7-2017

DyScale: A MapReduce Job Scheduler for Heterogeneous Multicore Processors

Feng Yan
College of William and Mary, fyan@cs.wm.edu

Ludmila Cherkasova
lucy.cherkasova@hp.com

Zhuoyao Zhang
zhuoyao@seas.upenn.edu

Evgenia Smirni
College of William and Mary, esmirni@cs.wm.edu

Recommended Citation

Yan, Feng; Cherkasova, Ludmila; Zhang, Zhuoyao; and Smirni, Evgenia, DyScale: A MapReduce Job Scheduler for Heterogeneous Multicore Processors (2017). *IEEE TRANSACTIONS ON CLOUD COMPUTING*, 5(2).
10.1109/TCC.2015.2415772

DyScale: A MapReduce Job Scheduler for Heterogeneous Multicore Processors

Feng Yan, *Member, IEEE*, Ludmila Cherkasova, *Member, IEEE*, Zhuoyao Zhang, *Member, IEEE*, and Evgenia Smirni, *Member, IEEE*

Abstract—The functionality of modern multi-core processors is often driven by a given power budget that requires designers to evaluate different decision trade-offs, e.g., to choose between many slow, power-efficient cores, or fewer faster, power-hungry cores, or a combination of them. Here, we prototype 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 "slow" and "fast" cores for multi-class priority scheduling. Since the same data can be accessed with either "slow" or "fast" slots, spare resources (slots) can be shared between different resource pools. Using measurements on an actual experimental setting and via simulation, we argue in favor of heterogeneous multi-core processors as they achieve "faster" (up to 40 percent) processing of small, interactive MapReduce jobs, while offering improved throughput (up to 40 percent) for large, batch jobs. We evaluate the performance benefits of DyScale versus the FIFO and Capacity job schedulers that are broadly used in the Hadoop community.

Index Terms—MapReduce, Hadoop, heterogeneous systems, scheduling, performance, power

1 Introduction

To offer diverse computing and performance capabilities, the emergent modern system on a chip (SoC) may include heterogeneous cores that execute the same instruction set while exhibiting different power and performance characteristics. The SoC design is often driven by a power budget that limits the number (and type) of cores that can be put on a chip. The power constraints force designers to exploit a variety of choices within the same power envelope and to analyze decision trade-offs, e.g., to choose between either many slow, low-power cores, or fewer faster, power hungry cores, or to select a combination of them, see Fig. 1. A number of interesting choices may exist, but once the SoC design is chosen, it defines the configuration of the produced chip, where the number and the type of cores on the chip is fixed and cannot be changed.

Intuitively, an application that needs to support higher throughput and that is capable of partitioning and distributing its workload across many cores favors a processor with a higher number of slow cores. However, the latency of a time-sensitive application depends on the speed of its sequential components and should benefit from a processor with faster cores to expedite the sequential parts of the computation. This is why a time-sensitive application may favor a SoC processor with faster cores, even if these are few. A SoC design with heterogeneous cores might offer the best of both worlds by allowing to benefit from heterogeneous processing capabilities.

MapReduce 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. Hadoop was originally 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 job completion in half. This rule is applicable to jobs that need to process large datasets and that consist of a large number of tasks. Processing these tasks on a larger number of nodes (slots) reduces job completion time. Efficient processing of such jobs is "throughput-oriented" and can be significantly improved with additional "scale-out" resources.

When multiple users share the same Hadoop cluster, there are many interactive ad-hoc queries and small MapReduce jobs that are completion-time sensitive. In addition, a growing number of MapReduce applications (e.g., personalized advertising, sentiment analysis, spam detection) are deadline-driven, hence they require completion time guarantees. To improve the execution time of small MapReduce jobs, one cannot use the "scale-out" approach, but could benefit using a "scale-up" approach, where tasks execute on "faster" resources.

- *F. Yan and E. Smirni are with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23187. E-mail: {fyan, esmirni}@cs.wm.edu.*
- *L. Cherkasova is with the Hewlett-Packard Labs, Palo Alto, CA 94304. E-mail: lucy.cherkasova@hp.com.*
- *Z. Zhang is with the Department of Computer Science, University of Pennsylvania, Philadelphia, PA 19104. E-mail: zhuoyao@seas.upenn.edu.*

Manuscript received 2 June 2014; revised 18 Dec. 2014; accepted 12 Feb. 2015. Date of publication 23 Mar. 2015; date of current version 7 June 2017. Recommended for acceptance by G. Agrawal. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TCC.2015.2415772

Fig. 1. Different choices in the processor design.

A typical perception of a MapReduce processing pipeline is that it is disk-bound (for small and medium Hadoop clusters) and that it can become network-bound on larger Hadoop clusters. Intuitively, it is unclear whether a typical MapReduce application under normal circumstances can benefit from processors with faster cores. To answer this question we performed experiments on 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 to 3.3 Ghz, with each core frequency set separately). While the achievable speedup across different jobs varies, many jobs achieved speedup of 1.6-2.1 thanks to the faster processors. Such heterogeneous multi-core processors become an interesting design choice for supporting different performance objectives of MapReduce jobs.

Here, we design and evaluate DyScale, a new Hadoop scheduler that exploits capabilities offered by heterogeneous cores for achieving a variety of performance objectives. These heterogeneous cores are used for creating different virtual resource pools, each based on a distinct core type. These virtual pools consist of resources of distinct virtual Hadoop clusters that operate over the same datasets and that can share their resources if needed. Resource pools can be exploited for multi-class job scheduling. We describe new mechanisms for enabling "slow" slots (running on slow cores) and "fast" slots (running on fast cores) in Hadoop and creating the corresponding virtual clusters. Extensive simulation experiments demonstrate the efficiency and robustness of the proposed framework. Within the same power budget, DyScale operating on heterogeneous multi-core processors provides significant performance improvement for small, interactive jobs comparing to using homogeneous processors with (many) slow cores. DyScale can reduce the average completion time of time-sensitive interactive jobs by more than 40 percent. At the same time, DyScale maintains good performance for large batch jobs compared to using a homogeneous fast core design (with fewer cores). The considered heterogeneous configurations can reduce completion time of batch jobs up to 40 percent.

There is a list of interesting *opportunities* for improving MapReduce processing offered by heterogeneous processor design. First of all, both *fast* and *slow* Hadoop slots have the same access to the underlying HDFS data. This eliminates the data locality issues that could make heterogeneous Hadoop clusters comprised of *fast* and *slow* servers[1] being inefficient [2]. However, when each node consists of heterogeneous core processors, then any dataset (or any job) can be processed by either *fast* or *slow* virtual resource pools, or their combination. Second, the possibility of task (job) migration between slow and fast cores enables enhancing performance guarantees and more efficient resource usage compared to static frameworks without the process migration feature. Among the challenges are *i)* the implementation of new mechanisms in support of dynamic resources allocation, like migration and virtual resource pools, *ii)* the support of accurate job profiling, especially, when a job/task is executed on a mix of fast and slow slots, *iii)* the analysis of per job performance trade-offs for making the right optimization decisions, and *iv)* increased management complexity.

This paper is organized as follows. Section 2 provides background of MapReduce processing. Section 3 gives a motivating example and discusses the advantages of the scale-out and scale-up approaches. Section 4 introduces the DyScale framework. Section 5 evaluates DyScale using measurements on actual machines and via simulation on a diverse variety of settings. Section 6 outlines related work. Section 7 summarizes our contribution and gives directions for future work.

2 MapReduce Background

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. After that, it applies the user-defined reduce function to the merged value list to produce the aggregate results (this is called the reduce phase). Then the reduce output is 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. Each worker node is configured with a fixed number of map and reduce slots, and these slots are managed by the local TaskTracker. The TaskTracker periodically sends heartbeats to the master JobTracker via TCP handshakes. The heartbeats contain information such as 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. Popular job schedulers include FIFO, Hadoop Fair scheduler (HFS) [4], and Capacity scheduler [5]. FIFO is the default and schedules MapReduce jobs according to their submission order. This policy is not efficient for small jobs if large jobs are also present. The Hadoop Fair Scheduler aims to solve this problem. It allocates on average the same amount of resources to every job over time so that small jobs do not

1. Note that the concept of heterogeneous cores within a single processor is very different from heterogeneous servers, where different servers have different capacity and performance. Hadoop clusters that include heterogeneous servers do have a variety of problems with traditional data placement and related unbalanced data processing as has been shown in [2].

TABLE 1
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	2,400	360	2%
10	4,800	NA	2%

suffer from delay penalties when scheduled after large jobs and large jobs do not starve. The Capacity scheduler offers similar features as the HFS but has a different design philosophy. It allows users to define different queues for different types of jobs and to configure a percentage of share of the total resources for each queue in order to avoid FIFO's shortcomings.

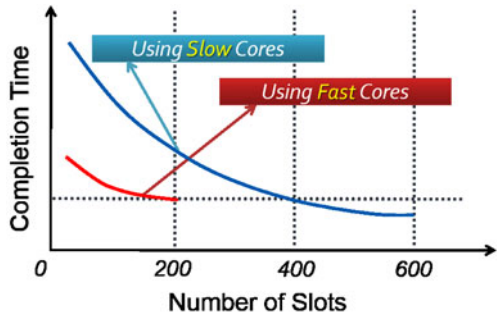
The assignment of tasks to slots is done in a greedy way: assign a task from the selected job J immediately whenever a worker reports to have a free slot. At the same time, a data locality consideration is taken into account: if there is a choice of available slots in the system to be allocated to job J, then the slots that have data chunks of job J locally available for processing are given priority [4]. If the number of tasks belonging to a MapReduce job is greater than the total number of processing slots, then 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. Counters help profile the job performance and provide important information for designing new schedulers. We utilize the extended set of counters from [6] in DyScale.

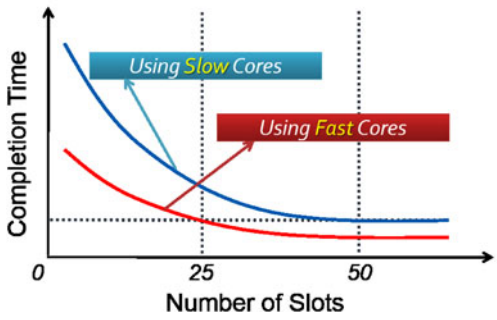
3 Motivating Example: Scale-Out Versus Scale-Up Approaches

Workload characterization based on Facebook and Yahoo jobs [4], [7] shows that a MapReduce workload typically can be described as a collection of "elephants" and "mice". Table 1 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. Most jobs are quite small (mice): 88 percent of jobs have less than 200 map tasks, but there is a small percentage of the large jobs (elephants) with up to thousands of map tasks and hundreds of reduce tasks.

Pig jobs [8] present a different case of a MapReduce workload with large and small jobs. Pig offers a high level SQL-like abstraction on top of Hadoop. Pig queries are composed of MapReduce workflows. During the earlier stages of the workflows, the datasets to be processed are usually large, and therefore, they correspond to "large" job



(a) Large *Job1* Processing.



(b) Small *Job2* Processing.

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

processing. After some operations such as "select" and "aggregate", the amount of data for processing may be significantly reduced, and the jobs in the second half of the workflow can be considered "small".

Different types of jobs may favor different design choices. For example, large jobs may benefit from processors with many slow cores to obtain better throughput, i.e., to execute as many tasks in parallel as possible in order to achieve better job completion time. Small jobs may benefit by processors with fewer fast cores for speeding-up their tasks and for getting an improved job completion time. Therefore, 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.

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

Motivating example. Assume that we have a Hadoop cluster with 100 nodes. Under the same power budget each node can have either *two fast* cores or *six slow* cores. We configure the Hadoop cluster with one map slot and one reduce slot per core. The slots that are executed on the fast or slow cores are called *fast* and *slow slots* respectively. Therefore, the system can have either *200 fast* map (reduce) slots or *600 slow* map (reduce) slots in total. Let us consider the following two jobs:

- *Job1* with 4,800 map tasks (i.e., similar to jobs in the 10th group of Table 1),
- *Job2* with 50 map tasks (i.e., similar to jobs in the fourth group of Table 1).

Assume a map task of *Job1* and *Job2* requires T amout of time to finish with a *fast* slot, and $2 \cdot T$ to execute with a *slow* slot. Let us look at the job completion time of *Job1* and *Job2* as a function of an increased number of fast or slow slots that are allocated to the job.

The scenarios that reflect possible executions of *Job1* and *Job2* are shown in Figs. 2a and 2b, respectively. The graphs in Fig. 2 are drawn based on calculations using the analytic model for estimating the completion time of a single job [9]. Completion times are graphed as a function of the number of slots allocated to a job. The figure illustrates that both jobs achieve lower completion times with a higher number of allocated slots.

When a large *Job1* is executed with fast slots, and *all 200 fast* slots are allocated to the job, then its completion time is: $4800 \cdot T/200 = 24 \cdot T$, i.e., it takes 24 rounds of T time units. The best job completion time is achieved when using *all 600 slow* slots. In this case, *Job1* finishes in $4{,}800 \cdot 2 \cdot T/600 = 16 \cdot T$, i.e., it takes eight rounds of $2 \cdot T$ time units. The job completion time with slow slots is 30 percent better than with the fast slots. Thus, using a larger number of slow slots leads to a faster completion time.

The a small *Job2* shown in Fig. 2b cannot take advantage of more than 50 slots, either slow or fast, because it only has 50 tasks. In this case, when *Job2* is executed with fast slots, it takes $50 \cdot T/50 = T$ time units to complete. If executed with slow slots, the completion time is $50 \cdot 2 \cdot T/50 = 2 \cdot T$ units, which is twice longer than using fast slots. The small jobs are usually interactive and thus are time sensitive. For such jobs, 50 percent of a completion time improvement represents a significant performance opportunity.

From the example, it is clear that large batch jobs (similar to *Job1*) can achieve better performance when processed by a larger number of slow slots, while the smaller jobs (like *Job2*) can execute faster on a smaller number of fast slots. Such diverse demands in MapReduce jobs indicate that traditional homogeneous multi-core processors may not provide the best performance and power trade-offs. This motivates a new Hadoop scheduler design with a tailored slot assignment to different jobs based on their scale-out and scale-up features.

4 DyScale Framework

We propose a new Hadoop scheduling framework, called DyScale, for efficient job scheduling on the heterogeneous multi-core processors. 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 existing virtual resource pools.

4.1 Problem Definition

The number of *fast* and *slow* cores is SoC design specific and workload dependent. Here, we focus on a given heterogeneous multi-core processor in each server node, and the problem of taking advantage of these heterogeneous capabilities, especially compared to using homogenous multi-core processors with the same power budget. Our goal is twofold: 1) design a framework for creating virtual Hadoop clusters with different processing capabilities (i.e., clusters with *fast* and *slow* slots); and 2) offer a new scheduler to support jobs with different performance objectives for utilizing the created virtual clusters and sharing their spare resources. The problem definition is as follows:

Input:
- C: cluster size (number of machines)
- N_f: number of fast cores on each machine
- N_s: number of slow cores on each machine
- S: job size distribution
- A: job arrival process

Output:
Sched: schedule of Map/Reduce task placement

Objective:
minimize$_{\{Sched\}}$ Job Completion Time (*Sched*)

A natural first question is why a new Hadoop scheduler is a necessity and why the default Hadoop scheduler can not work well. To answer this question, we show the performance comparison under the same power budget of using the default Hadoop scheduler on heterogenous and homogenous multi-core processors respectively, and also our DyScale scheduler with the same heterogenous multi-core processors, see Fig. 3. The details of the experiment configurations are given in Section 5.3. The important message from Fig. 3 is that the default Hadoop scheduler cannot use well the heterogenous multi-core processors and may even perform worse than when using it on a cluster with homogenous multi-core processors with the same power budget due to the random use of fast and slow cores.

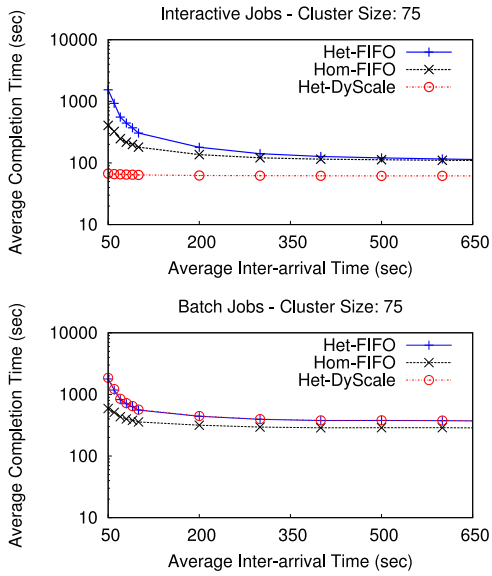


Fig. 3. The completion time of interactive jobs and batch jobs under different configurations: heterogenous cluster using FIFO, homogenous cluster using FIFO and heterogenous cluster using DyScale.

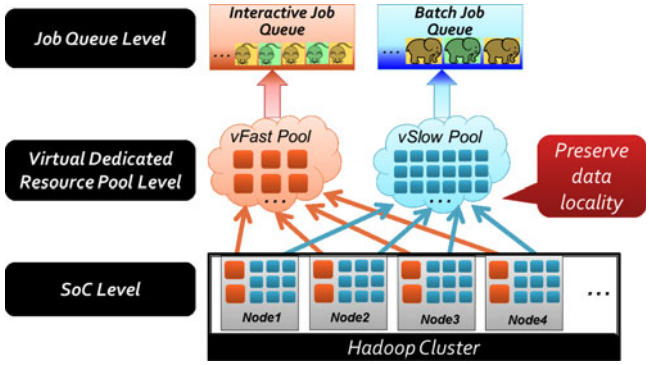


Fig. 4. Virtual resource pools.

4.2 Dedicated Virtual Resource Pools for Different Job Queues

DyScale offers the ability to schedule jobs based on performance objectives and resource preferences. For example, a user can submit small, time-sensitive jobs to the *Interactive Job Queue* to be executed by fast cores and large, throughput-oriented jobs to the *Batch Job Queue* for processing by (many) slow cores. This scenario is shown in Fig. 4. It is also possible for the scheduler to automatically recognize the job type and schedule the job on the proper queue. For example, small and large jobs can be categorized based on the number of tasks. A job can be also classified based on the application information or by adding a job type feature in job profile.

To allocate resources according to the above scenario, a dedicated virtual resource pool has to be created for each job queue. For example, as shown in Fig. 4, fast slots can be grouped as a Virtual Fast (vFast) resource pool that is dedicated to the Interactive Job Queue. Slow slots can be grouped as a Virtual Slow (vSlow) resource pool that is dedicated to the Batch Job Queue.

The attractive part of such virtual resource pool arrangement is that it *preserves data locality* because both 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.[2] To support a virtual resource pool design, the TaskTracker needs additional mechanisms for the following functionalities:

- 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;
- to maintain the mapping information between a task and the assigned slot type.

The TaskTracker always starts a new JVM for each task instance (if the JVM reuse feature in Hadoop is disabled). It is done such that a JVM failure does not impact other tasks or does not 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 a set of cores. The TaskTracker calls *spawnNewJVM* class to spawn a JVM in a new thread. The CPU affinity can be specified during spawn to force the JVM to run on the desired fast or slow core.

An additional advantage of using the CPU affinity is that it *can be changed during runtime*. If the *JVM reuse* feature is enabled in the Hadoop configuration (note, that the JVM reuse can be enabled only for the tasks of the same job), the task can be placed on a desired core by changing the CPU affinity of the JVM.

The mapping information between tasks and cores is 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 fast or slow.

The JobTracker needs to know whether the available slot is a slow or fast slot to make resource allocation decisions. DyScale communicates this information through the *heartbeat*, which is essentially a RPC (Remote Procedure Call) between the TaskTracker at a worker node and the JobTracker at the master node.

The 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, then 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. DyScale's Scheduler also needs to distinguish the slot type. There are four types of slots: *i)* fast map, *ii)* slow map, *iii)* fast reduce, and *iv)* slow reduce.

In the DyScale framework, the Scheduler interacts with the *JobQueue* by considering 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 allocates the available slot to the first job in the queue.

Different policies exist for ordering the jobs inside the *JobQueue* as well as different slot allocation policies. The default policy is FIFO. The job ordering/resource allocation depends on the performance objectives and can be defined by the Hadoop Fair Scheduler [4] or the ARIA SLO-driven scheduler [9]. DyScale can be easily augmented with additional policies for improving fairness, meeting completion time objectives, or other metrics. The JobTracker puts a list of current actions, such as LAUNCH_TASK, in the TaskTrackerAction list to tell the TaskTracker what to do next through the *heartbeatResponse*.

4.3 Managing Spare Cluster Resources

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 the Virtual Shared (vShare) Resource pool to utilize spare resources. As shown in Fig. 5, the spare slots are

2. Note the difference of this approach compared to node level heterogeneity, where data may reside on different node types, and therefore, it leads to data locality issues as data is not always available on the desired node types.

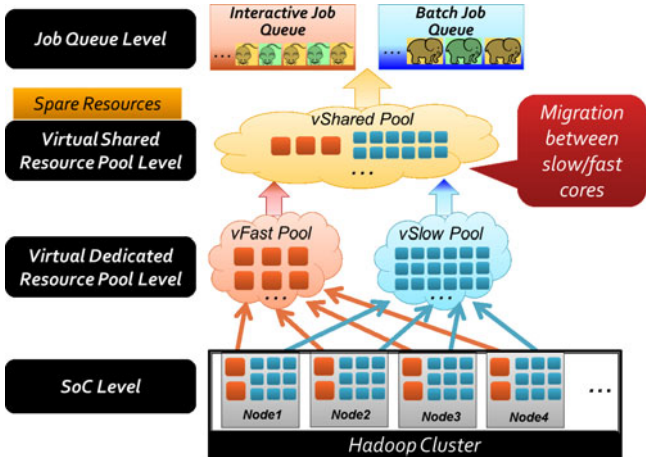


Fig. 5. Virtual shared resource pool.

put into the vShare pool. Slots in the vShare resource pool can be used by any job queue.

The efficiency of the described resource sharing could be further improved by introducing the *TaskMigration* mechanism. For example, the jobs from the *InteractiveJobQueue* can use spare slow slots until the future fast slots become available. 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 temporarily spare fast slots if the *InteractiveJobQueue* is empty. These resources are returned by migrating the batch job from the fast slots to the released slow slots when a new interactive job arrives.

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

DyScale can support SLOs by adding priorities to the queues and by allowing different policies for ordering the jobs inside each queue. For example, let the interactive jobs have deadlines to meet. The batch jobs are the best-effort jobs. When there are not enough fast slots for interactive jobs, these jobs can be given priority for using the available slow slots. This can be supported by the vShared resource pool and task migration.

5 Case Study

In this section, we first present our measurement results with a variety of MapReduce applications executed on a Hadoop cluster configured with different CPU frequencies. Then, we analyze and compare simulation results based on synthetic Facebook traces, that emulate the execution of the Facebook workload on a Hadoop cluster to quantify the effects of homogeneous versus heterogeneous processors. We also analyze the DyScale scheduler performance under different job arrival rates and evaluate its performance advantages in comparison to the FIFO and Capacity [5] job schedulers that are broadly used by the Hadoop community.

TABLE 2
Application Characteristics

Application	*Input data (type)*	*Input data (GB)*	*Interm data (GB)*	*Output data (GB)*	*#map, red tasks*
1.TeraSort	Synth	31	31	31	450, 28
2.WordCount	Wiki	50	9.8	5.6	788, 28
3.Grep	Wiki	50	$3x10^{-8}$	$1x10^{-8}$	788, 1
4.InvIndex	Wiki	50	10.5	8.6	788, 28
5.RankInvIndex	Wiki	46	48	45	768, 28
6.TermVector	Wiki	50	4.1	0.002	788, 28
7.SeqCount	Wiki	50	45	39	788, 28
8.SelfJoin	Synth	28	25	0.15	448, 28
9.AdjList	Synth	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

5.1 Experimental Testbed and Workloads

We use an eight-node Hadoop cluster as our experimental testbed. Each node is a HP Proliant DL 120 G7 server that employs the latest Intel Xeon quad-core processor E31240 @ 3.30 Ghz. The processor offers a set of controllable CPU frequencies varying from 1.6 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 dedicated for system usage and six additional 300 GB disks dedicated to Hadoop and data. The servers use 1 Gigabit Ethernet and are connected by a 10 Gigabit Ethernet Switch. We use Hadoop 1.0.0 with one dedicated server as JobTracker and NameNode, and the remaining seven servers as workers. We configure one map and one reduce slot per core, i.e., four map slots and four reduce slots per each worker node. The HDFS blocksize is set to 64 MB and the replication level is set to 3. We use the default Hadoop task failure mechanism to handle task failures.

We select 13 diverse MapReduce applications [2] to run experiments in our Hadoop cluster. The high level description of these applications is given in Table 2.

Applications 1, 8, and 9 use synthetically generated data as input. Applications 2 to 7 process Wikipedia articles. Applications 10 to 13 process Netflix ratings. The intermediate data is the output of map task processing. This data serves as the input data for reduce task processing. If the intermediate data size is large, then more data needs to be shuffled from map tasks to reduce tasks. We call such jobs *shuffle-heavy*. Output data needs to be written to the distributed storage system (e.g., HDFS). When the output data size is large, we call such jobs *write-heavy*. *Shuffle-heavy* and *write-heavy* applications tend to use more networking and IO resources.

Selected applications for our experiments represent a variety of MapReduce processing patterns. For example, TeraSort, RankInvIndex, SeqCount, and KMeans are both *shuffle-heavy* and *write-heavy*. Grep, HistMovies, HistRatings, and Classification have a significantly reduced data size after the map stage and therefore belong to the *shuffle-light* and *write-light* category. In addition, some applications including Classification and KMeans are

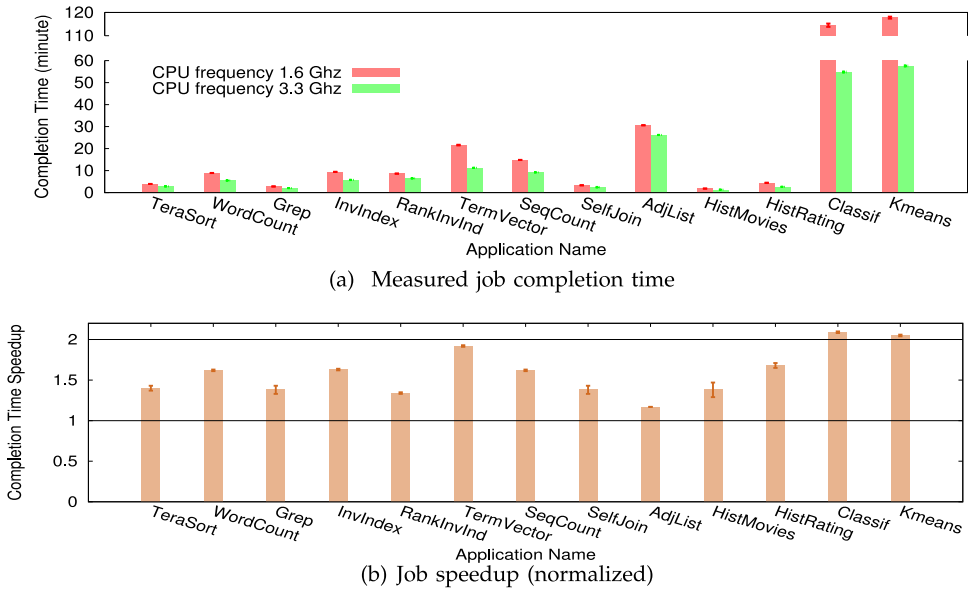


Fig. 6. Measured job completion time and speedup (normalized) when the CPU frequency is scaled-up from 1.6 to 3.3 GHz.

computation-intensive because their *map* phase processing time is orders of magnitude higher than other phases. The selected applications exhibit different processing patterns and allow for a detailed analysis on a diverse set of MapReduce workloads.

5.2 Experimental Results with Different CPU Frequencies

Since the heterogeneous multi-core processors are not yet available for provisioning a real testbed and performing experiments directly, we need to understand how execution on "fast" or "slow" cores may impact performance of MapReduce applications. Here we aim to empirically evaluate the impact of "fast" and "slow" cores on the completion time of representative MapReduce applications. We mimic the existence of fast and slow cores by using the CPU frequency control available in the current hardware. These experiments are important, because Hadoop and MapReduce applications are considered to be disk-bound, and intuitively, it is unclear what is the performance effect of different CPU frequencies.

We run all applications from Table 2 on our experimental cluster using two scenarios: *i)* CPU frequency of all processors is set to 1.6 Ghz for emulating "slow" cores, and *ii)* CPU frequency of all processors is set 3.3 Ghz, e.g., two times faster, for emulating "fast" cores. We flush memory after each experiment and disable write cache to avoid caching interference.

All measurement experiments are performed five times. We show the mean and the variance, i.e., the minimal and maximal measurement values across the five runs. This comment applies to the results in Figs. 6, 8, and 9.

Fig. 6 summarizes the results of our experiments. Fig. 6a shows the completion times for each job. Note the gap in the Y-axis that is introduced for better visualizing of all 13 applications in the same figure: the map task durations of Classification and Kmeans are much higher compared to the other 11 applications. Fig. 6b 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. Speedup of 1 means no speedup, i.e., the same completion time. Few jobs have a completion time speedup of 1.2 to 1.3, while the majority of jobs enjoy speedups of 1.6 to 2.1.

To better understand the above, 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, see Fig. 7. The map task *read*s the data, applies the *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 is comprised by the *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 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. By default, three copies are written to different worker nodes.

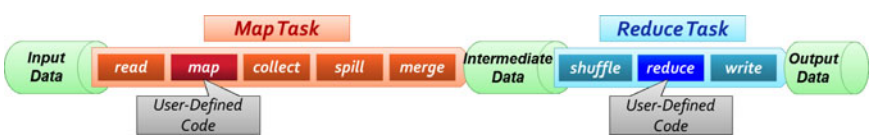
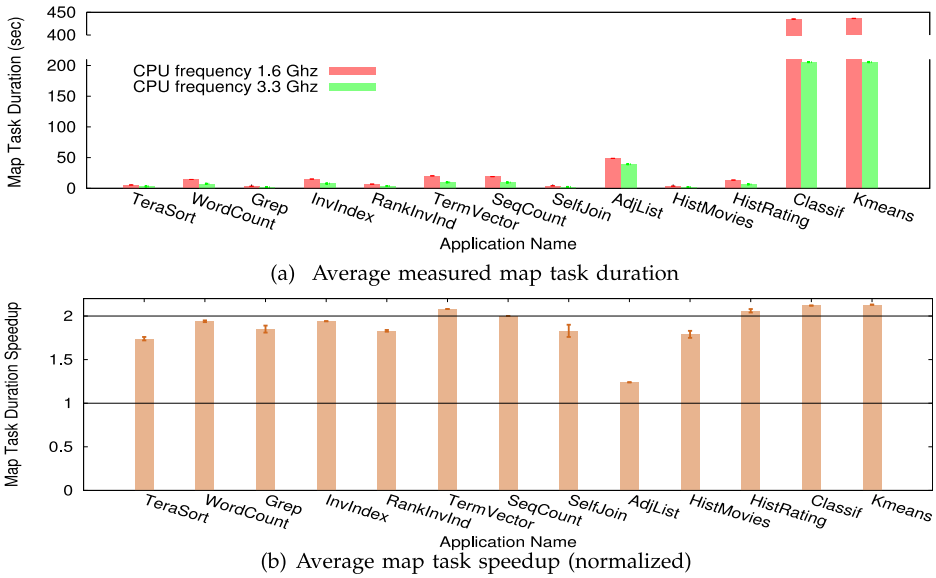


Fig. 7. Map and reduce tasks processing pipeline.



(a) Average measured map task duration

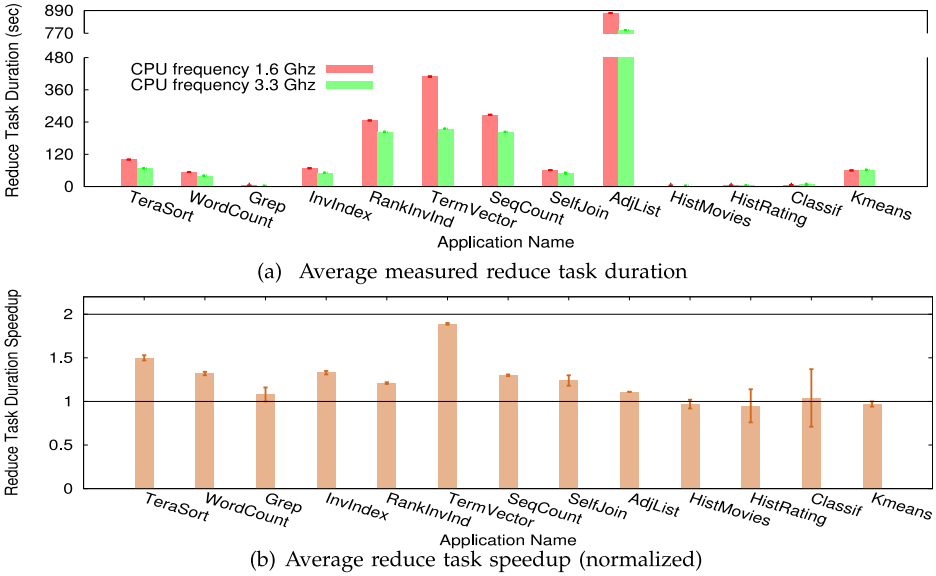
(b) Average map task speedup (normalized)

Fig. 8. Average measured map task duration and normalized speedup of map tasks in the experiments when the CPU frequency is scaled-up from 1.6 to 3.3 Ghz.

We report the average measured map task durations with CPU frequencies of 1.6 and 3.3 Ghz in Fig. 8a and the reduce task durations in Fig. 9a. 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). These *shuffle* and *write* portions of the processing time influence the outcome of the overall application speedup.

Our analysis reveals that the map task processing for different applications have a similar speedup profile when executed on a 3.3 Ghz CPU. In our experiments, this speedup is close to two across all 13 applications, see Fig. 8b. However, the *shuffle* and *write* phases in the reduce stage often show very limited speedup across applications (on average 20 percent, see Fig. 9b) due to different amount of data processed at this stage.

By looking at the results in Figs. 8b-9b, one may suggest the following *simple* scheduling policy for improving MapReduce job performance and taking advantage of heterogeneous multi-processors. Run map tasks on faster cores and reduce tasks on slower cores. However, performance of many large jobs is critically impacted not only by the type of slots allocated to the job tasks, but by the number of allocated slots. For example, if each processor has two *fast* cores and six *slow* cores then the proposed *simple* scheduling will not work as expected: using only *fast* cores for processing map tasks result in degraded performance for large jobs compared to their processing by using the available *slow* cores as has been shown in the related motivating example in Section 3. Therefore, to efficiently utilize the heterogeneous multi-core



(a) Average measured reduce task duration

(b) Average reduce task speedup (normalized)

Fig. 9. Average measured reduce task duration and normalized speedup of reduce tasks in the experiments when the CPU frequency is scaled-up from 1.6 to 3.3 Ghz.

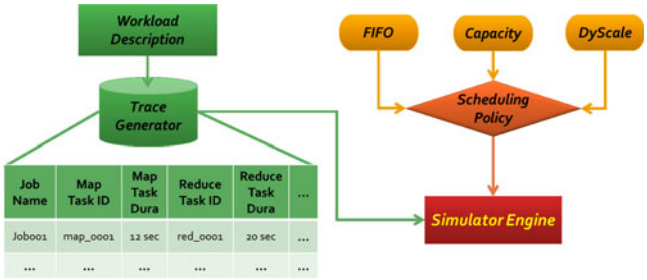


Fig. 10. Simulator design.

processors, one needs to consider a number of factors: 1) the type and the size of the job, workload job mix, jobs arrival rate, user performance objectives, and 2) the composition of the heterogeneous multi-processor, i.e., the number of *fast* or *slow* cores per processor, as well as the Hadoop cluster size.

5.3 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 [10] and a synthetic Facebook workload [4]. In addition, simulation allows more comprehensive sensitivity analysis. 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 either homogeneous or heterogeneous multi-core processors.

The event-driven simulator SimMR consists of the following three components, see Fig. 10:

- A *Trace Generator* creates a replayable MapReduce workload. In addition, the *Trace Generator* 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 to conduct sensitivity analysis of new schedulers and resource allocation policies applied to different workload types.
- The *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 extend SimMR[3] to emulate the DyScale framework. We also extend SimMR to emulate the Capacity scheduler [5] for homogeneous environments. We summarize the three schedulers used in this paper below:

- FIFO: the default Hadoop scheduler that schedules the jobs based on their arrival order.
- Capacity: users can define different queues for different types of jobs. Each queue can be configured with a percentage of the total number of slots in the cluster, this parameter is called *queue capacity*. This scheduler has an *Elasticity* feature that allows free resources to be allocated to a queue above its capacity to prevent artificial silos of resources and achieve better resources utilization.
- DyScale: we use two different versions: *i)* the *basic version* without task migration and *ii)* the *advanced version* with the migration feature enabled.

We approximate the performance and power consumption of different cores from the available measurements of the existing Intel processors [11], [12] executing the PARSEC benchmark [13]. 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 [14]. We additionally differentiate the performance of map and reduce tasks on the simulated processors by using our experimental results reported in Section 5.2. We summarize this data in Table 3.

With a power budget of 84 W, we choose three multi-core processor configurations, see Table 4. In our experiments, we simulate the execution of the Facebook workload on three different Hadoop clusters with multi-core processors. For sensitivity analysis, we present results for different cluster sizes of 75, 120, and 210 nodes as they represent interesting performance situations.

We configure each Hadoop cluster with one map and one reduce slot per core,[4] e.g., for a Hadoop cluster size with 120 nodes, the three considered configurations have the following number of map and reduce slots:

- the *Homogeneous-fast* configuration has 480 fast map (reduce) slots,
- *Homogeneous-slow* configuration has 2,640 slow map (reduce) slots, and
- the *Heterogeneous* configuration has 360 fast map (reduce) slots and 1,080 slow map (reduce) slots.

We generate 1,000 MapReduce jobs according to the distribution shown in Table1, with a three-fold increase in the input datasets,[5] Jobs from the first to the fifth group are small interactive jobs (e.g., with less than 300 tasks) and the remaining jobs are large batch jobs. The interactive jobs are 82 percent of the total mix and the batch jobs are 18 percent. The task duration of the Facebook workload can be best fit with a LogNormal distribution [16] and the following parameters: LN(9.9511, 1.6764) for map task duration and LN(12.375, 1.6262) for reduce task duration.

First, 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 all cluster resources. For the *heterogeneous* configuration, the SimMR implementation supports the vShared resource pool so that a job can use

3. SimMR accurately reproduces the original job processing: the completion times of the simulated jobs are within 5 percent of the original ones, see [10].

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

5. In our earlier conference paper [15], we have evaluated DyScale on a smaller workload defined by Table1 and smaller size Hadoop clusters with 25, 40, and 70 nodes. To test the scalability of the solution, we increased the application datasets and the Hadoop clusters for processing.

TABLE 3
Processor Specifications

Type	Processor Name	Tech.	Frequency	Power per Core	Normalized 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

both fast and slow resources. Results are plotted in Fig. 11. Each row shows three graphs that correspond to the clusters with 75, 120, and 210 nodes respectively. The graphs show the average completion time of interactive jobs (top row) and batch jobs (bottom row).

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 the necessary resources, these jobs cannot take advantage of the extra slots available in the system. For such jobs, fast slots are the effective way to achieve better performance (scale-up approach).

For batch jobs, as expected, the scale-out approach shows its advantage since batch jobs have a large number of map tasks. The *Homogeneous-slow* configuration consistently outperforms *Homogeneous-fast*, and can be almost twice faster when the cluster size is small (e.g., 75 nodes). The interesting result is 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 fast and slow cores present an interesting design point. It can significantly improve the completion time of interactive jobs with the same power budget. 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 additional fast slots in the vShared resource pool supported by DyScale.

5.4 Simulation Results with Arrival Process

In this section, we conduct further experiments for comparing the performance of different configurations under varying job arrival rates. We use the same experimental setup as in Section 5.3. We use exponential inter-arrival times to drive the job arrival process and vary the average of the inter-arrival time between 50 and 1,000 sec (between 50 and 100 sec with a step of 10 sec, and between 100 and 1,000 sec with a step of 100 sec). We analyze three scenarios:

TABLE 4
Processor Configurations with 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

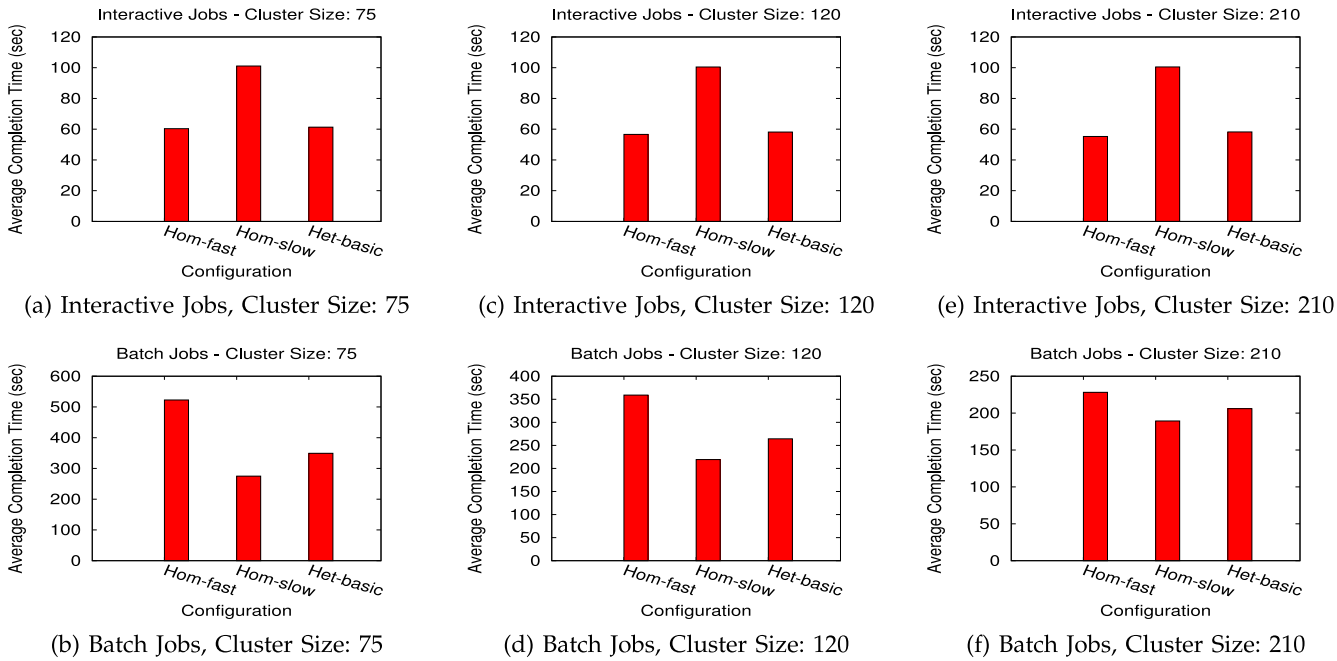
- *Scenario 1*. We compare the job completion times of DyScale (used in the *Heterogeneous* cluster configuration) with *FIFO* (used in both *Homogeneous-slow* and *Homogeneous-fast* cluster configurations).
- *Scenario 2*. We compare the job completion times of DyScale (used in the *Heterogeneous* cluster configuration) with *Capacity* (used in both *Homogeneous-slow* and *Homogeneous-fast* cluster configurations).
- *Scenario 3*. We compare the performance of DyScale with migration enabled and disabled to illustrate how a task migration feature can provide additional performance opportunities.

Fig. 12 illustrates the performance comparison of *DyScale vs FIFO* (*Scenario 1*). The completion times of interactive jobs (top row) for both *Homogeneous-slow* and *Homogeneous-fast* cluster configurations with FIFO are much higher than for the *Heterogeneous* configuration with DyScale. The *Homogeneous-fast* configuration is very sensitive to the cluster size and is least resilient to high arrival rates. The *Heterogeneous* configuration with DyScale consistently provides best performance for interactive jobs.

For batch jobs (second row in Fig. 12), the *Heterogeneous* configuration with DyScale is slightly worse than the *Homogeneous-slow* configuration because batch jobs have more slots to use in *Homogeneous-slow* configuration. However, it outperforms the *Homogeneous-fast* configuration by up to 30 percent.

Overall, the *Heterogeneous* configuration with the DyScale scheduler shows very good and stable job completion times compared to both *Homogeneous-slow* and *Homogeneous-fast* cluster configurations with the FIFO scheduler. It is especially evident under higher loads, i.e., when the inter-arrival times are small and traffic is bursty. Overall, system performance for the *Heterogeneous* configuration with the DyScale scheduler is very robust. When the inter-arrival time becomes larger (i.e., under light load), the observed performance gradually converges to the case when each job is executed in isolation, and the completion times are similar to the results shown in Fig. 11.

Fig. 13 illustrates the performance comparison of *DyScale vs Capacity* (*Scenario 2*). The Capacity Scheduler is configured with two queues for interactive jobs and batch jobs, respectively. Each queue capacity is determined based on the ratio of the interactive and batch jobs in the Facebook workload, see Table 5. We can see that the performance of interactive jobs (shown in top row) of Fig. 13 is supported better with the Capacity Scheduler compared to FIFO (the previous experiments shown in top row of Fig. 12). The completion times of interactive jobs for *Heterogeneous* configuration is slightly worse than for the *Homogeneous-fast* configuration, but much better than for *Heterogeneous-slow*,



(a) Interactive Jobs, Cluster Size: 75 (c) Interactive Jobs, Cluster Size: 120 (e) Interactive Jobs, Cluster Size: 210

(b) Batch Jobs, Cluster Size: 75 (d) Batch Jobs, Cluster Size: 120 (f) Batch Jobs, Cluster Size: 210

Fig. 11. Completion time of interactive and batch jobs under different configurations.

by up to 40 percent. For batch jobs, the *Heterogeneous* configuration is slightly worse than *Homogeneous-slow*, but it outperforms the *Homogeneous-fast* configuration by up to 30 percent. Again the *Heterogeneous* configuration with DyScale provides an interesting solution and exhibits a flexible support for job classes with different performance objectives compared to homogeneous-core configurations with either FIFO or Capacity schedulers.

Finally, we compare the *basic* DyScale (no task migration) and the *advanced* DyScale (with the task migration feature) and present the results for *Scenario 3* in Fig. 14. We can see that the migration feature always brings additional performance improvement for both interactive and batch jobs because it allows more efficient use of the cluster resources. When the cluster size is small, such feature provides a higher performance boost for interactive jobs, see Fig. 14a. In this case, there is only a limited number of fast slots, and the chance is higher that some interactive job is allocated to a slow slot. Task Migration allows migrating tasks when fast slots become available, and utilizes fast slots more efficiently.

When the cluster size increases, the task migration is more beneficial for batch jobs, see Fig. 14f. In this case, there are more fast slots in the system and the batch jobs can utilize them. However, when an interactive job arrives, the fast slots occupied by batch jobs can be released by migrating batch tasks to slow slots.

In summary, the *Heterogeneous* configuration with the DyScale scheduler allows to achieve significantly improved performance for interactive jobs while maintaining and

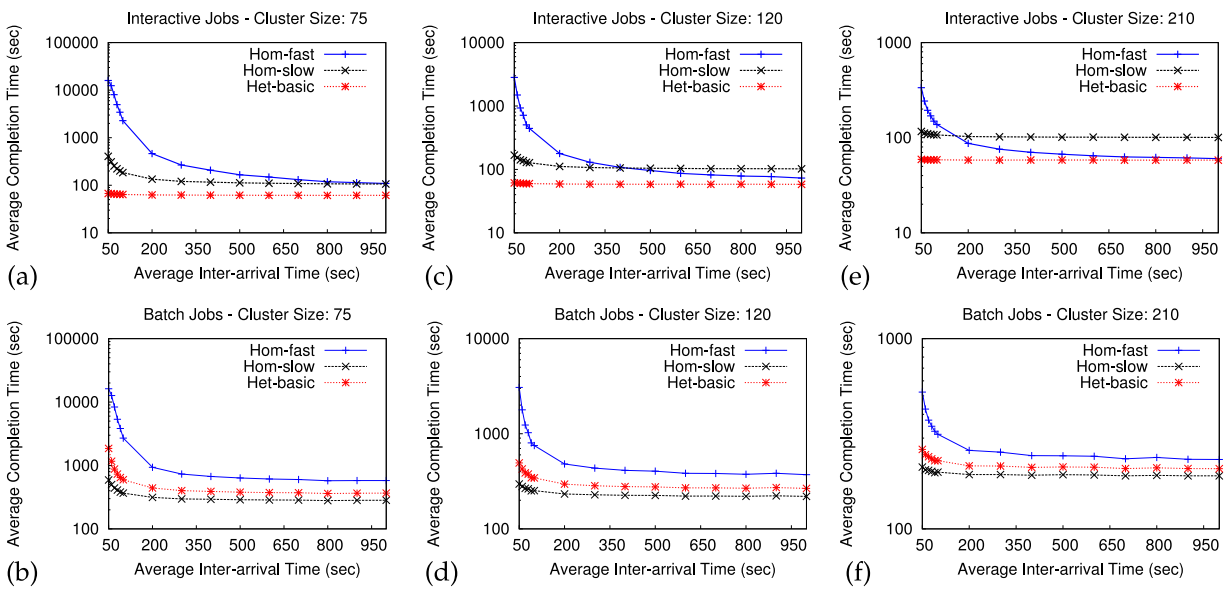


Fig. 12. DyScale versus FIFO scheduler: the completion time of interactive jobs and batch jobs under different configurations, (a)-(b) the Hadoop cluster with 75 nodes, (c)-(d) the Hadoop cluster with 120 nodes, (e)-(f) the Hadoop cluster with 210 nodes.

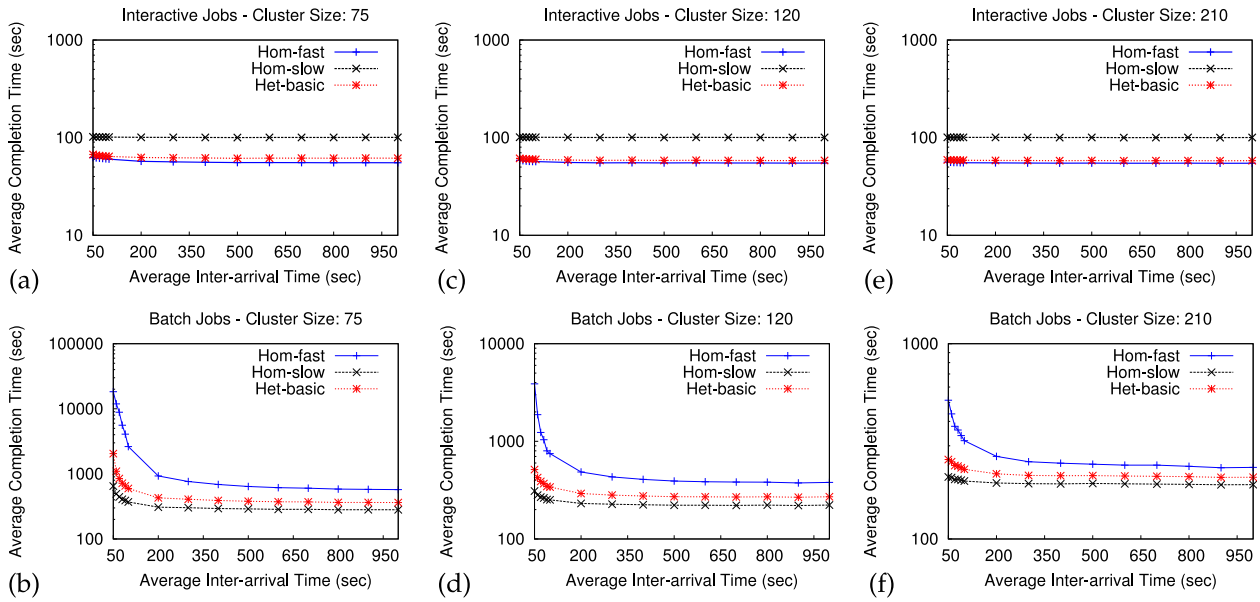


Fig. 13. DyScale versus Capacity Scheduler: the completion time of interactive jobs and batch jobs under different configurations, (a)-(b) the Hadoop cluster with 75 nodes, (c)-(d) the Hadoop cluster with 120 nodes, (e)-(f) the Hadoop cluster with 210 nodes.

improving the performance of batch jobs compared to both *Homogeneous-slow* and *Homogeneous-fast* configurations with different job schedulers.

6 Related Work

There is a body of work exploring power and performance trade-offs using heterogeneous multi-core processors. Some papers focus on the power savings aspect, e.g., Rakesh et al. [17], while others concentrate on the performance aspect, see [18], [19] that examine techniques such as monitoring, evaluating thread performance, and dynamically mapping threads to different core types. Shelepov and Fedorova [20] propose using architecture signatures to guide thread scheduling decisions. The proposed method needs to modify the applications for adding the architecture signatures, therefore it is not practical to deploy. These proposed techniques focus on improving the overall chip-level throughput. The work in [21] explores the per-program performance in addition to the overall chip level throughput when using heterogeneous multi-core processors.

General efforts for power and performance trade-offs focus on a single machine while Hadoop is a distributed framework and needs to manage a cluster environment. It is difficult to apply such traditional techniques for Hadoop. Here, we aim to support different performance objectives for classes of Hadoop jobs, which requires an exact control of running different types of slots in different cores, therefore dynamical mapping of threads to cores is not suitable here.

Performance analysis and optimization of MapReduce processing in the heterogeneous server environment is the subject of several works. The authors in [22], [23] compute the remaining time of each task and use speculative execution to accelerate the "slow" task to reduce the heterogeneity's negative impact. This technique is applicable to our case as well, especially for managing shared spare resources formed by different types of slots.

Load-balancing and load re-balancing approaches in a heterogeneous cluster is used in [2], [24] to allow the faster node to get more data, such that reduce tasks finish approximately at the same time. Xie et al. [25] use data placement to optimize performance in heterogeneous environments. Faster nodes store more data and therefore run more tasks without data transfer. Gupta et al. [26] 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. Lee et al. [27] propose to divide the resources into two dynamically adjustable pools and use the new metric "progress share" to define the share of a job in a heterogeneous environment so that better performance and fairness can be achieved. This approach only allocates resources based on the job storage requirement. Polo et al. [28] modify the MapReduce scheduler to enable it to use special hardware like GPUs to accelerate the MapReduce jobs in the heterogeneous MapReduce cluster. Jiang and Agrawal [29] developed a MapReduce-like system in heterogeneous CPU and GPU clusters.

All the above efforts focus on the server level heterogeneity in Hadoop cluster. In the case of Hadoop deployment on heterogeneous servers, one has to deal with data locality and balancing the data placement according to the server capabilities. One of the biggest advantages of Hadoop deployed with heterogeneous processors is that both *fast*

TABLE 5
Capacity Scheduler: *Queue Capacity* Configurations (in the Brackets, We Provide the Number of Slots in Each Queue for the Cluster with 120 Nodes as an Example)

Configuration	Interactive-Queue capacity (total slots for cluster size 120)	Batch-Queue capacity (total slots for cluster size 120)
Homogeneous-fast	18% (87)	82% (393)
Homogeneous-slow	18% (453)	82% (2,067)

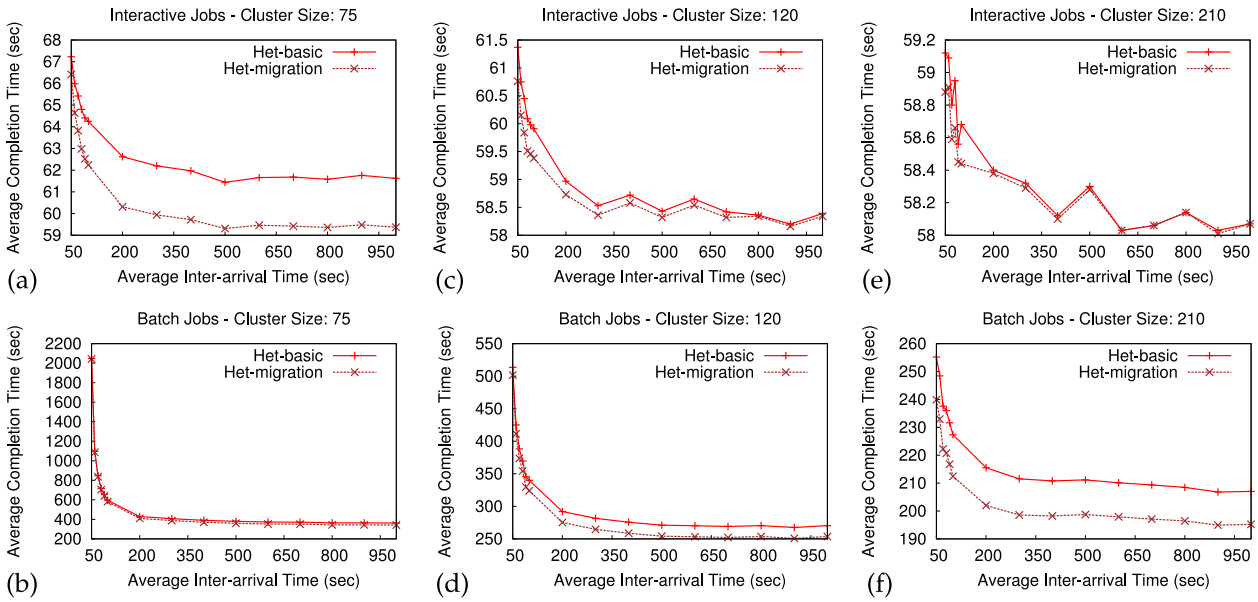


Fig. 14. Comparison of *basic DyScale* and *DyScale with migration* for *Heterogeneous* configuration: the completion time of interactive jobs and batch jobs under different cluster sizes, (a)-(b) cluster of 75 nodes, (c)-(d) cluster of 120 nodes, (e)-(f) cluster of 210 nodes.

and *slow* slots have a similar access to the underlying HDFS data that eliminates data locality issues.

Ren et al. [11] 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 multi-core processors. In [15], the basic idea of using heterogeneous multi-core processors for MapReduce processing is outlined and some initial evaluation results are presented. The current extended version of this paper provides a more detailed description of the scheduling DyScale framework and presents a comprehensive performance evaluation study.

7 Conclusions

In this work, we exploit the new opportunities and performance benefits of using servers with heterogeneous multi-core processors for MapReduce processing. We present a new scheduling framework, called DyScale, that is implemented on top of Hadoop. DyScale enables creating different virtual resource pools based on the core-types for multi-class job scheduling. This new framework aims at taking advantage of capabilities of heterogeneous cores 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. MapReduce jobs can be submitted into different queues, where they operate over different virtual resource pools for achieving better completion time (e.g., small jobs) or better throughput (e.g., large jobs). It is easy to incorporate the DyScale scheduler into the latest Hadoop implementation with YARN [30], as YARN has a pluggable job scheduler as one of its components.

In the future, once the servers with heterogeneous multi-core processors become available, we plan to conduct more testbed experiments using DyScale and a variety of job ordering scheduling policies for achieving fairness guarantees or job completion objectives. Also, using models from earlier work [31], we plan to quantify the impact of node and slot failures on the job completion time as the impact of failed fast or slow slots may be different. Similarly, the allocation of additional slots for re-running failed tasks may impact job completion times and can be supported by special policies in DyScale.

Acknowledgments

This work was completed during F. Yan's internship at HP Labs. E. Smirni and F. Yan are partially supported by US National Science Foundation (NSF) grants CCF-0937925 and CCF-1218758. F. Yan is the corresponding author.

References

[1] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2012.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing mapreduce on heterogeneous clusters," in *Proc. ASPLOS*, 2012, vol. 40, no. 1, pp. 61–74.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. EuroSys*, 2010, pp. 265–278.

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

[6] Z. Zhang, L. Cherkasova, and B. T. Loo, "Benchmarking approach for designing a mapreduce performance model," in *Proc. 4th ACM/SPEC Int. Conf. Perform. Eng.*, 2013, pp. 253–258.

[7] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *Proc. SOCC*, 2012, p. 4.

[8] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a highlevel dataflow system on top of mapreduce: The pig experience," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1414–1425, 2009.
[9] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic resource inference and allocation for mapreduce environments," in *Proc. 8th ACM Int. Conf. Autonomic Comput.*, 2011, pp. 235–244.
[10] A. Verma, L. Cherkasova, and R. H. Campbell, "Play it again, SimMR!" in *Proc. Int. IEEE Conf. Cluster Comput.*, pp. 253–261, 2011.
[11] S. Ren, Y. He, S. Elnikety, and S. McKinley, "Exploiting processor heterogeneity in interactive services," in *Proc. 10th ACM Int. Conf. Autonomic Comput.*, 2013, pp. 45–58.
[12] 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, pp. 105–114, 2012.
[13] C. Bienia, S. Kumar, J. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," Princeton Univ., Princeton, NJ, USA, Tech. Rep. TR-811-08, 2008.
[14] (2013). PassMark Software. CPU Benchmarks. [Online]. Available: http://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+E3-1240+%40+3.30GHz
[15] F. Yan, L. Cherkasova, Z. Zhang, and E. Smirni, "Optimizing power and performance trade-offs of mapreduce job processing with heterogeneousmulti-core processors," in *Proc. IEEE 7th Int. Conf. Cloud Comput.*, Jun., 2014, pp. 240–247.
[16] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Campbell, "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle," in *Proc. NOMS*, 2012, pp. 900–905.
[17] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *ACM SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 64, 2004.
[18] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Proc. 39th Int. Symp. Comput. Archit.*, 2012, pp. 213–224.
[19] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proc. 3rd Conf. Comput. Frontiers*, 2006, pp. 29–40.
[20] D. Shelepov and A. Fedorova, "Scheduling on heterogeneous multicore processors using architectural signatures," in *Proc. Workshop Interaction Between Oper. Syst. Comput. Archit.*, 2008.
[21] K. Van Craeynest and L. Eeckhout, "Understanding fundamental design choices in single-ISA heterogeneous multicore architectures," *ACM Trans. Archit. Code Optimization*, vol. 9, no. 4, p. 32, 2013.
[22] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proc. OSDI*, 2008, vol. 8, no. 4, pp. 29–42.
[23] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, "SAMR: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment," in *Proc. IEEE 10th Int. Conf. Comput. Inf. Technol.*, 2010, pp. 2736–2743.
[24] R. Gandhi, D. Xie, and Y. C. Hu, "Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters," in *Proc. 2013 USENIX Annu. Tech. Conf.*, 2013, pp. 61–66.
[25] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *Proc. IPDPS Workshops: Heterogeneity Comput.*, 2010, pp. 1–9.
[26] S. Gupta, C. Fritz, B. Price, R. Hoover, J. De Kleer, and C. Witteveen, "ThroughputScheduler: Learning to schedule on heterogeneous Hadoop clusters," in *Proc. ICAC*, 2013, pp. 159–165.
[27] G. Lee, B.-G. Chun, and R. H. Katz, "Heterogeneity-aware resource allocation and scheduling in the cloud," in *Proc. 3rd USENIX Workshop Hot Topics Cloud Comput.*, 2011, pp. 4–4.
[28] J. Polo, D. Carrera, Y. Becerra, V. Beltran, J. Torres, and E. Ayguadé, "Performance management of accelerated mapreduce workloads in heterogeneous clusters," in *Proc. 41st Int. Conf. Parallel Process.*, 2010, pp. 653–662.
[29] W. Jiang and G. Agrawal, "Mate-CG: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 644–655.
[30] Apache. (2013). Apache Hadoop Yarn. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[31] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for mapreduce jobs with performance goals," in *Proc. 12th ACM/IFIP/USENIX Middleware Conf.*, 2011, pp. 165–186.

Feng Yan received the BS degree in computer science from Northeastern University in 2008 and the MS degree in computer science from The College of William and Mary in 2011. Currently he is working towards the PhD degree of computer science at the College of William and Mary. His current research interests are distributed systems, performance tools, MapReduce, priority scheduling, cloud computing, and storage systems. He is a member of ACM and the IEEE.

Ludmila Cherkasova is a principal scientist at HP Labs, Palo Alto. Her current research interests are in developing quantitative methods for the analysis, design, and management of distributed systems (such as emerging systems for Big Data processing, internet and enterprise applications, virtualized environments, and next generation data centers). She is an ACM distinguished scientist and is recognized by Certificates of Appreciation from the IEEE Computer Society and USENIX Association. She is a member of the IEEE.

Zhuoyao Zhang received the bachelor's and master's degrees in computer science from Fudan University in 2006 and 2009, respectively, and the PhD degree in computer and information science from the University of Pennsylvania in 2014. Her research interests include large scale distributed systems and cloud computing. She is now a member of the infrastructure team in Google. She is a member of the IEEE.

Evgenia Smirni is a professor of computer science at the College of William and Mary, Williamsburg, VA. She holds a diploma degree in computer science and informatics from the University of Patras, Greece in 1987 and the PhD degree in computer science from Vanderbilt Universityin 1995. Her research interests include queuing networks, stochastic modeling, resource allocation, storage systems, cloud computing, workload characterization, and modeling of distributed systems and applications. She is an ACM distinguished scientist, and a member of the IEEE and the Technical Chamber of Greece.