MR.NITRO: Distributed Accelerators for MapReduce on CMPs

Abraham Addisie†, Rawan Abdel-Khalek†, Ritesh Parikh‡, and Valeria Bertacco†

†University of Michigan
{abrahad, rawanak, valeria}@umich.edu

‡Intel Corporation
parikh@intel.com

ABSTRACT

MapReduce is a commonly used programming model that provides a simple and high-performance implementation of data-intensive applications by separating the workload into a map stage, where data is organized into a uniform key-value pairs format, and a reduce stage, where these key-value pairs are aggregated to generate the desired outcome. The execution of MapReduce on chip multi-processors (CMP) entails the use and management of complex data structures. These data structures limit the performance benefits enabled by the parallel architecture.

In this work, we propose to equip each core in the design with a hardware accelerator module that frees the core from the frequent memory accesses and the hash function computations required by the MapReduce framework. Our experimental evaluation on a 64-core design indicates that our solution provides over a 3 times speedup, averaged over several applications, compared with a best-in-class software implementation.

1. INTRODUCTION

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. Today, MapReduce frameworks are deployed in a wide range of performance-critical applications, *e.g.*: image processing, web-search engines, genomics, *etc.* In image processing, image recognition is powered by MapReduce applications, with impact in domains ranging from security to medicine and beyond. In web-search applications, MapReduce is deployed in a wide variety of algorithms that deliver the search speed and quality that we have come to expect today: finding search keywords in pages, ranking pages when presenting search results, *etc.*

The goal of our work is to boost the performance of MapReduce applications running on multi-core architectures by leveraging a network of distributed hardware accelerators, thus breaking the performance barriers that make some of the problems above impractical.

MapReduce abstracts a workload through two basic primitives: a map function that parses the input data and emits it in the form of intermediate key-value (kv) pairs and a reduce function that aggregates each set of intermediate kv-pairs associated with the same key. In a CMP-based framework, input data is allocated equally to each core, which then applies both the ***map*** function on each input, and the ***combine*** function, aggregating data locally as much as possible. The kv-pairs obtained are then ***partitioned*** among the cores, so that each one is responsible for carrying out the reduce function on some of the keys. The kv-pairs are then ***shuffled*** through the interconnect and ***reduced*** in a distributed fashion.

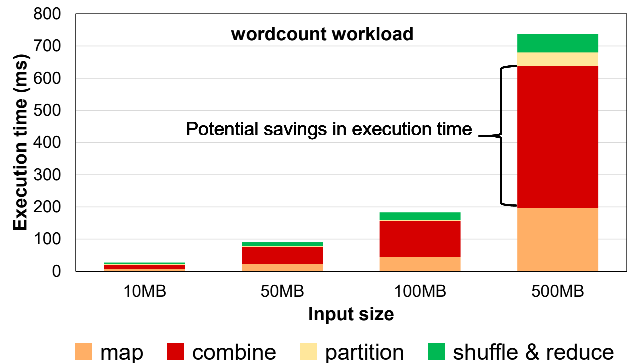

Figure 1: **Motivating study**. The chart plots the execution time's breakdown for *wordcount* with a range of input dataset sizes, running on the Phoenix++ framework [1] on an 8-core Intel Core i7-4790K machine. Each stage was isolated by leveraging the synchronization barrier and by executing multiple times with distinct termination points. Note that, while we measure the longest time among all cores, in practice the load is fairly balanced and each core computes approximately for the same amount of time in each stage. As it is evident form the plot, the combine stage drastically dominates the overall execution. By eliminating this stage alone, the overall execution time would be slashed by over 2x.

Motivating Study. As a motivating study, we analyzed the execution of MapReduce on a *wordcount* application over a range of input data sizes. Figure 1 plots the distribution of the execution time of each of the stages listed above for each of the data sets considered. In this study the application was implemented in the Phoenix++ framework [1], a state-of-the-art framework for MapReduce on chip multi-processors. We ran the experiment on an 8-core Intel i7-4790K and we derived the time taken by each stage by completing multiple runs, where we excluded all but one stage (all but two in the case of shuffle & reduce). For instance, to compute the execution time of the combine stage, we first measured the time to reach the synchronization barrier at the end of the combine stage. Then we executed the framework again, but forced each core to execute only the map stage. By subtracting the difference, we then derived the time spent in the combine stage alone.

The findings of our study suggest that the greatest opportunity for improvement lies in eliminating the combine stage, which accounts for over half of the total execution time. Thus, the goal of our solution is to offload this stage completely onto the hardware accelerators, so that it can be

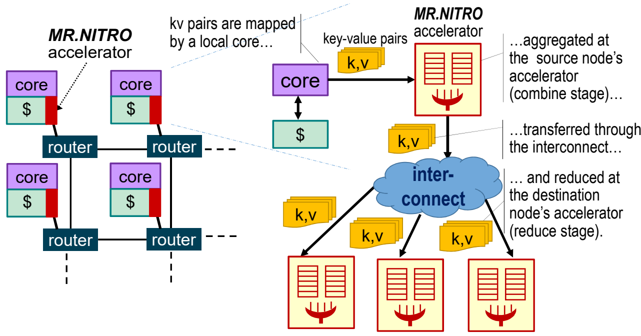

Figure 2: **MR.NITRO deployed in a CMP architecture**. Left side - MR.NITRO adds a local accelerator to each CMP's node to carry out MapReduce tasks. Right side - Each accelerator is responsible for completing the combine stage, aggregating kv-pairs emitted by the local core (source aggregation). Source-aggregated kv-pairs are then transferred among the accelerators through the CMP's interconnect. At the destination node, accelerators execute the reduce stage, so that there is only one kv-pair per key.

executed concurrently with the map stage running on the cores. We further managed to offload the partition and reduce stages to the accelerators, bringing further performance benefits to our solution.

In summary, MR.NITRO makes the following novel contributions:

- A novel distributed-accelerator solution for MapReduce, providing over a 3 times performance improvement on average, compared to CMP frameworks.
- Our distributed accelerators partially reduce intermediate kv-pairs at their source nodes, saving significant interconnect traffic by reducing transfers to the remote accelerators.
- MR.NITRO is transparent to the MapReduce programming interface, thus it preserves its simplicity to the application's developer.

2. RELATED WORK

The MapReduce programming model was originally introduced by Google [2] to provide efficient execution of data-intensive applications on a cluster of commodity-machines. With the adoption of chip-multiprocessor (CMP) architectures, several MapReduce frameworks have been proposed targeting these systems [3, 1]. In particular, Phoenix++ [1] is an optimized implementation of MapReduce for multi-core systems. It provides a simple programming interface for users, while internally managing the execution of the MapReduce tasks. However, for most applications, Phoenix++ relies on the use of complex data structures, limiting the performance gains that can be attained on these distributed architectures. MR.NITRO offloads the time-consuming and complex data accesses, as well as the reduce function execution, to special-purpose accelerators.

Similarly, [4] offloads the reduce stage to a single, centralized FPGA-based accelerator. This accelerator receives mapped kv-pairs from all cores. Then it performs in-hardware aggregation and stores final results into its internal memory. The main drawback of this solution is that it is not scalable over a large number of cores. It also uses a cuckoo hash function to implement key lookups. The function is applied repetitively until a free entry in the scratchpad (organized as a hashtable) is found. While this approach provides a better usage of the bounded-size scratchpad, it potentially leads to infinite loops of hash function computations [4]. On the other hand, we provide a scalable and distributed accelerator solution, where each CMP core is augmented with an accelerator module.

3. THE ACCELERATOR DESIGN

The focus of this work is to boost the performance of MapReduce applications by leveraging hardware acceleration. In general, CPU cores are most suitable in managing data transfers and organization, while systematic data processing can be completed more efficiently through a dedicated hardware structure, as an accelerator. Consequently, in our solution, the map stage of MapReduce is carried out by CMP's cores, which retrieve the input data from memory, parse it, and emit kv-pairs directly to their local accelerator. The accelerator then leverages its scratchpad memory to aggregate the pairs' values. Once all kv-pairs have been received and combined by the source node's accelerator, each source accelerator computes independently the partitioning of keys over the destination accelerators. It then proceeds in transferring all of its kv-pairs to the correct destination which, in turn, leverages its scratchpad memory to complete the reduce stage. Note that the same hardware module serves both the roles of source and destination accelerator simultaneously. Figure 2 shows the deployment of our distributed accelerators in a CMP architecture.

Each MR.NITRO's accelerator comprises a scratchpad memory, organized into two partitions, one to serve kv-pairs incoming from the local processor core (source scratchpad), and one to serve kv-pairs incoming from the interconnect (destination scratchpad). Each scratchpad is completed by a small "victim scratchpad", similar to a victim cache that stores data recently evicted from the main scratchpad. The accelerator also includes dedicated hardware to compute a range of reduce functions, used both to aggregate data in the source scratchpad and in the destination one. The hardware logic to compute hash functions, both for indexing the scratchpads and for partitioning kv-pairs over destination nodes, completes the design.

Note that both source and destination scratchpad memories are of fixed size, and as a result it may not be always possible for them to store all the kv-pairs that they receive. When a source scratchpad cannot fit all the kv-pairs, it defers their aggregation to the reduce stage, by transferring them to the destination accelerator. When a destination scratchpad encounters this problem, it transfers its kv-pairs to main memory, and then lets the destination core be in charge of carrying out the last few final steps of the reduce function. Both scratchpads resort first to a small victim scratchpad before folding into transferring out kv-pairs.

Figure 3 provides a schematic of the architecture described. 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 value from the key, and use it to index the scratchpad memory. The latter determines which destination node is responsible for reducing this kv-pair: if the local node is also the destination node (*node_is_dest*), then we send the kv-pair to the destination scratchpad, along with the hash index and an enable signal, otherwise we send it to the source scratchpad.

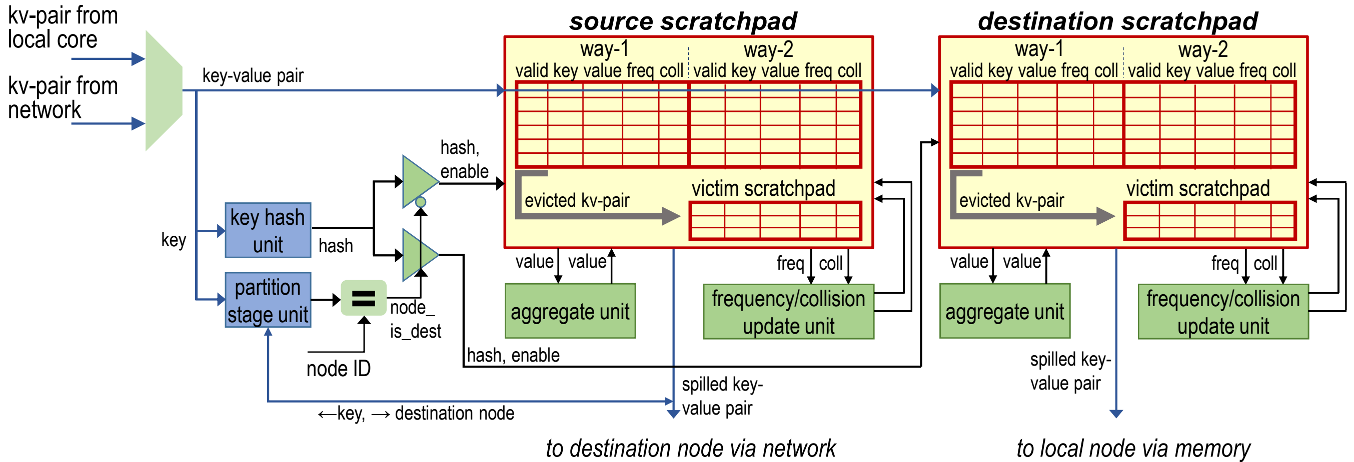

Figure 3: **MR.NITRO's accelerator architecture**. Each accelerator includes two separate *scratchpad memories*, organized as hashtables, to aggregate kv-pairs incoming from the local core or the interconnect. The *scratchpad memories* are organized as 2-way associative caches, augmented with a small *victim scratchpad*. The *aggregate units* are responsible for aggregating values for kv-pairs stored in the scratchpads. The *frequency/collision update units* enforce our novel kv-pair replacement policy. Finally, each accelerator includes a *key hash unit* to compute a hash value for each incoming key, and a *partition stage unit*, responsible for deriving the destination node ID in charge of applying the reduce function to each unique key. Note that kv-pairs evicted from the scratchpad memories are transferred out to either their destination node or memory, so that the destination node can complete the reduce stage.

Note that all kv-pairs incoming from the network will be aggregated at the destination scratchpad. In addition, some of the pairs incoming from the local core may also be aggregated at the destination scratchpad if the local core is both the source and the destination for that pair. Each scratchpad is internally organized as a 2-way cache augmented with a small victim cache. Associated with each scratchpad is an *aggregate unit*, responsible for deploying the specified reduce function to combine two kv-pairs with the same key. Each scratchpad 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 scratchpad, it is transferred out through the network, either to another compute node or to memory. kv-pair may be spilled because it is evicted from the victim scratchpad or because it collided with another entry in the scratchpad and did not qualify to replace it.

4. SYSTEM INTEGRATION

In our solution, each accelerator is tightly coupled with its local core. At the beginning of the execution, each core sends a start command to its accelerator and, based on the type of MapReduce application, it configures the accelerator to use one of the two hash functions we provide in the *key hash unit* (this setup could be easily extended to encompass a broad set of hash functions). The core also sends the initial address of a memory region that the accelerator shall use to store the reduced kv-pairs. The core can then begin to transmit mapped kv-pairs to its accelerator. It then waits for a completion signal from the accelerator, in the form of an interrupt (IPC), upon which it can retrieve the final, reduced kv-pairs from the shared memory region.

Note that the memory region that the core shares with the accelerator is also used as temporary storage for kv-pairs spilled from the destination scratchpad. When this situation arises, the processor core is responsible for completing the reduce function by aggregating the results from the accelerator with the spilled kv-pairs.

System configuration	Accelerator configuration
64 cores, 1GHz	maximum key size: 64 bits
L1 D & D cache size: 16KB	maximum value size: 32 bits
L1 D and I cache lat.: 1ns	scratchpads: 2-way set assoc.
L2 cache size: 128KB	# entries per scratchpad: 256
L2 cache latency: 12ns	scratchpad size = 18KB
memory type: ddr3_1600	# freq/coll counters: 8 bits

Table 1: **Characteristics of our architecture setup.**

If the accelerator requires additional temporary storage, it requests it to the processor core via an interrupt. All communication from a core to its local accelerator is carried out through store instructions to a set of memory-mapped registers within the accelerators, while accelerators communicate to the core via interrupts. Each accelerator is also directly connected to the network interface to send/receive kv-pairs to/from the other accelerators and the memory.

5. EXPERIMENTAL EVALUATION

To evaluate MR.NITRO, we developed two experimental setups: one for a baseline CMP architecture, and one for a CMP architecture augmented with MR.NITRO's hardware accelerators. To cope with the unreasonable amount of time and resources required to simulate big data workloads on cycle-accurate simulators, we developed a specialized experimental setup for our architectures. We modeled the baseline CMP solution as a 64-node CMP in an 8×8 mesh topology, with 64 cores and 4 memory nodes at the corners of the mesh. We created this design in the Gem5 simulation infrastructure. We ported the Phoenix++ framework [1] to Gem5 using "m5threads".

For our proposed solution, we still leveraged the Gem5 infrastructure. In addition, we modeled MR.NITRO's accelerators separately using Python. Our accelerator model is capable of providing a cycle-accurate simulation, tracking the state of the scratchpad memories, and the times of spilling events. It is used to simulate both the com-

bine/partition stages and the reduce stage. Finally, we used BookSim, a cycle-accurate network simulator to simulate the shuffle stage, matching the network configuration. Details of the architecture for both experimental setups are reported in Table 1.

Our MR.NITRO setup carries out each stage of MapReduce separately: first we simulate the map stage of each individual core using Gem5. We track these transfer events in simulation and generate a timed trace of kv-pairs emissions from the cores. Then we invoke our accelerator models to carry out the combine stage: we simulate all updates to the scratchpads and track all collisions and the times of kv-pair spilling events to the interconnect. We apply this simulation individually to each accelerator. We then simulate the interconnect exchanges during the combine (because of spilling) and the shuffle stages in BookSim. Finally, our accelerator model is used again to simulate the reduce stage and determine if there is any spilling to memory. In case of kv-pair spilling to memory, the last stage of aggregation is handled by the cores. Consequently, we determine the amount of time it takes to aggregate the spilled kv-pairs with those produced by the accelerators by simulating in Gem5 one more time.

Workload Characteristics. Table [2] provides information on all the workloads we considered in our evaluation.

name	description	#keys
hist	*histogram*: computes a histogram. Input data: an RGB bit image	768
lr	*linear regression*: linear data fitting. Input data: bidimensional points	5
wc	*wordcount*: counts the frequency of words. Input data: multiple documents.	68152-257225
pvc	*page view count*: counts the frequency of web page views. Input data: random page list.	10529
minmax	*min/max*: finds min/max. Input data: temperatures for all locations in a region	28514
avg	*average*: finds average. Input data: temperatures for all locations in a region	28514

Table 2: **Experimental workloads.** We selected these workloads because they represent a wide range of MapReduce applications.

Performance Evaluation. Figure 4 reports the speedup of MR.NITRO against the baseline CMP running Phoenix++. We report both the speedup that we could achieve with an infinitely large scratchpad, and that of the actual 256-entries scratchpad of our model. While the actual speedup varies with the application, ranging from 150% to 550%, the average speedup over all applications we studied is 320%.

histogram and *linear regression*: In the baseline solution without accelerators, these two applications do not use hash function computations because the number of their unique keys is small and known *a priori*, so a fixed-size array can carry out the combine stage. This optimization improves the performance of the baseline execution, reducing our room for speedup. The speedup of *linear regression* is relatively better because this application is less memory-intensive than *histogram*.

page view count: the speedup of this application is limited by the execution time of the map stage, which requires relatively more parsing than other applications.

A number of applications have speedups that are fairly

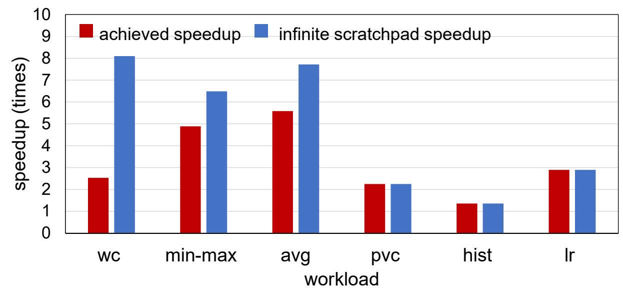

Figure 4: **MR.NITRO performance speedup for a range of workloads.** The chart plots the speedups over a baseline CMP execution of MapReduce, for our 256-entry scratchpad design and for an unbounded-size scratchpad design.

unaffected by the size of the scratchpads. However, a few, namely *wordcount*, *min/max* and *average* suffer from their limited size of the scratchpads, which leads to a lot of kv-pair spilling into the interconnect, thus overloading the reduce stage with additional aggregation and leveraging the slow processor cores for some of the reduce stage computation. This aspect is most pronounced in *wordcount*, which has the largest number of unique keys among all of our applications, sometimes by several orders of magnitude.

Area Overhead. The largest components of our accelerators are the two scratchpads. We modeled those and the cores's caches in Cacti and found that the area overhead of the accelerators' storage accounts only for 9.2% of the total storage.

6. CONCLUSIONS

MR.NITRO is a novel distributed hardware accelerator solution, capable of offloading the compute-intensive portion of MapReduce-based applications from the cores of a CMP to their local accelerators. The system is highly scalable with the number of cores. We found experimentally that our solution provides over a 3 times speedup on average over a pure CMP-based solution. We estimated the silicon footprint of MR.NITRO to account for less than 10% of the local cache storage.

7. ACKNOWLEDGEMENT

This work is sponsored in part by C-FAR, a funded center of STARnet, a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA.

8. REFERENCES

[1] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular MapReduce for shared-memory systems," in *Proceedings of the Second International Workshop on MapReduce and Its Applications*, 2011.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[3] M. Lu *et al*, "Optimizing the MapReduce framework on Intel Xeon Phi coprocessor," in *Big Data, 2013 IEEE International Conference on*, 2013.

[4] C. Kachris, G. Sirakoulis, and D. Soudris, "A reconfigurable MapReduce accelerator for multi-core all-programmable SoCs," in *System-on-Chip (SoC), 2014 International Symposium on*, 2014.