Multi-Objective Job Placement in Clusters

Sergey Blagodurov[12], Alexandra Fedorova[13], Evgeny Vinnik[1], Tyler Dwyer[13], and Fabien Hermenier[4]

[1]Simon Fraser University*

[2]Advanced Micro Devices, Inc.

[3]The University of British Columbia

[4]INRIA and the University of Nice - Sophia Antipolis

ABSTRACT

One of the key decisions made by both MapReduce and HPC cluster management frameworks is the placement of jobs within a cluster. To make this decision, they consider factors like resource constraints within a node or the proximity of data to a process. However, they fail to account for the degree of collocation on the cluster's nodes. A tight process placement can create contention for the intra-node shared resources, such as shared caches, memory, disk, or network bandwidth. A loose placement would create less contention, but exacerbate network delays and increase cluster-wide power consumption. Finding the best job placement is challenging, because among many possible placements, we need to find one that gives us an acceptable trade-off between performance and power consumption. We propose to tackle the problem via multi-objective optimization. Our solution is able to balance conflicting objectives specified by the user and efficiently find a suitable job placement.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design Studies; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

Keywords

Performance evaluation, scheduling, virtualization.

1. INTRODUCTION

Important computations, like search, analytics, and scientific simulations, run on distributed computing clusters. The computations typically run as jobs comprised of communicating processes. Scheduling of processes inside the cluster is done by a job management framework. Although there are many job management frameworks that vary widely in their implementation (more than a dozen exist for popular programming interfaces like MapReduce and MPI), they all have one thing in common. Among the many decisions revolving around scheduling, they must consider where in the cluster to place the processes of running jobs.

Placement of processes is guided by different considerations, such as the proximity of data for MapReduce or the node's physical memory limitations. Unfortunately, there is yet another very important dimension of placement that existing frameworks do not consider, and that is the degree of collocation of the processes on the cluster nodes. Collocation is a measure of how tightly we pack the processes on a single node, and broadly speaking there are two strategies: slot round-robin (SlotRR) and node round-robin (NodeRR). A slot usually refers to a CPU core in the high performance computing (HPC) setting, whereas in MapReduce clusters, the maximum number of slots per node may be bound by the memory requirements of workers. The SlotRR policy keeps assigning processes to the same node until all slots are occupied; only then the next idle node is chosen. The NodeRR policy assigns processes to a node only until it fills a certain fraction of slots on that node; then it moves to the next available node [38, 43]. SlotRR results in a more "tightly packed" placement of processes so there is more sharing of intra-node resources, which can result in contention and poor performance, but on the other hand, the communication latencies are smaller and we end up using fewer nodes for the job. The unused nodes could then be turned off to save power [17]. As we will show next, the degree of collocation has a crucial impact on performance and energy consumption.

Consider performance data in Figure 1. We use Intel HiBench [41] together with Purdue's PUMA benchmark suites [16] as a representative MapReduce load and SPEC MPI2007 [12], and SPEC HEP (the official benchmark for clusters processing LHC data [11]) as a representative HPC job mix. We use the *India* cluster described in Section 4. The figure shows the performance difference of the NodeRR policy[1] relative to the SlotRR. Higher numbers mean performance improvement, measured as the reduction in the job completion time. We observe that there are very large differences in performance between the two policies, and the magnitude and the direction of the effect is highly job dependent. Some jobs run as much as 80% faster under NodeRR while others run as much as 60% slower. There are also jobs that are unaffected by the type of assignment – these jobs can be tightly packed onto a small number of nodes, allowing us to reduce the total number of powered up nodes in the cluster without the impact on performance.

The reasons why some jobs run better under NodeRR while others prefer SlotRR are as follows. Processes assigned to the same

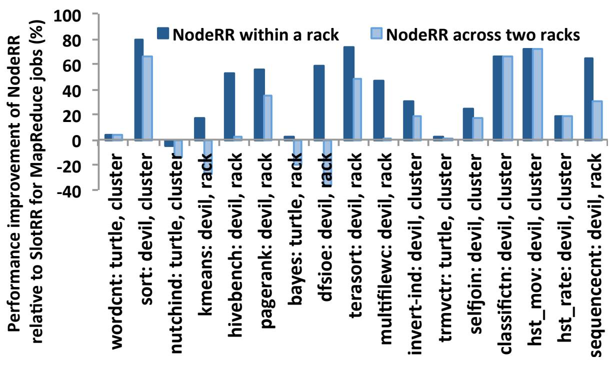
*The work was done while Sergey, Tyler, and Alexandra were affiliated with SFU. Present affiliations are also given.

SC '15, November 15-20, 2015, Austin, TX, USA

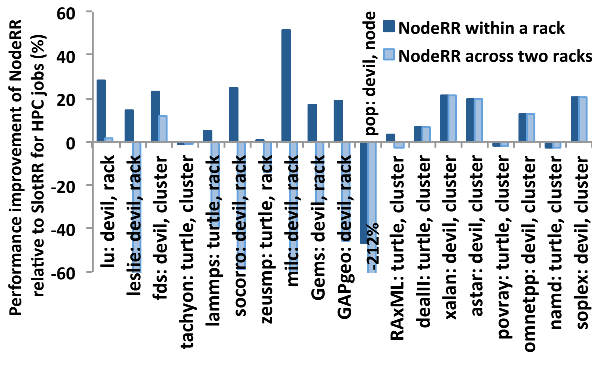
© 2015 ACM. ISBN 978-1-4503-3723-6/15/11…$15.00

DOI: http://dx.doi.org/10.1145/2807591.2807636

[1]NodeRR places job's processes on half of the slots on one node, before moving to the next node.



(a) Intel HiBench and Purdue PUMA (MapReduce workload).



(b) SPEC MPI2007 and SPEC HEP (HPC workload).

Figure 1: Comparison of SlotRR and NodeRR process-to-node assignments.

node using SlotRR may compete for shared resources, such as CPU caches, memory, disk, or network bandwidth. The resulting performance degradation can be as large as 3× relative to when a process runs contention-free [20]. On the other hand, if processes communicate intensively, they may suffer from network latency under the NodeRR policy. In addition to performance, there is the consideration of energy consumption: a NodeRR assignment can leave many slots unused, and the cluster as a whole ends up using more nodes (and more power) than it would under the SlotRR assignment.

It is clear that collocation should be an important part of process placement decisions. And yet, existing frameworks are not equipped to make effective choices. Many challenges lie herein. If our goal is to obtain the best performance, how do we know what has a greater negative impact on the job's completion time: contention for shared resources if we use SlotRR or network latency experienced under NodeRR? Sometimes, however, performance is not the primary goal; for instance, the user may wish to minimize the peak power consumption in the cluster. Most of the time, however, the situation is even more complicated in that the user wants to find some kind of "right" trade-off between the job's running time and power consumption. This makes deciding on optimal process collocation extremely challenging, and so it is not surprising that existing cluster management framework have not found an effective way to address this problem.

Our goal was to efficiently find a placement of cluster jobs that would balance the factors affecting performance and power in accordance with user preferences. To that end, we propose to model the problem as a *multi-objective optimization* (MO). This formulation allows us to represent the problem as a collection of *multiple objectives* and balance them according to user preferences. In the implementation described in this paper, we chose to balance the intra-node resource contention, network delays, and power consumption. However, our solution is general enough that it can be used with other objectives, if desired. ***The key contribution of our work is a general process placement solution for clusters that can be configured to satisfy multiple conflicting objectives.***

In a large cluster, the number of possible job placements can be very large (millions or even billions). So, it was crucial to find the one meeting user demands quickly and efficiently. To that end, our solution includes a *new multi-objective optimization algorithm* that relies on problem-domain specific information to find a job placement meeting user-defined criteria more efficiently than generic solvers. For the first time, we combine two distinct approaches to solve a multi-objective scheduling problem that make up for each other's disadvantages into a single solver. Our solution includes the visualization tool (adapted from prior work) that displays the extent of trade-offs between the chosen objectives as a Pareto front and enables administrator to choose the points on that front that he or she deems acceptable. We use this input to configure automatic multi-objective solver for finding the placement that meets user preferences.

We evaluated our solution called *ClavisMO* in real FutureGrid and Grid 5000 datacenters with up to 16 nodes (192 cores), simulated up to 3000 nodes of Facebook installations and demonstrated that it achieved energy savings of up to 46% (18% on average) and performance improvement of up to 60% (20% on average).

The rest of this paper is organized as follows: Section 2 presents *ClavisMO*, and Section 3 provides salient details about its multi-objective solver. Section 4 shows the experimental results, Section 5 discusses related work, and Section 6 concludes.

2. CLAVISMO: THE IMPLEMENTATION

Figure 2 shows a schematic view of a typical cluster management framework. The changes that we make to this "vanilla" framework are highlighted in bold.

Our goal was to create a solution that could be easily integrated with existing cluster management frameworks. To that end, we chose not to modify the scheduling algorithms within the framework, treating them as a "black box," and to rely on external performance monitoring and virtualization to migrate the processes within the cluster. Using virtualization allows for a seamless integration with the cluster schedulers like DRF [35], Hadoop's LATE [71], BEEMR [25] or delay scheduling [70], HPC's Maui, or Moab. These schedulers treat virtual containers as nodes, and perform job queue management as usual, without modifications. In parallel, our *ClavisMO* moves the containers across physical nodes according with the placement plan suggested by its solver. Based on our experimental results, we chose to have two containers per node.

For virtualization, we chose to use a container-based solution for Linux, OpenVZ, because it has less overhead than full VMs like Xen or KVM according to prior studies [10, 62, 46, 31, 42] and our own experiments. The container-based virtualization is an emerging trend for HPC and MapReduce in the cloud as the low overhead is paramount for distributed workloads: Google Cloud Platform, Amazon EC2 Container Service, and other large datacenter players are heavily investing in it [5, 1].

While the topic of virtualizing MapReduce workload is not novel, the topic of virtualizing HPC workload is. To that end, we have implemented live migration of HPC jobs across the physical nodes in a cluster. Such migration will not work with out-of-the-box virtualization solutions, because HPC jobs are tightly coupled and will often crash when attempting to migrate some of its processes. To prevent that, follow the guidelines at our *ClavisMO* website [2].

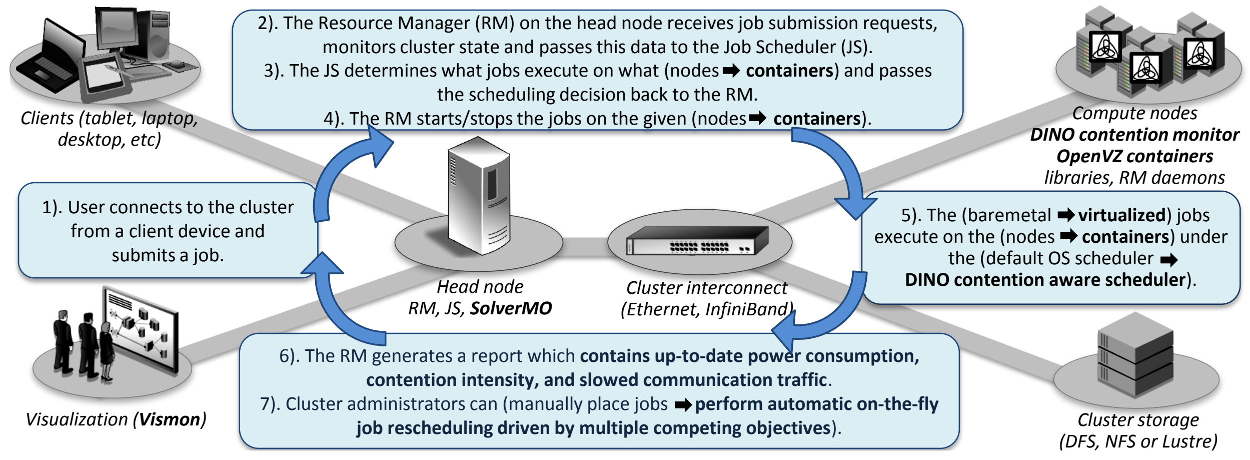

2). The Resource Manager (RM) on the head node receives job submission requests, monitors cluster state and passes this data to the Job Scheduler (JS).
3). The JS determines what jobs execute on what (nodes ➡ **containers**) and passes the scheduling decision back to the RM.
4). The RM starts/stops the jobs on the given (nodes ➡ **containers**).
Clients (tablet, laptop, desktop, etc)
Compute nodes
DINO contention monitor
OpenVZ containers
libraries, RM daemons
1). User connects to the cluster from a client device and submits a job.
Head node
RM, JS, **SolverMO**
Cluster interconnect (Ethernet, InfiniBand)
5). The (baremetal ➡ **virtualized**) jobs execute on the (nodes ➡ **containers**) under the (default OS scheduler ➡ **DINO contention aware scheduler**).
6). The RM generates a report which **contains up-to-date power consumption, contention intensity, and slowed communication traffic**.
7). Cluster administrators can (manually place jobs ➡ **perform automatic on-the-fly job rescheduling driven by multiple competing objectives**).
Visualization (**Vismon**)
Cluster storage (DFS, NFS or Lustre)

Figure 2: Cluster setting with *ClavisMO* modifications highlighted in bold.

Since we rely on virtualization to relocate the processes of a running job, we assume that the processes run within a *virtual container*. So, for the rest of the text, we will use the term *containers* when referring to the placement of processes.

In order to make a multi-objective scheduling decision, the solver (described in detail in Section 3) needs the data on the resource intensity and communication overhead for the running workload. In the rest of this section, we expand on the implementation aspects of these two modules.

Quantifying shared resource intensity: *ClavisMO* needs to measure how intensely the processes use shared resources. In the current implementation, *ClavisMO* addresses the following resources:

(a) *Chip-wide shared memory resources*, such as last-level caches, memory controllers, memory bandwidth, and prefetch hardware. Prior work showed that when processes running on the same chip compete for these resources, their performance can degrade by as much as 3× relative to running contention-free [20, 72, 21]. Directly measuring the amount of contention is very difficult on modern hardware. The best known approach is to approximate it using *memory intensity*, which is simply the rate of off-chip memory requests generated by a process or thread [72].

(b) *Network and disk IO*. *ClavisMO* measures the IO intensity via two metrics known to work well from the DeepDive experience [54]. The network intensity is described by the idle CPU cycles due to outstanding packets in the Snd/Rcv queue. The contention for disk is characterized by a percentage of time the given process was waiting for the disk IO.

The metrics enable us to quickly characterize the resource intensity of each process online, simply by measuring their values using hardware performance counters and standard system tools. We classify processes based on their resource intensity into "animalistic" categories adopted from an earlier study [69]: Tasmanian *devils* or *tazes* and *turtles*. Tazes suffer performance degradation in excess of 10% when more than one are co-scheduled on the slots of the same node because they aggressively compete for the shared resources, while turtles are peaceful, mostly compute intensive or IO-bound jobs, that can co-exist with each other and with tazes. Figure 1 shows the animal classes for the representative MapReduce and HPC workloads.

The metric values and performance degradations needed to delineate the categories are hardware specific, but we found them to be rather stable across several different makes and models of multicore processors we tested. Moreover, the categories can be quickly established using well known techniques. For example, we have adopted a training module proposed elsewhere [30] that can quickly identify the proper thresholds for a particular server architecture. Alternatively, the descriptive metric values can be learned during application execution, as suggested by the DeepDive framework [54]. Both methods are fully complementary to *ClavisMO*.

In addition to using the degree of resource intensity in cluster-wide scheduling decisions, *ClavisMO* also uses it for node-local scheduling, in order to avoid placing competing tazes on the same chip whenever we have more than one running on the same node. For that purpose, we use the DINO scheduler from [20, 21].

Quantifying communication overhead: We identify the jobs whose performance is lagging because of accessing *a slow link*: a part of the cluster interconnect that provides an inferior communication performance to the job. We consider two forms of communication overhead typically present in the real datacenters:

(a) When the processes of the job are spread across multiple nodes, as opposed to being collocated within a node, they communicate via a cross-node interconnect, which is usually much slower than the internal node interconnect. This form of communication overhead only matters for jobs that can fit within a node, as bigger jobs will access cross-node interconnect regardless of their placement.

(b) When the job processes communicate via a slower channel that links several racks together. Due to the geographical distribution of racks and the cost of the equipment, the delay and capacity of the cross-rack channels is often much worse than that of the links within each rack [37], so if a process communicates off-rack it can experience a slowdown.

We use the amount of traffic exchanged via a slow link as the metric for communication sensitivity. With the classification techniques described above [30, 54], we account for the two-level datacenter topology by placing the workloads into three communication related categories (Figure 1). The jobs of the *node* category are small traffic-intensive codes, whose processes benefit from being consolidated entirely within a node. The jobs of a *rack* category do benefit from running within a rack, while the jobs marked as *cluster* are neutral about the topology due to low communication. *ClavisMO* uses traffic accounting with iptables [14] to measure on-the-fly the traffic exchanged by every process on the system and detect those that use a slow link to communicate.

Many long-running Google jobs were found to have relatively stable resource utilizations [63]. We found the same to be true for many HPC workloads. However, both shared resource contention and the rate of network exchange in our work are tracked *on-the-fly* and without any workload modifications. This allows *ClavisMO* to handle dynamically changing workloads with phases in their execution.

3. CLAVISMO: THE SOLUTION

ClavisMO represents the problem of process placement as a multi-objective optimization. There are many ways to solve MO problems [28, 7]. In *a priori* methods, preference information is first asked from the decision maker and then a solution best satisfying these preferences is found. *Scalarizing* is a popular approach for implementing *a priori* methods, in which a multi-objective optimization problem is formulated as a combination of single-objective optimization problems (e.g., minimizing a weighted sum of goals) [8, 7]. The advantage of *a priori* methods is that they are *automatically* guided by predetermined numerical weights which allows them to find a small number of desired solutions. As such, these methods can be used by an automatic scheduling framework. The disadvantage is that it is often hard to determine the right values for the weights.

In contrast, *a posteriori* methods do not require any preferential weights. They produce a Pareto front of solutions for the task: each one of these solutions is better than the rest with respect to at least one objective. These methods generate a large, diverse set of solutions from which the user can choose. Although there is no need to specify numerical weights, the downside is that they require manual involvement to pick the most suitable solution.

Our idea in *ClavisMO* is to combine the two approaches together: first, we use a popular a posteriori genetic algorithm (MOGA) to generate a Pareto front of possible solutions for a representative cluster workload. These solutions are then visualized, allowing the cluster administrator to select job placements that are deemed acceptable for a particular cluster. This step needs to be performed only *once*. We believe this is a reasonable assumption for HPC, where operators often know their workload. For MapReduce, the Google trace analysis [63] has also shown that many jobs represent repeated programs (e.g., periodic batch jobs or programs being tweaked while being developed), thus providing a natural opportunity for resource prediction. *In choosing particular placements for a representative workload, the system administrator expresses his or her preferences with respect to balancing the optimization objectives.* The chosen schedules are then used to calibrate the weights of the automatic solver that finds solution for the actual cluster workload on-the-fly during cluster operation. This is done through a well-studied linear scalarization. We next describe these two parts of *ClavisMO*.

3.1 Visualizing Pareto Fronts and Calibrating Weights

To find a Pareto front we use *gamultiobj* solver from Matlab. *Gamultiobj* finds minima of multiple functions using the non-dominated sorting genetic algorithm (NSGA-II) [28, 50]. In its search, the algorithm favors solutions that are "genetically better" (have minimum objective values as of yet). It then tries to crossover those solutions to find the better ones. Besides solutions with better "genes," the algorithm also favors solutions that can help increase the diversity of the population (the spread in goal values) even if they have a lower fitness value, as it is important to maintain the diversity of population for convergence to an optimal solution. We used the custom population type for the solver and adopted a partially matched crossover and mutation operators to our problem.

To help the user guide the weight calibration, we need to demonstrate to them the extent of trade-offs resulting from various degrees of collocation. To that end, we visualize the Pareto front of potential trade-offs using a tool called *Vismon*. Vismon was created to facilitate analysis of trade-offs in decisions related to fisheries [23], and we adopted its source code to our domain. Figure 4 shows the Pareto front that resulted from running *gamultiobj* on an hour-long Hadoop workload. The communication traffic and job submission rate for this workload are taken from historical Hadoop traces from a 600 node production Facebook cluster [13, 26]. We further enriched the traces with the real performance and resource contention data from FutureGrid and Grid 5000 datacenters (see Section 4).

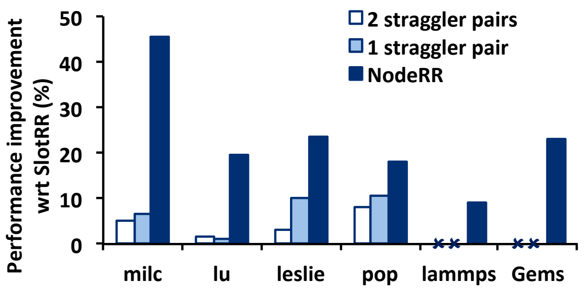


Figure 3: Relative comparison of SlotRR and assignments with varying degree of contention.

Every placement is characterized by its power consumption, communication overhead and intra-node resource contention. Hence, to visualize these three properties in 2D we need three (${}_3C_2$) screens, each one showing how each placement trades off each pair of normalized objectives. The axes show the objectives and the dots represent the placements. The objectives are defined as follows:

P: The total power consumption of the schedule. The lower this value for the schedule, the more energy efficient the schedule is.

D: *Diversified intensity* – this is a measure of contention. It is computed as a cumulative difference between *resource intensities* on each node. We provide details of how we measure resource intensities via dynamic online profiling in Section 2, and in the following text we refer to highly resource-intensive processes as *tazes* and to low-intensive processes as *turtles*. Higher values of diversified intensity correspond to more tazes being isolated from other tazes (collocated with turtles or placed alone on a node). This contributes to low shared resource contention on a node. For a better placement, this goal needs to be maximized.

C: *Diversified sensitivity* to the traffic exchanged via a slow link is defined as a number of communication sensitive processes placed remotely from the rest of their job. We explain what communication sensitive is in Section 2. Lower values identify robust placements with low communication overhead.

The last two objectives are subject to a constraint: in order to get the performance boost from a decrease in contention or in communication latencies, we need to provide it for all the containers within a job. For example, we need to isolate all the tazes within a job from other tazes to provide a performance boost. If there is at least one taz container within a job that is collocated with another taz, there will likely be no performance benefit for the job. This is known as the *straggler's problem* in MapReduce: the lagging container slows down the entire job [71]. For HPC workloads it is illustrated in Figure 3, where the respective taz jobs were running on 8 containers. We compared the following cases relative to SlotRR (4 nodes, 2 tazes per node): *2 straggler pairs* (4 nodes, 1 taz per node and 2 nodes, 2 tazes per node), *1 straggler pair* (6 nodes, 1 taz per node and 1 node, 2 tazes per node), and NodeRR (8 nodes, 1 taz per node). When there is at least one lagging pair, the performance benefit from contention mitigation is very limited, sometimes jobs even hang due to divergence of running times in their components.

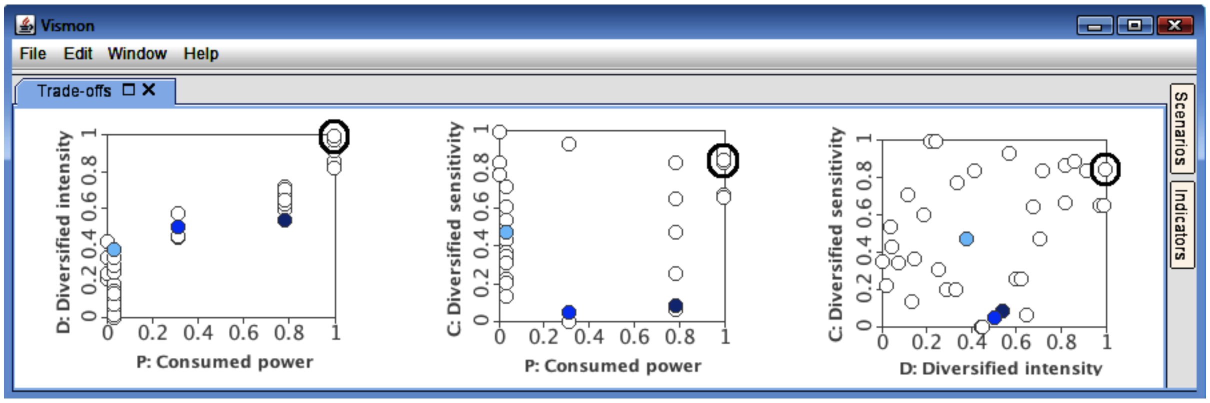


Figure 4: Choosing solutions from Pareto front with Vismon (the chosen solutions are colored).

Armed with the information presented by Vismon, the system administrator must pick the schedules (by clicking the points on the screens) that best express his or her preferences with respect to balancing among the objectives. Here the administrator is guided by the knowledge of performance of representative workloads on his or her cluster. In our case, we assume that the administrator has seen the data similar to what we presented in Figure 1a for Hadoop, and so he or she knows that: (a) contention can be a serious problem, and (b) communication overhead is typically not large with the exception of a few sensitive jobs. Also, for the sake of the example, we assume that in this case the administrator is more concerned with performance than energy.

With these goals and information in mind, the administrator is to pick at least three acceptable schedules on each screen in Figure 4. The colored points are those selected by the administrator. Notice how the administrator chose solutions that: (a) cover the entire spectra for the power objective; (b) provide consistently good values for the important contention factor, and (c) have the diversified sensitivity that varies, but is not exceedingly high. The administrator avoids some points, like the one circled on Figure 4, which while resulting in low contention also gives us many communication affected processes. According to Figure 1a, this will likely result in the performance degradation.

Once the acceptable schedules are chosen, *ClavisMO* proceeds with calibrating the numerical values for weights α, β and γ of the weighted sum equation:

$$\alpha P_n - \beta D_n + \gamma C_n = F \tag{1}$$

where F is the objective function that is to be minimized, P_n, D_n, and C_n are normalized goal values. The weights can be obtained from schedules chosen by the system administrator in many ways. In *ClavisMO*, each weight is inversely proportional to the standard deviation of its goal spread from schedules #1-3: the more spread out the goal is, the less important its value for the cluster and so lower the weight. For the normalized goals only the relative value of the weights matter [15], so we use 0.22, 1.00, and 0.36 as the weights in this case.

3.2 The Placement Algorithm

SolverMO is the component of *ClavisMO* that computes the approximation to the optimal schedule using a popular open-source solver Choco [67]. Finding a multi-objective job placement is similar to the vector bin packing problem [55]. Here we also need to pack multi-dimensional items (containers) into bins (nodes). Containers are multi-dimensional, because they have three attributes: contention class, communication category and power consumption, and we need to account for these attributes while packing the bins. Our problem complicates vector bin packing in that the goal is not to pack the items as tightly as possible, but to find a placement that balances the degree of collocation with other performance- and energy-related factors.

Vector bin packing is NP-hard even for a single dimension. We tackle the problem by creating a custom branch-and-bound *enumeration search tree* with *AbstractLargeIntBranchingStrategy* class of Choco, and then traversing this tree (Figure 5). On each tree level we evaluate the placement of a particular container. The branches are possible nodes that this container can be placed on. Traversing this tree can take a very long time, many hours, and even days. We compute scheduling decisions online, so we only have a few *minutes* to complete the traversal. To address this problem, we *use domain-specific knowledge to prune those parts of the tree that are unlikely to contain good solutions*.

There are two ways in which we can eliminate parts of the traversal: *custom branching* and *custom bounding*. A generic solver will create as many branches in the tree as necessary to explore *all* assignments of containers to nodes. However, from our point of view, not all of these assignments are interesting to consider. To eliminate useless assignments, we create a ***custom branching strategy***, described below. Another problem with a generic solver is that it will traverse *every* branch it creates. However, by looking at the initial placement of containers that was done prior to reaching a particular branching decision, we can sometimes already predict that following a particular branch any further will not result in a better scheduling assignment than the incumbent (the best assignment found so far). In that case, it is better to skip that branch altogether, pruning a large part of the tree. In order to predict whether a particular branch is worth skipping we design several ***custom bounding functions*** that we also describe below.

1) *Custom branching.* For our scheduling task, it is redundant to evaluate all possible assignments of containers to nodes. In constraint programming, this is often referred to as symmetry breaking. For example, suppose that we need to decide on possible node values for placing a container Y1 from Figure 5 which is classified as a communicating turtle. There are already three containers instantiated before that one in the tree: one is a turtle Y0 communicating with Y1, and the rest two are non-communicating tazes X0 and X1. All three previously assigned containers are alone on their respective nodes. Additionally, three nodes are currently idle – no container is assigned to them. So, when deciding how to branch Y1, we only need to explore three branches: one idle node, one node with a communicating turtle, and one node with a taz. We do not explore other branches: the placements they offer will not result in a different weighted sum outcome and hence can be pruned. Such are the dotted branches in Figure 5. We refer to the branches

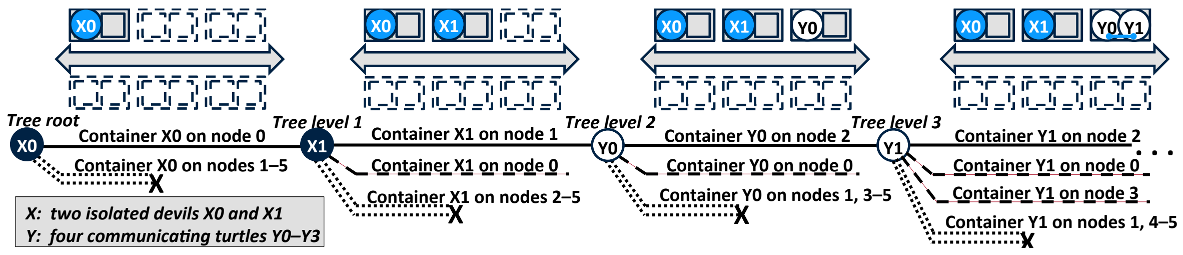


Figure 5: Example of a Branch-and-Bound enumeration search tree created by *SolverMO*.

that need to be evaluated as *fundamentally different*. They are pictured as dashed and solid lines in Figure 5 (the solid lines mark the currently evaluated solution).

In a custom branching strategy, we also decide what branches to evaluate first. This is called *branch prioritization* and it is important when the weights for each objective component in Equation 1 are different. For instance, exploring branches which result in separated tazes before branches that lead to collocated tazes could give a better solution sooner if the contention is more important than power. Figure 5 illustrates branch prioritization for $\gamma > \beta > \alpha$.

2) *Naive bounding function.* The idea behind the *Naive* bounding function, is that we estimate each addend from Equation 1 independently of other addends. For example, to calculate the total power consumption, we assume that all the containers are packed on as few nodes as possible regardless of their contention class or communication patterns. Similarly, for the diversified intensity we assume that each "tazish" container is put on a node by itself, unless no idle nodes are present. To estimate the total amount of traffic exchanged over slow links, we assume that the small jobs can fit within a node and that off-rack traffic is only experienced by the jobs that cannot fit within a rack.

3) *Elaborate bounding function.* The naive bounding function is not very precise in its estimate: in the real world, the three goals are interdependent. For example, decreasing the power consumption usually increases the number of collocated tazes. The *Elaborate* bounding function is based on a more strict simplification according to which the addends from Equation 1 are estimated from a potential workload placement. Let us return to the example from Figure 5. We see that in the path currently taken by the solver (the solid lines), four containers (X0, X1, Y0, Y1) are already instantiated. To estimate the best possible F that can be obtained in the subtree stemming from Y1, we first calculate the possible range for the number of collocated turtle pairs tt and the number of collocated taz-turtle pairs dt that can be formed with yet unassigned containers Y2 and Y3. The number of collocated taz pairs dd for this problem is determined by tt and dt, so each pair (tt, dt) describes how the unassigned containers of a particular class are placed with respect to other containers. In our case, $tt \in \{0, 1\}$ (both Y2 and Y3 are turtles and the rest of the turtles Y0 and Y1 are already together); and $dt \in \{0, 1, 2\}$ (because the two already assigned tazes X0 and X1 can share a node with a turtle).

We then iterate through the potential workload placements represented by pairs (tt, dt) and estimate the three goals for each of the pairs. The estimations are not independent anymore, but are based on a particular class distribution within the cluster, which makes them more precise. The best estimated F across all iterations is the value returned by the elaborate function. To make the calculation fast, we process each iteration in its own thread, which takes advantage of the multiple CPUs and works very well in practice.

4) *Elaborate fathoming.* The bounding function in each bud node estimates the best future values for addends from (1). Fathoming is a branch-and-bound technique that piggybacks on these calculations and attempts to greedily find a good schedule that is in accordance with the bounding predictions. This is done as follows. We know that the elaborate bounding function predicts the number of collocated turtles tt in the best schedule. Knowing this, the fathoming generates a greedy schedule by taking two random unassigned turtles, placing them on an idle node and then removing those turtles from further consideration. After this action, the value of tt is decremented. Fathoming proceeds until $tt = 0$ or no more turtles are available, after which taz-turtle pairs are taken and so on. Once the fathoming schedule is formed, the solver compares it to the incumbent: if the fathom is better than the incumbent, the fathom replaces incumbent.

5) *Approximate bounding* prunes more than the bounding function suggests: now a branch is pruned if the bounding function estimate is greater than only K% of the incumbent. K<100 can further increase the number of tree levels explored (hence the confidence in the incumbent, something a greedy heuristic cannot provide), but the solver may miss a better incumbent. However, if a solution with a better weighted sum does exist within the pruned tree nodes, it differs from the best weighted sum by no more than (100-K)%.

4. EVALUATION

In this section, we first evaluate the efficiency of our multi-objective solver and then present experiments with *ClavisMO* deployed on cluster installations.

4.1 Solver efficiency

Table 1 provides the evaluation data for our customized solver. The average results are shown for more than 5,000 runs of the solver. In each run, we terminated the search after 60 seconds and obtained the best weighted sum found so far. We also experimented with longer running times (600 and 6000 seconds) to see if this will give us better solutions. We varied the custom features enabled in the solver (Section 3.2) to evaluate their impact. We also varied the weights assigned to each objective from 1 to 10 with the increment of 1 (i.e., we minimized the weighted sum with the weights set to {1 1 1}, {1 1 2}, {1 2 1}, {1 2 2} and so on). We used resource intensity and communication patterns of our HPC programs. We configured the workload to have up to 1024 containers simultaneously running on the cluster with the number of nodes to be the same as the maximum number of containers. The number of nodes and containers was sufficiently large that it was impossible to find the truly optimal weighted sum within a feasible period of time. So, to evaluate the solution, we compare the weighted sum found by each solver to the best one found across all solver runs.

Row 0 shows the results for a simple solver that randomly searches for a solution and upon completion gives the best one it could find. We then evaluate the solutions with progressively more optimization features (rows 1–5). The metrics of importance are: *weighted sum relative to the best* (smaller is better) and *average fraction of tree levels explored* (higher is better).

The results suggest that the proposed optimization features are largely beneficial in terms of either approaching the best weighted

Row	Features enabled:							Performance (up to 128 containers per job):			
	Best Random	Custom branching?	Naïve bounding function?	Elaborate bounding function?	Elaborate fathoming?	Prune top % branches?	Time to solve, sec	Weighted sum relative to the best one	Average tree nodes evaluated (solutions tried for random)	Average fraction of tree nodes pruned	Average fraction of tree levels explored
0	yes						60	3.14x	505078		
1		no	no	no	no	no	60	5.18x	338789	0.00	0.00
2		yes	no	no	no	no	60	1.29x	663692	0.00	0.01
3		yes	yes			no	60	1.29x	3600690	0.41	0.04
4		yes		yes	no	no	60	1.23x	2267173	0.81	0.67
5		yes		yes	yes	no	60	1x	376533	0.80	0.93
6		yes		yes	yes	yes	60	1.03x	58055	0.88	0.96
7		yes		yes	yes	yes	600	0.99x	6934349	0.89	0.97
8		yes		yes	yes	yes	6000	0.99x	71722876	0.90	0.97

Table 1: Solver evaluation (custom branching strategy).

Cluster	Nodes	Cores	CPU count per node, make, model	RAM	Disk	Network
Parapluie (Grid 5000) [22]	8	192	4 AMD 6164 HE	48GB	232GB	1GbE, InfiniBand
India (FutureGrid) [34]	16	128	2 Intel X5570	24GB	434GB	1GbE, InfiniBand

Table 2: Clusters used for experiments.

sum or in the percentage of the problem space explored. The most efficient solver is the one that uses all the optimization features and the most sophisticated bounding function (row 5). This solver reduces the weighted sum by 3× on average relative to the random solver and explores 93% of the search tree.

In row 6, we show approximate bounding with K=99%. This further increases the number of tree levels explored by 3%, but does not correspond to a better weighted sum. This means that, if a solution with a better weighted sum exists within these 3% of the nodes, it differs from the best weighted sum by no more than 1%. We also note that, due to our branch prioritization, it is unlikely that the best solution is located at the end of the tree traversal. Rows 7 and 8 provide results for longer solver executions. As can be seen, longer execution times only marginally improve the solution, so we stick with the 60-second cut-off.

We believe re-evaluating scheduling assignments every minute is often enough to optimize cluster-wide schedules of most long running jobs, especially if the new schedule requires cross-node live migration of the job containers. While *ClavisMO* can also improve performance of short jobs locally within the node with the DINO scheduler, we note that prior studies have shown that tasks which start or stop within five minutes account for less than 10% of the overall Google cluster utilization [63]. For such jobs, it is better to not perform cluster-wide scheduling. The short jobs can be identified based on their requested walltime (in case of HPC), or their progress so far (from Hadoop logs). Many clusters also allow for a separate long-running queues or production priorities [63] which have a much higher proportion of long-running jobs.

4.2 Experimental configuration

Table 2 shows the two clusters used for our experiments. In the MapReduce setting, we used Hadoop's default scheduler as the job scheduler and Torque as the resource manager (it is also used in Hadoop On Demand [4]), Intel HiBench, and Purdue's PUMA as the workload. In the HPC setting, we used Maui as the job scheduler, Torque as the resource manager, SPEC MPI2007 and SPEC HEP as the workload. All nodes were running Linux 2.6.32.

We compare performance and energy consumption under three job schedulers: *Baseline*, *Expert*, and *ClavisMO*. *Baseline* is the state-of-the-art scheduler used in HPC clusters. With *Baseline*, the cluster is configured to use either SlotRR or NodeRR policy. The policy is chosen by the system administrator based on their knowledge about the performance of representative workloads on a particular cluster. For instance, for the Parapluie cluster, the administrator would use performance data similar to that shown in Figure 1 to decide that NodeRR would be a more favorable policy. Given the limitation of state-of-the-art cluster schedulers, we expect that *Baseline* would often create schedules that lead to poor performance and energy waste. Therefore, *Baseline* should be considered as the lower bound.

Many cluster users and operators report real cluster sizes of only a few dozen servers [59, 36]. We have performed experiments on real clusters of similar size, and, due to its scale, we assume that job placement may sometimes be performed by a knowledgeable system administrator based on workload characteristics (contention-intensive or not, communication-intensive or not). Note that this *Expert* is different from *ClavisMO* approach, because our solution asks administrator for the input on goal priorities only *once* during the cluster operation, while *Expert* is allowed to intervene every time a new job arrives. Since typical HPC clusters do not use virtualization, we assume that the *Expert* is unable to migrate jobs after they began execution (due to lack of virtualization support or lack of time). Nevertheless, clever placement of jobs prior to beginning of execution gives an important advantage, so we expect to see very good performance and energy with the *Expert* scheduler.

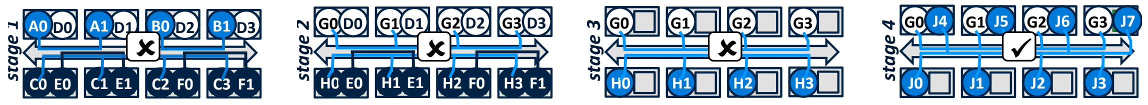
It is impractical to expect job scheduling to be performed manually by a human on a sizeable installation. Hence, when evaluating *ClavisMO* for a large scale Hadoop installation in Section 4.5, we compare it with a greedy heuristic called *Dot-Product* [55], the most efficient vector-packing heuristic known to us.

To mimic a typical batch-job environment, we create a job queue inside the cluster. Jobs are scheduled to run in the order of their arrival. Some jobs have dependencies: they cannot run before the job they depend on completes. In the figures that follow, we will show the placement of jobs in the cluster across several stages. A stage begins when a new set of jobs is assigned to run and ends when those jobs complete.

4.3 HPC workload on Parapluie

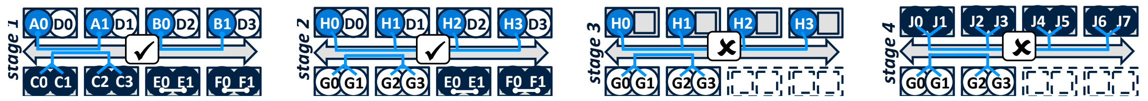
Figure 6 shows the placement of jobs under *Baseline*. There are four stages, each lasting six hours, giving us a day-long cluster-wide experiment. Job containers are shown in different colors to indicate their resource intensity. Turtle containers are white. Taz containers collocated with other taz containers (suffering from intra-node resource contention) are shown in black. Taz containers collocated with turtle containers or running alone (no contention) are shown in blue. Containers of *node* and *rack* category (sensitive to slow traffic) are connected by a solid line.

Baseline can stumble into good placements (e.g., when it collocates the taz containers of A and B with the turtle containers of



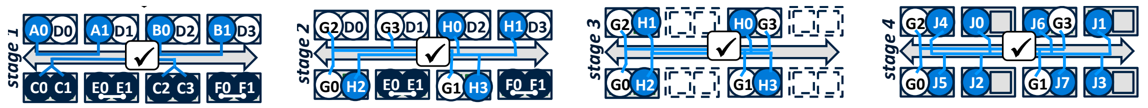
(a) A–F run on the cluster. (b) G, H start only after A–C. (c) Jobs D–F have finished. (d) J starts only after job H.

Figure 6: Baseline NodeRR experiments on Parapluie cluster.



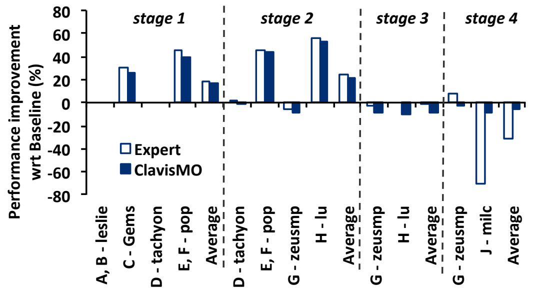
(a) Expert manually consolidates network-bound E, F. (b) D is put together with H which makes H run faster. (c) Expert cannot collocate H, G after D has finished. (d) Expert is unable to spread J apart due to lack of space.

Figure 7: Expert on Parapluie cluster, requires offline profiling and regular human involvement.

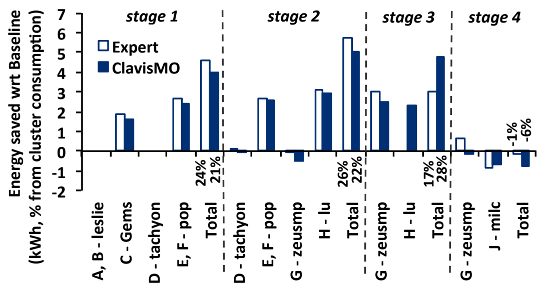


(a) *ClavisMO* dynamically consolidates E and F. (b) H is collocated with D and G to make H run faster. (c) G and H are consolidated on-the-fly to save power. (d) *ClavisMO* rearranges J and G to make J run faster.

Figure 8: *ClavisMO* automatic experiments on Parapluie cluster.



(a) The performance results of Parapluie runs.



(b) The energy measurements from Parapluie runs.

Figure 9: Experimental results on Parapluie cluster.

D (stage 1, Figure 6a), or when it spreads the taz containers of J apart and fills in the gaps with the turtle containers of H (stage 4, Figure 6d)). At the same time, there are many situations when it creates a bad placement. For example, separating the containers of communication-intensive E and F hurts their performance (stages 1 and 2). Collocating A and B with D results in a later collocation of two turtles (D and G), which is wasteful considering that those turtles could be collocated with the taz H instead (stage 2), reducing resource contention for H. Finally, lack of virtualization and online profiling make it impossible to detect that G and H in stage 3 are complimentary and could be collocated to save power.

The *Expert* scheduler (Figure 7) is able to detect that E and F need to be consolidated (stage 1, Figure 7a). It also makes sure that taz H is placed with turtle D upon its launch and that the containers of G stay together and do not mix with E and F even though this could reduce contention (stage 2, Figure 7b). This results in a significant (up to 60%) speed up for the job execution in stages 1 and 2 (Figure 9a). Total energy savings associated with the jobs completed ahead of time reach 6 kWh (Figure 9b). However, good decisions made by *Expert* initially affect the later schedules. In stage 3, the well-placed jobs D, E, and F have terminated, leaving a poor schedule in which G and H consume more power than required. In stage 4, the previously placed G prevents us from spread J to use more nodes, thus causing the straggler's problem and resulting in performance loss of more than 70% (Figure 9a).

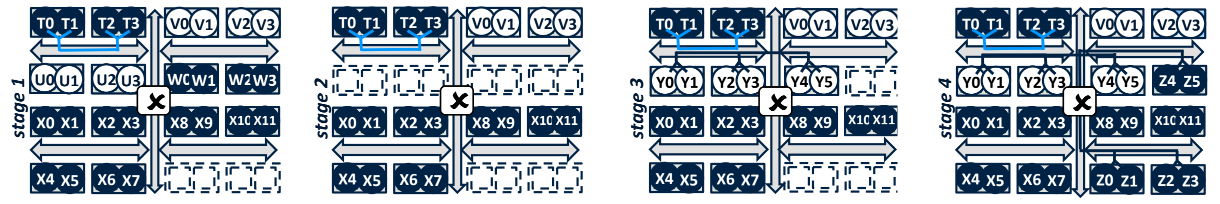
ClavisMO is able to find reasonable solutions at every stage automatically (Figure 8). If the initial placement is poor, *ClavisMO* migrates the container to achieve a better mapping. Migration is performed without affecting the Maui job queue. Besides stages 1 and 2, *ClavisMO* is also able to reach the most energy-efficient solution in stage 3 (Figure 8c), almost 2 kWh better than that of *Expert* in total. In stage 4, it is able to prevent significant performance loss by dynamically reshuffling containers of F and H across the cluster (Figure 8d). Interestingly, the energy consumption of the schedule found by *ClavisMO* in stage 4 is similar to that of the *Expert*, which uses fewer nodes, because *Expert* severely slows down J (by more than 60%), making it run longer and consume more energy.

4.4 MapReduce workload on India cluster

In the India cluster, nodes are divided among 4 racks that are connected via a slow link. We use the *netem* kernel module to create a multi-rack installation [9]. In such a cluster, the administrator is primarily concerned with avoiding the communication overhead of accessing the remote racks. We also assume that the power is as important as performance, and so SlotRR is chosen as the *Baseline*.

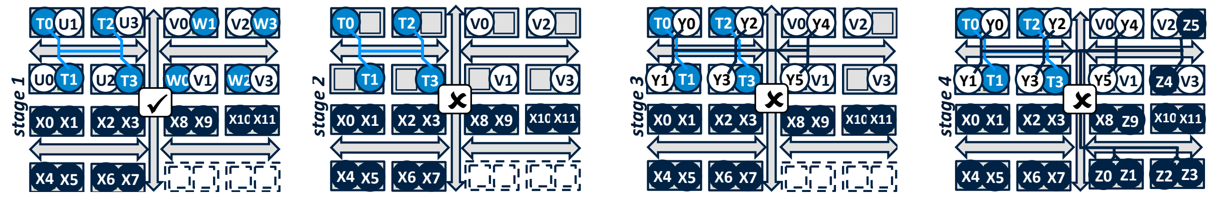
It is well known that servers consume a significant amount of power when they are idle, often more than 50% of peak power [6]. This is indeed the case on both Parapluie and India. At the same time, modern low idle power servers could also be deployed [49]. To show how *ClavisMO* adapts its decisions to changes in hardware, we here assume that the idle power consumption on India has been reduced from 50% of the peak power, to only 25%.

Job placement under *Baseline* is shown in Figure 10. The job queue in these runs is comprised of 7 Hadoop jobs, T through Z. It is not uncommon for MapReduce workloads to utilize huge parts of the cluster, so the biggest job X (classifictn) in our testbed requires 37% of the computing facilities (12 containers). The sheer size of this job makes it difficult to improve its performance in stage 1. The taz jobs T and W, on the other hand, can receive more than 40% boost if we collocate them with turtles U and V. Both *Expert*



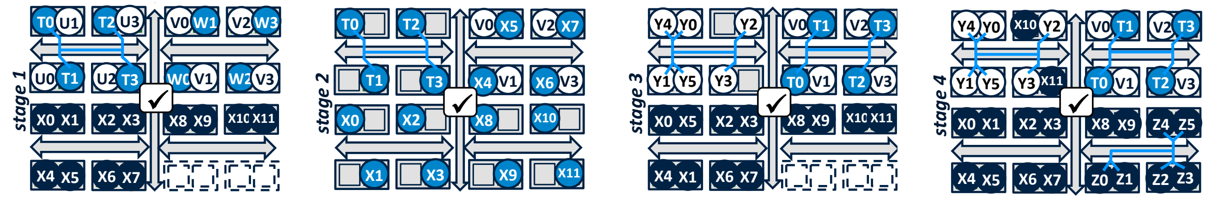
(a) The cluster runs jobs T–X. (b) U and W have finished. (c) Y's adv. reservation began. (d) Z's adv. reservation began.

Figure 10: Baseline SlotRR experiments on India cluster.



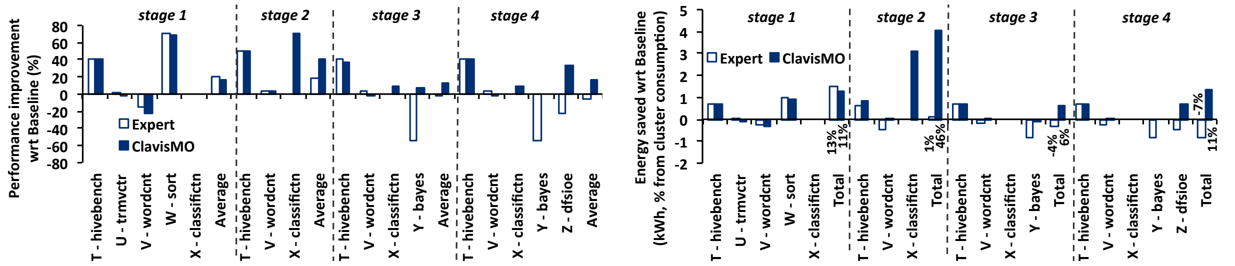
(a) Expert manually shuffles T, U and V, W to boost T and W. (b) Expert cannot spread X apart to increase performance. (c) The job Y now has to suffer remote communication. (d) Expert cannot reshuffle X and Z. Z sends data off-rack.

Figure 11: Expert on India cluster, requires offline profiling and regular human involvement.



(a) *ClavisMO* dynamically collocates T, U and V, W. (b) *ClavisMO* boosts X's performance. (c) *ClavisMO* puts network-bound Y within a rack. (d) *ClavisMO* reshuffles X and Z to avoid off-rack traffic for Z.

Figure 12: *ClavisMO* automatic experiments on India.



(a) The performance results of India cluster runs. (b) The energy measurements from India cluster runs.

Figure 13: Experimental results on India cluster.

and *ClavisMO* do just that (Figures 11a and 12a).

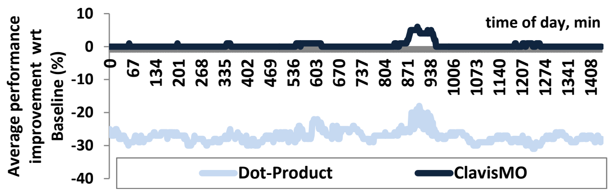
Despite our cluster being nearly fully utilized most of the time, in stage 2 it happens to have a pocket of six idle nodes (Figure 10b). This is because jobs Y and Z that should run next have advanced reservations and are waiting for their time slot. This opportunity is seized by *ClavisMO* , which spreads the containers of X on different nodes (Figure 12b), giving X a 62% boost in performance (Figure 13). By letting X use more nodes, *ClavisMO* also saves 3.1 kWh of energy relative to *Baseline*, because X now runs much faster. The gains in energy are significant (46% of the total cluster consumption) due to using dynamic power management, and *ClavisMO* is aware of that, because its weights were calibrated using the power data from the "energy-savvy" India cluster. If a similar situation had occurred on the the Parapluie cluster, where idle power consumption is much larger, *ClavisMO* would have consolidated X, because energy cost outweighs performance benefits in that case.

The jobs Y and Z on stages 3 and 4 arrive at the time when every rack on the cluster is partially loaded. As a result, neither *Baseline* nor *Expert* are able to place them entirely within a rack, triggering slow cross-rack communication. Only *ClavisMO* is able to avoid the worst case by reshuffling containers of already running jobs, thus freeing some rack space for the newcomers (Figure 13a).

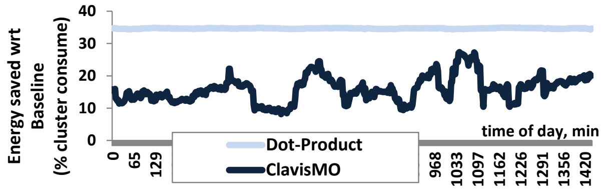
4.5 Large scale trace-driven analysis

To conclude, we follow a common distributed systems approach [19, 27, 56, 35, 59] of complementing our real cluster experiments with large scale trace-driven *ClavisMO* evaluation. The datacenter simulators are known to be very slow when it comes to simulating a scalable cluster installation [25]. To describe the contention- and communication-related behavior of installations with many thousands of jobs, we opted instead for a simple and fast in-house, trace-driven analysis. We bootstrapped it with the communication rate and job mix from a day long historical Hadoop traces of the production Facebook clusters. The two clusters have 600 or 3000 nodes and execute 6 or 24 thousands Hadoop jobs daily [13, 26]. We fill in the gaps about resource contention data by executing Hadoop jobs on our Parapluie cluster under *ClavisMO*. For that, we use the Hadoop configuration, shared by Facebook [3], and run a representative MapReduce workload from Figure 1a.

With this information, we then create the workload placement

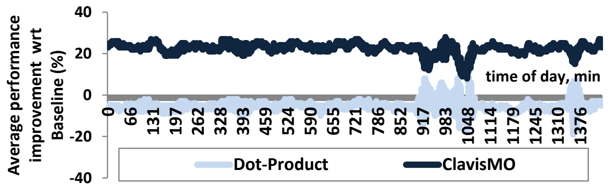


(a) The trace-driven analysis of performance.

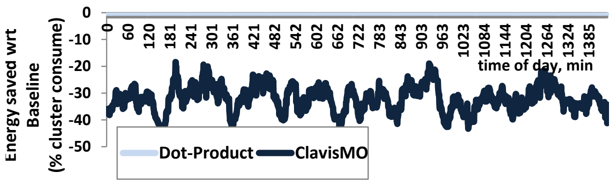


(b) The trace-driven analysis of energy savings.

Figure 14: Trace-driven analysis of *ClavisMO* on a 600-node Hadoop installation (Baseline=NodeRR).



(a) The trace-driven analysis of performance.



(b) The trace-driven analysis of energy savings.

Figure 15: Trace-driven analysis of *ClavisMO* on a 3000-node Hadoop installation (Baseline=SlotRR).

schedules with *Baseline*, *ClavisMO*, and *Dot-Product* solvers. We use the weights calculated as in Section 3. The results are given on Figures 14 and 15. We note that the workload is on 86% comprised of isolated jobs of single container (likely because users split embarrassingly parallel programs into many distinct tasks [63]), many of which are tazes. Because of that, there is little room for performance improvement relative to the NodeRR scenario which is the default for Parapluie and for our simulated environment. However, *ClavisMO* is notably good at avoiding the worst case performance, as demonstrated relative to the *Dot-Product* heuristic.

The reason why *Dot-Product* struggles is because it is greedy; it looks only at one node at a time. This approach has been shown to work well when the goal is to fit within the allowed machine capacities. However, the task of addressing shared resource contention effects is more challenging than that, because the jobs can experience degradation due to sharing, even though the node capacities are perfectly met. This is what *dot-product* stumbles into by packing the workload on as small number of nodes as possible, subject to their resource constraints. By doing so on a cluster with high idle power consumption, *dot-product* is still able to provide energy benefits, when it turns the idle nodes off. However, it is important to remember that the weights for this cluster were setup with having performance as the most important goal. Due to the multi-objective nature of *ClavisMO*, it is able to avoid performance degradation, while saving power when possible.

5. RELATED WORK

One key feature of *ClavisMO* is that it addresses intra-node resource contention in clusters. Among existing systems, Q-Clouds [53] is the most similar in that respect. Q-Clouds tracks resource contention among co-located VMs, but instead of separating VMs to different physical nodes, it allocates extra CPU cycles to VMs suffering from contention (as was also proposed in [33]). Q-Clouds is not designed to balance multiple objectives.

There are a number of systems that, like *ClavisMO*, target job placement, but focus on different aspects, mainly CPU and physical RAM capacity, and use different management strategies. Sandpiper [68] detects hotspots on physical servers and relieves them by migrating one of more VMs to a different server. The approach taken by Sandpiper can be characterized as local and reactive. *ClavisMO*, by contrast, aims to achieve a global objective and proactively adjusts VM placement to meet this goal. The generality of *ClavisMO*'s multi-objective optimization framework allows adding other objectives, such as minimizing CPU and memory overload as in Sandpiper, cooling power [17] or cost [57].

We are aware of two systems that, like *ClavisMO*, address process placement and balance multiple objectives: Quincy [44] and Entropy [39]. Quincy aims to locate a task close to a file server holding its data, and balances objectives such as fairness, latency, and throughput. Quincy represents the scheduling problem as a graph and reduces it to the min-flow problem. *ClavisMO* uses a classical multi-objective framework. Both are valid solutions, but the unique property of *ClavisMO* is its ability to interact with the user in calibrating the system.

Entropy [40] also uses Choco to find a process placement solution under constraints. Its goal is to minimize the number of utilized nodes while meeting CPU and memory constraints. *ClavisMO*'s key difference is that it adapts Choco to use the domain specific knowledge, thus reducing the time of problem space traversal.

Importance sampling [66] can be used to further refine the Pareto front of *ClavisMO*. It can also be utilized as an effective fathoming technique during the placement stage.

The current implementation of *ClavisMO* improves energy use in clusters using process placement. An alternative approach is to devise a novel hardware platform, customized for a particular type of workloads. Most prominent examples of such work include FAWN that couples low-power embedded CPUs and small amounts of local flash storage to improve the performance of IO intensive jobs [18, 32], and Triton that shows how to design a balanced hardware platform tuned to support a large scale sort application [61, 60]. Process placement is complementary to hardware solutions.

The inherent flexibility and absence of capital investment costs have made cloud an attractive platform for bursty HPC workloads [24, 64, 58, 51] and some educational HPC clusters [45, 29]. The downsides are higher operational costs and virtualizatrion overheads [46, 31, 42].

There is a plethora of work addressing time and queue management of HPC clusters to achieve various performance and energy objectives [36, 48, 47, 65, 52, 50]. None of it touched upon the subject of process collocation.

6. CONCLUSION

We have shown that *ClavisMO* saves energy and improves performance, because in scheduling jobs it balances multiple goals: minimizing contention for shared resources, reducing communication distance, and avoiding resource idleness. Underlying *ClavisMO* is a custom multi-objective solver that can be enhanced to address additional performance and energy-related goals.

7. ACKNOWLEDGMENT

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. SPEC MPI is a registered trademark of Standard Performance Evaluation Corporation (SPEC). Linux is a registered trademark of Linus Torvalds. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

8. REFERENCES

[1] Amazon ECS. *[Online] Available: https://aws.amazon.com/ecs/*.

[2] ClavisMO project webpage. *[Online] Available: http://hpc-sched.cs.sfu.ca/*.

[3] Facebook has the world's largest Hadoop cluster. *[Online] Available: http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html*.

[4] Hadoop On Demand. *[Online] Available: http://hadoop.apache.org/docs/stable/hod_scheduler.html*.

[5] Linux Operating Systems. *[Online] Available: https://cloud.google.com/compute/docs/operating-systems/linux-os*.

[6] Microsoft's top 10 business practices for environmentally sustainable data centers. *[Online] http://cdn.globalfoundationservices.com/documents/MSFTTop10BusinessPracticesforESDataCentersAug12.pdf*.

[7] Multi-objective optimization. *[Online] Available: http://en.wikipedia.org/wiki/Multi-objective_optimization*.

[8] Multi-objective optimization by Kevin Duh. *[Online] Available: http://www.kecl.ntt.co.jp/icl/lirg/members/kevinduh/papers/duh11multiobj-handout.pdf*.

[9] Network emulation with netem kernel module. *[Online] Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem*.

[10] Open VirtualiZation. *[Online] Available: http://en.wikipedia.org/wiki/OpenVZ*.

[11] SPEC High Energy Physics. *[Online] http://w3.hepix.org/benchmarks/doku.php/*.

[12] SPEC MPI2007. *[Online] Available: http://www.spec.org/mpi/*.

[13] SWIM repository. *[Online] Available: https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository*.

[14] Traffic accounting with iptables. *[Online] http://openvz.org/Traffic_accounting_with_iptables*.

[15] Using Weighted Criteria to Make Decisions. *[Online] Available: http://mathforum.org/library/drmath/view/72033.html*.

[16] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar. PUMA: Purdue MapReduce Benchmarks Suite. Technical report, 2013.

[17] F. Ahmad and T. N. Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. ASPLOS XV, pages 243–256, 2010.

[18] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14, 2009.

[19] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. NSDI '13.

[20] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-Aware Contention Management on Multicore Systems. In *USENIX ATC*, 2011.

[21] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, December 2010.

[22] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *Int. J. High Perform. Comput. Appl.*, 20(4):481–494, Nov. 2006.

[23] M. Booshehrian, T. Muller, R. M. Peterman, and T. Munzner. Vismon: Facilitating Analysis of Trade-Offs, Uncertainty, and Sensitivity In Fisheries Management Decision Making. *Comp. Graph. Forum.*

[24] A. G. Carlyle, S. L. Harrell, and P. M. Smith. Cost-Effective HPC: The community or the cloud? CLOUDCOM, 2010.

[25] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy efficiency for large-scale MapReduce workloads with significant interactive analysis. EuroSys '12, pages 43–56, 2012.

[26] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, Aug. 2012.

[27] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. *SIGCOMM Comput. Commun. Rev.*, 41(4), 2011.

[28] K. Deb and D. Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., 2001.

[29] F. Doelitzscher, M. Held, C. Reich, and A. Sulistio. ViteraaS: Virtual Cluster as a Service. CLOUDCOM '11.

[30] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei. A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 83:1–83:11, 2012.

[31] Y. El-Khamra, H. Kim, S. Jha, and M. Parashar. Exploring the Performance Fluctuations of HPC Workloads on Clouds. In *CLOUDCOM*, 2010.

[32] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, 2011.

[33] A. Fedorova, M. I. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, pages 25–38, 2007.

[34] G. C. Fox, G. von Laszewski, J. Diaz, K. Keahey, J. Fortes, R. Figueiredo, S. Smallen, W. Smith, and A. Grimshaw. FutureGrid - a reconfigurable testbed for Cloud, HPC and Grid Computing. 07/2012 2012.

[35] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, 2011.

[36] I. n. Goiri, R. Beauchea, K. Le, T. D. Nguyen, M. E. Haque, J. Guitart, J. Torres, and R. Bianchini. Greenslot: scheduling

energy consumption in green datacenters. SC ’11.

[37] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, pages 51–62, 2009.

[38] A. Gupta and G. Barua. Cluster schedulers. *[Online] Available: http://tinyurl.com/crdkq6f.*

[39] F. Hermenier, S. Demassey, and X. Lorca. Bin repacking scheduling in virtualized datacenters. *Principles and Practice of Constraint Programming–CP 2011*.

[40] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. L. Lawall. Entropy: a consolidation manager for clusters. In *VEE*, pages 41–50, 2009.

[41] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. *Data Engineering Workshops, 22nd International Conference on*, pages 41–51, 2010.

[42] K. Ibrahim, S. Hofmeyr, and C. Iancu. Characterizing the Performance of Parallel Applications on Multi-socket Virtual Machines. In *CCGrid*, 2011.

[43] S. IQBAL, R. GUPTA, and Y.-C. FANG. Job Scheduling in HPC Clusters. *[Online] Available: http://www.dell.com/downloads/global/power/ps1q05-20040135-fang.pdf.*

[44] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP ’09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[45] C. Ivica, J. Riley, and C. Shubert. StarHPC – Teaching parallel programming within elastic compute cloud.

[46] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. CLOUDCOM ’10.

[47] Y. Kessaci, N. Melab, and E.-G. Talbi. A pareto-based GA for scheduling HPC applications on distributed cloud infrastructures. In *HPCS*, 2011.

[48] O. Khalid, I. Maljevic, R. Anthony, M. Petridis, K. Parrott, and M. Schulz. Dynamic scheduling of virtual machines running HPC workloads in scientific grids. NTMS’09, 2009.

[49] S. Khan and A. Zomaya. *Handbook on Data Centers*. Springer New York, 2015.

[50] X. Li, L. Amodeo, F. Yalaoui, and H. Chehade. A multiobjective optimization approach to solve a parallel machines scheduling problem. *Adv. in Artif. Intell.*, 2010:2:1–2:10, Jan. 2010.

[51] T. Mastelic, D. Lucanin, A. Ipp, and I. Brandic. Methodology for trade-off analysis when moving scientific applications to cloud. In *CloudCom*, 2012.

[52] A. Mutz and R. Wolski. Efficient auction-based grid reservations using dynamic programming. In *SC*, 2008.

[53] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys ’10, pages 237–250, 2010.

[54] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *USENIX ATC*, 2013.

[55] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. Heuristics for Vector Bin Packing. *[Online] Available: http://research.microsoft.com/apps/pubs/default.aspx?id=147927*, 2011.

[56] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. SIGCOMM ’12.

[57] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the electric bill for internet-scale systems. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM ’09, pages 123–134, 2009.

[58] D. Rajan, A. Canino, J. A. Izaguirre, and D. Thain. Converting a High Performance Application to an Elastic Cloud Application. In *CLOUDCOM*, 2011.

[59] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: an I/O-efficient MapReduce. SoCC ’12, pages 13:1–13:14, 2012.

[60] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: a balanced large-scale sorting system. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI’11, 2011.

[61] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced and Energy-Efficient Large-Scale Sorting System. In *ACM Trans. Comput. Syst.*, 2013.

[62] N. Regola and J.-C. Ducom. Recommendations for Virtualization Technologies in High Performance Computing. In *CLOUDCOM*, 2010.

[63] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. SoCC ’12.

[64] E. Roloff, M. Diener, A. Carissimi, and P. O. A. Navaux. High Performance Computing in the cloud: Deployment, performance and cost efficiency. In *CloudCom*, 2012.

[65] I. Takouna, W. Dawoud, and C. Meinel. Energy efficient scheduling of hpc-jobs on virtualize clusters using host and vm dynamic configuration. *SIGOPS Oper. Syst. Rev.*, 2012.

[66] A. N. Tantawi. A scalable algorithm for placement of virtual clusters in large data centers. MASCOTS, pages 3–10. IEEE Computer Society, 2012.

[67] T. C. Team. Choco: An open source Java constraint programming library. *3rd constraint solver competition*.

[68] T. Wood, P. Shenoy, and Arun. Black-box and Gray-box Strategies for Virtual Machine Migration. NSDI’07, pages 229–242.

[69] Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *CMP-MSI*, 2008.

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

[71] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI’08, pages 29–42, 2008.

[72] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS*, 2010.