Enhancement of Xen's Scheduler for MapReduce Workloads

Hui Kang[(1)] Yao Chen[(1)] Jennifer Wong[(1)] Radu Sion[(1)] Jason Wu[(2)]

[(1)]CS Department
SUNY Stony Brook University
Stony Brook, NY 11794
{hkang, ychen, jwong, sion}@cs.sunysb.edu

[(2)]CS Department
Cornell University
Ithaca, NY 14853
wuja@cs.cornell.edu

ABSTRACT

As the trends move towards data outsourcing and cloud computing, the efficiency of distributed data centers increases in importance. Cloud-based services such as Amazon's EC2 rely on virtual machines (VM) to host MapReduce clusters in order to process large amounts of data with off-the-shelf systems. However, current VM scheduling does not provide adequate support for MapReduce workloads, resulting in degraded overall performance. For example, when multiple MapReduce clusters run on a single physical machine, the existing VMM scheduler does not guarantee fairness across clusters.

In this work, we present the MapReduce Group Scheduler (MRG). The MRG scheduler implements three mechanisms to improve the efficiency and fairness of the existing VMM scheduler. First, the characteristics of MapReduce workloads facilitate batching of I/O requests from VMs working on the same job, which reduces the number of context switches and brings other benefits. Second, because most MapReduce workloads incur a significant amount of I/O blocking events and the completion of a job depends on the progress of all nodes, we propose a two-level scheduling policy to achieve proportional fair sharing across both MapReduce clusters and individual VMs. Finally, the proposed MRG scheduler also operates on symmetric multi-processor (SMP) enabled platforms. The key to these improvements is to group the scheduling of VMs belonging to the same MapReduce cluster.

We have implemented the proposed scheduler by modifying the existing Xen hypervisor and evaluated the performance on Hadoop, an open source implementation of MapReduce. Our evaluations, using four representative MapReduce benchmarks, show that the proposed scheduler reduces context switch overhead and achieves increased proportional fairness across multiple MapReduce clusters, without penalizing the completion time of MapReduce jobs.

Categories and Subject Descriptors

D.4.1 [**OPERATING SYSTEMS**]: Process Management–Scheduling

HPDC'11, June 8–11, 2011, San Jose, California, USA.

General Terms

Measurement, Performance

Keywords

Scheduling, Virtualization, Cloud computing, MapReduce

1. INTRODUCTION

The MapReduce model was proposed for parallel processing of arbitrary large amounts of data [7]. The key concept is to break a job into small tasks which can then be run in parallel on multiple machines, which enables scalability to very large clusters of inexpensive commodity computers. With the emergence of cloud computing infrastructures, including Amazon's EC2, Eucalyptus [9], Cloudera's CDH [5], just to name a few, virtual machines (VMs) have become an attractive entity for hosting MapReduce workloads. For example, instead of setting up complicated hardware and software configurations for large clusters of MapReduce nodes, users can simply choose the number of Amazon EC2 nodes. Each node is pre-installed with the Hadoop framework [1], a popular open source implementation of the MapReduce model. These nodes, in fact, are virtual machine instances with various capacities in terms of the number of CPUs, disk spaces, memory sizes, etc. Charges are billed on the amount of resources being used, i.e., from the time job flow begins processing until it is terminated. The use of virtual machines to deploy MapReduce tasks provides security between users, ease of development process, and resource savings.

MapReduce clusters consist of a large number of nodes for redundancy and fault tolerance. Each node in the cluster is assigned a certain number of data chunks, which are in turn duplicated on several nodes based on a distributed filesystem [11]. A MapReduce task is divided into two phases: *map* and *reduce*. During the map phase, the input data is broken down into smaller problem instances for several nodes to work on. In the reduce phase, the results from the smaller problems are combined to obtain the solution for the original input. Although the input data blocks for each node could be different, the operation defined by the programmer is identical for all nodes running the MapReduce job. In practice, data centers build a MapReduce cluster for a user by deploying the MapReduce nodes as VMs. To improve scalability, reliability, and security, VMs in the same MapReduce cluster tend to be distributed across different physical machines as shown in Figure 1. This is done for three main reasons. First, the capacity of each physical machine limits the number of MapReduce VMs it can host. The size of a typical MapReduce cluster ranges from tens to hundreds of nodes. Next, if any failure occurs on a single physical machine, no single MapReduce cluster is affected. Finally, each VM can be isolated, to some extent, from a security breach on another physi-

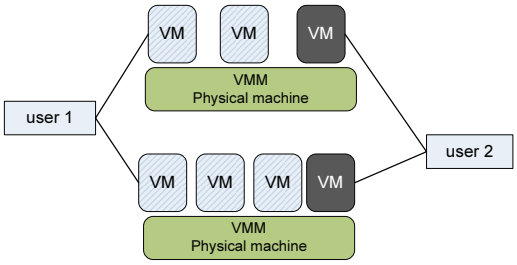


Figure 1: Multiple MapReduce clusters deployed in a virtualized data center. VMs with the same shade belong to one cluster.

cal machine. With multiple MapReduce cluster nodes deployed on each physical machine, as well as the heterogeneity of VM capacities, it is imperative that the underlying virtual machine monitor (VMM), aka hypervisor, efficiently and fairly schedules each VM, and therefore each data center user.

Xen provides a powerful open source solution for hosting multiple VMs by virtualizing different resources on the same physical machine. Its default scheduling mechanism is a credit scheduler. One key benefit of the credit scheduler is that the scheduler provides proportional fair share CPU time to each VM based on its CPU allocations. Unfortunately, although the credit scheduler enforces fairness and responsiveness to each VM, the characteristics of typical MapReduce workloads make such strategies inefficient and unpredictable. First, the large number of I/O requests significantly increases the context switch overhead when scheduling/descheduling VMs on the physical CPU. Second, when multiple MapReduce clusters are co-located on a single physical machine, fairness can not be guaranteed to each cluster with VM-level fairness. As a result, the completion of a job for a MapReduce cluster in terms of overall execution time and associated cost may increase depending on the MapReduce node allocation and the workload of other co-located MapReduce jobs.

In this work, we propose a new VMM scheduler called MapReduce group scheduler (MRG), which provides MapReduce cluster fairness, reduces unnecessary domain context switching, and scales to symmetric multi-processor machines. We find that in a MapReduce cluster, VMs have homogeneity in their behaviors as they are running the same map and reduce functions. For this reason, MRG sorts the VMs in the run queue based on their priorities, as well as the pending I/O operations. Batching the I/O operations from several VMs achieves the benefit of reducing context switch and potential energy costs. Fairness across multiple clusters is strictly enforced by assigning weights to both clusters and their VMs. In addition, current platforms are multi-core architectures; our scheduler is designed to work effectively as a load-balancing symmetric multi-processor (SMP) scheduler.

The rest of the paper is organized as follows: Section 2 provides an overview of background information. Then we provide a motivational example that illustrates the problem the default credit scheduler has with MapReduce workloads. In Section 4, we describe the goals and characteristics of our enhanced scheduler. We follow with a description of the proposed design and implementation details of our scheduler in Xen. Section 6 provides the details of the setup of our experiments, followed by evaluations of the MRG scheduler. Section 7 discusses the limitations and adding similar mechanisms in other hypervisors. Before we conclude, in Section 8 we present the related research in this area.

2. BACKGROUND

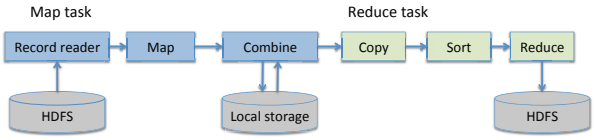


Figure 2: Detailed phases of a MapReduce job.

There are two primary motivating factors for the proposed MRG scheduler. The first is the data-intensive nature and node homogeneity of MapReduce workloads. The second is the lack of fairness provided by the default virtual machine scheduler on MapReduce workloads.

2.1 MapReduce Workload Characteristics

There are two aspects that differentiate a MapReduce workload from traditional web services in a data center. The first aspect is that most MapReduce applications are *data-intensive* and can be divided into multiple subtasks, which results in a large number of I/O-bound tasks at each node that involve input, output, and intermediate data; Figure 2 illustrates the seven phases of execution of a MapReduce job. As described earlier, the frequent I/O requests from each VM lead to additional overhead on the VMM; each I/O operation puts the VM into the wait queue, and delivery of a pending event by the driver domain to one VM lifts it to the top of the CPU run queue. This can result in limiting the responsiveness towards other VMs [16]. Since the combined outputs from all cluster nodes form the final result to a MapReduce job, an increased waiting time of any of the VMs can degrade the performance of the whole MapReduce cluster. In addition, this I/O interrupt on the VMM also affects some CPU-bound jobs. For example, suppose that a set of VMs from two different MapReduce clusters, a CPU-bound task (C_{cpu}) and an I/O-bound task ($C_{I/O}$) are allocated to a single physical machine. Assume the scheduler selects and runs a VM from C_{cpu} and completes its work quickly, the cluster must still wait for the completion of the other C_{cpu} VMs to complete the job. Before the scheduler can allocate the CPU to the next VM, the driver domain satisfies a batch of I/O requests from a number of VMs in $C_{I/O}$. As a result, these $C_{I/O}$ VMs are boosted to the top of the run queue, delaying the completions of the remaining C_{cpu} VMs. Additionally, if all the $C_{I/O}$ VMs can not complete their operations, the $C_{I/O}$ task also does not benefit from the scheduler biasing mechanism.

The second aspect of MapReduce is the homogeneity of each node in a MapReduce cluster. Traditional applications hosted in data center are typically three-tier web services, including multiple web, application, and database servers; each tier can consist of one or many VMs. The mixture of these different VMs facilitates a general-purpose VMM scheduler which can provide fairness to all the VMs running on the physical machine. A MapReduce cluster, however, processes one job at each time round; jobs are run in FIFO order of submit time. All nodes in the cluster perform the same map and reduce function most of time. Although a general-purpose VMM scheduler such as the credit scheduler can guarantee fairness to each VM on the physical machine, it is ignorant of the homogeneity of VMs in the same MapReduce cluster. For example, three VMs in one cluster are co-located on a single physical machine. During their *map* operation phase, all three VMs read the input data and store the intermediate results on their local disks. The I/O requests issued from the three VMs interrupt each other when the driver domain is processing the I/O requests. The

interleaved scheduling of the privileged and unprivileged domains incurs unnecessary context switches, imposing overhead on the virtualization. In contrast, if the scheduler knows that these VMs belong to the same cluster and issue I/O requests at almost the same time, it can delay the scheduling of the driver domain after VMs in this cluster, allowing these I/O operations to be batched at dom0's block layer to reduce the cost of context switches. A side benefit of this deferred scheduling of the driver domain is the energy savings gained from batching multiple I/O processing [28, 31]. When multiple MapReduce clusters share the resources of a physical machine the effect is compounded, and the current VMM scheduling methods do not provide fairness to these clusters.

2.2 Virtual Machine Scheduling in Xen

The default scheduler in Xen is the credit scheduler. Each VM is allotted a set number of credits which are used to schedule a VM on the CPU. The credit scheduler assigns one of four priorities to each VM: *idle*, *under*, *over*, and *boost*. The *idle* state is assigned to VMs which do not require the CPU at the moment (e.g. blocked waiting for I/O). A VM is assigned the *under* state if it still has credits remaining. A VM is assigned the *over* state if it has gone over its allotted credits. A VM is assigned the *boost* state if it receives an I/O event while it is in the *idle* state [21]. The order of priorities from highest to lowest is *boost*, *under*, *over*, and *idle*. VMs are sorted in the run queue based on priorities. When a VM is being descheduled after its allocated time slice, it loses a set number of credits and yields the CPU to the next VM in the run queue. The scheduler gives preference to VMs which have not consumed all its credits, but only by priority, not the actual number of credits.

In Xen, domain0 (dom0) is a special driver domain that acts as a layer of abstraction between the underlying physical hardware and the other domains (the VMs). Hence, dom0 handles all of the I/O processing. Whenever a VM makes an I/O request (i.e. hard disk, network, etc.), a virtual interrupt occurs and the request is passed along to dom0. When dom0 is scheduled to the CPU, it handles any pending I/O requests. Additionally, when an I/O request is completed, the VM is boosted to the head of the run queue in order to handle the I/O. This split driver model offers much flexibility in hosting a wide variety of guest operating systems and using most of the hardware, which have already been supported by the guest in dom0 [10].

A context switch occurs when the scheduler schedules another VM to run (i.e. the state of the running VM is saved and the state of another VM is resumed), similar to process context switching [30]. Context switches in virtualization incur additional overhead over standard context switching. When dom0 handles I/O events for the other VMs, a context switch to dom0 is necessary in order for the I/O requests to be processed. Since CPU resources are used when dom0 handles these I/O events, there is a CPU overhead associated with each I/O request and consequently, each context switch to dom0 [3]. It has been shown that scheduling dom0 too often leads to a higher context switch overhead because fewer I/O requests will be handled per scheduling of dom0 [4].

The overhead in context switches between VMs parallels the overhead associated with context switches between processes on the operating system level [29]. Context switches disturb the effectiveness of the system caches. For example, context switches have a severe negative impact on the translation lookaside buffer (TLB). Each time a context switch occurs, the TLB must be flushed. As a result, the utilization of the TLB decreases and more TLB misses occur. Since each TLB miss requires several slower memory accesses, context switches can significantly reduce the throughput of a system [30]. In a related study [22], as application workloads increase, the average response time increases by as much as six times in Xen. The performance overhead is due to L2 cache misses [22]. Once again, this can be attributed to context switching: some portion of the L2 cache must be re-populated after each context switch. Thus, there are additional memory accesses as well as a larger strain on the CPU. The effect of cache affinity on performance is extensively studied in [15].

Although there are negative effects of context switching, VMs running I/O-bound process can benefit from fast context switch between VMs. When a VM issues an I/O request, it is moved to the *idle* state. As a result the scheduler selects another VM or the driver domain to run on the CPU. Once the I/O request is satisfied, additional context switches are necessary to schedule the waiting VM such that it can continue. Thus, the increased rate of context switching between I/O tasks increases the rate at which I/O-bound requests are made.

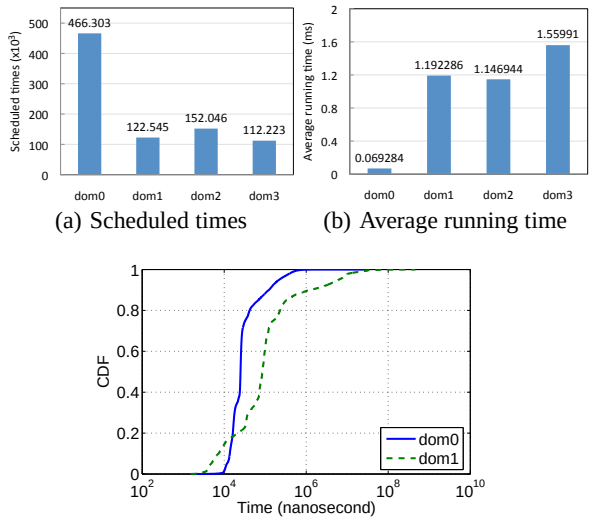
3. MOTIVATIONAL EXAMPLE

In this section we present a concrete example to illustrate the main drawbacks of the existing default credit scheduler when applied to MapReduce workloads. Figure 3(a) and (b) show the scheduled times and average running time of four domains on the same physical machine during a five minute profiling under MapReduce workloads. In this experiment, dom0 is the driver domain and the other three domains (dom1-3) belong to two MapReduce clusters. One cluster (dom1 and dom2) ran a word count application and the other one (dom3) a grep application. As Figure 3(a) illustrates the time scheduled to dom0 by the credit scheduler is four times greater than the other domains. This occurs because dom1-3 delegate their I/O operations to dom0 which functions like a proxy; the sum of scheduling times for the three domains is approximately the same as that of dom0.

The average running time of each domain (Figure 3(b)) is inversely proportional to its scheduled times. Because the average running time of dom0 is much less than 30ms, which is the default time quantum of the credit scheduler, we know that dom0 spends most of its time on I/O processing. The cumulative density functions (CDF) of the running time for all domains (dom1-3) exhibt the same behaviors as dom0. Figure 3(c) and (d) represent the CDFs for dom0 and dom1. In both cases, more than 90% of the running times are less than 1ms, which is significantly smaller than the 30ms time quantum. Since each scheduling of a domain corresponds to a context switch, we can conclude that almost every descheduling of an unprivileged domain (i.e, dom1-3) incurs a scheduling of the driver domain (dom0). While low latency for an I/O event is necessary for each domain, the default credit scheduler does not consider when each domain in the MapReduce cluster should send an I/O request to dom0 and thus results in inefficient and excess context switches.

Figure 4 illustrates an excerpt from a trace file of the above experiment under two cases, illustrating the root causes to the above problem with the default Xen scheduler on a MapReduce workload. In case 1, dom1 and dom2 attempt to issue I/O requests to dom0 and when a request is processed, that VM is boosted to the head of the run queue, preempting other domains. As described in Section 2.1, if the scheduler has cluster group support and dom1 and dom2 have similar task behaviors (e.g. frequency of I/O requests) their I/O requests can be batched. By deferring dom0 in the scheduler until after dom2, two I/O requests are processed by a single scheduling of the dom0 kernel. In this case, the device controller can be used effectively and unnecessary context switches by the scheduler are eliminated.

Another problem, illustrated in case 2 of Figure 4, is that the



(a) Scheduled times (b) Average running time

(c) CDF of running time

Figure 3: Motivating example. (a) Scheduled times for each domain in 5 minutes. (b) Average run time for each domain in 5 minutes; (c) CDF of running time for dom0 and dom1.

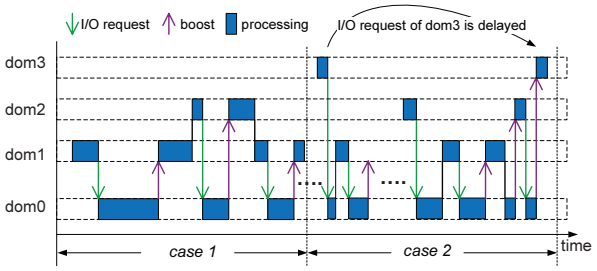


Figure 4: Excerpt of the trace file of domain scheduling. Case 1: interleaving of domains issuing I/O requests; case 2: delayed execution (I/O request of dom3 is delayed by dom1 and 2.)

scheduler continuously processes I/O for the VMs in one cluster and delays the execution of VMs in the other one. Even worse, if some domains in a cluster are scheduled first and then interrupted by a large number of scheduling of VMs from a different cluster, the remaining subtasks may be considered as stragglers and some backup task will need to be executed by MapReduce. Fundamentally, these problems are caused by the lack of knowledge about the relationship between VMs and clusters, as well as the limitation of fairness at the VM-level.

4. MRG SCHEDULER

In order to address the aforementioned limitations of the default credit scheduler on MapReduce workloads, we present the MRG scheduler. The scheduler has three main goals:

- to reduce the domain switch overhead: The scheduler incorporates the knowledge of the homogeneous behaviors of all cluster nodes to re-order the run queue in favor of batching I/O requests and reducing the number of context switches.
- fairness: We consider the fairness in terms of the MapReduce cluster, as well as each single VM.
- scalability on a SMP-enabled machine: The scheduler should behave well with virtual CPU migration or affinity on a SMP machine.

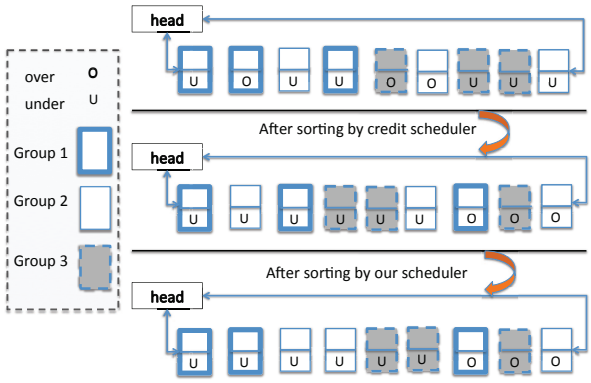


Figure 5: Sorting VCPUs in the run queue (We do not show dom0, boost, and idle state for simplicity.). The run queue is a doubly linked list. The scheduler picks up the VCPU to run on the PCPU from head of the run queue. Each VCPU has the priority of *over* or *under*.

In this section we present the new MRG scheduler in terms of the modifications made on the default credit based scheduler.

4.1 Cluster Grouping

The virtual CPU (VCPU) of each VM is scheduled to run on the physical CPU (PCPU) based on its state, priority, and the availability of the PCPU. When a VCPU's state becomes runnable, it is inserted in the run queue of the PCPU. Xen's credit scheduler sorts the VCPUs from high priority to low and then for VCPUs with the same priority, the credit scheduler places them in FIFO order. In order to take advantage of the characteristics of the MapReduce workload, we propose to additionally group the VMs belonging to the same MapReduce cluster together in the run queue.

Periodically, the MRG scheduler sorts the VCPUs in the run queue such that the high priority VCPUs are lifted to the first portion of the run queue. If two VCPUs have the same priority, FIFO priority is given. This indicates that VMs in different MapReduce group can be interleaved. Thus, in addition to the priority and time when entering the run queue, the proposed scheduler considers the group information which is applicable to the MapReduce VMs. In the proposed scheduler, VMs in the same cluster are grouped together if they have the same priority assignment. As an example in Figure 5, the default credit scheduler divides the run queue of a PCPU into two regions based on the VCPU priority of *over* or *under*. VMs in *under* priority are higher than *over* as they still have credits available. The proposed scheduler then reorders the VCPUs within each priority region. As a result, VMs in the same cluster are placed together in the same priority region.

As described in Section 3, VMs are interrupted by the driver domain due to a temporary boost of I/O processing, leading to inefficient scheduling. We address this issue by considering the needs of dom0 differently than other VMs. Because VMs in the same MapReduce cluster group exhibit similar operations, if one VM in the group issues I/O requests and enters the *blocked* state, the probability that other VMs in the same cluster will issue I/O requests as well is high. Hence, dom0 should be deferred to after a cluster group of VMs. On the other hand, if a boosted dom0 is deferred until after a group of VMs which are not performing any I/O requests, the latency and response time of other I/O-pending VMs will be affected. We propose to remedy this issue by predicting a

VM's likelihood to perform any I/O operation. We use the running time of the VCPU in its last scheduled cycle to predict whether it will issue any I/O request in a short time after it is to be scheduled in the next run cycle. This is motived by the data-intensive characteristic of MapReduce workloads. If the running time of a VCPU is smaller than threshold S, this means that the VCPU is more likely to issue an I/O request and block in the next cycle. In this paper, we set S to 10^5ns as more than half of the running time of a VM is less than 10^5ns as shown in Figure 3. At the same time, S should not be a large number to prevent delaying dom0 aggressively. The MRG scheduler places the VCPUs from the cluster group prior to dom0 in order to batch the I/O requests together, and therefore additionally reduce the number of context switches and improve I/O response latency.

4.2 Cluster Fairness

In order to provide fair scheduling to each MapReduce cluster group and prevent VM starvation, it is necessary to first provide proportional fair CPU time to each group on the physical machine and then to each VM belonging to a group. For example, VMs from two clusters C_1 and C_2 are co-located on a physical machine; C_1 consists of four VMs and C_2 two VMs. Suppose all VMs have the same resource allocations. Then C_1 has $\frac{2}{3}$ of the CPU power and C_2 uses the remaining $\frac{1}{3}$. When C_1 is scheduled, the VMs in C_1 will each run consecutively. VMs from C_2 will not be picked by the scheduler until the allocated CPU time for C_1 is used. The key insight behind group scheduling is that by letting the scheduler track the running time of each cluster and allowing context switches only in the same cluster, tasks in the same group proceed at nearly the same rate. A side benefit is that if the MapReduce cluster is running on a network filesystem, more savings can be made in terms of cache coherence and memory sharing [18].

Our proposed MRG scheduler uses a two-level scheduling hierarchy. The terminology credit is still used in our work to denote the CPU allocation unit, the same as in the credit scheduler. At the first level, the scheduler allocates credits across groups using the weighted fair sharing. The weight of a cluster group is the sum of each VM's credit in this group. The user can also allocate credits to each group. At the second level, the scheduler allocates its PCPU among the VMs in one group using each VM's credit. Note when the user explicitly allocates credits to a group, the credits of the VMs in that group are only used as weights for the second-level intra-group scheduling. In this case, the credit assigned to VM i is normalized as $C_g \times (C_i / \sum C_j)$, where C_g is the group credit and $C_i / \sum C_j$ is the fraction of total VCPU credits in the group for VM i.

Figure 6 shows an example of the scheduling hierarchy. The credits for the three groups are respectively 600, 300, and 900, which means the CPU sharing with a 2:1:3 ratio. Within group 1, the ratio of sharing is 2:1 and the two VMs are allocated $\frac{2}{9}$ and $\frac{1}{9}$ of the overall CPU time. When group 1 is scheduled to run on the PCPU, the scheduler only considers the two VMs in group 1 and allocates corresponding CPU time to them. Group 1 yields the CPU to VMs in other groups in the run queue when it consumes all of its credits. The scheduler guarantees that each VM is given the CPU time as its weighted shares as long as there is a running task in it.

The two-level scheduling strategy, providing the capability of specifying CPU sharing in both group and VM level, allows the cloud provider and the user to follow their own resource allocation rules, without worrying about each other. The provider allocates credits to each group based on the demand of the group user. In turn, the group user who has a set of VMs to form a MapReduce cluster can allocate credits to each VM.

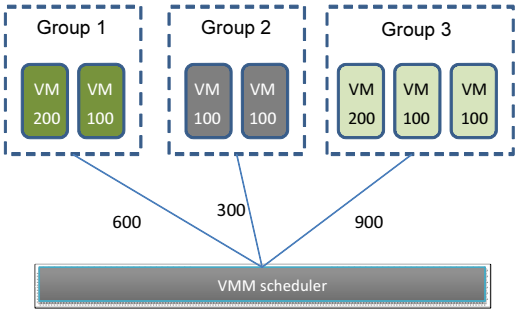


Figure 6: Example of two-level scheduling hierarchy. Group 1, 2, and 3 have the credit of 600, 300, and 900. The sharing ratio between groups is 2:1:3. In each group, CPU time is allocated to individual VMs using their credits (number in each VM) as sharing weight.

It is possible for VMs in one group to issue too many I/O requests in a batch in which case all VMs in the group are in the *blocked* state. In order to eliminate wasted CPU cycles, we introduce a timeout counter t to each group. The timeout starts when the scheduler finds that all VMs in the group are in I/O blocking state and the group has remaining credit. Once the timeout expires and there is still no runnable VM, the group must yield the CPU to the next group in the run queue. Section 4.2.1 analyses how to set t properly.

There are two priorities defined for a group: *under* and *over*. *Under* means the group has remaining credit, while *over* denotes the group has used up its credit. Like the existing scheduler, *under* has a higher priority than *over*. For VMs, we use the same priority relationship as the credit scheduler.

Two-level scheduling can be accomplished by extending the method in Section 4.1, which we propose to group VCPUs belonging to the same group in the run queue and assign them different priorities. First, the group, which the VM on the head of the run queue belongs to, is selected as the current running group on the PCPU. Second, because VCPUs vary in their remaining credits, VCPUs in the same group can be spread across different priority regions as shown in Figure 5. Because the scheduler is only allowed to select VCPUs in the same group, it must check each region for any available VCPU. As described in Section 5, our implementation achieves a time complexity of ($O(n)$) in picking the next VCPU from the run queue.

4.2.1 *Analysis of timeout*

In this section, we explore how to set an appropriate value of the timeout counter t, which affects the fairness among different cluster groups and I/O latency. If t is set too low, the MRG scheduler can switch between clusters of VCPUs quickly, prior to completion of pending I/O requests, and therefore cluster-level fairness cannot be guaranteed. The other extreme, i.e, a large t value, can make the entire system underutilized by blocking cluster groups from executing when all VCPUs in the running cluster are blocked. In between these two extremes, the scheduler must balance the trade-off between two factors: cluster-level fairness and utilization. These two factors are directly correlated with VCPU credits and I/O blocking time. Therefore, they are used to derive the value for timeout t.

First, we define credit-remaining ratio (CRR) for each group as

$$\texttt{CRR} = \frac{\sum credit\ remaining\ of\ each\ dom\ in\ the\ group}{Total\ credit\ of\ the\ group}. \quad (1)$$

CRR is the fraction of CPU credits that are remaining for the group. With CRR close to 1, it means the group is far from using up its assigned CPU credits and therefore it can wait for some time, whereas when CRR approaches 0, it means the group has consumed most of its CPU credit and we tend not to wait for a long time. We define t_w as the wakeup latency between the time when all VCPUs in a group are blocked and any of them is boosted. We initialize t_w to 1 ms and then update the value only when all VCPUs of the current running group are blocked. By combining CRR and t_w, we calculate the timeout as follows:

$$t = t_w \times \texttt{CRR}. \quad (2)$$

Since t_w is changed only on the condition that all VCPUs of the running group are blocked by a set of consecutive I/O requests, this update imposes minimal scheduler overhead. As described in Section 4.1, a group of blocked VCPUs are placed in front of dom0, which can process a subset of the tasks from the batch of requests. As a result, it is very likely that at least one of the blocked VCPU can be boosted.

4.3 Symmetric Multi-Processor Support

In a SMP-enabled platform, each PCPU has its own run queue of VCPUs. Load balancing across each core is crucial for efficient CPU utilization. VCPUs can be migrated from one core to another to improve performance. Two factors must be considered by the scheduler prior to migrating a VCPU. First, migrating loads from other cores should not cause severe starvation of VCPUs which have already existed in the run queue. Second, the costs of migrating a VCPU include giving up a warm cache and resorting VCPUs in the run queue.

In the current implementation of the credit scheduler, load balancing across cores is invoked when the running VCPU yields the PCPU. If the priority of the next VCPU in the current run queue, `snext`, is higher than OVER, it is allowed to run on the CPU; then no VCPU migration occurs. Otherwise, the scheduler first checks the run queue of its peer CPUs to see whether migrating a VCPU can improve the responsiveness of the overall system. The search begins from the core on the same socket and iterates the VCPUs of its run queue. If there is a VCPU in the peer's run queue having a priority higher than `snext` and its cache is not hot on the peer CPU, this VCPU can be migrated to the current CPU run queue. The VCPU's cache is considered hot, if its last scheduled time is less than 1ms.

We propose a basic modification to the existing load balancing algorithm in order to support the cluster groups. We allow load balancing only if the cluster group of the next VCPU in the current run queue is of *over* priority. For convenience, we will use *vc* to denote the next VCPU in the current run queue and *g* to denote its group. We first consider the case that the group of the next VCPU in the peer's run queue is different from *g*. If the priority of the peer's group is higher than *g*, we steal the next VCPU from the peer's run queue if that VCPU's priority is higher or equal to *under*. Since there is only two priority options for a group, the peer's group must have an *under* priority. In this situation, no matter which priority *vc* has, continuing running it will further exceed the CPU sharing of its group. But for the peer group with *under* priority, stealing an *under* VCPU to run on the current PCPU immediately will not only reduce its waiting time in the run queue, but also increase the progress of other *over* VCPUs in the group if there is any.

If the priorities of the two groups are the same, i.e., both are in *over* priority, the scheduler continues to compare the priorities of the two next VCPUs of the run queues. Here, we apply the same rule as the credit scheduler uses, i.e., the scheduler only steals VCPU from other queues which has a higher priority than *vc* and *vc*'s priority is smaller than *under*. Besides the benefits mentioned earlier, for MapReduce workloads this results in limiting the variance of running time among VCPUs in the same group.

In the second case when the group of the next VCPU in the peer's run queue is the same as *g*, we directly compare the two next VCPUs. When the group is of the *under* priority, we use the credit scheduler's rule again to reduce the waiting time of *under* VCPUs in the run queue. However, when the current group is *over*, we must not starve other groups with *under* priority in the current queue. Thus, we only steal VCPUs with *boosted* priority.

Cache affinity must also be considered. Traditionally cache affinity is exploited by the OS scheduler to improve performance; with a high cache affinity most of a process's data and instructions are accumulated in the cache and thus the process runs more efficiently when being scheduled to this core again. Because of the data-intensive nature of MapReduce workloads, it is crucial that the VMM scheduler can achieve a desirable cache affinity level for such workloads.

A coarse granularity cache management algorithm is proposed in [19] for real-time tasks: the scheduler which those real-time VCPUs bind with migrates real-time VCPUs in a coarser-granularity (e.g., 1 second) to balance their loads based on CPU usage history across cores. The MRG scheduler borrows the feature of coarse granularity from [19] to achieve the balance between load balancing and cache affinity. That is for any VCPU of a MapReduce cluster, MRG invokes VCPU migration once every second, instead of 10ms. Coarse granularity load balancing is necessary, in order to give a VCPU a higher chance to be scheduled on the current CPU and to gain benefits from cache affinity, especially for the I/O heavy VMs, like MapReduce nodes.

5. DESIGN AND IMPLEMENTATION

In this section, we provide details of the design and implementation of the scheduler for Xen version 3.4.1[1]. We demonstrate that the proposed scheduler can be easily integrated with the existing credit scheduler based on its per-CPU run queues. Because there is no modification to the guest operating system, any open source and closed source OS can be used to host the MapReduce node as a virtual machine with the proposed scheduler. First, we added two configurable variables to the guest domain: `group_id` and `group_credit` to support MapReduce cluster groups. VMs with the same `group_id` belong to a single MapReduce cluster. The `group_id` is then assigned to the VCPU(s) owned by the VM. The basic algorithm is presented in Figure 7.

As described in Section 4.1, each PCPU has its own run queue to sort the VCPUs in the priority order. To support grouping VCPUs with the same `group_id` in the run queue, it is necessary to link VCPUs in the same group together. Furthermore, in the run queue, VCPUs in the same group can be in different priority regions. Therefore, it is necessary to track all VCPUs based on its running PCPU, as well as its `group_ID`. A double linked list is used for each group as shown in Figure 8. When sorting VCPUs in

[1]The modification is downloadable as a patch to Xen v3.4.1 (changeset 19717) at `http://ntl.cewit.stonybrook.edu/mrg-scheduler`

```
// A vcpu becomes runnable
Insert_VCPU_runqueue(vcpu):
   pcpu = current;
   if vcpu is dom0:
       check if vcpu can be deferred;
   else:
       insert vcpu to its group based on priority;

// every 10ms
Sched_tick:
   sort vcpu in the runqueue;

// every 30ms
Sched_acct:
   allot credits to VCPUs of VMs in each group;
   if timer=1 second:
       run load balancing algorithm;
```

Figure 7: Pseudo code for MRG scheduler.

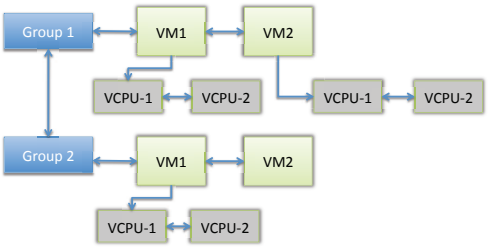


Figure 8: Linked list of VCPUs for each group

the run queue, the scheduler can use the group list to quickly find the VCPUs with the same `group_ID`. When one group is scheduled to run on the PCPU, the scheduler can locate the VCPUs in this group, which may be in discrete priority regions. In addition, recall that the scheduler needs to decide whether an unprivileged VCPU can be inserted in front of dom0's VCPU based on its last running time. A `last_rtime` field is added to each VCPU to record the previous running time.

Once the scheduler has constructed the data structures for all VCPUs and groups, the two-level scheduling mechanism is used to coordinate the VCPUs to run on the PCPU. Each VCPU and group are initialized by assigning proportional weights of credits from the configuration files. In the main function, the scheduler updates the credits for VCPUs and groups and sorts VCPUs in the run queue of the PCPU. In practice, these two events of sorting VCPUs in the run queue and credit accounting are separated (Figure 7). The former occurs in every scheduling tick (10ms) only if any of the VCPU's credits have been updated, which happens in each accounting period (30ms). Since the existing credit scheduler already assigns constant weights for credit accounting, our implementation retains these weights as default. After sorting the VCPU in the run queue, the head of the run queue is the next one to be scheduled. With the two-level scheduling policy, VCPUs in the same group are allowed to run on the PCPU as long as there are remaining credits for the group and the blocking timer has not expired. The scheduler tries stealing a VCPU to the current run queue every second if the priority condition is satisfied. Note that all the modifications to the VMs configured as MapReduce nodes do not affect VMs of other types. If the scheduler finds the VCPU does not have a `group_id`, this VCPU is treated by the default scheduling.

6. EXPERIMENTAL EVALUATION

We begin by evaluating the performance of a single MapReduce cluster group running on a single physical machine. This solely evaluates the improvement of group clustering (Section 4.1) within the scheduler which is the underlying principle of the other two improvements. We compare the results of four benchmark applications with Xen's default credit scheduler (baseline) and analyze the root causes of the improvements. Subsequently, we evaluate the proposed MRG scheduler running multiple cluster groups hosted on three physical machines and mixing different benchmarks.

All experiments were performed on Dell servers, running the modified Xen hypervisor (v3.4.1). The VMs run Fedora 8 and kernel v2.6.18.8 in all cases with one virtual CPU, 512MB memory, and 40G disk space. We run Hadoop-0.20.2 on the VMs with its default configuration, except for setting the replication factor of data blocks to 2.

We use `xentrace` to capture the virtualization events such as context switches between VMs, scheduled times, VCPU block wait, and running time. `xentrace` is able to output the trace buffer from the hypervisor level to the user level in a binary format. We trace only the events related to VM scheduling when running `xentrace`. They are runstate changes, domain wake, schedule switch, and so forth.

The benchmark workloads we use throughout the experimental evaluations are from the examples shipped with the Hadoop distribution and a SQL-like query in Hive [26] – a data warehouse infrastructure built on Hadoop. We choose three types of benchmarks from Hadoop: *word count*, *grep*, and *sort*. Word count counts the number of occurrences of each word in the input file. Word count is I/O-intensive, while grep searching for a regular expression is limited by CPU resources. The sort benchmark in the experiment uses the merge-sort algorithm; it reads chunks of data to each map node, generating intermediate data, and comparing the results. Thus, although the sort application has a lot of I/O operations, it still consumes much more CPU resources than word count. We choose these workloads because they represent typical MapReduce applications and exercise different system metrics for evaluating the performance of Hadoop. Moreover, they are the building blocks of software frameworks such as searching, indexing, pattern mining, and optimizing advertisement. At last, we use a Hive query to a database, including join, select, and insert operations. The Hive query (hql) differs from the previous benchmarks in that the query will be parsed into several MapReduce jobs by the query compiler and represent more advanced data structure of MapReduce. The dataset and Hive query script can be downloaded at the project website [2]. The configurations of the four applications are listed in Table 1. The input data to word count, grep, and sort are generated by the tools shipped with the Hadoop, while the Hive database is populated from a local disk file. We choose a wide variety of data input sizes for these jobs such that the execution time ranges from approximately 10 to nearly 80 minutes.

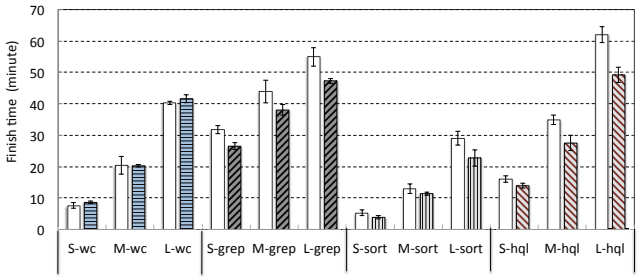
6.1 Single Cluster Group

We begin by running a 4-node single cluster of MapReduce VMs on a single physical machine, which is the machine type 1 of Table 2. In total there are five VMs running on the machine, including dom0. A single core on the physical machine is used to run all VMs, as well as dom0. By doing so we can study the effects of grouping VMs on reducing the number of unnecessary context switches caused by boosting dom0 and other VMs. We run the four

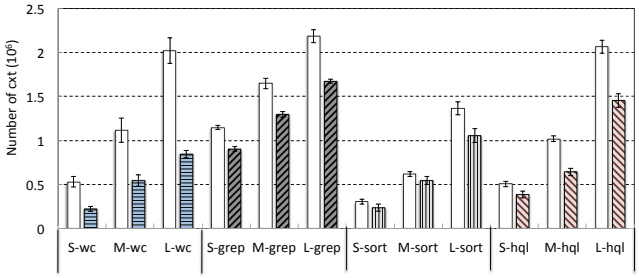
[2] `http://ntl.cewit.stonybrook.edu/mrg-scheduler`

Table 1: Application types and input sizes for the workloads

Applications	*Word count*		*Grep*		*Sort*		*Hql*	
Category	Label	Input size	Label	Input size	Label	Input size	Label	Table size (# entries)
Small	S-wc	2G	S-grep	10G	S-sort	512M	S-hql	100,000
Medium	M-wc	5G	M-grep	15G	M-sort	1G	M-hql	200,000
Large	L-wc	10G	L-grep	20G	L-sort	2G	L-sql	400,000

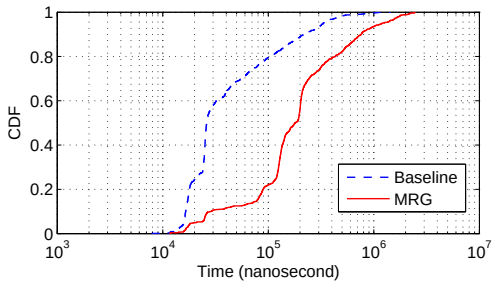


(a) Finish time

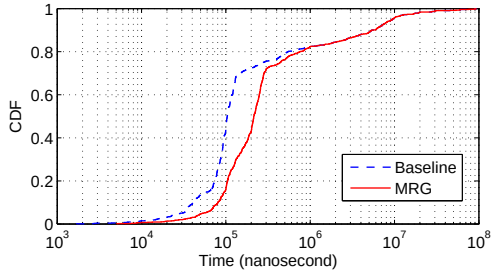


(b) Number of context switches

Figure 9: Finish time and number of context switches for word count, grep, sort, and Hive benchmarks. Each benchmark run with small, medium, and large input sizes. (a) finish time; (b) number of context switches. White bars show the results using the baseline, while results of the MRG scheduler are plotted in different patterns to denote each application.



(a) dom0



(b) dom1

Figure 10: CDF of running time for dom0 and dom1 using the baseline and MRG scheduler.

types of benchmark applications using the baseline and the proposed MRG scheduler and measure the workload completion time and the number of context switches during the executions. We also vary the job size of each benchmark to see how the proposed scheduler scales when the job size increases. The results are presented in Figure 9. White bars show the results using the baseline, while results of the MRG scheduler are plotted in different patterns to denote each application. All results are the average of 10 runs; error bars show the standard deviations calculated.

The first observation from the results is that for all types of applications, the proposed scheduler is able to reduce the overhead of context switches, without penalizing the job finish times of the MapReduce jobs. However, the effects on the finish time and reducing the number of context switches vary depending on the application type. For example, for the word count of all input sizes the MRG scheduler can reduce the number of context switches by more than a half, while the savings for the grep application is around 22%. Second, we see that batching I/O requests improves the job completion time for grep, sort, hql by about 25% of all input sizes, but not for word count. This is due to the fact that word count is more I/O-heavy, compared with the other applications. Therefore, each node has more I/O requests to issue to dom0. As a result, when dom0 of word count is scheduled, it can process more batched I/O requests than that of grep and sort, reducing more interleaved context switches between other VMs and dom0. On the other hand, since processing of an I/O from a VM at dom0 could be postponed after other VMs are inserted, the savings from batching I/O will be offset by this delay. However, given the significant improvements on the context switch overhead, the adverse effect on the finish time for word count is negligible.

Figure 10 compares the cumulative distributions of running time of two VMs for a medium input size sort application, which is profiled for 10 minutes during the middle of execution using the baseline and MRG scheduler. Note that the *x*-axis is on a log scale. There are two interesting observations here. First, we find that for dom0 (Figure 10(a)), the MRG scheduler has running times concentrated at values larger than 10^5ns, which are much longer than the default credit scheduler. That is 80% of the running time of dom0 are less than or equal to 10^5ns with the baseline, while about 20% fall in that range with MRG. Second, the difference between the CDFs of the default and proposed scheduler for the unprivileged domain (dom1), shown in Figure 10(b), is not as significant as dom0. This is because that MRG scheduler is clustering I/O requests from a group of VMs in a MapReduce cluster before the real device driver starts to process them. Thus, the MRG scheduler makes dom0 run longer to process more I/O when dom0 is scheduled. In addition, provided that the completion times are not impacted by the proposed scheduler, the increased average running

Table 2: VM deployment in the multiple cluster testbed.

Consolidation level	group 1	group 2	Machine type
3 VMs / host	2	1	1
5 VMs / host	3	2	2
4 VMs / host	2	2	2
total	7	5	–

Machine type 1: Pentium D 2.8GHz, 2MB L2 cache, and 3G memory
Machine type 2: Core 2 Duo 3.0GHz, 6MB L2 cache, and 6G memory

times for individual domains indicates that the number of context switches must be decreased, confirming our previous results.

6.2 Multiple Cluster Groups

In this section we performed studies by deploying two MapReduce clusters across three physical machines to evaluate the overall performance of the MRG scheduler. Table 2 summarizes the placement of VMs in the testbed. We use physical machines with different capacities and have various consolidation levels to resemble the practical allocation of provisioning multiple MapReduce clusters, like Amazon EC2. In particular, since VMs in a cluster are placed across several physical machines, machine capability and network traffic will affect the performance of MapReduce jobs. In Table 2, the first column lists the number of VMs hosted on the physical machines. The second and third columns are the number of VMs belonging to each cluster. All physical machines run the same patched Xen and OS version, the same for all guest domains. In total, cluster 1 and 2 consist of 7 and 5 VMs, respectively. We also enable the two cores on the machine in this experiment to show the effect of the proposed SMP load balancing support.

The CPU resource allocation to these VMs is described as follows. On three physical machine each cluster is assigned 500 credits and all VMs are configured to have the same credit of 256. Thus, two clusters have equal share of the CPU resource on each physical machine, which in turn allocate the same CPU resources to each VM in the same group. VMs of different cluster groups, however, may have different allocated CPU resources. For example, according to the two-level credit assignment policy, the two clusters on the second physical machine (row 2 in Table 2) have equal share of CPU time, while the ratio of CPU time of VMs in cluster 1 to those in cluster 2 is 2:3.

6.2.1 Performance

Figure 11 compares the finish time for different combinations of MapReduce jobs running on the two clusters using the baseline and the proposed scheduler. There are four groups of results (i-iv) in Figure 11; each group consists of one application job on each cluster with varying input sizes. For example, the two white bars of the rightmost four bars of group (i), represent the finish times of S-wc and S-grep by the baseline scheduler, the two colored bars by the MRG scheduler. As expected, although the proposed scheduler outperforms the baseline in overall finish time, the effects vary depending on the combinations of job types. For any combination with word count (group (i-ii) in Figure 11), MRG scheduler improves the finish time of the other group application. Since word count is the most I/O-heavy workload among the three types of jobs, its VCPU will be boosted more aggressively than VCPUs of VMs running the grep and sort tasks. As a result of the two-level scheduling policy, the VCPUs of grep and sort clusters can run their allocated time slots without being interrupted by VCPUs of other clusters. Thus, the disadvantage of boosting VCPU of I/O-heavy task is effectively limited. Similar to the single-group results, the

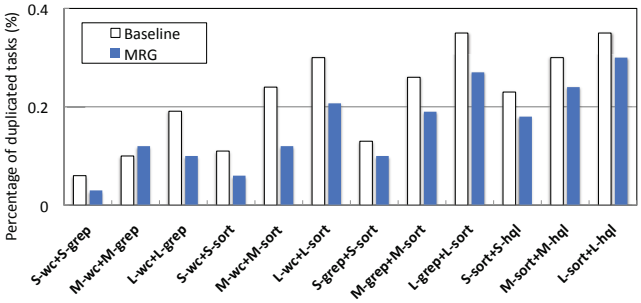


Figure 12: Percentage of duplicated tasks.

performance of word count does not change too much. For the job combinations of grep, sort, and hql shown in group (iii) and (iv), the MRG scheduler outperforms the baseline by $\sim 30\%$, on average.

A unique feature of the MapReduce framework is *speculative execution*, which provides high availability and robustness of the cluster [7]. However, speculative tasks are not free; the downsides are duplicated tasks and more resource contentions, especially in the virtualized environment. Although the MRG scheduler runs in the hypervisor level unlike other MapReduce scheduling algorithms, it has indirect impact on this metric. Thus, it is important to evaluate the number of speculative tasks.

A speculative task is launched when the progress of one running task is below average in either the map or reduce phase. A slow task can be caused by many reasons such as hardware failure, resource contentions, or misconfiguration. Although a speculative task is important to achieve fault-tolerance, it has negative impacts due to the extra energy and resource costs. Figure 12 shows the percentage of duplicated map tasks from the results of two MapReduce clusters. The MRG scheduler reduces duplicated tasks in all but one case. Furthermore, in Figure 12 the percentage of duplicated tasks is higher with the sort application running in one cluster. This is because the sort application generates a much larger number of intermediate and final results than word count and grep. An increased number of fetch failures within the map and reduce results spans the creation of a larger number of duplicated tasks. Overall these results imply that the proposed scheduler is able to balance the progress of map tasks running on each VM of the cluster.

Combining with the results of execution times (Figure 11), we can conclude that the MRG scheduler can deliver performance improvement over the default credit scheduler under various combinations of MapReduce clusters, while reducing the cost of context switching and the number of duplicated tasks. In addition, the loading balancing method of MRG allows VCPUs from the two cluster groups to perform well on SMP platforms.

6.2.2 Fairness

We evaluate the fairness between two MapReduce clusters on a physical machine by comparing the relative difference between the running time of one group by the scheduler with the time defined by a generalized processor sharing (GPS) scheduling algorithm [23]. In a GPS system, the amount of service time obtained by each process i during a time interval (τ_1, τ_2) is proportional to its CPU weight ψ_i. That is if process i and j are continuously runnable with fixed real numbers ψ_i and ψ_j, then GPS satisfies

$$\frac{T_i(\tau_1, \tau_2)}{T_j(\tau_1, \tau_2)} = \frac{\psi_i}{\psi_j}, \quad j = 1, 2, \cdots, N. \tag{3}$$

This concept can be easily extended to running multiple MapRe-

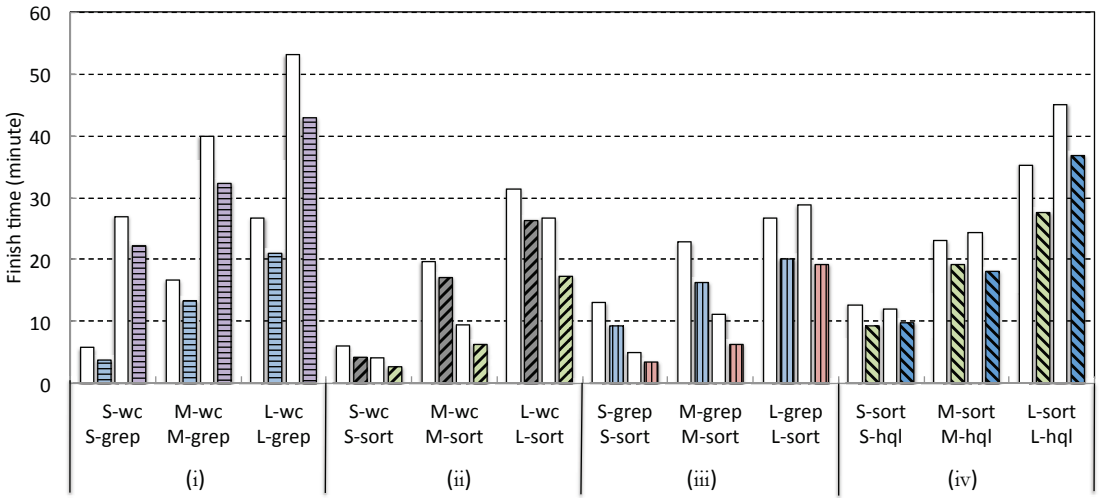


Figure 11: Finish times of two clusters running different combinations of MapReduce applications. (i) Cluster 1: word count, cluster 2: grep; (ii) Cluster 1: word count, cluster 2: sort; (iii) Cluster 1: grep, cluster 2: sort; (iv) Cluster 1: sort, cluster 2: Hive query. White bars show the results using the baseline, while results of the MRG scheduler are plotted in different patterns.

duce clusters on a physical machine. We consider clusters as processes and the credit assigned to each cluster as the weight used by the scheduler. Equation 3 can be transformed to

$$\frac{T_{C_i}(\tau_1,\tau_2)}{T_{C_j}(\tau_1,\tau_2)} = \frac{w_i}{w_j}, \quad j = 1, 2, \cdots, N, \tag{4}$$

where $T_{C_i}(\tau_1,\tau_2)$ denotes the CPU time that cluster C_i receives in interval $(\tau_1,\tau_2)$and w_i the credit assigned to cluster C_i.

In practice, however, a GPS scheduler is infeasible because a perfect fair scheduler requires that at least one VM in each cluster must run simultaneously and be scheduled with infinitely small quanta, which is infeasible. Therefore, a practical scheduler S, which aims to guarantee proportional fairness to each cluster, can be evaluated by the lag [2, 20]. However, there is a critical difficulty in using lag directly to evaluate the fairness of VMM schedulers under MapReduce workloads. Because MapReduce workloads do not have 100% CPU usage, the task running in each VM does not consume all the CPU time allotted. Here we defined a modified lag for cluster C_i running in the interval (τ_1,τ_2) as

$$L_i(t) = |T_{C_i,GPS}(\tau_1,\tau_2)\mu - T_{C_i,S}(\tau_1,\tau_2)|, \tag{5}$$

where μ is the average CPU utilization of the VMs of cluster C_i in (τ_1,τ_2). Thus the modified lag can be explained as the difference of CPU time scheduled between the normalized GPS scheduler and the practical scheduler S. Intuitively, the smaller L_i is, the fairer the scheduler S is. In our experiment, because the two clusters are allotted equal number of credits, we have $T_{C_i(\tau_1,\tau_2)} = T_{C_j(\tau_1,\tau_2)}$. This means ideally in any time interval, the two clusters should have the same amount of CPU time.

We compare L_i of the baseline and the MRG scheduler under the job combination of word count and sort, both with large input size. We run `xentrace` for 15 minutes in the middle of the benchmark run on the second physical machine (row 2 of Table 2). During the profiling period all VMs are running MapReduce tasks and keep nearly constant CPU utilization; the VMs running word count have 41% CPU utilization and VMs running sort have 67%. Once we have the trace outputs, which includes timestamps and VCPU's running times, we sum up the VCPU's running time belonging to each cluster for every 30 seconds. The sum is the actual CPU time

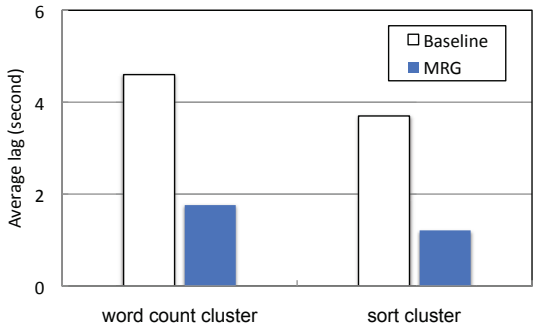


Figure 13: Average modified lag with standard deviation for two clusters running word count and sort.

a cluster receives in the 30-second interval. Because dom0's CPU utilization is low, we obtain word count cluster's and sort cluster's CPU time by GPS as about 6 and 10 seconds, respectively in the 30-second interval. Figure 13 reports the modified lags averaged over the sampling period. We see that the MRG scheduler achieves improvements for both clusters, implying better fairness. Furthermore, we observed the MRG scheduler offers an overall fairness to these MapReduce clusters. For the word count cluster, three VMs have their running time approximately about 2 seconds. The sum is also close to the running time allotted to the cluster by the GPS scheduler. For the sort cluster, the GPS scheduler assigns 10 seconds to the two VMs in a 30-second sampling period. We can see that the running times of sort-1 and sort-2 are around 5 seconds. From the results, we can also conclude that the two-level scheduling policy in MRG scheduler provides fairness to VMs of a MapReduce cluster.

7. DISCUSSION

Our work is motived by running MapReduce in a virtualized environment, which has become the dominant paradigm for large data processing in a very short time. Two key aspects of MapReduce workloads enable the MRG scheduler to perform well. First, all

nodes in a cluster run the same map and reduce function. Second, most MapReduce applications are I/O-intensive. The MRG scheduler performs well when these two conditions hold. However, as the MapReduce technology evolves, some of the conditions may change. For example, in a shared MapReduce cluster, nodes can perform different tasks to improve the cluster utilization by allocating slots to different jobs [33]. This breaks the first condition. At the same time, if some slots are running CPU-intensive tasks, which breaks the second condition, batching I/O will be ineffective. We leave generalizing the MRG scheduler to different MapReduce frameworks as future work.

Although MRG scheduler is designed inside the Xen hypervisor, we consider the possibility of adding similar capabilities to other options of hypervisors such as VMware ESX, KVM [17], and Hyper-V [14]. In fact, because the proposed MRG scheduler takes advantage of batching I/O in a privileged domain and grouping VCPUs of VMs in the same cluster, the architecture of those hypervisors must be considered. *Kernel-based virtual machine* (KVM) converts the Linux kernel into a hypervisor and makes use of many components of Linux kernel such as scheduler and memory management, instead of reimplementing them like Xen and ESX. KVM implements each VM as a regular process, scheduled by the standard Linux scheduler. Thus, we can adapt the Linux scheduler to the MRG scheduler by assigning a cluster ID to each VCPU process and batching I/O requests from VCPU processes in the same cluster. *Hyper-V*, which is Windows 2008 server's hypervisor, isolates virtual machines in terms of partitions. A root partition runs as a privileged partition like dom0 in Xen and is able to create a child partition as a virtual machine. As child partitions do not have access to hardware and the root partition performs all the privileged operations, it is possible to modify the root partition's scheduler to enable the MRG scheduler. VMware's *ESX server* has a quite different architecture than Xen, KVM, and Hyper-V. There is no privileged domain in ESX and thus all VMs run with the same priority. Instead, ESX includes a VMkernel, which provides the functionalities as resource scheduling, I/O stack, and device drivers. As a result, all I/O from VMs will be queued at VMkernel's virtual SCSI layer. In addition, ESX considers VCPUs as *world*s. Thus, reordering I/O could be implemented at the host's queue of the SCSI layer based on the VCPU worlds. In conclusion, the scheduling algorithms of the above hypervisors can also profit from the mechanism of the MRG scheduler.

8. RELATED WORK

In this section, we present the related work on CPU and I/O scheduling in operating systems and several job scheduling algorithms to improve the performance of MapReduce.

8.1 CPU and I/O scheduling

There is a long history of work in CPU scheduling in both academia and industry on supporting a variety of applications ranging from real-time applications [12], e.g., video/music player to resource intensive applications, e.g., web servers [8]. To combine the benefits of different ones, a hierarchy loadable scheduler (HLS) allows users to assign each application its preferred scheduler and organize them based on their priorities [24]. According to [24], the proposed MRG scheduler falls into the category of homogeneous hierarchical scheduler, which uses the same scheduling algorithm throughout the hierarchy. Homogeneous hierarchical schedulers provide a hierarchical isolation between groups of resource consumers, which fits the MapReduce model. However, it is difficult to assign CPU time to each VM under the existing HLS because higher priority processes may preempt lower ones. To address proportional sharing between VMs running simultaneously, assigning each VM some accurate number such as credit (in Xen) or shares (VMware ESX [27]) performs better than setting coarse-granular priorities as in traditional OSs, e.g., Linux, Solaris, and Mac OS X. In addition, group scheduling has been developed previously in Linux to improve the desktop interactivity, e.g., TTY-based and CFS group scheduling [6]; however, to our knowledge, this is the first time that group scheduling has been used directly for the scheduling of VMs and to guide their resource allocations.

Given the open source nature of the Xen hypervisor, a great deal of research has been done to advance it. Specifically, we would like to highlight efforts to improve Xen's support for near–real–time applications [19] and I/O intensive applications [13]. A near-real-time application is a task that does not require strict guarantees, but it still needs low latency and enough CPU resources within a certain amount of time. An example would be a server that handles enterprise telephony. Similar to our situation with MapReduce workloads, the Xen VMM is not aware of the characteristics of near-real-time tasks. Lee et al. [19] modified Xen's credit scheduler to provide better support for near-real-time workloads. Their primary modifications included a scheduling technique that inserted the VCPU of a near-real-time task into a position of the run queue where it would be scheduled before its desired deadline. Their experimental results indicate that the aforementioned modification improves the performance of near-real-time workloads without negatively impacting regular (non-real-time) workloads. In Hu's recent work [13], processor cores are divided into several subsets with each employing a specific scheduling algorithm to handle CPU-intensive or I/O-intensive workloads. However, this imposes the burden of run-time monitoring to distribute VCPUs to proper PCPUs.

Ye et al. [31] proposed the idea of batching hard disk I/O requests in a VM environment to reduce power consumption. By grouping I/O requests, the hard disk is able to remain in the idle state for longer periods of time, thereby lowering power usage. The idea was combined with two more techniques to further prolong the length of idleness: buffering to delay writes and flushing writes early. Their results from their implementation in Xen indicated a considerable savings in energy consumption with a small cost (i.e. increase) in disk access time.

On the operating system level, Zhuravlev et al. [35] examined the issue of shared resource contention in multicore processors. Their survey of different classification schemes for scheduling algorithms established that cache contention was not the only reason for performance degradation. Performance was also influenced by contention for the prefetching hardware and the memory controller and bus. They determined that the cache miss rate of an application was a good indicator of shared resource contention. Consequently, the scheduling algorithms they implemented aimed to allocate applications among the cores such that the miss rate is spread evenly.

8.2 MapReduce performance improvement

Job scheduling in the MapReduce architecture has been studied for improving the fairness and responsiveness, due to the poor performance of the default FIFO scheduling [25, 34, 32]. The approach proposed in this paper is fundamentally different from these existing work in that the MRG scheduler provides hypervisor-level scheduling which tries to fit the characteristics of MapReduce workloads. LATE [34] performs speculative tasks based on a more generic task progress rate, which takes into account node heterogeneity in a consolidated system. Sandholm and Lai [25] use regulated and user-assigned priorities to manage the MapReduce VMs and automatically adjust the resource allocation during run time

to meet the deadline or service level for different jobs. The delay scheduling in [32] solves the conflict between fairness and locality by delaying the launch of a non-local task for a certain amount of time when running multiple MapReduce jobs together. A commonality among these works is that they all consider running the MapReduce jobs in a shared virtualized environment. Thus, we believe that instead of conflicting with each other, the proposed work fills the gap between the high-level job management and the underlying VM scheduling.

9. CONCLUSION

We designed and implemented MRG scheduler, a new Xen scheduler for VMs running MapReduce workloads. The scheduler facilities MapReduce job fairness by introducing a two-level group credit based scheduling policy. Efficiency is improved through batching of I/O requests within a group and elimination of superfluous context switches. Additionally, the proposed mechanism also operates on SMP-enabled platforms. The MRG scheduler was implemented and tested on the Hadoop platform. Evaluations on MapReduce benchmarks show the MRG scheduler can deliver $\sim 30\%$ performance improvement over the default scheduler without runtime overhead, while simultaneously reducing the costs of context switching and duplicated tasks. In terms of fairness, the MRG scheduler provides MapReduce cluster group level fairness and fairness at the VM-level.

10. REFERENCES

[1] Amazon. Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce`.

[2] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Algorithmica*, volume 15, pages 600–625, 1996.

[3] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *USENIX*, pages 387–390, 2005.

[4] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. In *Sigmetrics/Performance*, pages 42–51, 2007.

[5] Cloudera. Cloudera's distribution for Hadoop. `http://www.cloudera.com/`.

[6] J. Corbet. TTY-based group scheduling. `http://lwn.net/Articles/415740/`.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Commun. ACM*, volume 51, pages 107–113, Jan 2008.

[8] K. J. Duda and D. C. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *SIGOPS Oper. Syst. Rev.*, pages 261–276, 1999.

[9] Eucalyptus. Eucalyptus systems. `http://www.eucalyptus.com/`.

[10] K. Fraser, S. Hand, I. Pratt, A. Warfield, R. Neugebauer, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *OASIS*, 2004.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[12] J. Goossens and R. Devillers. Feasibility intervals for the deadline driven scheduler with arbitrary deadlines. In *RTCSA*, pages 54–61, 1999.

[13] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia. I/O scheduling model of virtual machine based on multi-core dynamic partitioning. In *HPDC*, pages 142–154, 2010.

[14] Hyper-V. Microsoft Windows Hyper-V. http://www.microsoft.com/hyper-v-server/.

[15] V. Kazempour, A. Fedorova, and P. Alagheband. Performance implications of cache affinity on multicore processors. In *Euro-par*, pages 151–161, 2008.

[16] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for I/O performance. In *VEE*, pages 101–110, 2009.

[17] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Linux Symposim*, pages 225–230, 2007.

[18] H. Lagar-Cavilla, J. A. Whitney, A. Scannell, P. Patchin, and S. Rumble. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Eurosys*, pages 1–12, 2009.

[19] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the Xen hypervisor. In *VEE*, pages 97–108, 2010.

[20] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, pages 65–74, 2009.

[21] D. Ongaro, A. Cox, and S. Rixner. Scheduling I/O in virtual machine monitor. In *VEE*, pages 1–10, 2008.

[22] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance evaluation of virtualization technologies for server consolidation. In *TECH Report of HP*, 2007.

[23] A. K. Parekh and R. G. Gallager. Generalized processor sharing approach to flow control in integrated services networks, the single-node case. In *IEEE/ACM Transactions on Networking*, volume 1, pages 344–357, June 1993.

[24] J. D. Regehr. Using hierarchical scheduling to support soft real-time applications in general-purpose operating systems. In *Ph.D. Thesis*, 2001.

[25] T. Sandholm and K. Lai. MapReduce optimization using regulated dynamic prioritization. In *Sigmetrics/Performance*, pages 299–310, 2009.

[26] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, pages 996 – 1005, 2010.

[27] VMware White paper. VMware vSphere: the CPU scheduler in VMware ESX 4.1. 2010.

[28] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *OSDI*, pages 117–129, 2002.

[29] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the denali isolation kernel. In *OSDI*, pages 195–209, 2002.

[30] S. Yamada and S. Kusakabe. Effect of context aware scheduler on TLB. In *Workshop on Multi-Threaded Architectures and Applications*, pages 1–8, 2008.

[31] L. Ye, G. Lu, S. Kumar, C. Gniady, and J. H. Hartman. Energy-efficient storage in virtual machine environments. In *VEE*, pages 75–84, 2010.

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

[33] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user MapReduce clusters. In *Technical Report, EECS, Berkeley*, 2009.

[34] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, pages 29–40, 2008.

[35] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, pages 129–142, 2010.