Collaborative Accelerators for In-Memory MapReduce on Scale-up Machines

Abraham Addisie and Valeria Bertacco
Computer Science and Engineering, University of Michigan
Email: {abrahad, valeria}@umich.edu

Abstract— Relying on efficient data analytics platforms is increasingly becoming crucial for both small and large scale datasets. While MapReduce implementations, such as Hadoop and Spark, were originally proposed for petascale processing in scale-out clusters, it has been noted that most data centers processes today operate on gigabyte-order or smaller datasets, which are best processed in single high-end scale-up machines. In this context, Phoenix++ is a highly optimized MapReduce framework available for chip-multiprocessor (CMP) scale-up machines. In this paper we observe that Phoenix++ suffers from an inefficient utilization of the memory subsystem, and a serialized execution of the MapReduce stages. To overcome these inefficiencies, we propose CASM, an architecture that equips each core in a CMP design with a dedicated instance of a specialized hardware unit (the CASM accelerators). These units collaborate to manage the key-value data structure and minimize both on- and off-chip communication costs. Our experimental evaluation on a 64-core design indicates that CASM provides more than a 4x speedup over the highly optimized Phoenix++ framework, while keeping area overhead at only 6%, and reducing energy demands by over 3.5x.

I. Introduction

Both small and large companies, research institutes, and governmental agencies are increasingly relying on fast and efficient data analytics platforms. MapReduce [8] implementations, such as Hadoop [25] and Spark [29], are commonly deployed for this purpose. MapReduce is a programming paradigm that facilitates the parallel processing of large data sets and provides programmers with a simple abstraction to implement a wide range of data-intensive applications. While MapReduce was initially introduced to process multi-terabyte and petabyte data in scale-out clusters, most MapReduce workloads have a footprint in the GB range, as reported by [3]: indeed, data from analytics production clusters at Microsoft and Yahoo indicate that their median job has a 14GB input size and 90% of jobs in Facebook are smaller than 100GB. Today, a scale-up server enjoys substantially large memory and storage resources, being able to potentially accommodate most jobs in a data center and eliminating the communication overheads of scale-out solutions. Researchers at Microsoft have shown that a single-node scale-up optimized Hadoop framework is superior to its scale-out counterpart for performance/Watt and server density [3] and is competitive or better in terms of performance and cost. Similar conclusions have been reached for

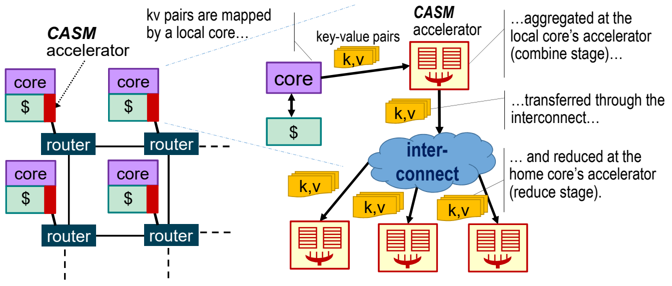

CASM
accelerator
kv pairs are mapped by a local core...
key-value pairs
CASM
accelerator
...aggregated at the local core's accelerator (combine stage)...
core
$
router
k,v
...transferred through the interconnect...
inter-connect
... and reduced at the home core's accelerator (reduce stage).

Fig. 1. **CASM deployed in a CMP architecture**. Left side - CASM adds a local accelerator to each CMP's core to carry out MapReduce tasks. Right side - Each accelerator aggregates kv-pairs emitted by the local core (local aggregation). Those kv-pairs are then transferred among the accelerators through the CMP's interconnect. At the home core, accelerators execute the reduce stage, so that, in the end, there is one kv-pair per key.

the Spark framework [28]. Indeed, for future data center scalability, it is crucial to design system architectures that can provide high performance with limited power and area budgets. More and more, because of their simplicity compared to traditional models like Pthreads, developers are turning to scale-up implementations of MapReduce, even for applications with small to medium data inputs [22, 27, 24]. In this work, we target the most optimized scale-up MapReduce framework available, Phoenix++ [24], analyze its inefficiencies and propose a novel accelerator architecture that overcomes those inefficiencies.

Phoenix++ underwent major revisions from previous releases [22, 27], and has been shown to be competitive against a hand-crafted Pthreads implementation. Phoenix++ provides a simple programming interface for users, while it manages internally the execution of the MapReduce tasks. At runtime, each core involved in an execution with the Phoenix++ framework must manage its own hashtable (a single centralized hashtable would lead to extreme access contention). Unfortunately, this approach creates pressure in the memory system causing high on- and off- chip communication. A monolithic concurrent hashtable (*i.e.*, no locks needed) as in [19] could remedy this problem. However, such a solution would provide limited performance benefits, as it relies on a traditional memory subsystem, which is suboptimal for CMP-based MapReduce applications.

The goal of this work is to **address the shortcomings of traditional CMP-based MapReduce implementations** through small additions to a baseline CMP architecture, so to be transparent to the application and the MapReduce framework.

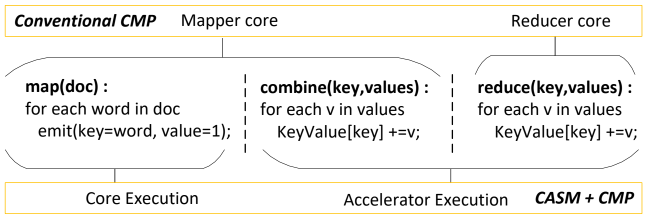


Fig. 2. **MapReduce for *wordcount***. *map* emits a kv-pair for each word, *combine* aggregates words emitted from a mapper, whereas *reduce* aggregates emitted words from all mappers. In CASM, *map* is executed by the cores, while *combine* and *reduce* are offloaded to the accelerators' network.

Specifically, we propose to augment existing CMP architectures with a network of small accelerators, in contrast with recent domain-specific approaches [13], which transfer the entire process to an accelerator. Fig.1 illustrates our proposed architecture, called CASM (Collaborative Accelerators for Streamlining Mapreduce). CASM comprises a network of accelerators, laid out alongside the cores, capable of delivering high computation performance locally, while collaboratively managing the application's data footprint, so as to minimize data transfers among the cores and to off-chip memory. Each accelerator contains two storage structures: a home scratchpad and a local scratchpad memory. The scratchpad memories (*SPM*) share some similarity with the home and local directories in a directory-based cache coherence protocol, but they are indexed by keys instead of memory addresses. The home SPMs collectively form a large cumulative on-chip memory to store (in MapReduce terminology) key-value pairs (*kv-pairs*). Each SPM is responsible for its own portion of the keys' space, capturing most keys of the input dataset – spilling to memory occurs rarely, only when the home SPMs cannot store all the keys. While home SPMs minimize off-chip memory access, most kv-pairs would still traverse the interconnect, potentially leading to performance degradation due to interconnect contention: local SPMs are used to combine kv-pairs locally, so as to slash contention on the interconnect.

Contributions. In summary, CASM's novel contributions are:

• a novel, distributed approach to hardware acceleration, through a network of collaborating units, leading to over 4x performance boost for communication-intensive applications.

• a scale-up implementation of our solution that is transparent to the MapReduce programming interface, thus preserving its simplicity for application developers.

• MapReduce acceleration at minimal area footprint (below 6%) and great energy savings (in excess of 3.5x), as shown by our simulation-based analysis [5], which compares CASM to a software-only CMP-based MapReduce framework.

II. Background and Motivation

MapReduce is a simple programming framework for data-intensive applications. Users can write complex parallel programs by simply defining map and reduce functions, while all the remaining parallel programming aspects, including data partitioning and data shuffle stages, are handled by the MapReduce infrastructure. In a typical scale-up MapReduce framework, the application's input data is first partitioned among the cores in the system. Then, each core runs the user-defined ***map*** function, which processes the input data and produces a list of intermediate kv-pairs, followed by a ***combine*** stage, which aggregates keys to partially reduce local kv-pairs, and thus conserves network bandwidth. Once this stage is complete, the intermediate data is ***partitioned*** and ***shuffled*** within the network, while the cores assume the role of reducers, so that all kv-pairs with the same key are transferred to a same reducer. In the final stage, each core executes the user-defined ***reduce*** function to complete the aggregation of its kv-pairs. As an example, Fig.2 provides the pseudo-code for *wc* (wordcount), a classic application with a wide-range of applications. *wc* computes the frequency of occurrence of each word in a document. In the MapReduce framework, the map function parses the input document and identifies all the words. It then emits each word as part of a kv-pair, with the word as the key and an initial value of 1. Combine partially aggregates kv-pairs at each core, before they are transferred to the reducer core. Finally, the reduce function collects all kv-pairs and sums up the values for each word, generating a final list of unique words along with their frequency.

Phoenix++: optimizations and inefficiencies. Phoenix++ is among the most optimized scale-up MapReduce framework for CMPs. One major optimization adopted by Phoenix++ is the interleaving of map and combine stages, which lowers memory pressure caused by large kv-pairs. In Phoenix++, kv-pairs are aggregated locally, immediately after they are mapped. The left part of Fig.3 illustrates the MapReduce steps for Phoenix++. At first, each core considers its own data segment, one input at a time, maps it into a kv-pair (orange) and combines with its current database of kv-pairs (red). Then it partitions aggregated kv-pairs over all the cores based on their key (yellow). At this point the threads are synchronized so that all cores switch from operating as local cores to home cores. kv-pairs are then transferred through the interconnect to the home core and reduced there (green). However, there are still two major inefficiencies in this approach: i) map and combine stages are executed in a sequential manner. If they were to run concurrently, as suggested in the right part of Fig.3, execution time would be significantly reduced. ii) Moreover, each combine function (one per map function) maintains its own kv data structure as a hashtable, thus keys are replicated in the on-chip caches, creating many off-chip memory accesses when that data no longer fits in cache.

Motivating Study. To gain insights on the execution bottlenecks of a real platform, we analyzed the execution of Phoenix++ with our experimental workloads (Section VI) on a 16-core Intel Xeon E5-2630 (2 threads per core) machine, using a range of input data sizes. Fig.4 plots the breakdown of execution time by MapReduce stage. Note, first of all, how map and combine dominate the overall execution time, and the combine stage contributes the majority of the total execution time for most benchmarks. We then analyzed in detail the execution of the combine stage for those benchmarks using Intel's VTune tool and collected the *Top-down Microarchitec-*

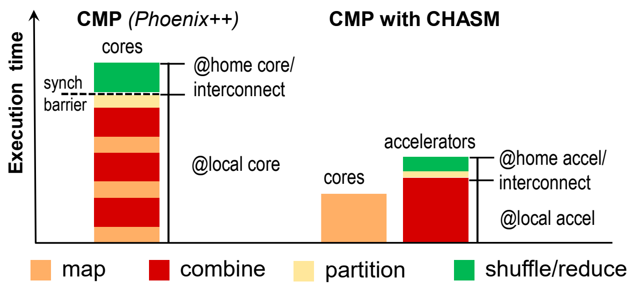


Fig. 3. **Execution flow of MapReduce**. Left side - in a typical CMP-based framework, the map, combine and partition stages execute on the local core. A barrier then synchronizes the execution, kv-pairs are shuffled through the interconnect and reduced at the home core. Right-side - When deploying CASM, cores are only responsible for the map stage. They then transfer kv-pairs to the local accelerator, which combines, partitions and transfers them to the home accelerator for the reduce stage.

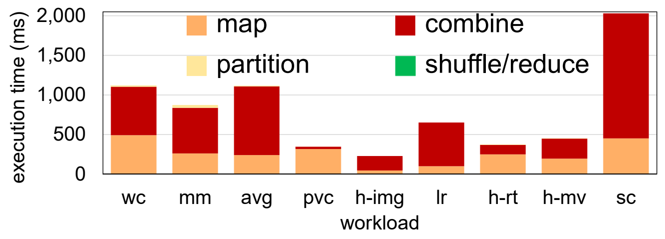


Fig. 4. **Motivating study**. Execution time breakdown for our MapReduce workloads, running Phoenix++ on a Xeon E5-2630 V3 machine with input datasets as in Section VI. Note how the combine stage dominates overall execution for most applications.

ture Analysis Method (TMAM) metrics [26]. VTune reported that most workloads were primarily back-end bounded (60% average value), that is, the main cause of performance bottlenecks was the inability for instructions to progress through the pipeline. Vtune could also indicate that, among the (lack of) resources that caused back-end bottlenecks, the time overhead in accessing memory was the primary one in our case. Based on this analysis, we suspected that the bottleneck could be due to the large number of data transfers occurring during the combine stage, driven by the need to maintain multiple, large and irregularly accessed hashtables (one per core), which often do not fit in on-chip storage. Thus, to assess the impact of these off-chip memory accesses, we setup a 64-core system on the gem5/garnet infrastructure, where all kv-pairs could be accommodated in on-chip storage, as it would be in an ideal case. This last analysis showed that the total data-transfer footprint differential between the real and the ideal system is over 9x.

III. Inside a CASM Accelerator

Each CASM's accelerator (see Fig.5) comprises a scratchpad memory (SPM), organized into two partitions, one to serve kv-pairs incoming from the local processor core (local SPM), and one to serve kv-pairs incoming from the interconnect (home SPM). Each SPM is complemented by a small "victim SPM", similar to a victim cache, which stores data recently evicted from the main SPM. The accelerator also includes dedicated hardware to compute a range of reduce functions, used both to aggregate data in the local SPM and in the home one. Logic to compute hash functions, both for indexing the SPMs and for partitioning kv-pairs over home accelerators, completes the design. Note that both local and home SPMs have fixed sizes, thus it may not be always possible to store all the kv-pairs that they receive. When a local SPM cannot fit all the kv-pairs, it defers their aggregation to the reduce stage, by transferring them to the home accelerator. When a home SPM encounters this problem, it transfers its kv-pairs to the local cache, and then lets the home core carry out the last few final steps of the reduce function.

When an accelerator receives a kv-pair from either its associated core or the network, it first processes the key through its *key hash unit* and through the *partition stage unit*. The purpose of the former is to generate a hash to use in indexing the SPM. The latter determines which home accelerator is responsible for reducing this kv-pair: if the local accelerator is also the home accelerator (*accel_is_home* signal), then we send the kv-pair to the home SPM, along with the hash value and an enable signal, otherwise we send it to the local SPM. Note that all kv-pairs incoming from the network will be aggregated at the home SPM. Pairs incoming from the local core will be aggregated at the home SPM only if the local core is also the home one. Each SPM is organized as a 2-way cache augmented with a small victim cache. Associated with each SPM is an *aggregate unit*, responsible for deploying the specified reduce function to combine two kv-pairs with the same key. Each SPM is also equipped with a dedicated unit, called *frequency/collision update unit*, to keep up to date the replacement policy information. Finally, note that, when a kv-pair is spilled from a SPM, it is transferred out through the interconnect, either to a home accelerator or to local cache. Spilling occurs because of eviction from the victim SPM, or because of losing in colliding with another entry in the SPM.

Hash Function Units. Our accelerator includes two hash computing units: the *key hash unit* and the *partition stage unit*. The former is used to compute the index value to access the SPM, which is organized as a 2-way associative cache. The latter uses a hash function to create a unique mapping from keys to an accelerator ID (*accel ID*), so that kv-pairs from each core can be distributed among the home accelerators for the final reduce stage, and each home accelerator is responsible for the reduce stage of all the keys hashed to its ID. We used an XOR-rotate hash for both units. We considered several alternatives for the *key hash unit* and found that XOR-rotate [16] is both the most compute efficient and has a low collision rate.

SPM (Scratchpad Memory). The accelerators' SPM is a fixed-size storage organized as a 2-way associative cache, where the set is determined by the *key hash unit*. Each entry in the SPMs stores the following fields: valid bit, key, the corresponding value to that key, frequency and collision values. When a kv-pair accessing the SPM "hits", that is, the keys of the SPM entry and the kv-pair are a match, we aggregate the two values by sending them to the aggregate unit, and then update the entry in the SPM. When a kv-pair "conflicts" in the SPM, that is, both stored keys are different, then we leverage our replacement solution to determine which kv-pair (the incoming one or one of those already stored) should be removed

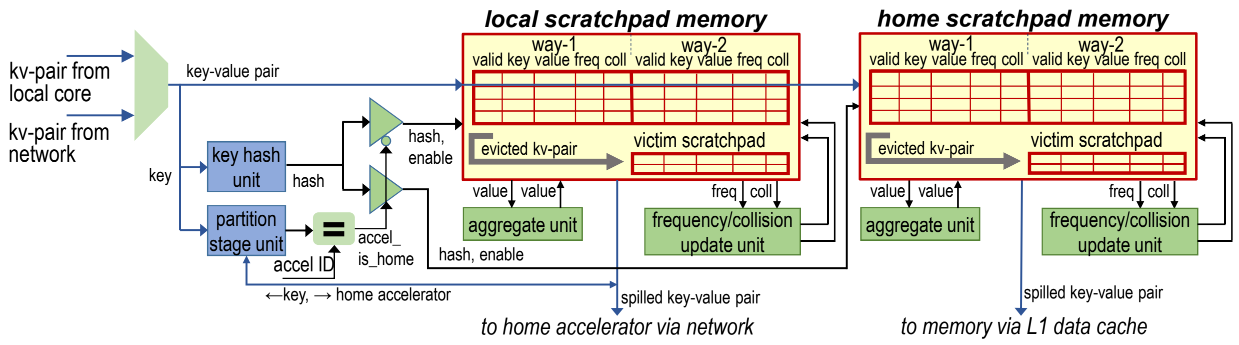


Fig. 5. **CASM's accelerator architecture**. Each accelerator includes two *SPMs*, organized as 2-way associative caches with small *victim SPMs*, indexed by a hash function, which aggregates kv-pairs incoming from the local core or the interconnect. The *aggregate units* aggregate values for kv-pairs stored in the SPMs. The *frequency/collision update units* enforce our kv-pair replacement policy. Finally, each accelerator includes a *key hash unit* to compute the hash of incoming keys, and a *partition stage unit*, responsible for deriving the ID of the home core in charge of reducing each unique key. kv-pairs evicted from the SPMs are transferred to their home core (from local SPM) or the local cache (from home SPM).

from the SPM and evicted to the victim SPM as discussed below. Of course, if there is an empty entry in the SPM corresponding to the current hash index, the incoming kv-pair will be stored there. We maintain separate local and home SPMs, one to store kv-pairs undergoing local aggregation, the other for kv-pairs in their final reduce stage. We keep them separate because, particularly for applications with a large number of unique keys, the home SPMs avoid key duplication and provide the equivalent of a large unified on-chip storage, minimizing kv-pair spills to memory. Local SPMs, on the other hand, are beneficial in avoiding network congestion. We also considered an alternative direct-mapped SPM organization, but dismissed it due to much higher collision rates.

kv-pair Replacement Policy. Since the SPMs are of limited size, it is possible for two distinct keys to be mapped to the same SPM entry and collide. Our replacement policy determines if a previously-stored key must be evicted and replaced with an incoming kv-pair. Upon storing a new entry in the SPM, its collision is initialized to 0 and its frequency to 1. Each time a new kv-pair is aggregated with a current entry, its frequency value is incremented, while the collision value remains unmodified. Each time there is a key conflict, that is, the incoming kv-pair has a different key than those stored in the SPM set, the collision is incremented for both kv-pairs in the set. Whenever an incoming kv-pair conflicts in the SPM, we analyze the two entries already in the SPM set to determine if one should be replaced. If, for either entry, the frequency is greater than the collision, then the entries are frequent ones and we simply update their collision values, but no replacement occurs. In this case, we send the new kv-pair to its destination (either its home accelerator or spilled into memory through the local cache). If, instead, collision exceeds frequency for one of the entries, then that entry is deemed infrequent, and it is replaced by the incoming kv-pair. If both entries are infrequent, we evict the one with the lowest frequency. Upon replacement, the new entry's frequency and collision values are reset.

Victim SPM . Depending on the sequence of keys accessing the SPMs, it is possible to incur thrashing, where a small set of keys keeps overwriting each other in the SPM. To limit the impact of this issue, we augment each SPM with a small, fully-associative "victim SPM": kv-pairs are stored in the victim SPM when they don't gain entry, or are evicted from the main SPM. All kv-pairs that are either evicted or rejected by the victim SPM are transferred to the home accelerator (from a local SPM), or to local cache (from a home SPM).

Aggregate Unit. The accelerator's *aggregate unit* implements the MapReduce reduce function. Our accelerator design supports several reduce operators, which cover a wide range of common MapReduce applications: we support addition, computing the maximum value, the minimum value, the average, and more. The average operation is implemented by separating it into two addition operations that are stored into the two halves of the data field: the first maintains the sum of values, and the second counts the total number of values. As a general rule, CASM requires that the reduce function be both commutative and associative, since the accelerators process kv-pairs independently from each other, and thus no ordering can be enforced on the kv-pair processing in the combine stage. Note that many practical applications satisfy this requirement and employ straightforward and common operators, as those we provide [7]. It is also possible to replace the *aggregate unit* with a reconfigurable logic block to provide further flexibility.

IV. System Integration

Fig.6 illustrates the interactions among all systems components during execution. At the start of an application, each core sends the type of aggregate operation and the pre-allocated memory region that the accelerator shall use for reduced kv-pairs at the end of the execution, and for potential spilled pairs. The core then begins sending mapped kv-pairs to its accelerator, completing this transfer with a *map completion* signal. Whenever an accelerator receives this signal from its local core, it sends all kv-pairs from its local SPM to their home SPMs, and then sends out a completion signal to all other accelerators in the system. After an accelerator has received completion signals from all other accelerators, it flushes out the contents of its home SPM to the local cache, and signals to its core that the processing has completed. At this point, the corresponding core retrieves the final, reduced, kv-pairs from memory and carries out a final reduce step, if any kv-pair was spilled during the execution. All communication from a

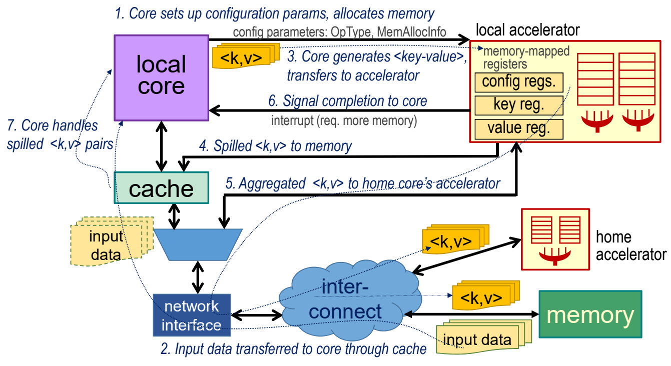

1. Core sets up configuration params, allocates memory
config parameters: OpType, MemAllocInfo
local accelerator
local core
<k,v>
3. Core generates <key-value>, transfers to accelerator
memory-mapped registers
config regs.
key reg.
value reg.
6. Signal completion to core
interrupt (req. more memory)
7. Core handles spilled <k,v> pairs
4. Spilled <k,v> to memory
cache
5. Aggregated <k,v> to home core's accelerator
input data
home accelerator
inter-connect
network interface
memory
2. Input data transferred to core through cache

Fig. 6. **System integration** showing the sequence of CASM execution steps.

core to its local accelerator is through store instructions to a set of memory-mapped registers, while accelerators communicate with the core via interrupts and shared memory. Each accelerator is also directly connected to the on-chip network interface to send/receive kv-pairs and synchronization commands to/from other accelerators and memory.

Cache Coherence. SPM storage is for exclusive access by its accelerator, and it is not shared with other units. Communication among the accelerators also happens via custom packets, instead of the CMP's coherence protocol's messages. Thus, CASM's read/write operations from/to the SPMs are transparent and oblivious to the CMP's coherence protocol. To handle spilling of kv-pairs to L1 cache, each accelerator, on behalf of its local core, writes the spilled kv-pair to the cache, handled by the CMP's coherence protocol. Note that, for each kv-pair, spilling is handled by only one home accelerator.

Virtual Memory. In accessing memory storage set up by the local core, each accelerator uses the same virtual memory space as its core, thus addresses are translated with the same page table and TLB as the process initiating the MapReduce application. Once the physical address is obtained, the access occurs by read/write to memory through the local cache.

Context Switching. During context switching of a process, the content of the local SPMs is flushed into their respective home SPMs, then the kv-pairs in the home SPMs are spilled to memory. Once context switching is complete, the accelerators stop issuing further requests to memory, so to avoid accessing stale data in page-tables and TLBs. Note that, when the context is restored, spilled kv-pairs do not have to be re-loaded to the SPMs, as their aggregation can be handled during the reduce step together, with other spilled kv-pairs.

V. Composite MapReduce Applications

Many MapReduce applications map to a single MapReduce task, for instance those considered in our evaluation (see Section VI). However, MapReduce has also been adopted for applications that involve multiple MapReduce tasks organized in a *pipeline*, as for collaborative filtering [30]. Other complex applications are those that compute their results *iteratively*, *e.g.* k-means [24]. Both execution flow structures are completely compatible with CASM, since CASM is embedded in the high-level framework, and final kv-pairs are copied from CASM's

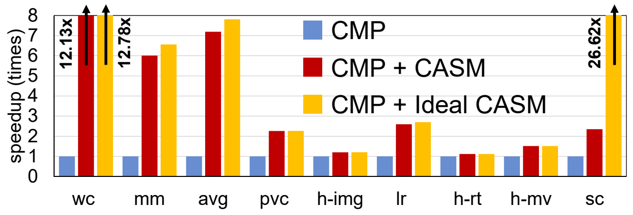


Fig. 7. **CASM performance speedup** over a baseline CMP execution of MapReduce. SPMs are 16KB for CASM, and infinite for the ideal variant.

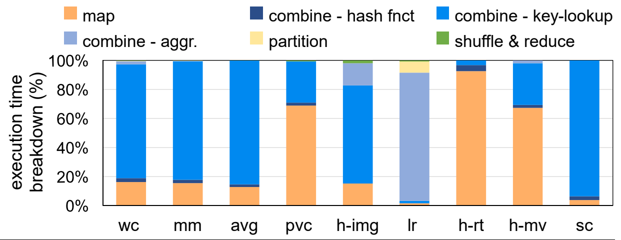
SPMs to the CMP's memory at the end of each MapReduce task. We leave further investigation of this family of applications for future work.

VI. Experimental Evaluation

In order to perform a detailed micro-architectural evaluation of a CASM-augmented, large-scale CMP architecture, we implemented our design with gem5 + Garnet [5], a cycle accurate simulation infrastructure. We ported the Phoenix++ framework [24] to gem5 using "m5threads" and carried out the simulations using the "syscall" mode. We modeled the baseline scale-up CMP solution as a 64-core CMP in a 8x8 mesh topology, with 4 DDR3 memory nodes at the corners of the mesh. Cores are OoO, 8-wide, equipped with a 16KB private L1 I/D cache, and shared 128KB L2. The interconnect uses 5-stage routers. CASM's SPMs are also 16KB, and use 8-entry victim SPMs. Each entry in the SPMs contains 64-bit key and value fields, along with an 8-bit field to implement our replacement policy. For 'average' reduce operations, the value field is partitioned into two 32-bit fields.

Workloads and Datasets. We considered several workloads in our evaluation: *wc* (wordcount, 68K-257K keys), *h-img* (histogram image, 768 keys) and *lr* (linear regression, 5 keys) are gathered from the Phoenix++ framework; while *sc* (counts frequency of 3-word sequences, 3.5M keys) *h-rt* (histogram ratings of movies, 5 keys) and *h-mv* (movies by histogram rating, 20K keys) are adopted from the PUMA benchmarks [2]. Finally, we developed *pvc* (page view count, 10K keys), *mm* (min-max), and *avg*, from scratch using the API of Phoenix++. The datasets come from the same framework as the workloads, except for *avg* and *mm* (we used a list of 28K cities and corresponding temperature logs) and *pvc*, which we generated. We used two families of datasets. In running the cycle-accurate gem5 simulations, we used datasets of 1GB for *h-rt* and *h-mv*, and 100MB for all others. The dataset footprint was driven by the limited performance of the gem5 simulator (*e.g.*, 15 days of simulation time for a 1GB input file). Moreover, datasets of similar size have been used to evaluate prominent scale-up MapReduce frameworks [22, 27, 24]. Hence, we determined that analyzing such datasets provided indeed useful and practical insights. To further evaluate the scalability of CASM to large-scale datasets, we carried out a study with dataset sizes of 30GB from [12] for *wc* and *sc*, 100GB from [2] for *h-mv* and *h-rt*, and 1.5GB from [24] for *h-img*. For other workloads, we generated 100GB datasets with the same number of keys.

Performance Evaluation. Fig.7 reports the speedup of CASM compared to the baseline CMP running Phoenix++. We report both the speedup that we could achieve with infinitely



	wc	mm	avg	pvc	h-img	lr	h-rt	h-mv	sc
map concurrency	1.09x	1.18x	1.15x	1.79x	1.20x	1.59x	1.12x	1.21x	1.04x
h/w acceleration	6.17x	3.12x	3.74x	1.13x	0.00x	1.32x	0.00x	0.00x	2.16x

Fig. 8. **Performance insights.** The plot reports a breakdown of MapReduce by stage for a baseline CMP framework. The table shows performance improvements provided by i) map stage concurrency alone, and ii) hardware acceleration alone.

large local and home SPMs (*i.e.*, no kv-pair collision occurs in either of the SPMs) and that of the actual setup (*i.e.*, 16KB SPMs). Note that most applications reach fairly close to their ideal speedup. From the figure, the ideal speedup ranges from 1.1x to 26x, while the speedups we observed with our implementation settings peak at 12x and average at 4x. Note that *wc*, *mm*, and *avg* have many unique keys; yet, CASM achieves an almost ideal speedup. *wc*, in particular, is an important kernel in MapReduce applications. *h-img*, *h-rt*, and *lr* have relatively few unique keys, which can comfortably fit in the baseline CMP's caches, reducing our room for improvement. The speedup of *lr* is relatively better because its map stage is less memory-intensive than that of *h-img* and *h-rt* (CASM executes concurrently with the map stage). *h-mv*'s speedup is limited despite its many unique keys, as its input data layout provides a cache-friendly access pattern, which benefits the baseline CMP. *pvc* entails fairly heavy parsing during mapping, thus limiting the speedup potential of CASM. Finally, it is clear that *sc*'s speedup is limited by the SPM size, because of its vast number of distinct keys, over 3 million. Yet, CASM's 16KB SPMs were able to deliver a 2x speedup.

Speedup Breakdown. To gain insights on the source of performance gains that CASM attains, we analyzed the execution stage breakdown of our testbed applications running on a Phoenix++/ CMP system. To gather the analysis data, this experiment leveraged our gem5-Garnet infrastructure. Fig.8 reports how execution time is partitioned among map, combine, partition and reduce stages. Moreover, we dissected the combine stage into i) hash function computation, ii) hash-key lookup, which entails memory accesses to walk the hashtable, and iii) data aggregation. Note that for most applications, the dominant time drain is the hash-key lookup. *pvc* presents a dominating map stage because of the extensive parsing of web addresses, while *lr* is dominated by data aggregation because it only uses five distinct keys, which can be easily mapped into registers. It can be noted that just optimizing the hash key lookup execution via a specialized hardware structure would provide significant overall performance benefits. The table in Fig.8 specifies the contribution of each source of performance improvement. As discussed in Fig.3, the map stage in CASM executes concurrently with the other MapReduce stages. The first row of the table reports the speedup we would obtain if this concurrency was the only source of performance improvement.

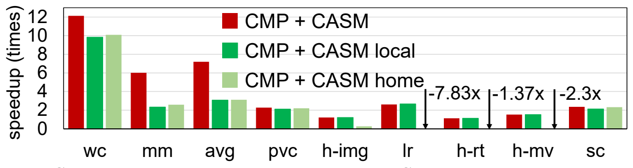


Fig. 9. **Speedup breakdown by home or local SPM.** CASM with local SPMs alone shows significant speedup on applications with few unique keys (no spilling to the home SPM). Home SPMs contribute to overall speedup mostly for applications with many keys and low key-access locality.

The second row of the table reports the speedup we would obtain if the map stage was serialized with the other stages, but combine, partition and reduce were all accelerated by CASM.

Data Transfers Analysis. To further explain sources of speedups, we tracked off-chip memory accesses for our baseline CMP, the CASM solution and an ideal solution. For the latter, we assumed that all unique keys could be accommodated in the SPMs, with no off-chip memory access for key-lookups. CASM reduces this traffic by 4.22x on average, while the ideal solution achieves a 9x traffic reduction. Such traffic reduction is possible because CASM's home SPMs experience minimal kv-pair spills (<4%), except for *sc*, which amount to 75% because of its large number of unique keys. CASM's ability to aggressively reduce off-chip memory accesses is the root of the vast performance benefits it provides, since most of our workloads spend most of their time performing key-lookups, which entail accesses to caches and memory. Furthermore, we found that CASM reduces interconnect latency by 7% on average, peaking at 15% for workloads such as *wc*.

SPM (scratchpad memory) Architecture Analysis. In our previous analysis, we evaluated the contribution to overall speedup by home and local SPMs. Fig. 9 provides the results of a comparative analysis when using only home SPMs, or only local SPMs – note that each SPM is still 16KB in size. When CASM uses only local SPMs, the benefits of local aggregation stand, and we still obtain acceptable speedups, although to a lower extent: local SPMs contribute to 2.75x of the speedup, on average. In particular, for applications with a moderate number of keys, such that they can all fit in the local SPM, local SPMs provide the majority of the overall performance benefit of CASM. For other applications, with many distinct keys, performance becomes more challenging using only local SPMs, because of the high rate of spilling to the home SPM. On the other hand, we found that having only home SPMs provides on average a 2.26x speedup. The performance boost here is extremely variable: applications with a large number of keys do best (*e.g.*, `wc`). Note that, in a few cases, the use of home SPMs alone leads to a slowdown, as much as -7x (`lr`), because of the high interconnect traffic generated in transferring all kv-pairs directly to the home SPMs, with no local aggregation.

Cache Size Sensitivity. Since CASM entails additional storage (32KB per accelerator in our experimental setup) over the L1 data caches, one might wonder if it were possible to achieve the same performance improvement by simply increasing the capacity of the cores' caches. To this end, we performed a cache-size sensitivity study by tracking the execution time of each application over a range of cache sizes. Fig.10 provides

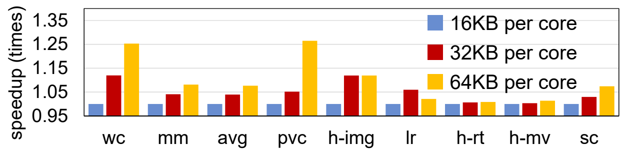


Fig. 10. **L1 cache sensitivity study.** Performance speedup over the baseline CMP, for varied L1 cache sizes. The performance improves by no more than 26% at L1=64KB.

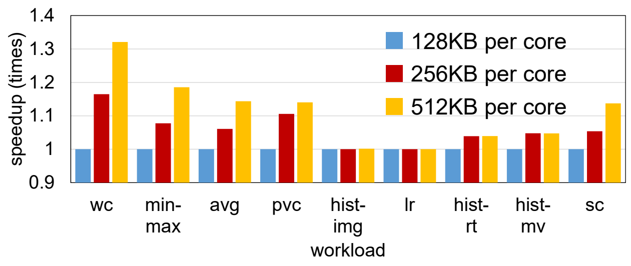


Fig. 11. **L2 cache sensitivity study.** Performance speedup over the baseline CMP, when sweeping the L2 cache size. The performance improves at most by 32% at L2=512KB.

a plot obtained by running our MapReduce applications on the baseline CMP, while sweeping the L1 data cache size from 16KB (the original baseline size) to 64KB. The plot shows that the largest performance gain is only 26%, corresponding to the largest L1 data cache considered, while running the `pvc` benchmark. Note that this solution point entails more storage than the total of our baseline L1 cache and the two SPMs embedded in the CASM accelerator. In contrast, with the addition of CASM, we can achieve an average of 4x speedup with less storage. Finally, we carried out a similar analysis for L2 caches. In this case, we swept the size of the total shared L2 cache from 8MB (128KB per core) to 32MB (512KB per core) and found that the largest performance gain was only 32% at 32MB when running `wc`, as reported in Figure 11.

Sensitivity to the Number of Cores. We evaluated the scalability of CASM over a range of cores in the underlying CMP, while always keeping the same 1:1 ratio between cores and accelerators. Figure 12 reports our findings for running `wc` on CMP+CASM systems from 8 to 64 cores. It can be noted that the speedup scales monotonically with the number of cores/accelerators. And with more accelerators, CASM approaches its ideal speedup. The reason behind this trend is the larger collective home scratchpad (the union of all the home scratchpads) that reduces the number of kv-pair spills.

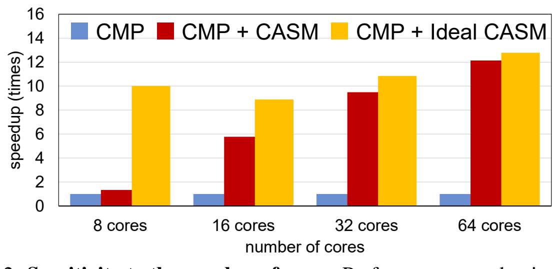


Fig. 12. **Sensitivity to the number of cores**. Performance speedup increases monotonically with the number of cores/accelerators in the system. The corresponding increase in global home-scratchpad storage (16KB per core) leads to a closer approximation of ideal speedups with more cores.

TABLE I
COLLISION RATES OVER A RANGE OF DATASET AND SPM SIZES

dataset size/	wc		mm/avg		pvc		sc	
SPM size	local	home	local	home	local	home	local	home
0.1-1GB / 16KB	24.14	3.51	46.35	1.92	3.99	0.00	91.44	75.31
30-100GB / 16KB	30.00	5.28	47.03	**1.63**	5.24	**0.00**	94.98	**74.51**
30-100GB / 32KB	24.16	3.96	**45.83**	**0.43**	**1.04**	**0.00**	93.47	**69.57**
30-100GB / 64KB	**18.71**	**3.01**	**41.25**	**0.10**	**0.63**	**0.00**	**91.19**	**65.17**

Scalability to Large Datasets. This study estimates the performance of CASM on the large datasets discussed earlier in this Section, ranging from 30 to 100GB. Since we could not carry out an accurate simulation, due to the limited performance of gem5, we used the collision rate on the local and home SPMs as a proxy for performance. Indeed, kv-pair collisions in local or home SPMs are the lead cause for contention in the interconnect; in addition, collisions in the home SPMs cause spilling to off-chip memory. As discussed above, off-chip memory accesses are the main source of performance benefits provided by CASM. In contrast, if an application generated no collisions, CASM's execution would almost completely overlap that of the map stage. Table I reports the collision rates for our applications, on all datasets we considered for them, both medium (0.1-1GB) and large (30-100GB). We compute a collision rate as the average number of kv-pairs collisions at the local/home SPM, divided by the total number kv-pairs. We only report four applications, because all others have a collision rate of 0%, irrespective of dataset size. The first row of the table reports the collision rate of the medium dataset – the one simulated in gem5 and reported in Fig.7. The other rows report the collision rate for the large datasets, over a range of SPM sizes. For the large datasets, we report in bold the SPM sizes that lead to a collision rate below that of our original setup with the medium dataset. Assuming that the baseline CMP maps large datasets as efficiently as medium datasets – a highly conservative assumption – CASM should achieve the same or better performance speedup as reported in Fig.7, when the collision rates at both SPMs fall below that of the medium datasets. Note that all workloads reach this low collision rate with 64KB SPMs. Note also that our analysis is based on a very conservative assumption that the baseline CMP would be able to provide the same performance for medium and large datasets, which is unlikely, as large datasets create further pressure on the baseline-CMP's memory subsystem, providing additional benefits for CASM.

Area, Power and Energy. We synthesized CASM's logic in IBM 45nm technology. We then setup Cacti with the same technology node to model the SPMs. We used McPAT to compute the same metrics for the other CMP node's components: cores, caches and interconnect. Our findings indicate that CASM accounts for approximately 6% of a CMP node's area and 1% of its peak power consumption. We derived energy consumption from average dynamic and leakage power and total execution time, using the tools detailed earlier and performance stats from gem5/Garnet. The performance speedup of CASM provides energy savings ranging from 1.17x to 11.8x, with an average of 3.5x. Since the power overhead of CASM is minimal (1%), it is natural to expect that high performance benefits translate into high energy savings.

VII. Related Work

Many MapReduce solutions targeting CMPs, GPUs and vector platforms have recently been proposed [22, 6, 24, 27, 18, 9, 10, 14]. CPU solutions rely on a traditional memory subsystem, which we found to be inefficient for most applications. Other architectures are optimized for compute-intensive applications, whereas, most MapReduce applications require large and irregularly accessed data structures, generating high off- and on-chip traffic. Several other works accelerate MapReduce using FPGA platforms: [23, 15], but their architecture does not allow for scaling to large CMP systems. Some recent works share some traits with CASM. For instance, [17] improves hash-key lookups, and [13] optimizes off-chip transfers, but do not tackle the other MapReduce bottlenecks. Other proposals, even if they had different goals, have proposed some architectural features that share some similarity with CASM: the software-defined caches of [4] partially resemble home scratchpads, [20] proposes a set of fine-grained accelerators, [11] provides a near-data processing architecture, [1] offers computation in cache, and [21] designs a novel reconfigurable architecture that is more flexible than domain-specific architectures. All of these solutions, while include valuable contributions, do not offers the full range of optimized design aspecs of CASM, and many have a different set of goals all together.

VIII. Conclusions

CASM is a novel collaborative hardware accelerator solution, capable of offloading the computation and communication portions of a scale-up implementation of MapReduce applications from the cores of a CMP to their local accelerators. The system is highly scalable with the number of cores, and provides over a 4x speedup on average over a CMP-based software solution, as we evaluated over a wide range of input dataset, up to 100GB in size. At the same time, it also provides energy savings of 3.5x. The cost of CASM is 6% in silicon area and 1% in peak power.

Acknowledgments

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[2] F. Ahmad *et al*. Puma: Purdue mapreduce benchmarks suite. 2012.

[3] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proc. SOCC*, 2013.

[4] N. Beckmann and D. Sanchez. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, 2013.

[5] N. Binkert *et al*. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[6] C.-T. Chu *et al*. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, 2007.

[7] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proc. NSDI*, 2012.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[9] K. Duraisamy *et al*. Energy efficient MapReduce with VFI-enabled multicore platforms. In *Proc. DAC*, 2015.

[10] W. Fang, B. He, Q. Luo, and N. Govindaraju. Mars: Accelerating MapReduce with graphics processors. *TPDS*, 2011.

[11] M. Gao, G. Ayers, and C. Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[12] P. Gutenberg. Accessed: 2017-11-05.

[13] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. MICRO*, 2016.

[14] T. Hayes *et al*. Future vector microprocessor extensions for data aggregations. In *Proc. ISCA*, 2016.

[15] C. Kachris, G. Sirakoulis, and D. Soudris. A reconfigurable MapReduce accelerator for multi-core all-programmable socs. In *Proc. ISSOC*, 2014.

[16] A. Klein. *Stream ciphers*. Springer, 2013.

[17] O. Kocberber *et al*. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proc. MICRO*, 2013.

[18] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. Goh. MrPhi: An optimized MapReduce framework on Intel Xeon Phi coprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 26(11), 2015.

[19] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: A cache-partitioned hash table. In *Proc. ACM SIGPLAN*, 2012.

[20] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr. Fine-grained accelerators for sparse machine learning workloads. In *Proc. ASP-DAC*, 2017.

[21] T. Nowatzki *et al*. Stream-dataflow acceleration. In *Proc. ISCA*, 2017.

[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. HPCA*, 2007.

[23] Y. Shan *et al*. FPMR: MapReduce framework on FPGA. In *Proc. FPGA*, 2010.

[24] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for shared-memory systems. In *Proc. MapReduce*, 2011.

[25] T. White. *Hadoop: The definitive guide*. 2012.

[26] A. Yasin. A top-down method for performance analysis and counters architecture. In *Proc. ISPASS,*, 2014.

[27] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proc. IISWC*, 2009.

[28] T. Yoo, M. Yim, I. Jeong, Y. Lee, and S.-T. Chun. Performance evaluation of in-memory computing on scale-up and scale-out cluster. In *Proc. ICUFN*, 2016.

[29] M. Zaharia *et al*. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[30] Z.-L. Zhao, C.-D. Wang, Y.-Y. Wan, Z.-W. Huang, and J.-H. Lai. Pipeline item-based collaborative filtering based on mapreduce. In *Proc. BDCloud*, 2015.