Dynamic Fractional Resource Scheduling for HPC Workloads

Mark Stillwell*, Frédéric Vivien*† and Henri Casanova*
*Department of Information and Computer Sciences
University of Hawai'i at Mānoa, Honolulu, U.S.A.
†INRIA, France

***Abstract*—We propose a novel job scheduling approach for homogeneous cluster computing platforms. Its key feature is the use of virtual machine technology for sharing resources in a precise and controlled manner. We justify our approach and propose several job scheduling algorithms. We present results obtained in simulations for synthetic and real-world High Performance Computing (HPC) workloads, in which we compare our proposed algorithms with standard batch scheduling algorithms. We find that our approach widely outperforms batch scheduling. We also identify a few promising algorithms that perform well across most experimental scenarios. Our results demonstrate that virtualization technology coupled with lightweight scheduling strategies affords dramatic improvements in performance for HPC workloads.**

I. Introduction

The standard method for sharing a cluster among High Performance Computing (HPC) users is batch scheduling. With batch scheduling, users wanting to run applications submit *job requests*. Each request is placed in a queue and waits to be granted an *allocation*, that is, a subset of the cluster's compute nodes, or simply *nodes*. The job, i.e., the running application, has exclusive access to these nodes for a bounded duration.

A problem with batch scheduling is that jobs are allocated integral numbers of nodes. Consequently, if a job uses only a fraction of a node's resource (e.g., half of the processor cores, a third of the memory), then the remaining fraction is wasted. It turns out that this is the case for many jobs in HPC workloads. For example, in a 2006 log of a large Linux cluster [1], more than 95% of the jobs use under 40% of a node's memory, and more than 27% of the jobs effectively use less than 50% of the node's CPU resource (due to time spent performing I/O or network communication and synchronization). This observation has been made repeatedly in the literature [2]–[6]. Some of these jobs could coexist on the same node while suffering only marginal performance degradation (e.g., due to cache interference). It may even be beneficial to force jobs to coexist on the same node in spite of reduced individual job performance in order to improve overall turn-around times and fairness. Batch schedulers take the opposite approach and use integral resource allocations with no time-sharing of nodes. As a result, jobs can be denied immediate access to the cluster in spite of cluster resources not being fully utilized, which is problematic from the perspective of the users.

A second problem with batch schedulers is that they do not optimize a user-centric objective function; it is known that there is a sharp disconnect between user concerns (low job turn-around time, fairness) and the schedules achieved in practice [7], [8]. Batch schedulers provide myriad configuration parameters by which cluster administrators can influence scheduling behaviors, but these parameters are not directly related to any desirable user-centric objective function.

We propose a novel approach that addresses both above problems. We address the problem of integral resource allocations by allowing resource allocations that are fractional (e.g., a job can be allocated only 70% of a resource). Furthermore, these allocations can be modified on the fly, by changing resource fractions allocated to a job and/or by migrating the job to different nodes. To address the second problem we define an objective performance metric. In the literature a traditional metric, which can be optimized both to achieve higher job performance and higher fairness among jobs, is the *stretch*, i.e., the slowdown factor experienced by a job because the whole platform is not dedicated to its use. The stretch is computed based on the job's execution time measured on a dedicated platform. Stretch optimization has been studied assuming that job execution times are known [9] or that reliable estimates are available [10]. These techniques are not widely used on HPC installations, in part because obtaining accurate job execution time estimates is difficult [11]. Consequently, we take a drastic departure from previous work: we do not assume any knowledge about job execution times, thereby freeing users of the need to estimate them. However, this mandates the use of an alternate optimization metric.

Our proposed approach, which we term *dynamic fractional resource scheduling* (DFRS), amounts to carefully controlled time-sharing that optimizes a clear objective and that is enabled by state-of-the-art virtual machine (VM) technology. We consider a non-clairvoyant, on-line scenario where jobs are submitted over time.

Our contributions in this work are:

- We propose several algorithms for solving the on-

line non-clairvoyant DFRS problem;
- We evaluate our algorithms via simulation using synthetic and real-world HPC workloads and compare them to standard batch scheduling algorithms;
- We show that our approach widely outperforms batch scheduling and we identify algorithms that perform well across most experimental scenarios.

This paper is organized as follows. In Section II we formalize the DFRS problem. In Section III we propose several DFRS algorithms. We describe our experimental procedure for evaluating these algorithms in Section IV and present results in Section V. Section VI discusses related work. Section VII concludes the paper with a summary of our results and a discussion of future directions.

II. Dynamic Fractional Resource Scheduling

To avoid the problems induced by integral allocations used in batch scheduling, we allocate *fractions* of resources. This amounts to time-sharing nodes between jobs. The classical time-sharing solution for parallel jobs is gang scheduling, which, because of its drawbacks (see Section VI), is used far less often than batch scheduling for HPC clusters. In this work we opt for time-sharing in an uncoordinated and low-overhead manner, which is enabled by virtual machine (VM) technology. Our hope is to circumvent the problems of batch scheduling without suffering from the drawbacks of gang scheduling.

A. System Overview and Use of VM Technology

We target clusters of *nodes* managed by a resource allocation system that relies on VM technology. The system responds to job requests by creating collections of VM instances on which to run the jobs. Each VM instance runs on a physical node under the control of a VM monitor that can limit its resource usage. All VM Monitors are in turn under the control of a VM Manager that specifies resource usage constraints for all instances. The VM Manager can also preempt instances, and migrate instances among physical nodes. Several groups in academia and industry are developing systems following or amenable to this conceptual architecture [12]–[17].

VM technology allows for accurate sharing of hardware resources among VM instances while achieving performance isolation. The Xen VM monitor [18] enables CPU-sharing and performance isolation in a way that is low-overhead, accurate, and rapidly adaptable [19]. Furthermore, sharing can be arbitrary. For instance, the Xen Credit CPU scheduler can allow three VM instances to each receive 33.3% of the total CPU resource of a dual-core machine [20]. This allows a multi-core physical node to be considered as an arbitrarily time-shared single core. Virtualization of other resources, such as I/O resources, is more challenging [21], [22] but is an active area of research [23], [24]. Recent work also targets the virtualization of full memory hierarchies (buses and caches) [25].

In this work we simply assume that accurate resource sharing and performance isolation can be achieved with VM technology: one can start a VM instance on a node and allocate to it reasonably precise fractions of the resources on that node. Given this capability, whether available today or in the future, our approach is generally applicable to many resource dimensions. In our experiments we include solely CPU and memory resources, the sharing of which is well supported by VM technology today.

Resource allocation decisions must be based on estimates of jobs' resource needs. One simple solution is to use VM instance monitoring mechanisms [26]. VM instance resource needs can also be discovered via a combination of introspection and configuration variation. With introspection, one can determine current CPU resource consumption by inferring process activity inside of VMs [27], and memory pressure by examining memory page eviction activity [28]. Another option is to use configuration variation, by which one can dynamically vary the amount of resources given to VM instances, track how they respond to the addition or removal of resources, and infer resource needs [27], [28].

B. Problem Statement

1) Platform and Job Models: We consider a homogeneous cluster based on a switched interconnect and with some type of network-attached storage. Users submit job requests to the cluster. Each job consists of one or more tasks to be executed in parallel, each task running within a VM instance. Our goal is to design algorithms to make sound resource allocation decisions. These decisions could include picking initial nodes for instances, setting resource consumption rates for each instance, migrating instances between nodes, preempting and pausing instances (by saving them to local or network-attached storage), and rejecting or postponing incoming job requests.

Each task has a *memory requirement*, expressed as a fraction of total node memory, and a *CPU need*, which is the fraction of available CPU cycles that the task needs to run at maximum speed. For instance, a task could require 40% of the memory of a node and would utilize 60% of the node's CPU resource in dedicated mode. We assume that these quantities are known and do not vary throughout job execution. Memory requirements could be specified by users or be discovered on-the-fly, along with CPU needs, using the discovery techniques described in Section II-A.

Memory capacities of nodes should not be exceeded.

In other words, we do not allow the allocation of a node to a set of tasks whose cumulative memory requirement exceeds 100%. This is to avoid the use of process swapping, which can have a hard to predict but almost always dramatic impact on task execution times. By contrast, we allow a node to be allocated to a set of tasks whose cumulative CPU needs exceed 100%. Let T denote the execution time of a task if it is given all its CPU need, say a fraction α of the CPU resource of a node. The CPU fraction actually allocated to the task can change over time, e.g., it may need to be decreased due to the system becoming more heavily loaded. When a task is given less than its desired CPU need its execution time is increased proportionally. The task completes once the cumulative CPU resource assigned to it up to the current time is equal to $\alpha \times T$. In this work we target HPC workloads, which mostly comprise regular parallel applications. Consequently, we assume that all tasks in a job have the same memory requirements and CPU needs, and that they must progress at the same rate. We enforce allocations to provide identical instantaneous CPU fractions to all tasks of a job, as non-identical fractions needlessly waste resources.

2) Performance Objective: A difficult question is that of the objective function to optimize. Recall that no such function is optimized by batch schedulers, which is a problem we wish to address. A metric commonly used to evaluate batch schedules post-mortem is the *stretch* (or slowdown) [29]. The stretch of a job is defined as its turn-around time divided by its turn-around time had it been alone on the cluster. For instance, a job that could have run in 2h on the dedicated cluster but ran instead in 4h due to competition with other jobs experienced a stretch of 2. In the literature a proposed way to optimize both for average performance and for fairness is to minimize the maximum stretch [29], as opposed to simply minimizing average stretch, which is prone to starvation [30]. Maximum stretch minimization is known to be theoretically difficult, as even in clairvoyant settings there does not exist any constant-ratio competitive algorithm [30]. Nevertheless, heuristics can lead to good results in practice [30]. Stretch minimization, and especially maximum stretch minimization, tends to favor short jobs. A known problem in practice is that real-world workloads contain many small jobs that simply fail at or soon after being launched, therefore introducing a strong bias in the metric. The common solution, which we adopt here, is to use a variant of the stretch metric called the *bounded stretch* (or "bounded slowdown", with the terminology in [31]). In this variant, the turn-around time of a job is replaced by the maximum of the turn-around time and of a threshold value. We set the threshold to 30 seconds, and hereafter we use the term stretch to mean bounded stretch. Finally, note that, given the definition of our objective function, we never reject job requests (to prevent infinite stretches). However, we may postpone them until the corresponding jobs can be scheduled on the cluster.

Our goal is to minimize the maximum job stretch. One difficulty here is that the stretch is computed based on job execution time on the dedicated cluster, which is not known. Current batch scheduling systems require that users provide execution time estimates, but it is well known that these estimates are typically (wildly) inaccurate [11]. Relying on them to optimize job stretch directly is thus a losing proposition. In this work, we take a radical departure from batch scheduling and simply do not assume *any* knowledge about job execution times! Instead, we define a new metric, the *yield*, which does not use job execution time. The *yield* of a job's task is the fraction of the CPU resource of the node allocated to the task divided by the task's CPU need. We contend that optimizing yield is more feasible than optimizing stretch given that, on the one hand, job execution time estimates are inaccurate, and, on the other hand, CPU needs can be discovered (see Section II-A). Since we assume that all tasks within a job have identical CPU needs and are allocated identical CPU fractions, they all have the same yield which is then the yield of the job.

Both the yield and the stretch capture an absolute level of job "happiness", and are thus related. The yield is a ratio of rates while the stretch is a ratio of times. The yield can, in fact, be seen as the inverse of an instantaneous stretch. In this work, we develop algorithms that explicitly maximize the minimum yield. The key, and intriguing, question is whether optimizing the yield is sufficiently related to optimizing the stretch so that DFRS outperforms standard batch scheduling in terms of stretch, or even produces schedules with good maximum stretch in the absolute sense. When presenting our results, we report on stretch values since they are the ones used in the literature to evaluate schedules.

In previous work we proposed an efficient algorithm for off-line minimum yield maximization [32]. Although interesting from a theoretical point of view, this off-line algorithm cannot be applied in practice to our on-line problem: job requests arrive non-deterministically and a resource allocation decision must be made upon the arrival of each request. Each time resources are allocated to the tasks of a new job, CPU fractions allocated to other tasks running in the cluster may need to be dynamically adjusted in an attempt to reach a new optimal maximum minimum yield.

Note that, in some cases, resources may not be fully utilized once the minimum yield has been maximized. Unused resource fractions can then be allocated to jobs as a way to improve average yield, which improves overall platform utilization. If the system is truly under-

subscribed, then one could temporarily turn off one or more nodes in order to save energy.

III. ALGORITHMS FOR DYNAMIC FRACTIONAL SCHEDULING

In this section we propose several DFRS algorithms. Our goal is twofold: develop an algorithm that delivers good performance and that is as low-overhead as possible. By starting from a simplistic algorithm and gradually introducing additional mechanisms (e.g., preemption, migration), we aim at determining what mechanisms are truly necessary. All the proposed algorithms (except DYNMCB8-STRETCH-PER, described hereafter) attempt to maximize the minimum yield directly. All the proposed algorithms follow the same underlying principle. While they do map tasks to a single node in a way that does not overcome that node's memory capacity, they often deliberately map to a single node tasks whose cumulative CPU need far exceeds the node's computational power. The hope is that the resulting performance degradation is offset by the added benefit of starting tasks earlier than in a batch scheduling scenario.

A. Greedy Algorithms

GREEDY – For a given job, this algorithm first identifies the nodes that have sufficient available memory to run at least one task of the job. Among these nodes, the one with the lowest total *CPU load* is allocated a task. We define the CPU load of a node as the sum of the CPU needs of all the tasks allocated to it. If, after this allocation, that node has insufficient remaining memory to accommodate another task, then it is removed from consideration. All tasks of the job are allocated to nodes in this manner. If one or more tasks cannot be allocated to a node, then the job is postponed using bounded exponential backoff: the job is reconsidered $\min(2^{12}, 2^{count})$ seconds later, where $count$ is the number of failed scheduling attempts for the job. Otherwise, all running jobs, including the new job, are given a CPU fraction corresponding to a yield of $1.0/\max(1.0, \Lambda)$, where Λ is the maximum CPU load over all nodes. This maximizes the minimum yield given the current allocation of jobs to nodes.

With GREEDY, once a job has been allocated to nodes there may be unused CPU resources on some nodes. We use an additional heuristic to take advantage of remaining CPU resource. This heuristic never decreases but may increase the resource allocation of a job. It selects the job with the lowest *total* CPU need (summed over the job's tasks) among all jobs whose tasks are allocated on nodes where some CPU resource remains unused. Increasing the yield of this job as much as possible provides the best cost/benefit ratio in terms of resource consumption vs. improvement to the overall average yield. This is repeated until no job can see its yield further increased. This simple heuristic aims at maximizing the average yield in a view to increasing cluster utilization. This heuristic is not optimal, but computes new CPU allocations quickly and improves the average stretch in our experiments. All algorithms (except DYNMCB8-STRETCH-PER) use this average yield optimization heuristic.

A weakness of GREEDY is its admission policy. If a short-running job is submitted to the cluster but cannot be executed immediately due to memory constraints, then it is postponed. However, since we assume no knowledge of job execution time, there is no way to correlate the duration of the postponing with job execution time. In fact, a job may be postponed forever, leading to unbounded maximum stretch. The only way to circumvent this problem is to force the admission of all newly submitted jobs, which may require that one or more running jobs be paused via preemption.

The important question then is: which jobs should be preempted? To answer this question we define a priority based on the *virtual time* of a job. The virtual time is essentially the total subjective execution time experienced by that job. Formally, this is the integral of its yield between its release time and the present moment. For example, a job that starts and runs for 10 seconds with a yield of 1.0, then is paused for 2 minutes, and then restarts and runs for 30 seconds with a yield of 0.5 has experienced 25 total seconds of virtual time ($10 \times 1.0 + 30 \times 0.5$). From the virtual time, we define the following priority: priority $= \frac{\max(30,\text{flow time})}{(\text{virtual time})^2}$ where the *flow time* of a job is the time elapsed since its submission. We always consider jobs for pausing or moving by increasing order of priority. Conversely, we always consider jobs for resuming by decreasing order of priority. A job that has not yet been allocated any CPU time (null virtual time) has an infinite priority. The presence of the flow time in the numerator ensures that any paused job will eventually be resumed, hence preventing starvation. The power of two in the priority function is used to increase the importance of the virtual time with respect to the flow time, thereby giving an advantage to short-running jobs. Although we do not give a theoretical justification for this power of two, experiments not reported in this paper have shown that the same priority function without the power of two leads to markedly inferior results. Note that, in the definition of the priority function, we use the same 30 sec bound as in the bounded stretch. This bound ensures that a job is never eligible for pausing shortly after it has begun. Using this priority function we can now propose two greedy algorithms that use preemption.

GREEDY-PMTN – Like GREEDY except that if an incoming job cannot be started then some of the running

jobs are paused. To do so, this algorithm goes through the list of currently running jobs in order of increasing priority and marks them as candidates for pausing until the incoming job could be started if all these candidates were indeed paused. It then goes through the list of these marked jobs in decreasing order of priority and determines for each whether it could instead be left running due to sufficient available memory. After this step, running jobs that are still marked as candidates for pausing are paused, and the new job is started. Paused jobs may be resumed at any future event (i.e., job submission or completion), provided there are sufficient resources. GREEDY-PMTN attempts to resume paused jobs in order of decreasing priority. A paused job may or may not be restarted on the nodes on which it was previously running.

GREEDY-PMTN-MIGR – Like GREEDY-PMTN, but with the added capability of moving rather than pausing running jobs. More precisely, jobs that are paused on one event (to make room for an incoming job) may be resumed to a different set of nodes during that same event, rather than waiting for a future event as in GREEDY-PMTN. This amounts to a job migration.

B. Non-Greedy Algorithms

The GREEDY algorithm and its variants construct resource allocations incrementally. We also consider algorithms that try to optimize the resource allocation globally. To this end, we build on the MCB8 algorithm [32], which is itself based on an algorithm proposed by Leinberger et. al. in [33], whose principle is explained in the next paragraph. Because we have two resource dimensions (CPU and memory), our resource allocation problem is related to the bi-dimensional version of bin packing, or bi-dimensional *vector packing*. One important difference between our problem and vector packing is that our jobs have fluid CPU needs. This difference can be addressed as follows. Consider a fixed value of the yield, Y, that must be achieved for all jobs. Fixing Y amounts to transforming all CPU needs into *CPU requirements*: simply multiply the CPU need by Y. The problem then becomes exactly vector packing, with a set of tasks with fixed CPU and memory requirements, that must each be mapped on an individual node. A binary search on Y is used to find the highest yield for which the vector packing problem can be solved (our binary search has an accuracy threshold of 0.01).

MCB8 is a bi-dimensional vector packing heuristic. It splits the tasks into two lists, with one list containing the tasks with higher CPU requirements than memory requirements and the other containing the tasks with higher memory requirements than CPU requirements. Each list is then sorted by non-increasing order of the largest of the two requirements. Initially one assigns the first task of one of the lists (picked arbitrarily) to the first node. Subsequently, one searches for the first task that can fit on that node from the list that goes against the current imbalance, e.g., if the node's available memory exceeds its available CPU resource (in percentage), one searches for a task whose memory requirement exceeds its CPU requirement. The purpose of this step of MCB8 is, on each node, to keep the total requirements of both resources in balance, so that one is not depleted while the other is still underutilized. If no task in this list fits on the node, then the same process is applied with the other list. When no task in either list can fit on the node, one repeats this process for the next node. If all tasks can be assigned in this manner then resource allocation is successful. We can now develop a family of resource allocation algorithms based on this procedure:

DYNMCB8 – This algorithms runs the MCB8 algorithm at every event (job submission or completion) over all jobs in the system. If no valid allocation of nodes to jobs can be found however small the minimum yield is, the job with the smallest priority is removed from consideration (and preempted if it was running), and the algorithm tries again. This strategy is aggressive and may lead to prohibitively large numbers of preemptions and migrations.

DYNMCB8-PER – Like DYNMCB8, but instead of invoking MCB8 at every event it invokes it periodically, every T seconds. Incoming jobs are placed in a waiting queue until the next scheduling event. The goal is to achieve the benefits of the aggressive DYNMCB8 while mitigating preemption and migration overhead. The name of the algorithm is suffixed with the scheduling period: DYNMCB8-PER-T.

DYNMCB8-ASAP-PER – Like DYNMCB8-PER, but instead of placing new jobs in a waiting queue it attempts to greedily schedule them immediately if possible given memory constraints. The goal of this algorithm is to gain the benefits of DYNMCB8-PER, while providing better response time and allowing potentially short jobs to run to completion between scheduling events.

DYNMCB8-STRETCH-PER – All previous algorithms try to maximize instantaneous yield. Instead, this algorithms tries to account for jobs' past histories to minimize an estimate of the maximum stretch. The algorithm follows the same general procedure as DYNMCB8-PER but with the following differences. At scheduling event i, since we assume no knowledge of job execution times, the best estimate of the stretch of job j is the ratio of its flow time (time since submission) to its virtual time: $\hat{S}_j(i) = flowtime_j(i)/vt_j(i)$. Assuming that a job continues running until scheduling event $i+1$, then $\hat{S}_j(i+1) = flowtime_j(i+1)/vt_j(i+1) = (flowtime_j(i) + T)/(vt_j(i) + y_j(i) \times T)$, where T is the scheduling period, and $y_j(i)$ is the yield that

DYNMCB8-STRETCH-PER assigns to job j between scheduling events i and $i+1$. Similar to the binary search on the yield, here we do a binary search to minimize $\hat{S}(i+1) = \max_j \hat{S}_j(i+1)$. At each iteration of the binary search, a target value $\hat{S}(i+1)$ is tested. From this value the algorithm computes the yield for any job j by solving the above equation for $y_j(i)$. If the computed yield is negative, the job is heuristically given a yield of 0.01 so that no job consumes memory without making progress. If the compute yield is greater than 1, then the job is given a yield of exactly 1 so that it is not allocated more than it can use. At that point, CPU requirements are defined and the MCB8 algorithm can be applied to try to produce a resource allocation for the attempted $\hat{S}(i+1)$ value. This is repeated until the lowest feasible such value is found. If the MCB8 algorithm cannot find a valid allocation for any value of estimated stretch, then a job is removed from consideration using the same criteria as the other algorithms and the algorithm tries again. While the previous algorithms uses a heuristic to improve the average yield, DYNMCB8-STRETCH-PER performs a similar last step to improve the estimated stretch.

All our results are for a 10-minute scheduling period ($T = 600$). Based on experiments with $T = 60$ and $T = 3600$, we found that $T = 600$ is sufficiently small to achieve results comparable to those using the much smaller period, and sufficiently large to lead to overhead comparable to that using the much larger period.

IV. SIMULATION METHODOLOGY

A. Discrete-Event Simulator

We have developed a discrete event simulator that implements our scheduling algorithms and takes as input a number of nodes and a list of jobs. Each job is described by a submit time, a required number of tasks, one CPU need and one memory requirement specification (since all tasks within a job have the same needs and requirements), and an execution time.

Our simulator makes it possible to specify the preemption and migration overhead. The question of properly accounting for this overhead is a complicated one. For this reason we provide two versions of each simulation experiment: one where the overhead is zero and one where this overhead is 5 minutes of wall clock time, which is justifiably high[1]. We call this overhead the *rescheduling penalty*. Note that none of the scheduling algorithms are aware of this penalty or try to schedule around it. In the real world there are facilities available that can allow for the live migration of a running task between nodes [35], but in order to avoid introducing additional complexity we make the pessimistic assumption that all migrations are carried out through a pause/resume mechanism. Note that the periodic algorithms described in Section III use a period larger than the rescheduling penalty. Experiments have shown that using periods shorter than the penalty leads to poor results due to job thrashing.

[1] Consider a 128-task job with 1 TB total memory, or 8 GB per task (our simulations are for a 128-node cluster). Current technology affords aggregate bandwidth to storage area networks up to tens of GB/sec for reading and writing [34]. Conservatively assuming 10 GB/sec, moving this job between node memory and secondary storage can be done in under two minutes.

B. Batch Scheduling Algorithms

We consider two batch scheduling algorithms: FCFS and EASY. FCFS (First Come First Serve algorithm), often used as a baseline comparator in the literature, holds incoming jobs in a queue and assigns them to nodes in order as nodes become available. EASY [36], which is representative of production batch schedulers, is similar to FCFS but enables backfilling to reduce resource fragmentation. EASY gives the first job in the queue a reservation for the earliest possible time it would be able to run under FCFS, but other jobs in the queue are scheduled opportunistically as nodes become available, as long as they do not interfere with the reservation for the first job. EASY thus improves on FCFS by allowing small jobs to run while large jobs are waiting for a sufficient number of nodes.

A drawback of EASY is that it requires estimations of job execution times. In our experiments we conservatively assume that EASY has perfect knowledge of job execution times. While this seems a best-case scenario for EACH, studies have shown that for some workloads some batch scheduling algorithms can, surprisingly, produce better schedules when using non-perfectly accurate execution times (e.g., using inaccurate user-provided estimates, multiplying the perfectly accurate estimate by a certain factor). We refer the reader to the discussion in [37] for more details. At any rate, in these studies the potential advantage of using inaccurate estimates is shown to be relatively small, while our results show that our approach outperforms EASY by orders of magnitude. Our conclusions thus still hold when EASY uses non-accurate execution time estimates.

C. Workloads

For part of our study we use synthetic workloads based on the model by Lublin et al. [38], augmented with additional information as described hereafter. There are a number of reasons for using synthetic workloads. Real workloads are often of poor quality, and do not usually contain all of the information that we require. Also, real workloads are for specific systems, while synthetic workloads are generated using a model instantiated from multiple systems, and so may be more representative.

Through the use of VM technology the CPUs of a multi-core node can be shared precisely and fluidly as a single resource [20]. Thus for each node the total amount of allocated CPU resource is simply constrained to be less than or equal to 100%. However, if the node is multi-core, then 100% CPU resource utilization can only be reached by a single task if that task is CPU-bound and is implemented using multiple threads. A CPU-bound sequential task could only use $100/n\%$ of the node's CPU resource, where n is the number of processor cores. In our synthetic trace experiments we arbitrarily assume quad-core nodes, thus meaning that a sequential task would use at most 25% of a node's CPU resource.

We know of no systematic study in the literature of CPU utilization levels for HPC jobs. We assume that the task in a one-task job is sequential, and, to err on the side of caution, that other tasks are multi-threaded. Furthermore, we assume that all tasks are CPU-bound: CPU needs of sequential tasks are 25% and those of other tasks are 100%. This is a pessimistic assumption for our approach as performance degradation due to sharing of CPU resources among jobs will be maximum.

The general consensus is that ample memory is available for allocating multiple tasks on the same node [2], [4]–[6], but no explicit model is available. We opt for a simple model suggested by data in Setia et. al. [3]: 55% of the jobs have tasks with a memory requirement of 10%. The remaining 45% of the jobs have tasks with memory requirements $10 \times x\%$, where x is an integer value uniformly distributed over $\{2,\ldots,10\}$.

We generated 100 distinct traces of 1,000 jobs using the Lublin model [38] and annotated them with CPU needs and memory requirements as described. The generated traces assume a 128-node cluster and thus contain jobs with between 1 and 128 tasks. Generally the time between the submission of the first job and the submission of the last job is on the order of 4-6 days. Next, in order to provide a way to systematically study how different algorithms perform on problems with different levels of difficulty, we multiplied the inter-arrival times of jobs in each generated trace by 9 computed constants in order to create 9 new traces with identical job mixes but offered load [4], or *load*, levels of 0.1 to 0.9 in increments of .1. Thus, from the 100 initial traces we created 900 scaled traces.

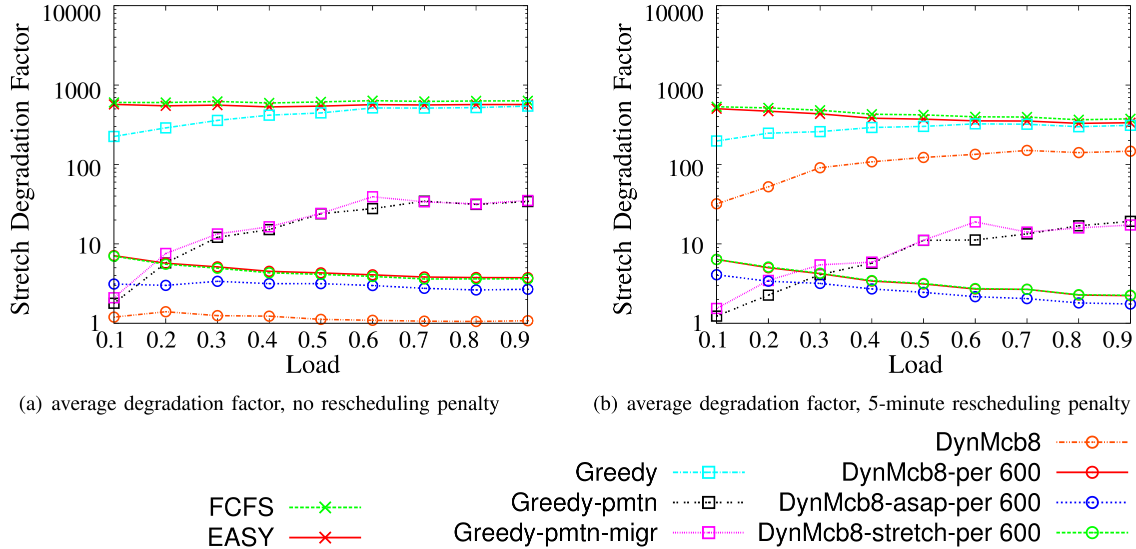
In addition to using synthetic workloads, we also perform experiments with a real-world workload from a well-established on-line repository [1]. Most logs provide standard information such as job arrival times, start time, completion time, requested duration, size in number of nodes, etc. For our experiments we choose the HPC2N workload from [1], which is a 182-week trace from a 120-node dual-core cluster running Linux. A primary reason for choosing this workload was that it contains almost complete information regarding memory requirements, while other workloads often contain no or very little such information.

The HPC2N workload required some processing for use in our experiments. The swf file format used by [1] contains information about the required number of "processors," but not the required number of tasks, and so this value had to be inferred. First, job per-processor memory requirements were set as the maximum of either requested or used memory as a fraction of the system memory of 2GB, with a minimum observed of 10%. Of the 202,876 jobs in the trace, only 2,142 ($\sim 1\%$) did not give values for either used or requested memory and we assigned them a value of 10%. For jobs that required an even number of processors and had a per-processor memory requirement less than 50% of the available node memory, we assumed that the job used a number of multi-threaded tasks equal to half the number of processors. In this case each task had a CPU need of 100% (recall that we pessimistically assume CPU-bound tasks) and the memory requirement was doubled from its initial per-processor value. For jobs requiring an odd number of processors or more than 50% of the available node memory per processor, we assumed that the number of tasks was equal to the number of processors and that each of these tasks had a CPU need of 50% (i.e., 1 core). These assumptions, once again, are detrimental to our approach and should benefit batch scheduling algorithms. Finally, we split the HPC2N workload into 182 1-week segments as processing the full workload through our simulator provides only one data point.

V. Experimental Results

For a given instance, and for each algorithm, we define the *degradation factor* as the ratio between the maximum stretch achieved by that algorithm on that instance and that achieved by the best algorithm for that instance. A value of 1 means that the algorithm is the best for that instance, while a value of 10 means that there was an algorithm in that experiment that achieved a maximum stretch 10 times lower.

Figure 1 plots average degradation factors vs. load for all experiments conducted using the scaled synthetic workloads, using a logarithmic scale on the y-axis. Figure 1(a) shows results when there is no rescheduling penalty. In this case, a striking but nevertheless expected result is that the DynMCB8 algorithm leads to average degradation factors averaging 1.16. This confirms that, without any overhead for pausing, resuming, or migrating jobs, solving the vector packing problem using an efficient heuristic is indeed best. The worst algorithm is FCFS, followed closely by EASY and then Greedy. The relatively poor performance of



(a) average degradation factor, no rescheduling penalty

(b) average degradation factor, 5-minute rescheduling penalty

Figure 1. Degradation factor vs. load for all our algorithms. Each data point shows average values over 100 instances.

GREEDY when compared to other DFRS algorithms is expected given that it can delay short jobs arbitrarily long, thereby increasing their stretch many-fold. More generally, this demonstrates that preemption mechanisms are necessary for DFRS. Our two remaining greedy algorithms, GREEDY-PMTN and GREEDY-PMTN-MIGR, improve significantly over batch scheduling (EASY results in degradation values nearly 28 times those of GREEDY-PMTN on the average) Surprisingly, GREEDY-PMTN leads to slightly better results than GREEDY-PMTN-MIGR even though there is no migration overhead in this set of experiments. Our main result is that, at least for nontrivial loads, the DYNMCB8-PER, DYNMCB8-ASAP-PER, and DYNMCB8-STRETCH-PER algorithms provide a large further improvement (GREEDY-PMTN results in degradation values 4.5 times those of DYNMCB8-PER on the average). The best of the three is DYNMCB8-ASAP-PER, particularly for low load. Interestingly, the DYNMCB8-STRETCH-PER algorithm, which attempts to minimize estimated maximum stretch, does not have better performance than DYNMCB8-PER. This provides some confidence that explicitly optimizing the yield is a good option for indirectly optimizing the stretch.

While Figure 1(a) corresponds to an ideal case, results in Figure 1(b) include a high 5-minute rescheduling penalty. We see that DYNMCB8 is no longer the best algorithm. It suffers from large overhead due to its aggressive use of preemption and migration, but still outperforms GREEDY and the batch scheduling algorithms. The DYNMCB8-PER, DYNMCB8-ASAP-PER, and DYNMCB8-STRETCH-PER algorithms are now the best for load levels higher than 0.3, while GREEDY-PMTN and GREEDY-PMTN-MIGR perform best for load levels below 0.3. DYNMCB8-ASAP-PER is the best algorithm for this set of experiments, outperforming DYNMCB8-PER by a slight margin, which outperforms DYNMCB8-STRETCH-PER by an even smaller margin.

To cross-validate our results, Table I shows aggregate degradation factor numbers for all experiments using our scaled synthetic traces (i.e., the results discussed earlier), using the original unscaled synthetic traces directly from the Lublin model, and using the real-world HPC2N trace. All results are for a 5-minute rescheduling penalty. We see a significant improvement in the average degradation experienced by the greedy algorithms for the HPC2N trace. Further investigation has shown that this trace contains a large number of short-duration serial jobs, which are less likely to benefit from the bin-packing approach than long-lived parallel jobs.

The maximum degradation factor is the result for the "worst trace" of each algorithm. Moving from average to the maximum degradation factor results, the main difference is that DYNMCB8-ASAP-PER becomes the clear winner by this metric, though all DFRS approaches provide an improvement over FCFS and EASY.

From these results, our main conclusion is that the DFRS algorithms provide drastic improvements over standard batch scheduling strategies for maximum stretch minimization. This holds even when there is a high penalty for preempting, resuming, and migrating jobs, and with all the conservative assumptions on the workload explained in Section IV. DYNMCB8-STRETCH-PER always has average results worse than DYNMCB8-PER, which confirms that optimizing the yield compares well to attempting to optimizing the stretch directly. The approaches based on multi-capacity bin packing

Table I
DEGRADATION FACTOR RESULTS FOR ALL SCALED SYNTHETIC TRACES, THE UNSCALED SYNTHETIC TRACES, AND THE REAL-WORLD HPC2N WORKLOAD. ALL RESULTS ARE FOR A 5-MINUTE RESCHEDULING PENALTY.

Algorithms	Scaled synthetic traces			Unscaled synthetic traces			Real-world trace		
	Degradation factor			Degradation factor			Degradation factor		
	avg.	*std.*	*max*	*avg.*	*std.*	*max*	*avg.*	*std.*	*max*
FCFS	435.32	239.37	1,470.30	448.74	238.38	1,128.44	469.51	581.12	4,422.00
EASY	392.78	227.85	1,454.26	410.06	226.41	989.91	402.93	500.13	3,136.92
GREEDY	284.07	222.33	2,288.65	239.76	217.47	1,128.32	134.32	268.66	2,129.98
GREEDY-PMTN	9.45	41.30	509.05	3.28	10.59	84.16	1.72	6.63	89.54
GREEDY-PMTN-MIGR	10.86	46.35	509.04	3.61	13.81	102.69	1.77	6.87	92.87
DYNMCB8	108.78	183.77	1,435.44	36.95	32.16	162.33	67.19	229.96	2,241.83
DYNMCB8-PER	3.55	2.62	16.38	4.83	3.90	21.30	14.73	64.41	668.24
DYNMCB8-ASAP-PER	2.62	1.66	12.77	3.28	2.03	11.00	4.23	3.05	24.68
DYNMCB8-STRETCH-PER	3.59	2.53	16.38	4.89	3.77	20.97	16.01	78.24	900.45

provide the best results for nontrivial loads when there is no migration penalty, and the periodic approaches perform well on both synthetic and real workloads even when there is a migration penalty. While purely greedy approaches performed best in the average case on the real-workload in the 5-minute migration penalty case, their worst case performance was significantly worse than that of DYNMCB8-ASAP-PER.

Preemption and/or migration, at least when used in conjunction with adequate algorithms, provide benefit in terms of stretch even with a high rescheduling penalty. However, one may wonder whether network and I/O resources are not overly taxed by preemption and/or migration activity. Table II shows preemption and migration costs for our synthetic workloads with high load (≥ 0.7), for all algorithms that use preemption and/or migration (only GREEDY-PMTN uses no migration). The main observation is that the average bandwidth consumption due to migration and preemption is only $0.60 + 0.26 = 0.86$ GB/sec on average, and only $1.31 + 0.77 = 2.08$ GB/sec for the worst trace, for DYNMCB8-PER. This corresponds to under 5 MB/sec per node on average and under 17 MB/sec per node for the worst trace. We conclude that these resource demands are well within the capacity of current technology, based on our previous discussion of I/O and the observation that 17 MB/sec is less than 1.5% of the peak bandwidth on a 10 Gigabit Ethernet switch. On average DYNMCB8-PER leads to under 46 preemptions and under 49 migrations per hour, with these numbers being around 110 and 142, respectively, for the worst trace. On average, each job is preempted and migrated about 8 and 6 times respectively. As expected the aggressive DYNMCB8 algorithm leads to the highest costs. GREEDY-PMTN and GREEDY-PMTN-MIGR have the lowest costs, but we have seen that they lead to poor maximum stretches. DYNMCB8-PER and DYNMCB8-ASAP-PER have comparable costs. Compared to them, the DYNMCB8-STRETCH-PER algorithm leads to up to 33% lower preemption costs but to more than 100% higher migration costs. We conclude that our approach, in particular when using the DYNMCB8-PER and DYNMCB8-ASAP-PER algorithms, does not impose unreasonable strain on network and I/O.

Another concern is the time needed to compute a resource allocation. We have instrumented DYNMCB8 to record each scheduling event. We ran the simulator on a system with a 3.2GHz Intel Xeon CPU and 4GB RAM against the 100 unscaled traces generated by the Lublin model. This gave us 197,808 observations. For 67.25% of them DYNMCB8 computed allocations for 10 or fewer jobs in less than 0.001 seconds. The remaining observations were for 11 to 102 jobs. Average compute time was about 0.25 seconds with the maximum under 4.5 seconds. Since typical job inter-arrival times are orders of magnitude larger [38], we conclude that DFRS is feasible in practice.

VI. RELATED WORK

Our work is related to gang scheduling [39], which also departs from batch scheduling by allowing time-sharing of compute resources. In gang scheduling, tasks in a parallel job are executed during the same synchronized time slices across cluster nodes. This requires distributed synchronized context-switching, which may require significant overhead and thus long time slices, although solutions have been proposed [40]. In this work we simply achieve time-sharing in an uncoordinated and low-overhead manner via VM technology.

A second drawback of gang scheduling is memory pressure, i.e., the overhead of swapping to disk [4]. In our approach we completely avoid swapping by enforcing that tasks may be assigned to the same node only if the node's physical memory capacity is not exceeded. This precludes some time sharing when compared to gang scheduling. However, this constraint is eminently sensible given the observation that many jobs in HPC workloads use only a fraction of physical memory.

Other works have explored the problem of scheduling jobs without knowledge of their execution time. The famous "scheduling in the dark" approach [41] shows

Table II
PREEMPTION AND MIGRATION COSTS IN TERMS OF AVERAGE BANDWIDTH CONSUMPTION, NUMBER OF PREEMPTION AND MIGRATION OCCURRENCES PER HOUR, AND NUMBER OF PREEMPTION AND MIGRATION OCCURRENCES PER JOB. AVERAGE VALUES OVER ALL SYNTHETIC TRACES WITH LOAD ≥ 0.7, WITH MAXIMUM VALUES IN PARENTHESES.

Algorithm	Bandwidth Consumption (GB / sec)		Frequency of Occurrence (# occurrences / hour)		# occurrences per job	
	pmtn	mig	pmtn	mig	pmtn	mig
GREEDY	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
GREEDY-PMTN	0.10 (0.27)	0.00 (0.00)	12.26 (39.24)	0.00 (0.00)	1.26 (4.44)	0.00 (0.00)
GREEDY-PMTN-MIGR	0.05 (0.14)	0.03 (0.12)	5.42 (20.88)	8.99 (34.56)	0.58 (2.64)	0.90 (2.58)
DYNMCB8	0.53 (1.48)	1.46 (2.90)	84.99 (294.48)	604.55 (1329.84)	9.32 (23.61)	71.54 (110.19)
DYNMCB8-PER	0.60 (1.31)	0.26 (0.77)	45.58 (110.16)	48.80 (141.84)	7.63 (32.32)	6.18 (20.77)
DYNMCB8-ASAP-PER	0.59 (1.34)	0.27 (0.77)	44.05 (105.84)	49.16 (142.92)	7.33 (30.87)	6.08 (20.35)
DYNMCB8-STRETCH-PER	0.38 (0.87)	0.51 (1.11)	33.78 (76.32)	88.32 (197.64)	7.15 (27.68)	16.11 (40.61)

that in the absence of knowledge giving equal resource shares to job is theoretically sound. We use the same approach by ensuring that all jobs achieve the same yield. Our problem is also related to thread scheduling done in operating system kernels, given that thread execution times are unknown. Our work differs in that we strive to optimize a precisely defined objective function.

Several previous works have explored algorithmic issues pertaining to bin packing and/or multiprocessor scheduling. Coffman studies bin stretching, a version of bin packing in which a bin may be stretched beyond its normal capacity [42]. Epstein studies the on-line bin stretching problem as a scheduling problem with the goal of minimizing makespan [43]. Our scheduling problem is strongly related to vector packing, i.e., bin packing with multi-dimensional items and bins. Vector packing has been studied from both a theoretical standpoint (i.e., guaranteed algorithms) [44], [45] and a pragmatic one (i.e., efficient heuristics) [33], [46], [47]. In this work we employ an algorithm based on a particular vector packing algorithms, MCB (Multi-Capacity Bin Packing), proposed by Leinberger et. al. [33].

Azar has studied on-line load balancing of temporary tasks on identical machines with assignment restrictions [48]. The problem therein is to assign incoming tasks to nodes permanently. Each task has a weight and a duration. The weight is known when the task arrives, but the duration is not known until task completion. The goal is to minimize the maximum load on any machine over all time instants. There is thus a connection to our goal of maximizing the minimum yield.

Finally, previous works have explored the use of VM technology in the HPC domain. The broad consensus is that VM overhead does not represent a barrier to mainstream deployment [49], [50]. Additional research has shown that the performance impact on MPI applications is minimal [51] and that cache interference effects do not cause significant performance degradation in commonly-used libraries such as LAPACK and BLAS [52]. Finally, it has been shown that current VM technology allows for preemption and checkpointing of MPI applications [53], as assumed in this work.

VII. CONCLUSION

We have proposed DFRS, a novel approach for job scheduling in a homogeneous cluster. We have focused on an on-line, non-clairvoyant scenario in which job execution times are unknown ahead of time. We have proposed several scheduling algorithms and have compared them to standard batch scheduling approaches using both synthetic and real-world workloads. In our simulations, our algorithms were given no knowledge of job execution times, while batch scheduling algorithms were provided with perfect estimates. We have found that DFRS provides dramatic improvement over batch scheduling in terms of schedule quality, where schedule quality is measured by a popular job performance metric, the maximum (bounded) stretch. The improvement shown in our results is likely to be larger in practice due to conservative assumptions in our evaluation methodology, including on the overhead due to preemptions and migrations. The improvement is the highest when using one of our resource allocation algorithms, DYNMCB8-ASAP-PER, which schedules jobs periodically and uses both preemption and migration. An important result in this work is that optimizing the yield, an instantaneous performance metric that does not rely on job execution times, is an effective way to (indirectly) optimize the stretch.

This work can be extended in several directions. Our scheduling algorithms could be improved with a strategy for reducing the yield of long running jobs as a way to improve fairness and further decrease maximum stretch. This strategy, inspired by thread scheduling in operating systems kernels, would be particularly useful for mitigating the negative impact of long running jobs on shorter ones. Also, mechanisms for implementing user priorities, such as those supported in batch scheduling systems, are needed. More broadly, a logical next step is to implement and benchmark our algorithms as part of a prototype virtual cluster management system, such as [13], that uses some of the resource need

discovery techniques described in Section II-A. This would naturally lead to a study of jobs with resource requirements that evolve over time.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] D. G. Feitelson. Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[2] ——, "Memory usage in the LANL CM-5 workload," in *Job Scheduling Strategies for Parallel Processing*, ser. LNCS, D. G. Feitelson and L. Rudolph, Eds. Springer-Verlag, 1997, vol. 1291, pp. 78–94.

[3] S. Setia, M. S. Squillante, and V. K., "The impact of job memory requirements on gang-scheduling performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 4, pp. 30–39, 1999.

[4] A. Batat and D. G. Feitelson, "Gang Scheduling with Memory Considerations," in *Proc. of the 14th Intl. Parallel and Distributed Processing Symp.*, 2000, pp. 109–114.

[5] S.-H. Chiang and M. K. Vernon, "Characteristics of a large shared-memory production workload," in *Proc. of the 7th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. LNCS, vol. 2221, 2001, pp. 159–187.

[6] H. Li, D. Groep, and L. Wolters, "Workload Characteristics of a Multi-cluster Supercomputer," in *Proc. of the 10th Intl. Workshop on Job Scheduling Strategies for Parallel Processing*, ser. LNCS, vol. 3277, 2005, pp. 176–193.

[7] C. B. Lee and A. Snavely, "Precise and Realistic Utility Functions for User-Centric Performance Analysis of Schedulers," in *Proc. of the Symp. on High-Performance Distributed Computing*, June 2007.

[8] U. Schwiegelshohn and R. Yahyapour, "Fairness in parallel job scheduling," *J. of Scheduling*, vol. 3, no. 5, pp. 297–320, 2000.

[9] M. A. Bender, S. Muthukrishnan, and R. Rajaraman, "Approximation algorithms for average stretch scheduling," *J. of Scheduling*, vol. 7, no. 3, pp. 195–222, 2004.

[10] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," in *Proc. of the 2002 Intl. Conf. on Parallel Processing Workshops*, Aug. 2002, pp. 514–522.

[11] C. Bailey Lee and A. Snavely, "On the User-Scheduler Dialogue: Studies of User-Provided Runtime Estimates and Utility Functions," *Intl. J. of High Perf. Computing Applications*, vol. 20, no. 4, pp. 495–506, 2006.

[12] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. Yocum, "Sharing Networked Resources with Brokered Leases," in *Proc. of the USENIX Technical Conf.*, June 2006.

[13] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker, "Usher: An Extensible Framework for Managing Clusters of Virtual Machines," in *Proc. of the 21st Large Installation System Administration Conf.*, Nov. 2007.

[14] L. Grit, D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecht, "Harnessing Virtual Machine Resource Control for Job Management," in *Proc. of HPCVirt2007*, Nov. 2007.

[15] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-source Cloud-computing System," in *Proc. of Cloud Computing and Its Applications*, Oct. 2008.

[16] "VirtualCenter," http://www.vmware.com/products/vi/vc.

[17] "Citric XenServer Enterprise Edition," http://www.xensource.com/products/Pages/XenEnterprise.aspx.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proc. of the 19th ACM Symp. on Operating System Principles*, 2003, pp. 164–177.

[19] D. Schanzenbach and H. Casanova, "Accuracy and responsiveness of CPU sharing using Xen's cap values," University of Hawai'i at Mānoa Department of Information and Computer Sciences, Tech. Rep. ICS2008-05-01, May 2008. [Online]. Available: http://www.ics.hawaii.edu/research/tech-reports/ICS2008-05-01.pdf

[20] D. Gupta, L. Cherkasova, and A. Vahdat, "Comparison of the Three CPU Schedulers in Xen," *ACM SIGMETRICS Performance Evaluation Review (PER)*, vol. 35, no. 2, pp. 42–51, September 2007.

[21] A. Warfield, S. Hand, T. Harris, and I. Pratt, "Isolation of Shared Network Resources in XenoServers," PlanetLab Project, Tech. Rep. PDN-02-2006, Nov. 2002.

[22] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel, "Concurrent Direct Network Access for Virtual Machine Monitors," in *Proc. of HPCA 2007*, Feb. 2007, pp. 306–317.

[23] *Proc. of the 1st USENIX Workshop on I/O Virtualization (WIOV)*, Dec. 2008.

[24] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in Virtual Machine Monitors," in *Proc. of the ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environment (VEE)*, March 2008.

[25] K. Nesbit, J. Laudon, and J. Smith, "Virtual Private Caches," in *Proc. of ISCA 2007*, 2007.

[26] D. Gupta, R. Gardner, and L. Cherkasova, "XenMon: QoS Monitoring and Performance Profiling Tool," Hewlett-Packard Labs, Tech. Rep. HPL-2005-187, 2005.

[27] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking Processes in a Virtual Machine Environment," in *Proc. of the USENIX Annual Technical Conf.*, June 2006.

[28] ——, "Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment," in *Proc. of the ASPLOS 2006 conference*, Oct. 2006.

[29] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and Stretch Metrics for Scheduling Continuous Job Streams," in *Proc. of the 9th ACM-SIAM Symp. On Discrete Algorithms*, Jan. 1998, pp. 270–279.

[30] A. Legrand, A. Su, and F. Vivien, "Minimizing the stretch when scheduling flows of divisible requests," *J. of Scheduling*, vol. 11, no. 5, pp. 381–404, 2008.

[31] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing*, ser. LNCS, D. G. Feitelson and L. Rudolph, Eds. Springer-Verlag, 1997, vol. 1291, pp. 1–34.

[32] M. Stillwell, D. Shanzenbach, F. Vivien, and H. Casanova, "Resource Allocation using Virtual Clusters," in *Proc. of CCGrid 2009*. IEEE, May 2009, pp. 260–267.

[33] W. Leinberger, G. Karypis, and V. Kumar, "Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints," in *Proc. of the Intl. Conf. on Parallel Processing*. IEEE, 1999, pp. 404–412.

[34] S. Cochrane, K. Kutzer, and L. McIntosh, "Solving the HPC I/O Bottleneck: Sun™ Lustre™ Storage System," Sun BluePrints™ Online, Sun Microsystems, April 2009.

[35] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proc. of the 2nd Symp. on Networked Systems Design and Implementation*, 2005, pp. 273–286.

[36] D. Lifka, "The ANL/IBM SP Scheduling System," in *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing, LCNS*, vol. 949, 1995, pp. 295–303.

[37] C. Bailey Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "Are User Runtime Estimates Inherently Inaccurate?" in *Proc. of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2004, pp. 253–263.

[38] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs," *J. of Par. and Dist. Computing*, vol. 63, no. 11, 2003.

[39] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems," in *Proc. of the 3rd Intl. Conf. on Distributed Computing Systems*, Oct. 1982, pp. 22–30.

[40] A. Hori, H. Tezuka, and Y. Ishikawa, "Overhead Analysis of Preemptive Gang Scheduling," in *Proc. of Workshop on Job Scheduling Strategies for Parallel Processing, LCNS*, vol. 949, June 2006, pp. 217–230.

[41] J. Edmonds, "Scheduling in the Dark," in *Proc. of the 31st annual ACM STOC*, 1999, pp. 179–188.

[42] E. G. J. Coffman and G. S. Lueker, "Approximation Algorithms for Extensible Bin Packing," *J. of Scheduling*, vol. 9, no. 1, pp. 63–69, 2006.

[43] L. Epstein, "Bin Stretching Revisited," *Acta Informatica*, vol. 39, no. 2, pp. 97–117, 2003.

[44] C. Chekuri and S. Khanna, "On multi-dimensional packing problems," *SIAM J. on Computing*, vol. 33, no. 4, pp. 837–851, 2004.

[45] N. Bansal, A. Caprara, and M. Sviridenko, "Improved approximation algorithms for multidimensional bin packing problems," in *Foundations of Computer Science*, 2006, pp. 697–708.

[46] K. Maruyama, S. K. Chang, and D. T. Tang, "A general packing algorithm for multidimensional resource requirements," *Intl. J. of Computer and Information Sciences*, vol. 6, no. 2, pp. 131–149, 1977.

[47] N. Roy, J. S. Kinnebrew, N. Shankaran, G. Biswas, and D. C. Schmidt, "Toward effective multi-capacity resource allocation in distributed real-time and embedded systems," in *Proc. of ISOCR 2008*, 2008.

[48] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts, "On-Line Load Balancing of Temporary Tasks," *J. of Algorithms*, vol. 22, no. 1, pp. 93–110, 1997.

[49] W. Huang, J. Liu, B. Abali, and D. Panda, "A Case for High Performance Computing with Virtual Machines," in *Proc. of the 20th ACM Conf. on Supercomputing*, June 2006, pp. 125–134.

[50] C. Macdonell and P. Lu, "Pragmatics of Virtual Machines for High-Performance Computing: A Quantitative Study of Basic Overheads," in *Proc. of the 2007 High Performance Computing and Simulation Conf.*, June 2007.

[51] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems," in *Proc. of the 1st Intl. Work. on Virtualization Techn. in Distr. Computing*, 2006.

[52] L. Youseff, K. Seymour, H. You, J. Dongarra, and R. Wolski, "The impact of paravirtualized memory heirarchy on linear algebra computational kernels and software," in *Proc. of the HPDC 2008*, Jun. 2008.

[53] W. Emeneker and D. Stanzione, "Increasing Reliability through Dynamic Virtual Clustering," in *Proc. of the High Availability and Performance Computing Workshop*, 2006.