Optimizing Power and Performance Trade-offs of MapReduce Job Processing with Heterogeneous Multi-Core Processors

Feng Yan[1,2], Ludmila Cherkasova[1], Zhuoyao Zhang[3], and Evgenia Smirni[2]

[1] Hewlett-Packard Labs, lucy.cherkasova@hp.com
[2]College of William and Mary, fyan,esmirni@cs.wm.edu
[3]University of Pennsylvania, zhuoyao@seas.upenn.edu

***Abstract*—Modern processors are often constrained by a given power budget that forces designers to consider different trade-offs, e.g., to choose between either many slow, power-efficient cores, or fewer faster, power-hungry cores, or to select a combination of them. In this work, [1], we design and evaluate a new Hadoop scheduler, called DyScale, that exploits capabilities offered by heterogeneous cores within a single multi-core processor for achieving a variety of performance objectives. A typical MapReduce workload contains jobs with different performance goals: large, batch jobs that are throughput oriented, and smaller interactive jobs that are response-time sensitive. Heterogeneous multi-core processors enable creating virtual resource pools based on the different core types for multi-class priority scheduling. These virtual Hadoop clusters, based on "slow" cores versus "fast" cores can effectively support different performance objectives that cannot be achieved in a Hadoop cluster with homogeneous processors. Using detailed measurements and extensive simulation study we argue in favor of heterogeneous multi-core processors as they provide performance means for "faster" processing of the small, interactive MapReduce jobs (up to 40% faster), while at the same time offer an improved throughput (up to 40% higher) for large, batch job processing.**

***Keywords*-MapReduce; Hadoop; heterogeneous systems; scheduling; performance; power.**

I. Introduction

To offer diverse computing capabilities, the emergent modern system on a chip (SoC) might include heterogeneous cores that execute the same instruction set while exhibiting different power and performance characteristics. The current SoC design is often driven by a given power budget and needs to exploit a variety of choices within the same power envelope. These power constraints force designers to consider and analyze differrent decision trade-offs, e.g., to choose between either many slow, low-power cores, or fewer faster cores (which consume much more power per core), or to select a combination of them as shown in Figure 1.

Figure 1. Choices in the SoC design with the same power budget.

Intuitively, an application that needs to support a higher throughput and that is capable of partitioning and distributing its workload across many cores, favors a processor with many slow cores. However, the latency of time-sensitive application depends on the speed of its sequential components and benefits from a processor with fewer, faster cores to speedup the sequential parts of the computation. A SoC design with heterogeneous cores could offer the best of both worlds for applications that can take an advantage of these heterogeneous processing capabilities.

The MapReduce programming model and its open source implementation Hadoop offer a scalable and fault-tolerant framework for processing large data sets. MapReduce jobs are automatically parallelized, distributed, and executed on a large cluster of commodity machines. Originally, Hadoop was designed for batch-oriented processing of large production jobs. These applications belong to a class of so-called scale-out applications, i.e., their completion time can be improved by using a larger amount of resources. For example, Hadoop users apply a simple rule of thumb [1]: processing a large MapReduce job on a double size Hadoop cluster can reduce the job's completion time twice. Thus, efficient processing of such jobs is "throughput-oriented" and can be significantly improved with additional (scale-out) resources.

However, when multiple users are sharing the same Hadoop cluster, there are many interactive ad-hoc queries and small MapReduce jobs that are completion-time sensitive. Moreover, a growing number of MapReduce applications, e.g., personalized advertising, sentiment analysis, spam detection, real-time event log analysis, require completion time guarantees and are deadline-driven. For improving the execution time of small MapReduce jobs, scale-out does not work. Rather, one could use a "scale-up" approach, where the tasks comprising such jobs are executed on "faster" resources.

A typical MapReduce processing pipeline is disk-bound (for small and medium Hadoop clusters) and could become network-bound for larger Hadoop clusters. Therefore, it is unclear whether under normal circumstances a typical MapReduce application may benefit from processors with faster cores. To answer this question we performed experiments with a diverse set of MapReduce applications in a Hadoop cluster that employs the latest Intel Xeon quad-core processor (it offers a set of controllable CPU frequencies varying from 1.6 Ghz to 3.3 Ghz, and each core frequency can be set separately). While the achievable speedup across different jobs varies, the majority of jobs achieve speedup of 1.6-2.1 thanks to the code exhibits an improved performance when it is executed by faster processors. Therefore, the heterogeneous multi-core processors that have both fast and slow cores become an interesting design point for supporting

[1]This work was completed during F. Yan's internship at HP Labs. E.Smirni and F. Yan are partially supported by NSF grants CCF-0937925 and CCF-1218758.

different performance objectives of MapReduce jobs.

In this work, we design and evaluate a new Hadoop scheduler, called DyScale, that exploits capabilities offered by heterogeneous cores. It enables creating virtual resource pools based on the core types for multi-class priority scheduling. We describe new mechanisms for enabling "slow" and "fast" slots in Hadoop and for creating the corresponding virtual clusters. Extensive simulation experiments demonstrate the efficiency and robustness of the proposed framework. Within the same power budget, the DyScale scheduler that operates on the heterogeneous multi-core processors provides better performance for small, interactive jobs compared to using homogeneous processors with (many) slow cores. DyScale reduces the average completion time of time-sensitive interactive jobs by more than 40% while preserving good performance for large batch jobs. The considered heterogeneous configurations can reduce the completion time of batch jobs up to 40% as shown in our simulations.

There is a list of interesting opportunities for improving MapReduce processing offered by the heterogeneous processor design. First, both *fast* and *slow* Hadoop slots have a similar access to the underlying HDFS data. This eliminates the data locality issues that could make the heterogeneous Hadoop clusters comprised of the *fast* and *slow* servers[2] being inefficient [2], that is any dataset can be processed by either *fast* or *slow* virtual resource pools, or their combination. Second, the task (job) migration between slow and fast cores enables enhanced performance guarantees and more efficient resource use. To achieve the above, following challenges should be addressed: *i)* an implementation of new mechanisms in support of dynamic resources allocation, such as migration and virtual resource pools, *ii)* support for accurate job profiling (especially, when a job/task is executed on a mix of fast and slow slots), *iii)* analysis of per job performance trade-offs for making the right optimization decisions, and *iv)* increased management complexity. The remainder of the paper presents our results in more detail.

II. Background and Motivating Example

In the MapReduce model [3], computation is expressed as two functions: map and reduce. MapReduce jobs are executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*. The map and reduce tasks are executed by *map slots* and *reduce slots*.

In the *map stage*, each map task reads a split of the input data, applies the user-defined map function, and generates the intermediate set of key/value pairs. The map task then sorts and partitions these data for different reduce tasks according to a partition function.

In the *reduce stage*, each reduce task fetches its partition of intermediate key/value pairs from all the map tasks and sorts/merges the data with the same key (it is called the shuffle/sort phase). After that, it applies the user-defined reduce function to the merged value list to produce the aggregate results (it is called the reduce phase). Then, the reduce outputs are written back to a distributed file system.

Job scheduling in Hadoop is performed by a master node called JobTracker, which manages a number of worker nodes in the cluster. Each worker node in the cluster is configured with a fixed number of map and reduce slots, and these slots are managed by the local TaskTracker. The worker TaskTracker periodically connects to the master JobTracker to report current status and the available slots. The JobTracker decides the next job to execute based on the reported information and according to a scheduling policy. The popular job schedulers include FIFO, Hadoop Fair scheduler (HFS) [4], and Capacity scheduler [5]. The assignment of tasks to slots is done in a greedy way: assign a task from the selected job J immediately whenever a worker reports to have a free slot. At the same time, a data locality consideration is taken into account: if there is a choice of available slots in the system to be allocated to job J, then the slots that have data chunks of job J locally available for processing are given a priority in the assignment process [4].

If the number of tasks belonging to a MapReduce job is greater than the total number of slots available for processing the job, the task assignment takes multiple rounds, which are called *waves*. The Hadoop implementation includes *counters* for recording timing information such as start and finish timestamps of the tasks, or the number of bytes read and written by each task. These counters are sent by the worker nodes to the master node periodically with each heartbeat and are written to logs after the job is completed.

A. Motivating Example: Scale-out vs. Scale-up Approaches

According to a workload characterization based on Facebook and Yahoo experience [4], [6], a typical MapReduce workload can be described as a collection of "elephants" and "mice" jobs. Table II-A shows the number of map and reduce tasks and the percentage of these jobs in the Facebook workload [4]. In the table, different jobs are grouped into different bins based on their size in terms of number of map and reduce tasks. From the table, we can see most jobs are quite small (mice), e.g., 88% of jobs have less than 200 map tasks. At the same time, there is a small percentage of the large jobs (elephants) with up to thousands of map tasks and hundreds of reduce tasks.

These two different types of jobs may favor different design choices of multi-core processors. For example, the large jobs may benefit from processors with many slow cores to get a better throughput, i.e., to execute as many tasks in parallel as possible, and as a corollary, to achieve a better job completion time. While the small jobs may prefer the choice of processors with fewer fast cores for speeding-up their tasks execution and getting an improved job completion time. Therefore, the heterogeneous multi-core processors may offer an interesting design point because they bring a potential opportunity to achieve a win-win situation for both types of MapReduce jobs.

[2]Note, that the concept of heterogeneous cores within a single processors is very different from the heterogeneous servers. The Hadoop clusters that include the heterogeneous servers do face a variety of performance issues with balanced data placement across the heterogeneous servers.

Table I
JOB DESCRIPTION FOR EACH BIN IN FACEBOOK WORKLOAD.

Bin	Map Tasks	Reduce Tasks	# % Jobs
1	1	NA	38%
2	2	NA	16%
3	10	3	14%
4	50	NA	8%
5	100	NA	6%
6	200	50	6%
7	400	NA	4%
8	800	180	4%
9	2400	360	2%
10	4800	NA	2%

MapReduce applications are scale-out by design, which means the completion time is improved when more slots are allocated to the job see Figure 2. The job's scale-out is usually limited by the total number of slots in the system and the parallelism of the job. MapReduce applications may also benefit from "scale-up", e.g., a job may complete faster by using faster cores. The interesting question is how different MapReduce jobs may benefit from scale-out and scale-up of the slots in the system. To understand the possible trade-offs, let us consider the following example.

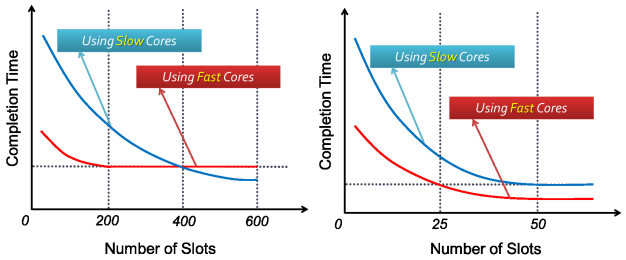
Motivating Example. Assume we have a Hadoop cluster with 100 nodes, each node with **two fast** cores and **six slow** cores. We configure the Hadoop cluster with one map and one reduce slots per core. The slots that are executed on the fast or slow cores are called *fast* and *slow slots* respectively. Therefore, the system has **200 fast** map (reduce) slots and **600 slow** map (reduce) slots in total.

Let us consider the following two jobs:

- *Job1* with 4800 map tasks (e.g., similar to jobs in the 10th group as shown in Table II-A),
- *Job2* with 50 map tasks (e.g., similar to jobs in the 4th group as shown in Table II-A).

Let a map task of *Job1* and *Job2* require time T to finish with a *fast* slot, and $2 \cdot T$ to execute with a *slow* slot (i.e., a fast slot is 2 times faster than a slow slot). Let us look at the job completion time of *Job1* and *Job2* as a function of the increasing number of fast or slow slots that can be allocated to the job during its execution.

The scenarios that reflect possible executions of *Job1* and *Job2* are shown in Figure 2(a) and Figure 2(b) respectively. The graphs in Figure 2 were drawn based on calculations [7] and illustrate that both jobs achieve better (lower) completion times with a higher number of slots (either fast or slow) allocated to the jobs during their execution. When a large *Job1* is executed with fast slots, and *all 200 fast* slots are allocated to the job then it leads to the following completion time: $4800 \cdot T/200 = 24 \cdot T$, i.e., it takes 24 rounds and each round takes T time. Since there are only 200 fast slots in the cluster – the remaining part of the red curve stays flat. The best job completion time is achieved when using *all 600 slow* slots. In this case, *Job1* finishes with the following time: $4800 \cdot 2 \cdot T/600 = 16 \cdot T$, i.e., it takes 8 rounds and each round takes $2 \cdot T$ time. The job completion time with slow slots is 30% better than with the fast slots available in the cluster, i.e., using a larger number of slow slots leads to a better completion time for this job.



(a) Large *Job1* Processing. (b) Small *Job2* Processing.

Figure 2. Processing MapReduce jobs *Job1* and *Job2* with *slow* or *fast* slots available in the cluster.

We can see that a small *Job2* shown in Figure 2 (b) cannot take advantage of more than 50 slots (either slow or fast) available in the cluster, because it only has 50 tasks. *Job2* achieves the best completion when using *50 fast* slots.

Therefore, a Hadoop cluster based on the heterogeneous multi-core processors (that exhibit different computing capabilities and power/performance ratios) could offer a new, flexible framework for supporting MapReduce jobs with different performance objectives.

III. DYSCALE FRAMEWORK

In this section, we present the implementation details and features of a new DyScale framework. First, we describe the DyScale scheduler that enables creating statically configured, dedicated virtual resource pools based on different types of available cores. Then, we present the enhanced version of DyScale that allows the shared use of spare resources among the existing virtual resource pools.

A. Dedicated Virtual Resource Pools

We would like to offer to the users the ability to schedule jobs based on the performance objectives and resource preferences. For example, a user can submit the small, time-sensitive jobs to the *Interactive Job Queue* to be executed by fast cores and the large throughput-oriented jobs to the *Batch Job Queue* for processing by (many) slow cores. This scenario is shown in Figure 3.

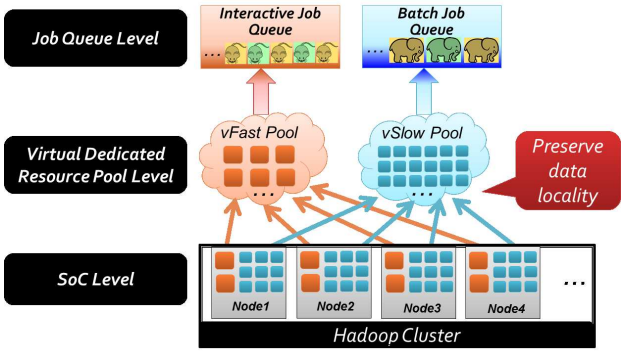


Figure 3. Virtual Resource Pools.

To allocate the resources according to the scenario described above, a dedicated virtual resource pool should be created for each job queue. As shown in Figure 3, fast slots

(running on fast cores) can be grouped into the Virtual Fast (vFast) resource pool that is dedicated for the Interactive Job Queue and the slow slots (running on slow cores) can be grouped as the Virtual Slow (vSlow) resource pool that is dedicated to serve the Batch Job Queue.

The attractive part of such virtual resource pool arrangement is that it *preserves data locality* because both the fast and slow slots have the same data access to the datasets stored in the underlying HDFS. Therefore, any dataset can be processed by either *fast* or *slow* virtual resource pools, or their combination.

To support a virtual resource pool design, the TaskTracker needs additional mechanisms for the following functionality:

- the ability to start a task on a specific core, i.e., to run a slot on a specific core and assign a task to it, and
- maintain the mapping information between a task and the assigned slot type.

TaskTracker always starts a new JVM for each task instance (if the JVM reuse feature in Hadoop is disabled) so that the JVM failure does not impact other tasks or take down the TaskTracker. Running a task on a specific core can be achieved by binding the JVM to that core. We use the *CPU affinity* to implement this feature. By setting the CPU affinity, a process can be bound to one or to a set of cores. TaskTracker calls *spawnNewJVM* class to spawn a JVM in a new thread. The CPU affinity can be specified during the spawn to force the JVM to run on the desired core. An additional advantage of using the CPU affinity is that it *can be changed during runtime*. Therefore, if the *JVM reuse* feature is enabled in the Hadoop configuration, then the task can be placed on a desired core by changing the CPU affinity of the JVM.

The mapping information between tasks and cores can be maintained by recording *(task_ID, JVM_pid, core_id)* in the TaskTracker table. When a task finishes, the TaskTracker knows whether the released slot is a fast or a slow one.

The JobTracker needs to know whether the available slot is a slow or fast one, in order to make resource allocation decisions. DyScale communicates this information through the *heartbeat*, which is essentially a RPC (Remote Procedure Call) between TaskTracker at a worker and JobTracker at the master node. TaskTracker asks the JobTracker for a new task when the current running map/reduce tasks are below the configured maximum allowed number of map/reduce tasks through a boolean parameter *askForNewTask*. If the TaskTracker can accept a new task, the JobTracker calls the Hadoop Scheduler for a decision to assign a task to this TaskTracker.

The Scheduler checks *TaskTrackerStatus* to know whether the available slots are Map or Reduce slots. In the DyScale case, the Scheduler needs also to distinguish the slot type (i.e., slow or fast). There are four types of slots: *1-2)* fast and slow map slots, *3-4)* fast and slow reduce slots.

In the DyScale framework, the Scheduler interacts with the *JobQueue* by using the information about the slot type, e.g., if the available slot is a fast slot, then this slot belongs to vFast pool and the *InteractiveJobQueue* is selected for a job/task allocation. After selecting the *JobQueue*, it takes the first job in the queue and allocates to it the available slot.

Different policies exist for ordering the jobs inside the *JobQueue* as well as different slot allocation policies. The default policy is FIFO.

B. Managing Spare Cluster Resources

The static resource partitioning and allocation may be inefficient if a resource pool has spare resources (slots) but the corresponding *JobQueue* is empty, while other *JobQueue*(s) have jobs that are waiting for resources. For example, if there are jobs in the *InteractiveJobQueue* and they do not have enough fast slots, then these jobs should be able to use the available (spare) slow slots.

We use Virtual Shared (vShare) Resource pool to utilize the spare resources. As shown in Figure 4, the spare slots are put into the vShare pool and the slots in vShare resource pool can be used by any job queue.

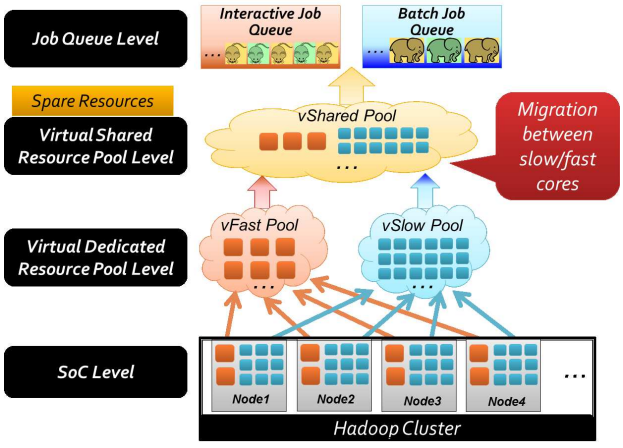


Figure 4. Virtual Shared Resource Pool.

The efficiency of the described resource sharing could be further improved by introducing the *TaskMigration* mechanism that allows task migration between slow and fast slots. For example, the jobs from the *InteractiveJobQueue* can use the spare slow slots until the fast slots become available. Then these tasks are migrated to the newly released fast slots so that the jobs from the *InteractiveJobQueue* always use optimal resources. Similarly, the migration mechanism allows the batch job to use temporally the spare fast slots if there are no waiting jobs in the *InteractiveJobQueue*. These resources can be returned by migrating the batch job from the fast slots to the released slow slots when a new interactive job arrives.

The DyScale framework allows to specify different policies for handling the spare resources. The migration mechanism is implemented by changing the JVM's CPU affinity within the same SoC as described earlier. By adding MIGRATE_TASK action in TaskTrackerAction list in heartbeatResponse, the JobTracker can inform the TaskTacker to migrate the designated task between slow and fast slots.

IV. Case Study

In this section, we present our experimental measurement results with a variety of MapReduce applications executed on the Hadoop cluster configured with different CPU frequencies and discuss why a typical MapReduce application may benefit from processors with slow or faster cores. We analyze and compare the simulation results based on synthetic Facebook traces that emulate the Facebook workload's execution on a Hadoop cluster.

A. Experimental Testbed and Workloads

We use an 8-node Hadoop cluster as our experiment testbed. Each node is a HP Proliant DL 120 G7 server that employs the latest Intel Xeon quad-core processor E31240 @ 3.30GHz. The processor offers a set of controllable CPU frequencies varying from 1.6 GHz to 3.3 GHz, and each core frequency can be set separately. The memory size of the server is 8 GB. There is one 128 GB disk for system usage and 6 additional 300 GB disks dedicated to Hadoop and data. The servers use 1 Gb Ethernet and connected by 10 Gb Ethernet Switch. We use Hadoop 1.0.0 with 1 dedicated server as JobTracker and NameNode and 7 servers as workers. We configure 1 map and 1 reduce slot per core, i.e., 4 map slots and 4 reduce slots per each worker node. The HDFS blocksize is set to 64MB and the replication level is set to 3. We use OpenJDK V1.6 and set the JVM heap size to 1 Gb.

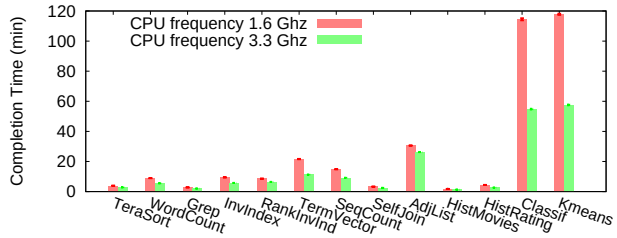
We select 13 diverse MapReduce applications [2] to run experiments in our Hadoop cluster. The high level description of these applications, e.g., application name, input data type and size, intermediate data size, output data size, number of map and reduce tasks are shown in Table II. Application 1, 8 and 9 use synthetically generated data as input data. Application 2 to 7 process Wikipedia articles. Application 10 to 13 process the Netflix movie ratings set as the input dataset.

Table II
Application characteristics.

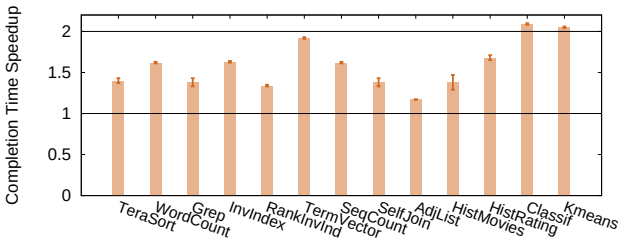
Application	*Input data (type)*	*Input data (GB)*	*Interm data (GB)*	*Output data (GB)*	*#map,red tasks*
1. TeraSort	Synthetic	31	31	31	450, 28
2. WordCount	Wikipedia	50	9.8	5.6	788, 28
3. Grep	Wikipedia	50	$3x10^{-8}$	$1x10^{-8}$	788, 1
4. InvIndex	Wikipedia	50	10.5	8.6	788, 28
5. RankInvIndex	Wikipedia	46	48	45	768, 28
6. TermVector	Wikipedia	50	4.1	0.002	788, 28
7. SeqCount	Wikipedia	50	45	39	788, 28
8. SelfJoin	Synthetic	28	25	0.15	448, 28
9. AdjList	Synthetic	28	11	11	507, 28
10. HistMovies	Netflix	27	$3x10^{-5}$	$7x10^{-8}$	428, 1
11. HistRatings	Netflix	27	$2x10^{-5}$	$6x10^{-8}$	428, 1
12. Classification	Netflix	27	0.008	0.006	428, 50
13. KMeans	Netflix	27	27	27	428, 50

B. Experimental Results with Different CPU Frequencies

We run the thirteen applications from Table II on our experimental cluster using two scenarios: *i)* the CPU frequency of all processors is set to 1.6 GHz and *ii)* the CPU frequency of all processors is set 3.3 GHz. We flush memory after each experiment and disable write cache to avoid the cache interference.



(a) Measured job completion time



(b) Job speedup (normalized)

Figure 5. Measured job completion time and speedup (normalized) when the CPU frequency is scaled-up from 1.6 GHz to 3.3 GHz.

All the measurement experiments are performed five times, and the measurement results are averaged. We also show the minimal and maximal measurement values across the 5 runs. This comment applies to the results in Figures 5 and 7.

Figure 5 summarizes the results of all experiments. Figure 5 (a) shows the completion time for each job running under different CPU frequencies. Figure 5 (b) shows the normalized results of the relative speedup obtained by executing the applications on the servers with 3.3.GHz compared to the application completion time on the servers with 1.6 GHz. A speedup of 1 implies no speedup, e.g., the same completion time. The achievable speedup across different jobs varies: a few jobs have 20-30% of completion time reduction, while a majority of jobs enjoy 1.6-2.1 times speedup.

To understand the reason behind different speedup across different applications, we performed further analysis at the phase level duration. Each map task processes a logical split of the input data (e.g., 64 MB) and performs the following steps: *read*, *map*, *collect*, *spill* and *merge* phases as shown in Figure 6. The map task *read*s the data, applies

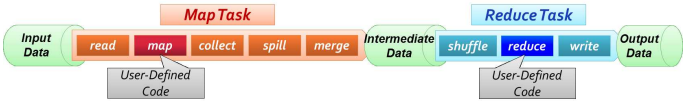


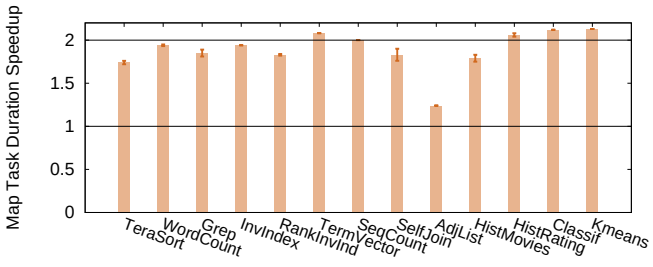
Figure 6. Map and Reduce Tasks Processing.

the user-defined *map* function on each record, and *collect*s the resulting output in memory. If this intermediate data is larger than the in-memory buffer, it is *spill*ed on the local

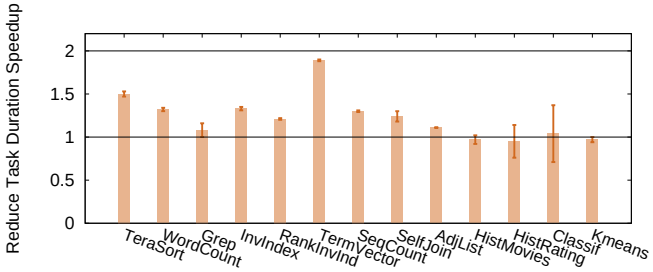
disk of the machine executing the map task and *merge*d into a single file for each reduce task.

The reduce task processing has *shuffle*, *reduce* and *write* phases. In the *shuffle* phase, the reduce tasks fetch the intermediate data files from the already completed map tasks and sort them. After all the intermediate data is shuffled, a final pass is made to merge sorted files. In the *reduce* phase, data is passed to the user-defined reduce function. The output from the reduce function is written back to the distributed file system in the *write* phase (three copies are written by default to different worker nodes.).

Our analysis reveals that *map task* processing for different applications have a similar speedup profile when executed by CPU with 3.3 GHz. This speedup is 2 times across all 13 applications (java-based processing is CPU intensive) as shown in Figure 7 (a).



(a) Map task speedup (normalized)



(b) Reduce task speedup (normalized)

Figure 7. Normalized speedup of map and reduce tasks when the CPU frequency is scaled-up from 1.6 GHz to 3.3 GHz.

The *shuffle* and *write* phases in reduce task processing show very limited speedup, only about 20% across the studied applications as shown in Figure 7 (b). For different applications the time spent in the shuffle and write phases is different and depends on the amount of intermediate data and output data written back to HDFS (i.e., whether the application is shuffle-heavy and/or whether it writes a large amount of output data such as *TeraSort*, *RankInvIndex*, *AdjList*, etc.). These shuffle and write portions of the processing time influence the outcome of the overall speedup for the application.

C. Simulation Framework and Results

As the heterogeneous multi-core processors are not yet readily available, we perform a simulation study using the extended MapReduce simulator SimMR [8] and a synthetic Facebook workload [4]. SimMR accurately reproduces the original job processing: the completion times of the simulated jobs are within 5% of the original ones [8]. Our goal is to compare the job completion times and to perform a sensitivity analysis when a workload is executed by different Hadoop clusters deployed on homogeneous or heterogeneous multi-core processors.

SimMR consists of the following three components:

- *Trace Generator* creates a replayable MapReduce workload and can create traces defined by a synthetic workload description that compactly characterizes the duration of map and reduce tasks as well as the shuffle stage characteristics via corresponding distribution functions. This feature is useful for sensitivity analysis of new schedulers and resource allocation policies applied to different workload types.
- *Simulator Engine* is a discrete event simulator that accurately emulates the job master functionality in the Hadoop cluster.
- *A pluggable scheduling policy* dictates the scheduler decisions on job ordering and the amount of resources allocated to different jobs over time.

We have extended SimMR to emulate the DyScale framework, i.e., the heterogeneous slow/fast cores that correspond to slow and fast Hadoop slots. We approximate the performance and power consumption of different cores from the available measurements of the existing Intel processors [9], [10] executing the PARSEC benchmark [11]. We observe that the Intel processors i7-2600 and E31240 (used in the HP Proliant DL 120 G7 server) are from the same Sandy Bridge micro-architecture family and have almost identical performance [12]. We additionally differentiate the performance of map and reduce tasks on the simulated processors by using the measured speedup from our experimental results reported in Section IV-B. We summarize all this data in Table IV-C.

With a power budget of 84W, we choose three multi-core processor configurations, see Table IV-C (there can be different combinations in configurations, but due to the interest of space, we show only one combination as an example here). In our experiments, we simulate the execution of the Facebook workload on three different Hadoop clusters with multi-core processors shown in Table IV-C. For sensitivity analysis, we present results for different cluster sizes of 25, 40, and 70 nodes as they reflect interesting performance situations.

We configure each Hadoop cluster with 1 map and 1 reduce slot per core [3], e.g., for a Hadoop cluster size with 40 nodes, the three considered configurations have the following number of map and reduce slots:

- the *Homogeneous-fast* configuration has 160 fast map (reduce) slots in total,
- the *Homogeneous-slow* configuration has 840 slow map (reduce) slots in total, and
- the *Heterogeneous* configuration has 120 fast map (reduce) slots and 360 slow map (reduce) slots in total.

[3]We assume that each node has enough memory to configure map and reduce slots with the same amount of RAM for different SOC configurations.

Table III
PROCESSOR SPECIFICATIONS.

Type	Processor Name	Tech.	Frequency	Power per Core	Norma -lized Power	Normalized (PARSEC) Performance	Normalized Map Task Performance	Normalized Reduce Task Performance
Type 1	i7-2600 Sandy Bridge	32 nm	3.4 GHz	21 W	1.0	1.0	1.0	1.0
Type 2	i5-670 Nehalem	32 nm	3.4 GHz	16 W	0.81	0.92	0.92	0.98
Type 3	AtomD Bonnell	45 nm	1.7 GHz	4 W	0.19	0.45	0.45	0.83

Table IV
PROCESSOR CONFIGURATIONS UNDER THE SAME POWER BUDGET OF 84 W.

Configuration	Type 1	Type 2	Type 3	Power
Homogeneous-fast	4	0	0	84 W
Homogeneous-slow	0	0	21	84 W
Heterogeneous	0	3	9	84 W

We generate 1000 MapReduce jobs according to the distribution shown in Table II-A. We consider jobs from 1st - 5th groups as small interactive jobs (e.g., with less than 100 tasks) and the jobs in the remaining five groups as large batch jobs. The interactive jobs are 82% of the total mix and the batch jobs are 18%. The task duration of the Facebook workload can be best fit with LogNormal distribution [13] and the following parameters: LN(9.9511, 1.6764) for map task duration and LN(12.375, 1.6262) for reduce task duration.

We perform a comparison of these three configurations when jobs are processed by each cluster in isolation: each job is submitted in the FIFO order, there is no bias due to the specific ordering policy nor queuing waiting time for each job, e.g., each job can use the entire cluster resources. For the *heterogeneous* configuration, the SimMR implementation supports the vShared resource pool so that a job can use both fast and slow resources of the entire cluster.

We plot the results in Figure 8, where the x-axis is the cluster size index of 25, 40, and 70 nodes respectively. The graphs show the average completion time of interactive jobs (top plot) and batch jobs (bottom plot).

For interactive jobs, *Homogeneous-fast* and *Heterogeneous* configurations achieve very close completion times and significantly outperform the *Homogeneous-slow* configuration by being almost twice faster. The small, interactive jobs have a limited parallelism and once their tasks are allocated necessary resources, these jobs cannot take advantage of the extra slots available in the system. For these jobs, the fast slots are the effective way to achieve better performance.

For batch jobs, as expected, the scale-out approach shows its advantage because batch jobs have a large number of map tasks. For example, the *Homogeneous-slow* configuration consistently outperforms the *Homogeneous-fast*, and can be almost twice as fast when the cluster size is small (e.g., 25 nodes). It is interesting to note that the *Heterogeneous* configuration is almost neck-to-neck with the *Homogeneous-slow* configuration for batch jobs.

By comparing these results, it is apparent that the heterogeneous multi-core processors with both fast and slow cores present an interesting design point. It may significantly improve the completion time of interactive jobs. The large batch jobs are benefiting from the larger number of the slower cores that improve throughput of these jobs. Moreover, the batch jobs are capable of taking advantage and effectively utilizing the additionally available fast slots in the vShared resource pool supported by the DyScale framework.

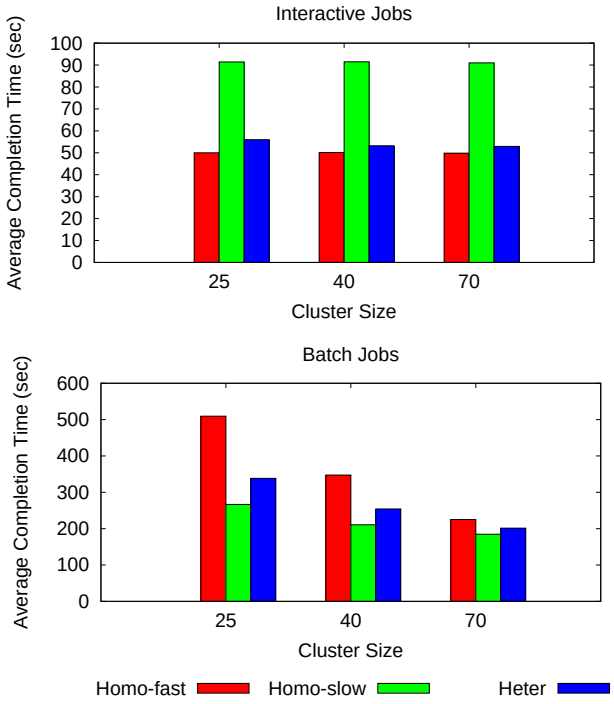


Figure 8. Completion time of interactive and batch jobs under different configurations.

V. RELATED WORK

There is a body of work focusing on performance analysis and optimization of MapReduce executions in heterogeneous environments.

Zaharia et al. [14], focus on eliminating the negative effect of stragglers on job completion time by improving the scheduling strategy with speculative tasks. This technique is applicable to our case as well, especially for operating with shared spare resources that are formed by different types of slots.

The Tarazu project [2] provides a communication-aware scheduling of map computation which aims at decreasing the communication overload when faster nodes process map tasks with input data stored on slow nodes. It also proposes a load-balancing approach for reduce computation by assigning different amounts of reduce work according to the node capacity.

Xie et al. [15] improve the MapReduce performance through a heterogeneity-aware data placement strategy: faster nodes store larger amount of input data. In this way, more tasks can be executed by faster nodes without a data transfer for the map execution. Gupta et al. [16] use off-line profiling of the jobs execution with respect to different heterogeneous nodes in the cluster and optimize the task placement to improve the job completion time. Polo et al. [17] show that some MapReduce applications can be accelerated by using special hardware. The authors design an adaptive Hadoop scheduler that assigns such jobs to the nodes with corresponding hardware.

Zhang et al. [18] explores the efficiency and performance accuracy of the *bounds-based* performance model introduced in the ARIA project [7] for predicting the MapReduce job completion times in heterogeneous Hadoop clusters and discuss factors that impact the MapReduce job performance in the Amazon EC2 environment.

In the case of Hadoop deployment on heterogeneous servers, one have to deal with data locality issues and balancing the data placement according to the server capabilities as presented in the earlier work cited above. One of the biggest advantages of Hadoop deployed with heterogeneous processors is that both *fast* and *slow* slots have a similar access to the underlying HDFS data that eliminates the data locality issues.

Ren et al. [9] consider heterogeneous SoC design and demonstrates that the heterogeneity is well suited to improve performance of interactive workloads (e.g., web search, online gaming, and financial trading). This is another example of interesting applications benefiting from the heterogeneous processors. In [19], the opportunities and challenges of using heterogeneous multi-core processors for MapReduce processing is discussed.

VI. Conclusions

In this work, we exploit new opportunities of using heterogeneous multi-core processors for MapReduce processing. We present a new framework, called DyScale, that is implemented on the top of Hadoop. DyScale enables creating different virtual resource pools based on the core-types for multi-class job scheduling. The new framework aims at taking the advantage of heterogeneous cores' capabilities for achieving a variety of performance objectives. DyScale is easy to use because the created virtual clusters have access to the same data stored in the underlying distributed file system, and therefore, any job and any dataset can be processed by either *fast* or *slow* virtual resource pools, or their combination. With DyScale, MapReduce jobs can be submitted into different queues, where they operate over different virtual resource pools for achieving a better completion time (e.g., small jobs) or a better throughput (e.g., large jobs).

There is an interesting application of the proposed framework for improving performance of Pig queries. Pig queries usually process large datasets (that corresponds to processing large jobs) and then have data transformation over small datasets in the second part of the workflow (that corresponds to processing small jobs). In the future, we plan to exploit the required analysis and automated job assignment to different resource pools for achieving an improved query completion time. We also plan to compare our scheduler with other existing schedulers such as Capacity Scheduler.

References

[1] T. White, *Hadoop:The Definitive Guide*. Yahoo Press.

[2] F. Ahmad *et al.*, "Tarazu: Optimizing MapReduce on Heterogeneous Clusters," in *Proceedings of ASPLOS*, 2012.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[4] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of EuroSys*, 2010.

[5] Apache, "Capacity Scheduler Guide," 2010. [Online]. Available: http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html

[6] S. Rao *et al.*, "Sailfish: A Framework For Large Scale Data Processing," in *Proceedings of SOCC*, 2012.

[7] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," in *Proc. of ICAC*, 2011.

[8] ——, "Play It Again, SimMR!" in *Proceedings of Intl. IEEE Cluster'2011*.

[9] S. Ren, Y. He, S. Elnikety, and S. McKinley, "Exploiting Processor Heterogeneity in Interactive Services," in *Proceedings of ICAC*, 2013.

[10] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley, "Looking back and looking forward: power, performance, and upheaval," *Commun. ACM*, vol. 55, no. 7, 2012.

[11] C. Bienia, S. Kumar, J. Singh, and K. Li, "The PARSEC benchmark suite:Characterization and architectural implications." in *Technical Report TR-811-08, Princeton University*, 2008.

[12] "PassMark Software. CPU Benchmarks," 2013. [Online]. Available: http://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+E3-1240+%40+3.30GHz

[13] A. Verma *et al.*, "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle," in *Proc. of IEEE/IFIP NOMS*, 2012.

[14] M. Zaharia *et al.*, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of OSDI*, 2008.

[15] J. Xie *et al.*, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *Proceedings of the IPDPS Workshops: Heterogeneity in Computing*, 2010.

[16] G. Gupta, C. Fritz, B. Price, R. Hoover, J. DeKleer, and C. Witteveen, "ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters," in *Proc. of ICAC*, 2013.

[17] J. Polo *et al.*, "Performance management of accelerated mapreduce workloads in heterogeneous clusters," in *Proceedings of the 41st Intl. Conf. on Parallel Processing*, 2010.

[18] Z. Zhang, L. Cherkasova, and B. T. Loo, "Performance Modeling of MapReduce Jobs in Heterogeneous Cloud Environments," in *Proceedings of IEEE CLOUD*, 2013.

[19] F. Yan, L. Cherkasova, Z. Zhang, and E. Smirni, "Heterogeneous cores for mapreduce processing: Opportunity or challenge?" in *Proc. of IEEE/IFIP NOMS*, May, 2014.