Resource-Aware MapReduce Runtime for Multi/Many-core Architectures

Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris
Microprocessors and Digital Systems Laboratory, ECE
National Technical University of Athens, Athens 15780, Greece
E-mail: {kiliakis, sxydis, dsoudris}@microlab.ntua.gr

***Abstract*—Modern multi/many-core processors exhibit high integration densities, e.g. up to several dozens or hundreds of cores. To ease the application development burden for such systems, various programming frameworks have emerged. The MapReduce programming model, after having demonstrated its usability in the area of distributed systems, has been adapted to the needs of shared-memory many-core and multi-processor systems, showing promising results in comparison with conventional multi-threaded libraries, e.g. pthreads. In this paper, we propose a novel resource-aware MapReduce architecture. The proposed runtime decouples map and combine phases in order to enhance the parallelism degree, while it effectively overlaps the memory-intensive combine with the compute-intensive map operation resulting in superior resource utilization and performance improvements. A detailed sensitivity analysis to the framework's tuning knobs is provided. The decoupled MapReduce architecture is evaluated against the state-of-art library into two diverse systems, i.e. a Haswell server and a Xeon Phi co-processor, demonstrating speedups on average up-to 2.2x and 2.9x respectively.**

***Index Terms*—MapReduce, Runtime Systems, Multi-cores**

I. Introduction

MapReduce (MR) forms a programming model coupled with an associated implementation for efficient processing of large datasets. Since MR was implemented [1], it has enjoyed much appreciation among the computing community, becoming the tool of choice for "big data" analytics frameworks [2], [3]. Among its advantages, is the ability to horizontally scale to petabytes of data, comprehensible programming semantics, and a high degree of fault tolerance. The success of MR has led to its adoption and continuous evolution on various platforms such as CPUs [4], [5], GPUs [6], and FPGAs [7] and towards differing optimization goals, i.e. energy-efficient MR architectures either in the form of software libraries [8] or as specialized hardware units for MR acceleration [9], [10].

Although MR was originally targeting cluster environments [1], the evolution of the computing infrastructures towards "fat" server implementations with tens of cores and hundreds of gigabytes of memory, the need for data analytics at the edge as well as the deeper understanding of "big data" workloads, challenge the assumption of MR applicability and profitability only for distributed cluster installations. Interestingly, Rowstron et al. [11] showed that at least two analytics production clusters (at Microsoft and Yahoo) have median job input sizes under 14 GB, and 90% of jobs on a Facebook cluster have input sizes under 100 GB. Thus, holding all data in-memory utilizing the advanced processing capabilities of current emerging multi-cores is not far-fetched.

A rich body of recent studies has targeted scale-up implementations of MR [4], [5], [12]–[15]. Processors integrating tens of cores have been commercialized and it is foreseeable that systems with higher densities will appear. Despite the various emerging programming frameworks, harnessing the potential of such systems remains challenging. For example, Phoenix [15] has shown that compared to hand-crafted pthreads implementations, MR improves applications' performance and scalability on shared-memory systems while simplifying the development process. However, existing implementations of MR runtimes for multi-cores are mainly focusing on exploiting architectural features of the underlying platforms, e.g. SIMD execution [16], cache hierarchies [17] or memory organization [4], silently neglecting the performance overheads due to resource contention of MR threads/tasks on the shared architectural resources. Recently, MR runtime optimizations for reducing the memory to disk communication in a single node are also investigated [12].

In this paper, we propose an efficient, resource-aware MR runtime that enables higher parallelism and resource utilization to be exploited over other state-of-art frameworks. We show that the typical structure of MR workflows and the inherent resource contention limit the execution's parallelism, especially for workloads that expose complementary characteristics. To address this limitation, we introduce the Resource-Aware MapReduce (RAMR) runtime that exploits fine-grained thread parallelism and effectively pipelines the map and combine phases, utilizing a low overhead, shared buffer. Despite fine-grained threading currently gaining a lot of attention [18], inter-thread data sharing introduces communication costs. RAMR takes into consideration the aforementioned issues by both i) reducing the contention on shared variables by enabling combiners to operate on blocks of elements and ii) minimizing the induced data exchange overhead, through a communication-aware thread-to-CPU pinning policy.

Although RAMR focuses mainly on optimizing the map-combine phase, it highly impacts the performance of MR workloads. Fig.1 reports the average run-time breakdown percentages of the de-facto suite for shared-memory MR [19] and we can see that the map-combine phase dominates the

This work has been partially funded by EU Horizon 2020 program under grant agreement No 825061 EVOLVE (https://evolve-h2020.eu/).

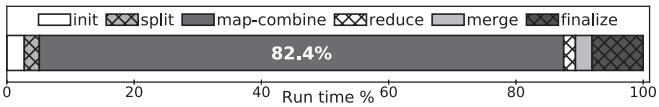


Fig. 1: Typical MapReduce application run time breakdown.

execution time (82.4%) of typical MR applications. We evaluate the efficiency of RAMR on two diverse multi/many-core systems, i.e. an Intel Haswell server and an Intel Xeon Phi co-processor, demonstrating speedups of up to 2.2x and 2.9x, respectively, compared to the state-of-art Phoenix++ [4] library. We further provide an in-depth sensitivity analysis over the most significant RAMR tuning knobs. Lastly, we identify the applications' characteristics that determine their suitability to RAMR, showing that the proposed solution performs better under more complex workloads and/or complementary workload types during the map/combine phases.

II. Related Work

In this paper, we focus on MR for shared-memory systems. The Phoenix library [15] adopted the traditional map, reduce, merge work-flow taking advantage of the shared memory property to communicate data between consecutive phases. Phoenix Rebirth [19] added NUMA-awareness and introduced the concept of combiners. Phoenix++ [4] improved combiners' functionality by applying the combine function after every map operation and explored various intermediate containers, each targeting a specific workload. Containers interface the map phase output with the reduce phase input and are responsible for grouping by key the emitted key-value pairs and storing them in a data-structure to be passed to the reducers.

Alternative shared-memory MR architectures have been proposed. Hone [20] designed an API and binary Hadoop compatible MR implementation for multicores, showing performance gains w.r.t. a 15-node Hadoop instance. Metis [5] focused on the container organization and developed an efficient data-structure that performs adequately for most applications. Tiled MR [17] argued that it is more efficient to iteratively process smaller chunks of data, than a large volume of data at once. Therefore in Tiled MR, a job is partitioned in smaller sub-jobs, which are processed and finally merged, reducing the memory footprint and leading to a more advantageous use of the memory hierarchy. MRPhi [16] adapted the MR framework to the needs of the Intel MIC architecture [21]. To utilize the wide vector units, the map phase was re-written in a SIMD friendly way. Map and reduce are pipelined to exploit the 4-way hyper-threading. Lastly, due to the limited memory resources, an atomically-accessed global container was favored instead of thread-local containers.

RAMR differentiates from the aforementioned frameworks in numerous ways. Map and combine are decoupled into separate phases, with their execution being overlapped in a pipelined fashion. Since the combine step has moved significant part of reducers' work to combine, the map/combine phase dominates the run-time, so overlapping map and combine is more prosperous than overlapping map and reduce, as suggested by MRPhi [16]. RAMR handles the inter-thread communication effectively by pinning co-operating threads

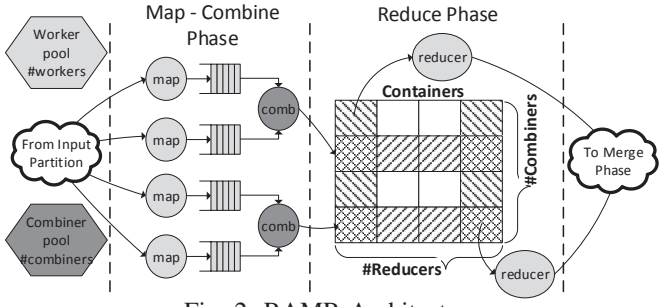


Fig. 2: RAMR Architecture

in close cores. Finally, RAMR is evaluated on two diverse systems, i.e. a high performance Xeon Phi co-processor and a NUMA Intel Haswell server.

III. RAMR Architecture

In this section, we describe the RAMR architecture. Fig. 2 depicts only the map-combine and reduce phases of RAMR, since the input partitioning and merging phases remain the same as in typical MR libraries [4], [19]. At first, the necessary initializations take place and two separate thread pools are instantiated. In Fig. 2, the top pool contains general-purpose workers and will be used to execute the tasks of map, reduce and merge. The thread pool on the bottom is used during the combine phase and contains a less or equal number of workers compared to the general-purpose pool.

The input partition phase follows, where the raw input is partitioned into splits using a user-specified partitioning function. The task size defines the number of splits that correspond to a task and is subject to tuning. Our analysis has shown that large task sizes result in substandard load balancing, while small task sizes result in non-negligible library overhead with regards to computation. In RAMR, the task size can be finely tuned via a set of environmental variables.

The map-combine phase begins after input partitioning. The map tasks are added in the task queues – one for each locality group. Map workers dequeue tasks from their local queue and apply the map function. The map function emits intermediate key-value pairs that would be, according to the traditional MR execution, directly combined and stored in the worker's container. Instead, in RAMR, to overlap the execution of map and combine functions, the intermediate data are pipelined through a set of Single-Producer/Single-Consumer (SPSC) queues. Combiners, operating concurrently with mappers, pop batches of key-value pairs out of the queue, apply the combine function and store the result in their private container. To allow multiple combiners to operate simultaneously, a separate container is allocated to each combiner.

When all input tasks have been processed, the map phase ends and the combiners are notified. Before exiting, combine workers consume any remaining data and empty their assigned queues. The rest MR execution remains unchanged.

A. Concurrent Shared Queues

Vital role for the effective pipelining and overlapping of map and combine phase plays the queue implementation, used to forward data from mappers to combiners.

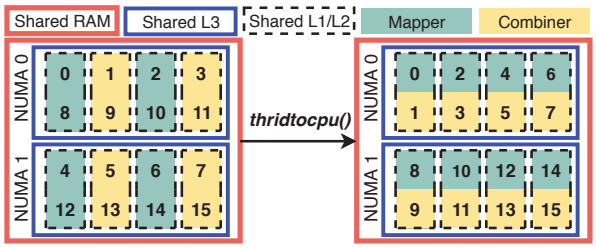


Fig. 3: The communication-aware pinning policy.

As shown in Fig. 2, every mapper writes in a separate queue so a single producer queue suffices. In addition, each combiner has explicit access to its set of assigned queues so a single consumer queue is also sufficient. Thus, a SPSC queue satisfies our specifications. A SPSC concurrent queue can be implemented without explicit synchronization mechanisms [22]. A fixed-size queue has been favored instead of a dynamically resizable queue because of the limited scalability and performance penalty imposed by dynamic memory allocators available in typical operating systems [23], [24].

After benchmarking several SPSC buffers in terms of concurrent read-write throughput, we settled on using the static-allocation queue implementation found in the Boost library [25]. Furthermore, with fine-tuning we discovered that a maximum capacity of five thousand elements achieves near-optimal (within 2%) performance across all test-cases. Building on top of Boost SPSC, we implemented additional features to improve the overall performance of RAMR.

Sleep on failed push. Pushing elements in the queue always succeed; discarding or overwriting elements violates correctness. This means that mappers that attempt to push data on a full queue, will have to block until space is freed. A busy-wait loop was originally used for this purpose. Instead, letting the mappers sleep after a failed trial improves runtime.

Batched reads. Instead of letting combiners consume single elements, the buffer is divided into blocks of elements that are processed contiguously. This reduces the mappers/combiners congestion on shared control variables and favors locality. We benchmark the effect of this optimization in section IV-C.

B. Resource-Contention Aware Pinning Policy

The extensive communication between mappers and combiners is posing additional pressure on the memory subsystem. In the absence of a careful design, this resource contention overhead can overshadow any potential performance gains.

To tackle this, we develop a contention-aware thread pinning policy. According to the ratio of mapper-to-combiner threads, a set of mapper queues is assigned to each combiner. This ratio is application dependent and is driven by the throughput (in processed elements/second) of the map and combine functions. For instance, a workload with equivalent map and combine processing rate requires equal number of mapper and combiner threads to operate steadily. Then, based on the system's layout, a positioning of threads to CPUs is devised that places combiners together with their assigned mappers in contiguous logical cores. Finally, using the *setaffinity()* system call, threads are pinned to CPU throughout the MR invocation.

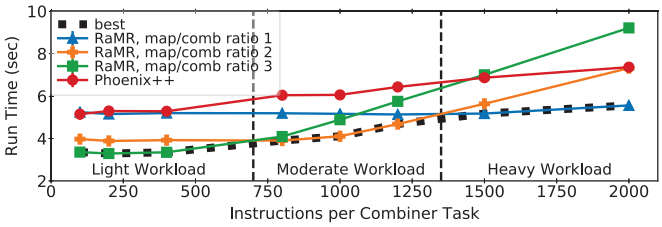


Fig. 4: Combine workload impact on the map/combine ratio.

Fig. 3 shows an example of our pinning policy in a CPU, similar to the one used in the experimental evaluation, with two NUMA nodes, two-way hyper-threading and four cores per NUMA node. We assume that the application requires a mappers/combiners ratio equal to one. The original mapping of CPU ids to physical resources is on the left. The function *thridtocpu()* re-maps the CPU ids to a sequence of numbers closely coupled in the physical layout. As a result, the mapper-combiner pairs $(2i, 2i+1), i \in 0..7$ communicate through the shared L1/L2 cache. Additionally, since the mapper-combiner pairs share the same physical core, the RAMR pinning policy enables more efficient resource utilization when the two phases exhibit complementary behaviour, i.e. CPU-intensive map with memory-intensive combine.

C. Workload-Aware Synthetic Test-Suite

The performance of RAMR is sensitive to the map/combine workload type and duration. In order to evaluate RAMR under variable map/combine workload combinations, we implemented a synthetic test-suite that allows for easy configuration of the type and intensity of the map and combine phases. We targeted CPU and memory-intensive workloads. A CPU-intensive workload is generated by combining computationally heavy trigonometric and exponential functions, which access contiguous, small datasets. On the contrary, to generate a memory-intensive workload, computationally light operations are applied on wide datasets with non-regular access pattern.

Fig. 4 demonstrates a use-case of the synthetic test suite. The map task is CPU-intensive with fixed intensity while the combine task is memory-intensive with variable intensity. The X-axis denotes the number of instructions per combine task (higher values correspond to heavier workloads) and the Y-axis shows the run time. The dashed line follows the best configuration in every region. We observe that for light combine workloads, a higher ratio performs best, meaning that a single combiner can handle multiple mappers (three in this case). Then, for a moderate combine workload, the optimal mappers/ combiners ratio drops to two. Lastly, for a heavy combine workload the optimal configuration has a ratio of one, i.e. equal number of combiners and mappers. The Phoenix++ [4] run-time has been included in Fig. 4, showing that in this synthetic benchmark with fixed CPU-intensive map and memory-intensive combine, RAMR performs better.

IV. Experimental Evaluation

A. Experimental Setup

Contrary to other MR runtimes, RAMR is experimentally evaluated on two systems with inherently diverse character-

TABLE I: Input sizes used in the experimental evaluation.

Test-case	Small		Medium		Large	
	HWL	PHI	HWL	PHI	HWL	PHI
HG	400MB	200MB	800MB	400MB	1.6GB	800MB
KM	400K	200K	800K	400K	2M	800K
LR	400MB	200MB	800MB	400MB	1.6GB	800MB
MM	500	300	800	500	1000	800
PCA	$2K \times 2K$		$3K \times 2K$		$4K \times 4K$	
WC	200MB	200MB	400MB	400MB	1GB	600MB

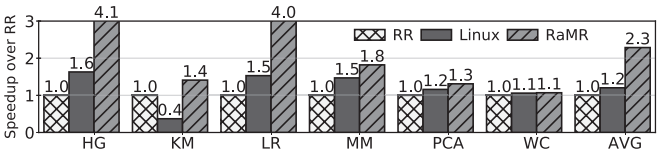


Fig. 5: Speedup of RAMR over the baseline policies.

istics – a multi-core, NUMA, server platform and a high performance many-core co-processor. Specifically:

Multi-core server. We used an Intel Haswell micro-architecture, dual-socket server, with 14 cores-per-socket, 2-way hyper-threading, 35MB of L3 cache per socket and 64GB of RAM. Each socket forms a separate NUMA node with 28 logical cores, so the system can run a total of 56 threads.

Many-core co-processor. The Xeon Phi co-processor is a relatively recent product family. The platform used for the measurements consists of an Intel Xeon processor connected to the Xeon Phi co-processor with 57 cores @1.1GHz, 28.5MB L2 cache and 6GB of RAM. At full capacity, Xeon Phi can run 228 hardware threads. All the applications were compiled on the host and executed natively on the co-processor.

The Phoenix++ [4] benchmark suite was used for the experimental evaluation to allow for direct comparison against RAMR. The applications derive from a wide range of computing domains such as enterprise (Word Count), scientific (Matrix Multiply[1]), artificial intelligence (KMeans, PCA, Linear Regression) and image processing (Histogram).

The input sizes used are summarized in Table I. As a system with greater potential, the Haswell setup was tested under heavier inputs than Xeon Phi. Nevertheless, the input sizes used in the evaluation of both platforms are larger or comparable with the sizes used by other competitive libraries. All three input flavors were used for the comparison of RAMR against Phoenix++ [4], while the the large input flavor was used for the intermediate analysis. Every plotted value corresponds to the average of 20 runs, with a standard deviation of $\approx 1\%$.

B. Contention-Aware Pinning Policy

In Section III-B we presented the RAMR thread pinning policy aware of the co-operation pattern between mapper and combiner threads. We consider two more thread scheduling policies for comparison: (i) Round-robin (RR), that pins threads to cores in a round-robin fashion without considering their role, and (ii) the default scheduler of the Linux kernel that allows thread migrations.

Fig. 5 shows the execution time speedup compared to RR for the six test-cases on the Haswell setup. The RAMR policy minimizes the distance in logical core units of co-operating threads, hence manages to keep within the same socket co-operating threads and reduce the communication overhead. RAMR achieves an average 2.28x and 2.04x speedup compared to RR and the Linux scheduler, respectively. In Fig. 5, we see that in HG and LR RAMR is exceptionally faster than the baseline. Due to their computationally "light" workload (see section IV-E), HR and LR are particularly sensitive to the queue communication overhead. Hence, the RAMR pinning policy significantly improves their performance.

[1]Matrix Multiply has been adapted to utilize the Map/Reduce semantics

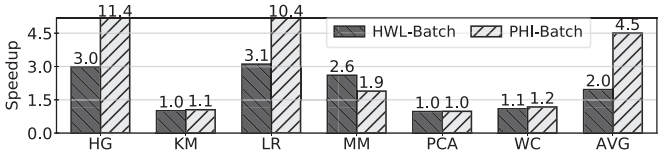
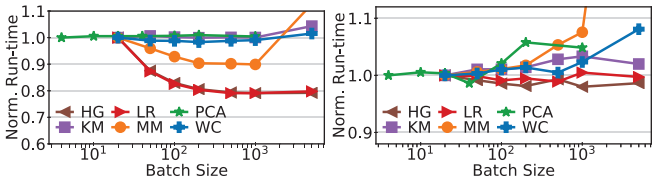


Fig. 6: Speedup of batched over single consume



(a) Intel Haswell (b) Intel Xeon Phi

Fig. 7: Batch size sensitivity

In Xeon Phi, a bidirectional ring inter-connects the memory-controllers of all cores. The L2 caches of each core contribute to a universally shared L2 cache. As a result, the performance gains from the optimized thread pinning policy were limited to 1–3% compared to both the RR policy and the Linux scheduler. Nevertheless, the RAMR policy outperforms the baseline policies in all available test-cases.

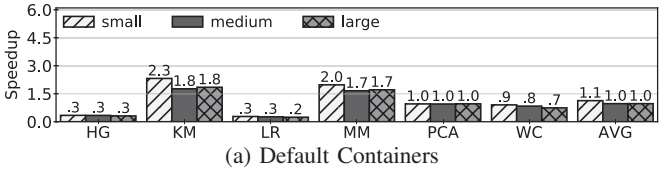
C. Batch Size Sensitivity Analysis

The batched consume method applies a functor f to *batch_size* contiguous elements, given that there are enough elements in the queue. Performing operations in batches benefits from spatial locality and reduces contention on shared control variables. The speedup of the batched consume method is depicted in Fig. 6 for Haswell (HWL) and Xeon Phi (PHI). The performance gain is remarkable, with speedups of up to 11.4x on Xeon Phi and up to 3.1x on Haswell.

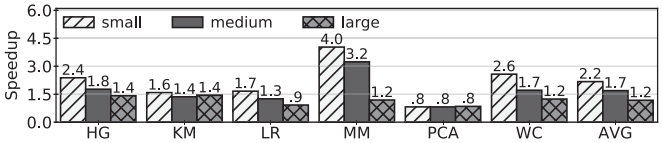
The batch size defines the number of elements processed contiguously by each consume operation. Larger batch sizes may increase combiners' idle time as combiners cannot start operating unless the queues are sufficiently filled. On the other hand, processing large amounts of contiguous data improves spatial locality. Fig. 7 shows the sensitivity of each test-case to the batch size on both platforms. The Y-axis shows the execution time normalized to the first data point of each curve. For Haswell, we observe that all applications profit from a 1000 elements batch size. In Xeon Phi, the test-cases profit from smaller batch sizes, ranging from 20 to 500 elements due to the much smaller amount of cache capacity per thread.

D. Performance Evaluation

Haswell Server. Fig. 8a, shows the RAMR execution time speedup over Phoenix++ for variable input sizes, when using

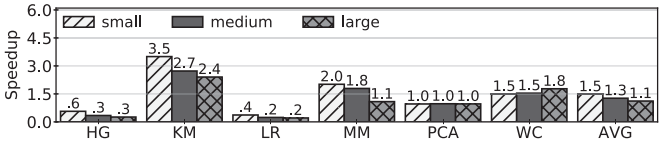


(a) Default Containers

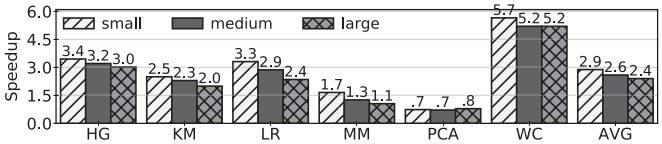


(b) Hash Containers

Fig. 8: RAMR Speedup over Phoenix++ on Haswell



(a) Default Containers



(b) Hash Containers

Fig. 9: RAMR Speedup over Phoenix++ on Xeon Phi

the default containers for every application. The default container for all applications is a thread-local fixed array structure as the range of keys is known a-priory, except WC that uses thread-local hash tables, which is more suitable for storing an arbitrary set of keys. KM and MM profit from RAMR and demonstrate speedups of 1.95x and 1.77x on average, respectively. PCA performs similarly to Phoenix++ and WC is slightly slower by 21.6%. However, HG and LR are outperformed by a factor of 3x and 3.8x on average, respectively. As we will explain in Section IV-E, this slowdown comes in agreement with our analysis due to the relatively "light" workload, causing the overhead induced by the shared-queue to dominate the run-time.

To stress the memory intensity of the combine phase we replace the containers with fixed-size hash tables in HG, KM, LR and WC, and regular hash tables in MM and PCA. The memory intensity is increased due to the hash calculation, dynamic memory allocation for new keys and non-regular data access. The results are summarized in Fig. 8b. RAMR outperforms Phoenix++ in 5 out of 6 testcases and is 1.57x faster on average. MM exhibits the highest speedup of 2.46x while PCA is outperformed by 20.4%.

Xeon Phi. Fig. 9a summarizes the RAMR run-time speedup against Phoenix++ for various input sizes (Table I), using the default intermediate containers. WC performs better by 1.59x on average, KM by 2.8x and MM by 1.52x. PCA performs similarly, while HG and LR are outperformed by 2.84x and 2.87x on average, respectively. The reason is their "light" workload, that lets the queue overhead overshadow any other benefits (see Section IV-E).

Again, to increase the workload intensity and stress the memory sub-system, in Fig. 9b, fixed-size hash containers were used in HG, KM, LR and WC, and regular hash containers in MM and PCA. The performance of RAMR improved compared to Phoenix++. In 5 out of 6 applications, RAMR is faster, with a maximum speedup of 5.34x and an average speedup of 2.6x.

E. Applications' Suitability to RAMR

Not every application benefits from RAMR. In this section, we reason about the applications' characteristics that influence their suitability to RAMR. The analysis is based on the following metrics:

$$ipb = \frac{inst.}{input(B)}, mspi = \frac{mem.stall}{inst.}, rspi = \frac{res.stall}{inst.}$$

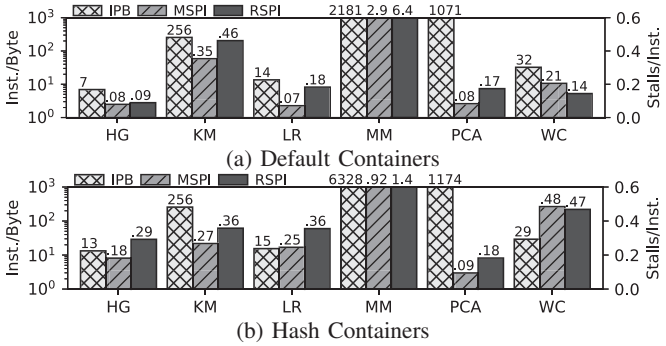
IPB is used to quantify the workload intensity. MSPI and RSPI represent the frequency of stalls due to the memory subsystem and lack of core resources, respectively. More specifically, memory-related stalls include stalled cycles due to L1 and L2 cache misses. Resource-related stalls can occur due to full ROB, no eligible RS entries or no space in the load/store buffer. All three metrics are only meaningful when used comparatively.

The suitability analysis is based on the following line of thought. RAMR decouples map and combine and lets them operate concurrently on separate threads. The data are pipelined through a set of SPSC queues from mappers to combiners. This introduces the cost of thread communication and queue manipulation. As a result, *lightweight applications, i.e. applications with low IPB value, are not likely to benefit from RAMR as the added cost dominates their run-time.*

On the other side, a high IPB value alone is not enough to indicate a well-fitted application to RAMR. If an application is not experiencing any stalls due to memory or core resources, then decoupling map and combine into separate threads, would not differ from executing map and combine sequentially on each thread, with an additional communication overhead. *In order to profit from a decoupled workload, the two decoupled parts should have complementary (CPU and memory/ resource-intensive) characteristics*, which is the case for many MR applications. MSPI and RSPI quantify the stalls' frequency due to memory and core resources. Applications with high MSPI/ RSPI values indicate sub-optimal hardware utilization and are good candidates for a decoupled and pipelined work-flow. As a consequence, *workloads with sufficient complexity, that suffer from frequent memory or resource stalls, are ideal candidates for our runtime.*

Fig. 10a presents the IPB, MSPI and RSPI metrics for the six target test-cases. The metrics were calculated based on hardware counters and concern the map/combine phase only.

The left Y-axis is used for the IPB, while the right one is used for MSPI and RSPI. For the intermediate key-value storage, the default container was used, which is a fixed-size array for all applications except WC that used a hash table. We observe that HG and LR are not good candidates for RAMR, as they demonstrate light workload and few stalls. On the other hand, KM and MM are suitable to RAMR, as their workload is complex and suffers frequently from stalled cycles. For PCA



(a) Default Containers

(b) Hash Containers

Fig. 10: Evaluating applications' suitability to RAMR.

and WC no safe conclusions can be drawn. PCA features high IPB value but rare stall cycles while in WC the metrics are not clearly indicating whether it will profit or not from RAMR.

In Fig. 10b the intermediate containers are replaced by fixed-size hash tables in HG, KM, LR and WC, and regular hash tables in MM and PCA. The use of hash containers, adds the hash calculation overhead, requires dynamic memory allocation and enforces a non-regular access pattern. Hence, an increase in the IPB, MSPI and RSPI metrics is expected. WC is a reasonable exception considering that in Fig. 10a a hash container was also used, so the hash table overhead has been already counted. In MM, stalls are reduced because when a fixed-size array is used, each worker thread allocates an array of sufficient capacity to store every element of the output array. However, only a small part of it is used as each mapper computes values for a limited range of keys. When switching to a hash table, the size of the container is adjusted so that it fits only the essential key-value pairs. This improves cache locality and reduces memory and resource stalls. For the same reason, KM has also slightly improved MSPI and RSPI metrics. In general, a KM map worker may not encounter points belonging to every cluster, so pre-allocating an array of size equal to the number of clusters is not necessary.

Examining Fig. 10b, we conclude that KM, MM and WC are suitable to RAMR, as they have both enough instruction intensity and frequent stalls. HG and LR experience memory and resource related stalls often, but still their workload complexity is relatively low. PCA will practically demonstrate the same behavior as with the default array container, because although the workload complexity is sufficient, the number of resource and memory stalls is very low.

Overall, the suitability analysis provided above is in good agreement with the reported, experimental results.

V. Conclusion

This paper presents RAMR, an alternative MR runtime for shared-memory systems. RAMR effectively decouples map and combine tasks in a resource-aware manner, eliminating the inherent inefficiency of the traditional MR runtime that requires serialization of map and combine tasks. The performance of RAMR was evaluated against the state-of-art Phoenix++ library on two multi/many-core platforms with diverse characteristics. We show that while in low intensity workloads, RAMR behaves similarly to Phoenix++, for increased workload intensity RAMR achieves average speedups of up-to 2.2x and 2.9x in Xeon Haswell and Xeon Phi, respectively, favoring applications with fairly complex workload that suffer from frequent resource or memory-related stalls.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, 2008.

[2] K. Shvachko, H. Kuang, S. Radia, R. Chansler, *et al.*, "The hadoop distributed file system.," in *MSST*, vol. 10, pp. 1–10, 2010.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, 2010.

[4] 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*, pp. 9–16, ACM, 2011.

[5] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing mapreduce for multicore architectures," in *Computer Science and Artificial Intelligence Laboratory, MIT, Tech. Rep*, Citeseer, 2010.

[6] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *IEEE TPDS*, vol. 22, 2011.

[7] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *18th annual ACM/SIGDA IS on FPGAs*, pp. 93–102, 2010.

[8] K. Duraisamy, R. G. Kim, W. Choi, G. Liu, P. P. Pande, R. Marculescu, and D. Marculescu, "Energy efficient mapreduce with vfi-enabled multicore platforms," in *2015 52nd DAC*, IEEE, 2015.

[9] A. Addisie and V. Bertacco, "Collaborative accelerators for in-memory mapreduce on scale-up machines," in *Procs of the 24th Asia and South Pacific Design Automation Conf.*, ACM, 2019.

[10] H. Xiao, H. Zhang, F. Ge, and N. Wu, "A mapreduce architecture for embedded multiprocessor socs," *IEICE Electronics Express*, 2016.

[11] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using hadoop on a cluster," in *HotCDP '12*, (NY, USA), ACM, 2012.

[12] G. Kaur, K. Vora, S. C. Koduru, and R. Gupta, "Omr: Out-of-core mapreduce for large data sets," in *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2018, (New York, NY, USA), pp. 71–83, ACM, 2018.

[13] A.-L. Georgiadis, S. Xydis, and D. Soudris, "Deploying and monitoring hadoop mapreduce analytics on single-chip cloud computer," in *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*, ACM, 2016.

[14] I. El-Helw, R. Hofman, and H. E. Bal, "Glasswing: Accelerating mapreduce on multi-core and many-core clusters," in *Procs of the 23rd IS on High-performance parallel and distributed computing*, ACM, 2014.

[15] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *IEEE 13th International Symposium on HPCA.*, pp. 13–24, 2007.

[16] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh, "Optimizing the mapreduce framework on intel xeon phi coprocessor," in *IEEE IC on Big Data*, 2013.

[17] R. Chen, H. Chen, and B. Zang, "Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling," in *Proc. of the 19th international conference on PACT*, ACM, 2010.

[18] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-aware thread management," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, (Berkeley, CA, USA), pp. 145–160, USENIX Association, 2018.

[19] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *IEEE International Symposium on Workload Characterization.*, pp. 198–207, 2009.

[20] K. A. Kumar, J. Gluck, A. Deshpande, and J. Lin, "Hone: Scaling down hadoop on shared-memory systems," *Proc. VLDB Endow.*, Aug. 2013.

[21] A. Duran and M. Klemm, "The intel many integrated core architecture," in *Int. Conf. on HPC and Simulation (HPCS)*, IEEE, 2012.

[22] L. Lamport, "Correctly executes multiprocess program," *IEEE transactions on computers*, vol. 28, no. 9, 1979.

[23] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *Proc. of the 9th Int. Conf. ASPLOS*, 2000.

[24] S. Xydis, A. Bartzas, I. Anagnostopoulos, D. Soudris, and K. Z. Pekmestzi, "Custom multi-threaded dynamic memory management for multiprocessor system-on-chip platforms," in *SAMOS Int. Conf.*, 2010.

[25] T. Blechmann, "boost::lockfree::spsc_queue," 2013.