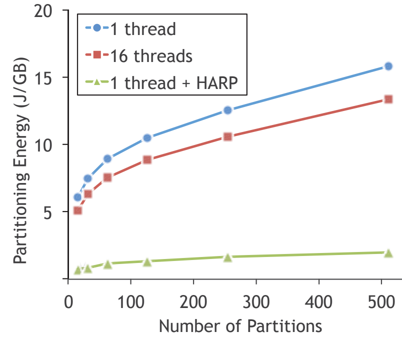
**Xeon Area and Power Estimates** The per-processor core area and power figures in the analyses that follow are based on Intel's published information and reflect our estimates for the system we used in our software partitioning measurements as described in Table 1.

**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 1 $GB$ table natively

[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]Scaled conservatively from the baseline design using area and power trends seen in Figures 16 and 17.



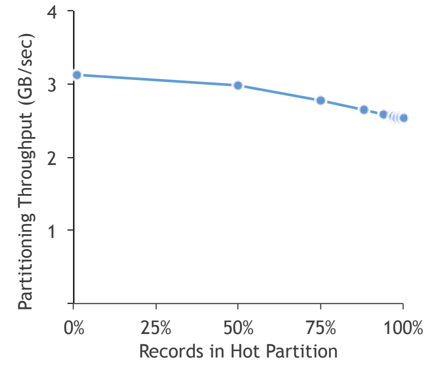




**Figure 12: A single HARP unit outperforms single threaded software from 7.8X with 63 or 255 partitions to 8.8X with 31 partitions, approaching the throughput of 16 threads.**

**Figure 13: HARP-augmented cores partition data using 6.3-8.7X less energy than parallel or serial software.**

**Figure 14: As input imbalance increases, throughput drops by at most 19% due to increased occurrence of back-to-back bursts to the same partition.**

on the Xeon server described in Table 1. All code was compiled using `gcc` 4.6.3 with -O3 optimization.

**Streaming Buffer Area and Power** We use CACTI [22] 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.

# 5. EVALUATION RESULTS

## 5.1 Area, Power and Performance

We evaluate the proposed HARP system in the following categories:

1. Throughput comparison with the optimistic software range partitioning from Section 2.2.
2. Area and power comparison with the processor core on which the software experiments were performed.
3. Non-performance partitioner desiderata.

For all evaluations in this section, we use the baseline configuration of HARP outlined in Section 4 unless otherwise noted.

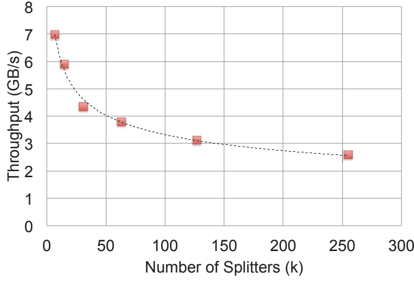
**HARP Throughput** Figure 12 plots the throughput of three range partitioner implementations: single-threaded software, multi-threaded software, and single-threaded software plus HARP. We see that HARP's throughput exceeds a single software thread by 6.5X–8.8X, with the difference primarily attributable 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 16 threads for the software implementation to match the throughput of the hardware implementation. At 3.13 $GB/sec$ per core with HARP, augmenting all or even half of the 8 cores with HARP would provide sufficient compute bandwidth to fully utilize all DRAM pins.

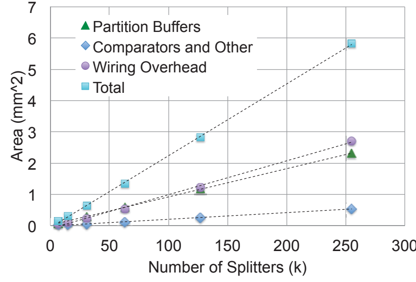
In terms of absolute numbers, the baseline HARP configuration achieved a 5.06 $ns$ critical path, yielding a design that runs at 198 $MHz$, delivering partitioning throughput of 3.13 $GB/sec$. This is 7.8 times faster than the optimistic single-threaded software range-partitioner described in Section 2.2.

**Streaming Throughput** Our results in Figure 11 show that C's standard library memcpy provides similar throughput to hand-optimized vector code, while scalar code's throughput is slightly lower. For comparison, we have also included the results of a similar experiment published by IBM Research [48]. Based on these measurements, we will conservatively estimate that the streaming framework can bring in data at 4.6 $GB/sec$ and write results to memory at 4.6 $GB/sec$ 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 3 quantifies the area and power 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 2.83 $mm^2$ 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 3. The proposed streaming framework adds 0.3 $mm^2$ area, consumes 10 $mW$ power for a baseline HARP configuration.



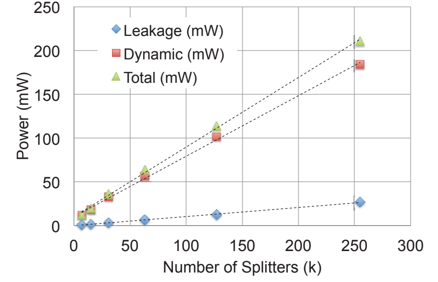
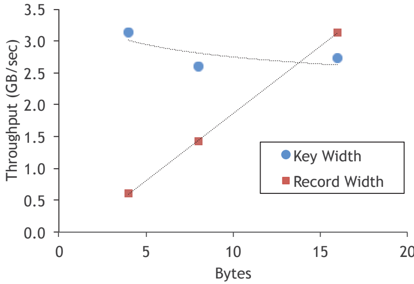




**Figure 15: HARP throughput is most sensitive to the number of partitions, dropping about 38% going from a 15-way to a 63-way partitioner.**

**Figure 16: HARP area scales linearly to the number of partitions because partition buffers dominate area growth and are scaled linearly with the number of partitions.**

**Figure 17: HARP power consumption also scales linearly with the number of partitions, on roughly the same linear scaling as area.**



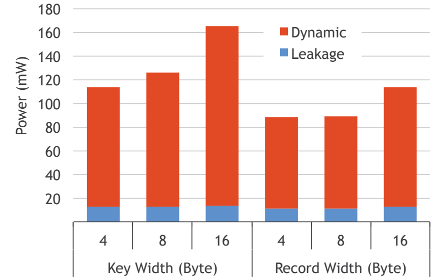



**Figure 18: HARP throughput increases linearly with record width because HARP partitions in record granularity. HARP throughput degrades mildly when key width increases.**

**Figure 19: HARP area is not particularly sensitive to key or record widths. Wiring overhead and partition buffers dominate area at over 80% of the total partitioner area.**

**Figure 20: HARP power consumption is slightly sensitive to key widths because the comparators are doubled in width when the key width doubles.**

**Energy Efficiency** From an energy perspective, this slight increase in power is overwhelmed by the improvement in throughput. Figure 13 compares the partitioning energy per GB of data of software (both serial and parallel) against HARP-based alternatives. The data show a 6.2–8.7X improvement in single threaded partitioning energy with HARP.

**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 14 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 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

**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 | | |

**Table 4: Parameters for HARP design space exploration with baseline configuration highlighted.**

seen experimentally in Figure 14, where throughput sinks from 3.13 $GB/sec$ with no skew to 2.53 $GB/sec$ in the worst-case.

## 5.2 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.

Below, we examine eleven different design points by holding two of the design parameters in Table 4 constant while varying the third. All reported throughputs are measured using a uniform random distribution of records to partitions. Figures 15 - 17 compare the throughput, area, and power as the number of partitions varies. Figures 18 - 20 show the same comparisons as number of key width and record width vary.

**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 Tradeoffs** 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 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 tradeoffs among the parameters we have explored and choose a design that provides high throughput, low area, and high energy efficiency while maintaining overall system balance.

# 6. RELATED WORK

**Streaming Computation** The last decade has seen substantial interest in software-based streaming computation with the development of new parallel languages [7, 17] and middleware support focused on portability and interoperability [11, 27, 39, 13].

The hardware support for streaming has been substantially more limited. The vast majority of streaming architectures, such as Cell's SPE [15], RSVP [10], or Piperench [16] are decoupled from the processing core and are highly tailored to media processing. The designs that most closely resemble HARP microarchitecturally are DySER [19] and ReMAP [52]. 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). While one might be able to program DySER to partition data, its full interconnect between functional units is overkill for partitioning's predictable data flow. ReMAP [52] has a very different goal, integrating reconfigurable fabric, called Specialized Programmable Logic (or SPL), to support fine-grained inter-core 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 which 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., non-temporal loads).

The SIMD vector extensions outlined above 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 [5, 32] and regular expression matching [45] and database kernels [55, 18, 31]. Our experiments in Section 2.2 using a state of the art SIMD range partitioning [46] indicate that vector-based traversal improves throughput somewhat but fails to fully saturate DRAM bandwidth.

These efforts demonstrate moderate speedups, at the cost of substantial programmer effort. One recent study of regular expression matching compared different strategies for acceleration [45]. 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 [4]. While 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. While FPGAs have been successfully used to accelerate a number of data intensive algorithms [35, 53, 36], 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 paper 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% [26]. This observation and others about the performance criticality of memory controller throughput [38] have inspired substantial research in memory scheduling (e.g., [42, 41, 47, 26, 14]). 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.

# 7. CONCLUSIONS

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.

We presented the design and implementation of HARP, a hardware accelerated range partitioner. HARP is able to provide a compute bandwidth of at least 7.8 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.

# 8. ACKNOWLEDGMENTS

The CAD infrastructure was partly supported by resources of Sethumadhavan's Comp Arch and Security Technology Lab (CASTL) which is funded through grants CNS/TC 1054844, FA 99500910389, FA 865011C7190, FA 87501020253 and gifts from Microsoft Research, WindRiver Corp, Xilinx Inc. and Synopsys Inc.. The authors also wish to thank Todd Austin, Doug Carmean, Stephen Edwards, Tim Paine, and the anonymous reviewers for their time and feedback.

# 9. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, 1999.

[2] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, 2011.

[3] Bluespec, Inc. Bluespec Core Technology. `http://www.bluespec.com`.

[4] H. Boral and D. J. DeWitt. Database machines: an idea whose time has passed? In *IWDM*, 1983.

[5] R. D. Cameron and D. Lin. Architectural support for SWAR text processing with parallel bit streams: the inductive doubling principle. In *ASPLOS*, 2009.

[6] Centrum Wiskunde and Informatica. `http://www.monetdb.org`.

[7] S. Chakraborty and L. Thiele. A new task model for streaming applications and its schedulability analysis. In *DATE*, 2005.

[8] D. Chatziantoniou and K. A. Ross. Partitioned optimization of complex queries. *Information Systems (IS)*, 32(2):248–282, 2007.

[9] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, 2008.

[10] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (RSVPTM). In *MICRO*, 2003.

[11] B. F. Cooper and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *CAC*, 2005.

[12] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: active low-power modes for main memory. In *ASPLOS*, 2011.

[13] M. Duller, J. S. Rellermeyer, G. Alonso, and N. Tatbul. Virtualizing stream processing. In *Middleware*, 2011.

[14] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel application memory scheduling. In *MICRO*, 2011.

[15] B. Flachs et al. A streaming processing unit for a CELL processor. In *ISSCC*, 2005.

[16] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: a co/processor for streaming multimedia acceleration. In *ISCA*, 1999.

[17] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[18] N. K. Govindaraju and D. Manocha. Efficient relational database management using graphics processors. In *DaMoN*, 2005.

[19] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA*, 2011.

[20] G. Graefe and P.-A. Larson. B-tree indexes and CPU caches. In *ICDE*, 2001.

[21] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4), 2011.

[22] HP Labs. `http://www.hpl.hp.com/research/cacti/`.

[23] IBM. DB2 Partitioning Features. `http://www.ibm.com/developerworks/data/library/techarticle/dm-0608mcinerney`.

[24] IBM. IBM What is big data? Bringing big data to enterprise. `http://www-01.ibm.com/software/data/bigdata/`.

[25] Intel Corporation. Intel® Xeon® Processor E5620. `http://ark.intel.com/products/47925`.

[26] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[27] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD*, 2006.

[28] N. P. Jouppi. Improvind direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

[29] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[30] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30(4), July/August 2010.

[31] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, Sept. 2011.

[32] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *HPCA*, 2012.

[33] K. T. Malladi, F. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz. Towards energy-proportional datacenter memory with mobile dram. In *ISCA*, 2012.

[34] Microsoft. Microsoft SQL Server 2012. `http://technet.microsoft.com/en-us/sqlserver/ff898410`.

[35] C. Mohan. Impact of recent hardware and software trends on high performance transaction processing and analytics. In *TPCTC*, 2011.

[36] R. Müller and J. Teubner. FPGAs: a new point in the database design space. In *EDBT*, 2010.

[37] MySQL. Date and time datatype representation. `http://dev.mysql.com/doc/internals/en/date-and-time-data-type-representation.html`.

[38] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, 2004.

[39] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, 2010.

[40] Oracle. Oracle Database 11g: Partitioning. `http://www.oracle.com/technetwork/database/options/partitioning/index.html`.

[41] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT*, 2007.

[42] S. Rixner. Memory controller optimizations for web servers. In *MICRO*, 2004.

[43] K. A. Ross and J. Cieslewicz. Optimal splitters for database partitioning with size bounds. In *ICDT*, pages 98–110, 2009.

[44] P. Saab. Scaling memcached at Facebook, Dec 2008. `https://www.facebook.com/note.php?note_id=39391378919`.

[45] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira. Accelerating business analytics applications. In *HPCA*, 2012.

[46] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In *DaMoN*, 2009.

[47] J. Shao and B. Davis. A burst scheduling access reordering mechanism. In *HPCA*, 2007.

[48] H. Subramoni, F. Petrini, V. Agarwal, and D. Pasetto. Intra-socket and inter-socket communication in multi-core systems. *IEEE Computer Architecture Letters*, 9:13–16, January 2010.

[49] Synopsys, Inc. 32/28nm Generic Library for IC Design, Design Compiler, IC Compiler. `http://www.synopsys.com`.

[50] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.

[51] Transaction Processing Performance Council. `http://www.tpc.org/tpch/default.asp`.

[52] M. A. Watkins and D. H. Albonesi. ReMAP: A reconfigurable heterogeneous multicore architecture. In *MICRO*, 2010.

[53] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *PVLDB*, 3(1):660–669, 2010.

[54] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, 2011.

[55] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, 2002.