9th IEEE/ACM International Symposium on Cluster Computing and the Grid

Resource Allocation using Virtual Clusters

Mark Stillwell[1] David Schanzenbach[1]
Frédéric Vivien[2,3,4,1] Henri Casanova[1]
[1]Department of Information and Computer Sciences,
University of Hawai‘i at Mānoa, Honolulu, U.S.A.
[2]INRIA, France
[3]Université de Lyon, France
[4]LIP, UMR 5668 ENS-CNRS-INRIA-UCBL, Lyon, France

Abstract— **We propose a novel approach for sharing cluster resources among competing jobs. The key advantage of our approach over current solutions is that it increases cluster utilization while optimizing a user-centric metric that captures both notions of performance and fairness. We motivate and formalize the corresponding resource allocation problem, determine its complexity, and propose several algorithms to solve it in the case of a static workload that consists of sequential jobs. Via extensive simulation experiments we identify an algorithm that runs quickly, that is always on par with or better than its competitors, and that produces resource allocations that are close to optimal. We find that the extension of our approach to parallel jobs leads to similarly good results. Finally, we explain how to extend our work to dynamic workloads.**

I. Introduction

The use of large commodity clusters has become mainstream for scientific applications, large-scale data processing, and service hosting. These clusters represent enormous investments and high utilization is paramount for justifying their costs (hardware, power, cooling, staff) [1]. There is therefore a strong incentive to share cluster resources among many applications and users. In general, resource sharing among competing instances of applications, or *jobs*, is difficult because different jobs have different needs that cannot all be accommodated simultaneously.

There are two main approaches to share clusters today. In the high-performance computing arena, the ubiquitous approach is "batch scheduling", by which requests for compute resources are placed in queues and wait to gain exclusive access to a subset of the platform for bounded amounts of time. In service hosting or cloud environments, the approach is to allow users to lease "virtual slices" of physical resources, enabled by virtual machine technology. The latter approach has several advantages, including O/S customization and interactive execution.

Both approaches dole out *integral* subsets of the resources, or *allocations* (e.g., 10 physical nodes, 20 virtual slices), which places inherent limits on cluster utilization. Indeed, even if an application can dynamically change the number of resources it uses (via "malleability" [2]), if the application uses only 80% of a resource then 20% of it are wasted. As a result, other applications are denied access to resources, or delayed, in spite of cluster resources not being fully utilized.

Another problem with integral allocations is that they make the resource allocation problem, i.e., the optimization of an objective function, theoretically difficult [3]. While it is possible to define objective functions that capture notions of performance and fairness [4], in practice no such objective function is optimized. For instance, batch schedulers provide instead a myriad of configuration parameters to tune resource allocation behavior using ad-hoc rules of thumb. Consequently, there is a sharp disconnect between the desires of users (low response time, fairness) and the schedules computed by batch schedulers [5,6]. A notable finding in the theoretical literature is that with job preemption and/or migration certain resource allocation problems become (more) tractable or approximable [3,7]. Unfortunately, preemption and migration are rarely used on production parallel platforms. The gang scheduling [8] approach allows entire parallel jobs to be context-switched in a synchronous fashion, but is known to have high overhead and, more importantly, to be susceptible to prohibitive memory pressure. Therefore while flexibility in resource allocations is desirable for solving resource allocation problems, affording this flexibility has not been successfully accomplished in production systems.

We claim that both limitations of current resource allocation schemes, namely, lowered utilization and lack of objective function, can be addressed *simultaneously* via fractional and dynamic resource allocations enabled by state-of-the-art Virtual Machine (VM) technology. Indeed, applications running in VM instances can be monitored so as to discover their true resource needs, and their resource allocations can be fractional and modified dynamically with low overhead. In this paper we:

- Formalize and justify a relevant resource allocation problem based on a simple system architecture and on the use of current technology (Section II);
- Establish the complexity of the problem and propose algorithms to solve it in the case of sequential jobs in a static workload (Section III);
- Evaluate our proposed algorithms in simulation and identify an algorithm that is efficient and close to

This work is partially supported by the U.S. National Science Foundation under award #0546688, and by the European Commission's Marie Curie Fellowship IOF #221002.

978-0-7695-3622-4/09 $25.00 © 2009 IEEE
DOI 10.1109/CCGRID.2009.23

optimal (Section IV);
- Extend our approach to parallel jobs (Section V);
- Formulate a resource allocation adaptation problem for dynamic workloads (Section VI).

II. Resource Allocation using Virtual Clusters

A. System Overview

We consider a homogeneous cluster based on a switched interconnect, and managed by a resource allocation system. The system responds to job requests by creating collections of VM instances called "virtual clusters" [9] on which to run the jobs. The VM instances run on physical hosts that are each under the control of a VM monitor [10–12]. The VM monitor can enforce specific resource consumption rates for different VM instances running on the host. All VM Monitors are in turn under the control of a VM Manager that can specify resource consumption rates for all of the VM instances running on the physical cluster. Furthermore, the VM Manager can enact VM instance migrations among physical hosts. The VM Manager thus provides the mechanisms for allocating to jobs only the resources they need when they need them. Resource allocation decisions are made by a Resource Allocator, regarding whether a job request should be rejected or admitted, regarding VM migrations, and regarding throttling resource consumption rates of each VM instance. Several groups in academia and industry are developing systems following this conceptual architecture [9,13–16]. Our goal is to design a sound resource allocation algorithm to be implemented at the heart of the Resource Allocator.

B. VM Technology to support Virtual Clusters

1) Resource sharing via virtualization: Virtualization should ideally allow accurate sharing of hardware among VM instances while achieving performance isolation. The work in [17], based on the Xen VM monitor [10], shows that CPU-sharing and performance isolation is low-overhead, accurate (on average less than 1% difference between effective and specified allocated resource fractions), and rapidly adaptable. Current VM monitors such as Xen also enable accurate and efficient sharing of memory space. VM instances can make use of other resources beyond CPU and memory, requiring for instance virtualization of network I/O resources [18–20]. I/O virtualization is an active area of research: the 1st Usenix Workshop on I/O Virtualization was in December 2008. See [21] for a discussion of current challenges and possible solutions. Another question is the virtualization of the memory hierarchy (buses and caches). Promising work provides hardware and software techniques for virtualizing microarchitecture resources [22], including full memory hierarchies [23].

We conclude that accurate sharing and performance isolation among VM instances along various resource dimensions is either available today or feasible and to be available in the near future.

2) Job resource need discovery via virtualization: With virtualization one can infer resource needs of VM instances through observation. The simplest solution is to use monitoring mechanisms, e.g., the XenMon facility [24]. VM instance resource needs can also be discovered via a combination of introspection and configuration variation. With introspection, one can for instance deduce CPU needs by inferring process activity inside of VMs [25], and memory pressure by inferring memory page eviction activity [26]. With configuration variation one can vary the amount of resources given to VM instances, track how they respond to the addition or removal of resources, and infer resource needs [25,26]. A combination of the above techniques can thus be used effectively to determine VM resource needs, defined as the amount of resources a VM instance would use if alone on a host.

C. Problem Statement

To formalize a first resource allocation problem for virtual cluster environments we make a number of assumptions. We only consider CPU-bound jobs that need a fixed amount of memory. Accurate virtualization of CPU and memory resources can be done today, and VM technologies to virtualize other resource dimensions are on the way (see Section II-B.1). We assume that job resource needs are known, perhaps specified by users, or, more likely, via discovery techniques (see Section II-B.2). We leave the study of erroneous resource need estimates for future work. For now we assume that each job requires only one VM instance. (We deal with parallel jobs in Section V.) We also assume that the workload is static: no jobs enter or leave the system, and jobs run forever and with unchanging resource needs. (We discuss dynamic workloads in Section VI.)

The difficult question for resource allocation is how to define precisely what a "good" allocation is. Allocation goodness should encompass both notions of individual job performance and notions of fairness among jobs, as defined by a precise metric. This metric can then be optimized, possibly ensuring that it is above some threshold (for instance by rejecting requests for new jobs).

We call our resource allocation problem VCSched and define it here formally. Consider $H > 0$ identical physical hosts and $J > 0$ jobs. For job i, $i = 1, \ldots, J$, let α_i be the fraction of a host's computational capability utilized by the job if alone on a physical host, $0 \leq \alpha_i \leq 1$. Let m_i be the maximum fraction of a host's memory needed by job i, $0 \leq m_i \leq 1$.

Let α_{ij} be the fraction of the computational capability of host j allocated to job i, $0 \leq \alpha_{ij} \leq 1$. If α_{ij} is in $\{0, 1\}$, then the model is that of scheduling with exclusive access to resources. Instead, we let α_{ij} take rational values to enable fine-grain resource allocation.

We can write a few constraints due to resource limitations. We have

$$\forall j \quad \sum_{i=1}^{J} \alpha_{ij} \leq 1 \qquad \text{and} \qquad \forall j \quad \sum_{i=1}^{J} \lceil \alpha_{ij} \rceil m_i \leq 1 \ ,$$

because the total CPU and memory fraction allocated to jobs on any single host cannot exceed 100%. Also, a job should not be allocated more resources than it can use and each job can run on only a single host, thus:

$$\forall i \quad \sum_{j=1}^{H} \alpha_{ij} \leq \alpha_i \qquad \text{and} \qquad \forall i \quad \sum_{j=1}^{H} \lceil \alpha_{ij} \rceil = 1 \; .$$

We wish to optimize a metric that encompasses user-centric notions of both performance and fairness. In the literature, a popular such metric is the *stretch* (also called "slowdown") [4], defined as the job's turn-around time divided by the turn-around time that would have been achieved had the job been alone in the system. Minimizing the maximum stretch has been recognized as a way to optimize average job turn-around time while ensuring that jobs do not experience high relative turn-around times. Consequently, it is a way to optimize *both* performance and fairness [4,7]. Because our jobs have no time horizons, we use a new metric, which we call the *yield* and which we define for job i as $\sum_j \alpha_{ij}/\alpha_i$. The yield of a job represents the fraction of its maximum achievable compute rate that is achieved (1 being the best value). We define problem VCSCHED as maximizing the minimum yield (which is akin to minimizing the maximum stretch).

One of our constraints is that a job runs on a single host, which is to say that there is no job migration. However, migration could be used to achieve better minimum yield. Assuming that migration can be done with no overhead or cost whatsoever, as often done in the literature, migrating jobs among hosts in a periodic steady-state schedule affords more flexibility for resource sharing, which could in turn be used to maximize the minimum yield further. We refer the reader to [27] for a 2-host 3-job example that demonstrates the use of migration. Unfortunately, the assumption that migration comes at no cost or overhead is not realistic. While VM migration is fast [28], it consumes network resources. It is not clear whether the pay-off of these extra migrations would justify the added cost, and one should probably place a judiciously chosen bound on the number of migrations. We leave this question for future work and use migration only for the purpose of adapting to dynamic workloads (see Section VI).

Finally, a key aspect of our approach is that it can be combined with resource management and accounting techniques. For instance, it is straightforward to add notions of user priorities, of resource allocation quotas, of resource allocation guarantees, or of coordinated resource allocations to VMs belonging to the same job.

III. SOLVING VCSCHED

VCSCHED is NP-hard in the strong sense via a reduction to 3-PARTITION [29] (the straightforward proof is available in a technical report [27]).

A. Mixed-Integer Linear Program (MILP) Formulation

VCSCHED can be formulated as a MILP, that is, a Linear Program (LP) with both rational and integer variables. Non-linear constraints given in Section II-C, i.e., ones involving $\lceil \alpha_{i,j} \rceil$, can be made linear by introducing binary integer variables, e_{ij}, set to 1 if job i is allocated to resource j, and to 0 otherwise. We can then rewrite the constraints in Section II-C as follows, with $i = 1, \ldots, J$ and $j = 1, \ldots, H$:

$$\forall i,j \qquad e_{ij} \in \{0,1\} \quad \alpha_{ij} \in \mathbb{Q} \tag{1}$$

$$\forall i,j \qquad 0 \leq \alpha_{ij} \leq e_{ij} \tag{2}$$

$$\forall i \qquad \textstyle\sum_{j=1}^{H} e_{ij} = 1 \tag{3}$$

$$\forall j \qquad \textstyle\sum_{i=1}^{J} \alpha_{ij} \leq 1 \tag{4}$$

$$\forall j \qquad \textstyle\sum_{i=1}^{J} e_{ij} m_i \leq 1 \tag{5}$$

$$\forall i \qquad \textstyle\sum_{j=1}^{H} \alpha_{ij} \leq \alpha_i \tag{6}$$

$$\forall i \qquad \textstyle\sum_{j=1}^{H} \frac{\alpha_{ij}}{\alpha_i} \geq Y \tag{7}$$

The objective is to maximize Y, i.e., to maximize the minimum yield.

B. Exact and Relaxed Solutions

In general, solving a MILP requires exponential time. We use a publicly available MILP solver, the Gnu Linear Programming Toolkit (GLPK), to compute the exact solution when there are few jobs and few hosts. When the instance is large we relax the problem by assuming that all variables are rational, converting the MILP into a rational LP, which can be solved in polynomial time in practice. The obtained solution may be infeasible but has two important uses. First, the achieved minimum yield is an upper bound on the solution of the MILP. Second, the rational solution may point the way toward good feasible solutions. Note that we do not need a linear program solver to compute the maximum minimum yield for the relaxed LP. Indeed, it is equal to $min\{H/\sum_{i=1}^{J} \alpha_i, 1\}$ and achieved by the trivial allocation: $e_{ij} = 1/H$ and $\alpha_{ij} = (Y/H)\alpha_i$, for all i and j.

C. Algorithms Based on Relaxed Solutions

We propose two algorithms, RRND and RRNZ, that use a solution of the rational LP as a basis and then round-off rational e_{ij} values. The trivial solution given in the previous section splits each job evenly across all hosts as all e_{ij} values are identical. Therefore it is a poor (in fact, the poorest) starting point for rounding off e_{ij} values. Instead we use GLPK to solve the LP and use the produced solution as a starting point.

1) Randomized Rounding (RRND): For each job i (taken in an arbitrary order), this algorithm allocates it to host j with probability e_{ij}. If the job cannot fit on the selected host because of memory constraints, then probabilities are adjusted and another attempt is made. If all jobs can be placed in this manner then the algorithm succeeds. Such a probabilistic approach has been used successfully in previous work [30].

2) Randomized Rounding with No Zero probability (RRNZ): One problem with RRND is that a job, i, may not fit (in terms of memory needs) on any of the hosts, j, for which $e_{ij} > 0$, in which case the algorithm would fail to generate a solution. To remedy this problem, we first set each zero e_{ij} value to ϵ, where $\epsilon << 1$ (we use $\epsilon = 0.01$). For those problem instances for which RRND provides a solution RRNZ should provide nearly the same solution. But RRNZ should also provide a solution for some instances for which RRND fails.

D. Greedy Algorithms

1) Greedy (GR): This algorithm first goes through the list of jobs in arbitrary order. For each job the algorithm ranks the hosts in non decreasing order by the sum of the maximum CPU needs of all jobs already assigned to a host. The algorithm then selects the first host that can satisfy the job's memory needs.

2) Sorted-Task Greedy (SG): This algorithm is similar to GR but first sorts the jobs in descending order by their memory needs so as to place larger jobs earlier.

3) Greedy with Backtracking (GB): We can modify the GR algorithm to add backtracking. Full-fledged backtracking, which guarantees finding a solution if it exists, requires potentially exponential time. Instead, we limit the number of job placement attempts to 500,000. Other simple options for bounding placement attempts are possible but, based on our experiments, do not work as well.

4) Sorted Greedy with Backtracking (SGB): SGB combines the SG and GB algorithms.

E. Multi-Capacity Bin Packing Algorithms

Resource allocation is akin to bin packing. There are however two important differences between our problem and bin packing. First, our job resource needs are dual, with both memory and CPU needs. Second, our CPU needs are not fixed but depend on the achieved yield. The first difference can be addressed by using "multi-capacity" bin packing heuristics [31]. The second difference can be addressed as follows. Consider an instance of VCSched and a fixed value of the yield, Y, that needs to be achieved for each job. By fixing Y, each job has fixed memory *and* fixed CPU needs, making it possible to apply multi-capacity bin packing heuristics directly. A binary search on Y is then used to find the highest value for which the problem can be solved.

For completeness, we give the principle of the algorithm in [31]. Jobs are split into two lists, with one list containing the jobs with higher CPU needs than memory needs and the other containing the jobs with higher memory needs than CPU needs. Each list is then sorted according to some to-be-defined criterion. One starts assigning jobs to the first host. Lists are scanned in order, searching for the first job that can "fit" on the host, which for the sake of this discussion we term a "possible job". Initially one searches for a possible job in any list. Subsequently, one searches for a possible job first in the list that goes against the current imbalance. For instance, say that the host's available memory capacity is 50% and its available CPU capacity is 80%, based on jobs assigned to it so far. One would first scan the list of jobs with higher CPU needs than memory needs. When no possible jobs are found in either list, one repeats this process for the next host. If all jobs can be assigned in this manner then resource allocation is successful.

While the algorithm in [31] sorts each list by descending order of the sum of the memory and CPU needs, there are other options. We experiment with sorting in ascending order of: the sum of the two needs (MCB1), the difference between the larger and the smaller of the two (MCB2), the ratio between the larger and the smaller of the two (MCB3), and the maximum of the two (MCB4). Sorting in descending order gives us four more algorithms (MCB5, MCB6, MCB7, and MCB8). MCB5 is the algorithm in [31].

F. Experimental Methodology

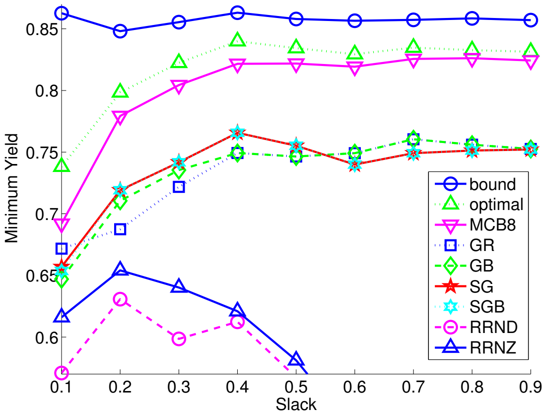
We conduct simulations on synthetic problem instances defined based on the number of hosts, the number of jobs, the total amount of free memory, or *memory slack*, in the system, the average job CPU need, and the coefficient of variance of both the memory and CPU needs of jobs. The memory slack is used rather than the average job memory need since it gives a better sense of how tightly packed the system is as a whole. In general (but not always) the greater the slack the greater the number of feasible solutions to VCSched.

Job CPU and memory needs are sampled from a normal distribution with given mean and coefficient of variance, truncated so that values are between 0 and 1. The mean memory need is defined as $H * (1 - slack)/J$, where $slack$ is between 0 and 1. The mean CPU need is taken to be 0.5, which in practice means that feasible instances with fewer than twice as many jobs as hosts have a maximum minimum yield of 1.0 with high probability. We do not ensure that every problem instance has a feasible solution.

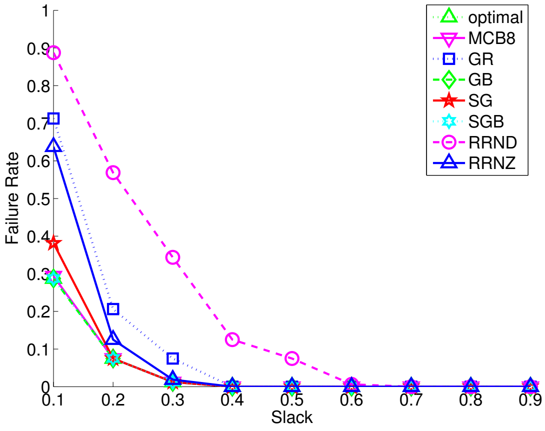
Two different sets of problem instances are examined. The first set of instances, "small" problems, includes instances with small numbers of hosts and jobs. Exact optimal solutions to most of these problems should be computable in a tractable amount of time (e.g., from a few minutes to a few hours). The second set of instances, "large" problems, includes instances for which the numbers of hosts and jobs are too large to compute exact optimal solutions.

For the small problem set we consider 4 hosts with 6, 8, 10, or 12 jobs. Slack ranges from 0.1 to 0.9 with increments of 0.1, while coefficients of variance for memory and CPU needs are given values of 0.25 and 0.75, for a total of 144 different problem specifications. 10 instances are generated for each problem specification, for a total of 1,440 instances.

For the large problem set we consider 64 hosts with sets of 100, 250 and 500 jobs. Slack and coefficients of variance for memory and CPU needs are the same as for the small problem set for a total of 108 different problems specifications. 100 instances of each problem specification were generated, for a total of 10,800 instances.



(a) Average Minimum Yield vs. Slack



(b) Failure Rate vs. Slack

Fig. 1. Minimum Yield and Failure rate for small problem instances.

IV. Experimental Results

We evaluate our algorithms based on three metrics: (i) the achieved minimum yield; (ii) the failure rate; and (iii) the run time. We also compare the algorithms with the exact solution of the MILP for small instances, and to the (unreachable upper bound) solution of the rational LP for all instances. The average minimum yield is computed based on successfully solved instances. Displayed values are averaged over all relevant problem instances.

Due to our large number of algorithms we first summarize results for our MCB algorithms. Overall MCB8 outperforms or is on par with the other MCB algorithms. Its minimum yield is on average 1.06% away from the best achieved minimum yield by any of the 8 algorithms for small problem instances, and 0.09% for all large problem instances. MCB5, used in [31], is a close second to MCB8. All these algorithms exhibit nearly equivalent run times. We conclude that MCB8 is the algorithm of choice and, to avoid graph clutter, we only include MCB8 results hereafter (full results are available in [27]).

A. Small Problems

Figure 1(a) shows the average minimum yield achieved versus the memory slack for our algorithms, the optimal MILP solution, and for the solution of the rational LP which is an upper bound of the MILP solution. The solution of the rational LP is only about 4% higher on average than the MILP solution, although it is significantly higher for slack values of 0.3 or smaller. The solution of the rational LP will be particularly interesting for large problem instances, for which we cannot compute the MILP solution. On average, the MILP solution is about 2% better than MCB8, and about 11% to 13% better than the greedy algorithms. All greedy algorithms exhibit roughly the same performance. RRND and RRNZ lead to results markedly poorer than the other algorithms, with expectedly RRNZ slightly outperforming RRND.

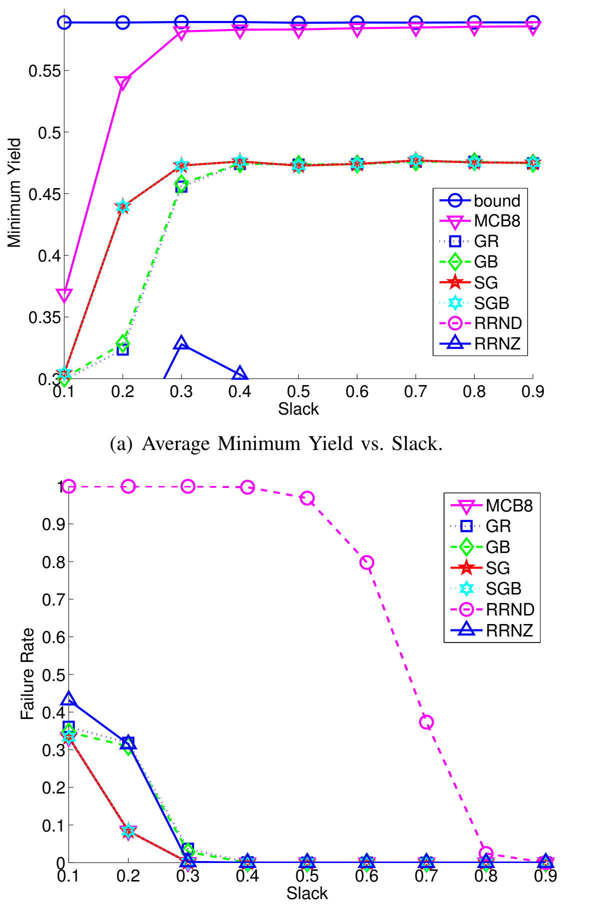
Figure 1(b) plots the failure rates of our algorithms. RRND has the worst failure rate, followed by GR and then RRNZ. There were a total of 60 instances out of the 1,440 for which GLPK could not find an optimal MILP solution. We see that MCB8, SG, and SGB have failure rates that are not significantly larger than that of GLPK. Out of the 1,380 feasible instances, the GB and SGB never fail to find a solution, MCB8 fails once, and SG fails 15 times.

We measured run times of the various algorithms on a 3.2GHz Intel Xeon processor. The computation time of the exact MILP solution is by far the largest, on average 28.7 seconds. There were 9 problem instances with solutions that took over 500 seconds to compute, and a single problem instance that required a little over 3 hours. For all small problem instances the average run times of the greedy algorithms are 20 to 30 microseconds, except for GB, which has an average runtime of 120 microseconds. MCB8 has an average runtime of 151 microseconds. RRND and RRNZ are slower, with average run times on the order of 2 milliseconds.

B. Large Problems

Figure 2 is similar to Figure 1, but for large instances, and thus does not show the optimal MILP solution. In Figure 2(a) we can see that MCB8 achieves far better results than any other algorithm. Furthermore, MCB8 is extremely close to the upper bound on optimal as soon as the slack is 0.3 or larger and is only about 8% away from this upper bound when the slack is 0.2. When the slack is 0.1, MCB8 is about 37% away from the upper bound, but in this case the upper bound is likely significantly larger than the optimal (see Figure 1(a)). The performance of the greedy algorithms relative to the upper bound is lower than for small instances, on average nearly 20% lower than the bound for slack values 0.3 or larger. RRNZ and RRND algorithms are again poor performers. RRND is not even visible on the graph and in fact fails to solve any instance for a slack lower than 0.4.

Figure 2(b) shows that GB has nearly as many failures as GR, and SGB has the same number of failures as SG. This suggests that 500,000 placement attempts when



(a) Average Minimum Yield vs. Slack.

(b) Failure Rate vs. Slack.

Fig. 2. Minimum Yield and Failure rate for large problem instances.

backtracking, which was more than sufficient for the small problem set, has little effect on the failure rate for the large problem set. RRND is the only algorithm with a significant number of failures for slack values larger than 0.3. SG, SGB and MCB8 exhibit the lowest failure rates, on average about 40% lower than that experienced by the other greedy and RRNZ algorithms, and more than 14 times lower than the failure rate of RRND.

On a 3.2GHz Intel Xeon RRND and RRNZ require roughly 650 seconds on average for 500 jobs. This large time is attributed to solving the rational LP using GLPK (which could be reduced significantly by using a faster solver, e.g., CPLEX). The greedy algorithms are the fastest, returning solutions in about 15 to 20 milliseconds for 500 jobs. This is to be compared to 500 milliseconds for MCB8, which is still acceptable in practice.

C. Discussion

The multi-capacity bin packing algorithm that sorts jobs in descending order by their largest resource need (MCB8) is the algorithm of choice. It outperforms or equals all other algorithms nearly across the board in terms of minimum yield and failure rate, while exhibiting low run times. The sorted greedy algorithms (SG or SGB) lead to reasonable results and could be used for very large instances, for which the run time of MCB8 may become too high. The use of backtracking led to performance improvements for small problem instances but not for large ones. This suggests that using a problem-size- or run-time-dependent bound on the number of branches to explore could be effective.

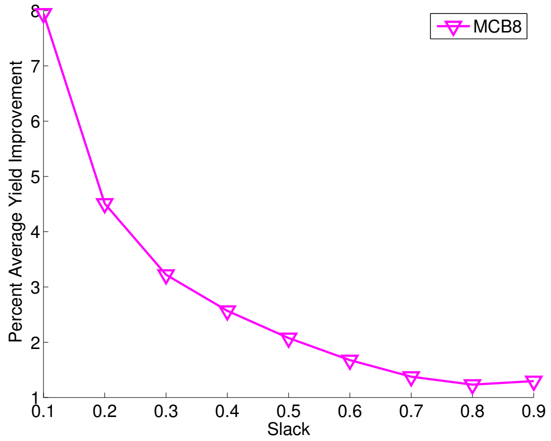


Fig. 3. Average Yield Percent Improvement vs. Slack for large problem instances for the MCB8 algorithm.

D. Optimizing Utilization

Once an allocation with a given minimum yield, say $\mathcal{Y}$, has been produced by any of our algorithms, there may be excess computational resources available. To maximize cluster utilization one can then maximize average yield while preserving $\mathcal{Y}$ as the maximum minimum yield. This maximization can be formulated as a MILP, simply replacing our objective function by the average yield and adding the constraint $Y \geq \mathcal{Y}$. Unfortunately, this MILP cannot be solved in polynomial time. We can however enforce that the placement of jobs onto the hosts not be modified, only their allocated CPU fractions. In this case we can can use the following optimal greedy algorithm. First, we set the yield of each job to $\mathcal{Y}$: $\alpha_{ij} = \alpha_i.\mathcal{Y}$. Then, for each host, we increase the CPU fraction of the job with the smallest CPU need α_i until either the host has no CPU capability left or the job's CPU need is fulfilled. In the latter case, we then apply the same scheme to the job with the second smallest CPU need on that host, and so on. The optimality of this process is easily proved via a typical exchange argument.

Figure 3 shows the average percentage improvement in average yield for MCB8 versus the memory slack, for large problem instances. Due to left-over resources after the minimum yield maximization phase, the average yield can be further increased by between 1% and 8%. The lower the slack, the more difficult the resource allocation problem, and thus the likelier it is that the minimum yield maximization was unable to fully utilize resources.

Even after average yield optimization, there may still be underutilized nodes if the cluster is overprovisioned. In this case one may wish to turn off nodes to save

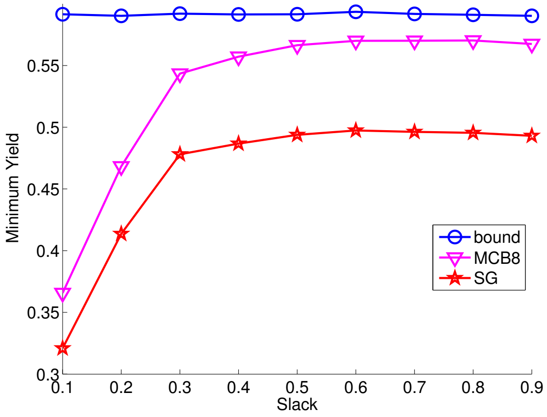


Fig. 4. Average Minimum Yield for large problem instances for parallel jobs.

energy [32]. Our approach is compatible with this strategy: simply determine potential allocations for various numbers of nodes (which is feasible because our algorithm for computing the allocation is fast). One can then pick the allocation that strikes any desirable compromise (e.g., turn on the smallest number of nodes so that the minimum yield is within some percentage of what it would be if all nodes were turned on).

V. Parallel Jobs

Users may want to split jobs into multiple tasks, either because they wish to use more CPU power in order to return results more quickly or because they wish to process an amount of data that does not fit comfortably within the memory of a single machine. We consider homogeneous parallel jobs, i.e., jobs whose tasks have identical resource needs. α_i is then the CPU fraction consumed by any task of job i if alone on a host. We define the yield of a parallel job as the sum of the yields of the job's tasks divided by the number of tasks in the job. Since the vast majority of parallel jobs make progress at the rate of their slowest task, we impose equal allocations for the tasks of the same job. All tasks in a parallel job then have the same yield, which is also the yield of the job. As a result, the algorithms described in Section III for sequential jobs can be used directly for parallel jobs by scheduling the tasks within the same job independently. (A complete MILP formulation of the resource allocation problem for parallel jobs can be found in a technical report [27].)

We present results only for our largest problem instances, as defined in Section III-F. To model the grouping of tasks into jobs we use the model in [33], i.e., a two-stage log-uniform distribution biased towards powers of two. We instantiate this model using the same parameters as in [33], assuming that jobs can consist of between 1 and 64 tasks.

Figure 4 shows results for SG and MCB8. We exclude all other algorithms as they were all shown to be at best as good as SG. The figure also shows the upper bound on the optimal yield obtained assuming that e_{ij} variables can take rational values. We see that MCB8 outperforms SG significantly and is close to the upper bound on the optimal for slacks larger than 0.3. Our results, not shown here due to lack of space (see [27]), also show that MCB8 and SG exhibit identical failure rates. In terms of run time, although SG is faster than MCB8 for 500 tasks by almost a factor of 32, MCB8 computes allocations in under half a second. Our conclusions are unchanged: MCB8 is the algorithm of choice while SG could be an alternate choice if the instance is very large.

Finally, average yield maximization for parallel jobs can be done using an algorithm similar to the greedy algorithm in Section IV-D. Jobs are considered in order of "computational size", i.e., the sum of their tasks' CPU needs. To increase a job's yield, one increases the yields of all its tasks equally until they reach 1 or the CPU capacity of one of the hosts running one of the tasks is fully consumed.

VI. Dynamic Workloads

Generally, job resource needs can change and jobs enter and leave the system. The schedule should then be adapted to reach a new good allocation. This adaptation can entail two types of actions: (i) modifying the CPU fractions allocated to jobs; and (ii) migrating jobs.

Actions of type (i) above can be done with virtually no overhead [17], and only involve computing new α_{ij} values without changing e_{ij} values. This can be done by first computing $Y_j = min\{1/\sum_{i=1}^{J} e_{ij}\alpha_i, 1\}$ for each host j, that is, the maximum minimum yield achievable on that host. One then sets $\alpha_{ij} = e_{ij}Y_j\alpha_i$ for each job i and each host j. To perform average yield optimization (as in Section IV-D for sequential jobs and Section V for parallel jobs), one needs to compute the minimum of Y_j over all hosts, that is the overall maximum minimum yield. All this can be done via exchanges of short control messages between the Resource Allocator, the VM Management System, and the VM Monitors.

Actions of type (ii) above can be done with low perceived job unresponsiveness [28], but may consume significant amounts of network bandwidth, raising the question of whether the adaptation is "worth it". Answering this question often uses a time horizon (e.g., "adaptation is not worthwhile if the workload is expected to change significantly in the next 5 minutes") [34]. In our setting we do not assume any knowledge of future evolution of job arrival, departure, and resource needs. While techniques to make predictions based on historical information have been developed [35], it is unclear that they provide sufficient accuracy to carry out precise cost-benefit analyses of various adaptation paths.

Faced with the above challenge we propose a pragmatic approach. We consider schedule adaptation that attempts to achieve the best possible minimum yield while not migrating more than some fixed number of bytes, B. If B is set to 0, then the adaptation will do the best it can without using migration whatsoever. B can be made large enough so that all jobs could potentially be migrated. The value of B can be chosen so that it achieves a reasonable trade-off between overhead and workload dynamicity (lower B for more dynamic workloads). Tuning workload adaptation behavior with a single parameter is simple, and thus desirable. Choosing the absolute best value for B is however system-, epoch-, and workload-specific.

One can easily formulate the resource allocation adaptation problem as a MILP. The idea is to consider the current allocation (i.e., e_{ij}^{old} and α_{ij}^{old}) as constants, and to compute a new allocation (i.e., e_{ij}^{new} and α_{ij}^{new}) that maximizes the minimum yield. The old allocation is used to construct an additional constraint that sums migrated bytes and that bounds this sum by B. We refer the reader to the technical report [27] for the full MILP, and we leave for future work the development of algorithms for solving it.

VII. Conclusion

We have proposed a novel approach for allocating cluster resources among competing jobs, relying on Virtual Machine technology, in a view to promoting cluster utilization and optimizing a well-defined and relevant metric. We have developed an algorithm that runs quickly, is always on par with or better than its competitors, and is close to optimal. We have then extended our approach to handle parallel jobs and proposed a pragmatic formulation of the resource allocation adaptation problem.

Future directions include the development of algorithms to solve the resource allocation adaptation problem, and experimentation with extant strategies for estimating job resource needs accurately. Our ultimate goal is to develop a new resource allocator as part of the Usher system [9].

References

[1] U.S. Environmental Protection Agency, "Report to Congress on Server and Data Center Energy Efficiency," http://repositories.cdlib.org/lbnl/LBNL-363E/, August 2007.

[2] G. Utrera, J. Corbalan, and Labarta, "Implementing Malleability on MPI Jobs," in *Proc. of PAACT*, 2004, pp. 215–224.

[3] P. Brucker, *Scheduling Algorithms*, 5th ed. Springer, 2007.

[4] M. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and Stretch Metrics for Scheduling Continuous Job Streams," in *Proc. of the Symp. On Discrete Algorithms*, 1998, pp. 270–279.

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

[6] 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*, 2007.

[7] 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.

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

[9] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker, "Usher: An extensible framework for managing clusters of virtual machines," in *Proc. of Lisa*, 2007.

[10] 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 ACM Symp. on Operating Systems Principles*, October 2003, pp. 164–177.

[11] "VMware," http://www.vmware.com/.

[12] "Microsoft Virtual PC," http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx.

[13] "Virtualcenter," http://www.vmware.com/products/vi/vc, 2008.

[14] "Citric XenServer Enterprise," http://www.xensource.com/products/Pages/XenEnterprise.aspx, 2008.

[15] L. Grit, D. Itwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecth, "Harnessing Virtual Machine Resource Control for Job Management," in *Proc. of HPCVirt*, 2007.

[16] P. Ruth, R. Junghwan, D. Xu, R. Kennell, and S. Goasguen, "Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure," in *Proc. of the IEEE Intl. Conf. on Autonomic Computing*, June 2006.

[17] D. Schanzenbach and H. Casanova, "Accuracy and Responsiveness of CPU Sharing Using Xen's Cap Values," Computer and Information Sciences Dept., University of Hawai'i at Mānoa, Tech. Rep. ICS2008-05-01, May 2008, available at http://www.ics.hawaii.edu/research/tech-reports/ICS2008-05-01.pdf/view.

[18] L. Cherkasova and R. Gardner, "Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor," in *Proc. of the Usenix Annual Technical Conference*, 2005.

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

[20] 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 the Intl. Symp. on High-Performance Computer Architecture*, February 2007.

[21] 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.

[22] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, and M. Valero, "Multicore Resource Management," *IEEE Micro*, May-June 2008.

[23] K. Nesbit, J. Laudon, and J. Smith, "Virtual Private Caches," in *Proc. of the Symp. on Computer Architecture*, 2007.

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

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

[26] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment," in *Proc. of Architectural Support for Programming Languages and Operating Systems*, October 2006.

[27] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource Allocation Using Virtual Clusters," http://www.ics.hawaii.edu/research/tech-reports/ics2008-09-15.pdf, Information and Computer Sciences Dept., University of Hawai'i at Mānoa, Tech. Rep. ICS2008-09-01, Sept. 2008.

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

[29] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[30] L. Marchal, Y. Yang, H. Casanova, and Y. Robert, "Steady-State Scheduling of Multiple Divisible Load Applications on Wide-Area Distributed Computing Platforms," *Intl. J. of High Performance Computing Applications*, vol. 20, no. 3, pp. 365–381, 2006.

[31] 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*, 1999, pp. 404–412.

[32] J. Chase, C. Anderson, P. Thakar, R. Doyle, and A. Vahdat, "Managing Energy and Server Resources in Hosting Centers," in *Proc. of the 18th ACM Symposium on Operating System Principles*, 2001, pp. 103–116.

[33] U. Lublin and D. Feitelson, "The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs," *J. of Parallel and Distributed Computing*, vol. 63, no. 11, 2003.

[34] R. L. Ribler, H. Simitci, and D. A. Reed, "The Autopilot Performance-Directed Adaptive Control System," *Future Generation Computer Systems*, vol. 18, no. 1, 2001.

[35] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling Using System-Generated Predictions Rather than User Runtime Estimates," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789–803, 2007.