Entropy: a Consolidation Manager for Clusters

Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, Julia Lawall

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Entropy: a Consolidation Manager for Clusters

Fabien Hermenier — Xavier Lorca — Jean-Marc Menaud — Gilles Muller — Julia Lawall

N° 6639

Septembre 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6639--FR+ENG

Entropy: a Consolidation Manager for Clusters

Fabien Hermenier* , Xavier Lorca* , Jean-Marc Menaud* , Gilles Muller† , Julia Lawall‡

Thème COM — Systèmes communicants
Projet OBASCO

Rapport de recherche n° 6639 — Septembre 2008 — 23 pages

Abstract: Clusters provide powerful computing environments, but in practice much of this power goes to waste, due to the static allocation of tasks to nodes, regardless of their changing computational requirements. Consolidation is an approach that migrates tasks within a cluster as their computational requirements change, both to reduce the number of nodes that need to be active and to eliminate temporary overload situations. Previous consolidation strategies have relied on task placement heuristics that use only local optimization and typically do not take migration overhead into account. However, heuristics based on only local optimization may miss the globally optimal solution, resulting in unnecessary resource usage, and the overhead for migration may nullify the benefits of consolidation.

In this paper, we propose the Entropy resource manager for homogeneous clusters, which performs consolidation based on constraint programming and takes migration overhead into account. The use of constraint programming allows Entropy to find mappings of tasks to nodes that are better than those found by heuristics based on local optimizations, and that are frequently globally optimal in the number of nodes. Because migration overhead is taken into account, Entropy chooses migrations that can be implemented efficiently, incurring a low performance overhead.

Key-words: Virtualization, Consolidation, Cluster, Reconfiguration, Migration

* Département Informatique, École des Mines de Nantes – INRIA, LINA, CNRS – firstname.lastname@emn.fr
† École des Mines de Nantes – INRIA – Gilles.Muller@emn.fr
‡ DIKU, University of Copenhagen – julia@diku.dk

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Entropy: un Gestionnaire de Consolidation pour Grappes

Résumé : Les grappes de serveurs fournissent un environnement de calcul puissant. Cependant, une partie de cette puissance est perdue par une allocation statique des tâches sur les nœuds de calculs qui ne tient pas compte de la variations de leurs besoins. En regroupant ces tâches dynamiquement, la consolidation permet de réduire le nombre de nœuds nécessaires à l'exécution des calculs, tout en éliminant les situations de saturations temporaires. Les stratégies de consolidation actuelle se focalisent sur une optimisation locale du placement des tâches et ne tiennent pas compte de l'impact des migrations. Ces heuristiques manquent la notion d'optimalité globale qui implique une consommation de resources qui n'est pas nécessaire. De plus, l'absence de considération des migrations réduit de manière notable les performances de la grappe, limitant ainsi l'interêt de la consolidation.

Cet article présente Entropy, un gestionnaire de consolidation pour grappes homogènes utilisant une approche basée sur la programmation par contraintes et tenant compte de l'impact des migrations. Notre approche permet la réalisation d'un agencement des tâches globalement meilleur par rapport aux approches classiques à base d'heuristiques. De plus, en tenant compte des migrations des tâches sur la grappe, l'impact de la consolidation sur les performances est diminuée.

Mots-clés : Virtualisation, Consolidation, Grappe , Reconfiguration, Migration

1 Introduction

Grid and Cluster computing are increasingly used to meet the growing computational requirements of scientific applications. In this setting, a user organizes a job as a collection of tasks that each should run on a separate processing unit (*i.e*, an entire node, a CPU, or a core) [6]. To deploy the job, the user makes a request to a resource broker, specifying the number of processing units required and the associated memory requirements. If the requested CPU and memory resources are available, the job is accepted. This static strategy ensures that all jobs accepted into the cluster will have sufficient processing units and memory to complete their work. Nevertheless, it can lead to a waste of resources, as many scientific computations proceed in phases, not all of which use all of the allocated processing units at all times.

Consolidation is a well-known technique to dynamically reduce the number of nodes used within a running cluster by liberating nodes that are not needed by the current phase of the computation. Liberating nodes can allow more jobs to be accepted into the cluster, or can allow powering down unused nodes to save energy. To make consolidation transparent, regardless of the programming language, middleware, or operating system used by the application, it is convenient to host each task in a virtual machine (VM), managed by a VM Monitor (VMM) such as Xen [1], for which efficient migration techniques are available [5]. Consolidation then amounts to identifying inactive VMs that can be migrated to other nodes that have sufficient unused memory. A VM that is inactive at one point in time may, however, later become active, possibly causing the node that is hosting it to become overloaded. A consolidation strategy must thus also move VMs from overloaded nodes to underloaded ones.

Several approaches to consolidation have been proposed [3, 7, 11]. These approaches, however, have focused on how to calculate a new configuration, and have neglected the ensuing migration time. However, consolidation is only beneficial when the extra processing unit time incurred for migration is significantly less than the amount of processing unit time that consolidation makes available. While migrating a single Xen VM can be very efficient, incurring an overhead of only between 6 and 26 seconds in our measurements, it may not be possible to migrate a VM to its chosen destination immediately; instead other VMs may first have to be moved out of the way to free sufficient memory. Delaying the migration of an inactive VM only causes unnecessary node usage. On the other hand, delaying the migration of an active VM that is running on a processing unit overloaded with n other VMs degrades the performance of those VMs for a period of time by a factor of n as compared to a non-consolidated solution, in which each VM always has its own processing unit. Increasing the number of VMs that need to migrate as compared to the amount of available resources only exacerbates these problems. Thus, it is essential that consolidation be as efficient and reactive as possible.

In this paper, we propose a new approach to consolidation in a homogeneous cluster environment that takes into account both the problem of allocating the VMs to the available nodes and the problem of how to migrate the VMs to these nodes. Our consolidation manager, *Entropy*, works in two phases and is based on constraint solving [2, 14]. The first phase, based on constraints describing the set of VMs and their CPU and memory requirements, computes a placement using the minimum number of nodes and a tentative reconfiguration plan to achieve

that placement. The second phase, based on a refined set of constraints that take feasible migrations into account, tries to improve the plan, to reduce the number of migrations required. In our experiments, using the NASGrid benchmarks [6] on a cluster of 39 AMD Opteron 2.0GHz CPU uniprocessors, we find that a solution without consolidation uses 24.31 nodes per hour, consolidation based on the previously-used First Fit Decreasing (FFD) heuristic [3, 17, 18] uses 15.34 nodes per hour, and consolidation based on Entropy uses only 11.72 nodes per hour, a savings of more than 50% as compared to the static solution.

The remainder of this paper is organized as follows. Section 2 gives an overview of Entropy. Then, Section 3 describes how Entropy uses constraint programming to determine the minimum number of nodes required by a collection of VMs, and Section 4 presetns how Entropy uses constraint programming to minimize the reconfiguration plan. Finally, Section 5 evaluates Entropy using experimental results on a cluster of the Grid'5000 experimental testbed, Section 6 describes related work, and Section 7 presents our conclusions and future work.

2 System Architecture

A cluster typically consists of a single node dedicated to cluster resource management, a collection of nodes that can host user tasks, and other specialized nodes, such as file servers. Entropy is built over Xen 3.0.3 [1] and is deployed on the first two. It consists of a reconfiguration engine that runs on the node that provides cluster resource management and a set of sensors that run in Xen's Domain-0 on each node that can host user tasks, *i.e.*, VMs.

The goal of Entropy is to efficiently maintain the cluster in a *configuration*, *i.e.* a mapping of VMs to nodes, that is (i) *viable*, *i.e.* that gives every VM access to sufficient memory and every active VM access to own processing unit, and (ii) *optimal*, *i.e.* that uses the minimum number of nodes. For this, the Entropy reconfiguration engine iteratively 1) waits to be informed by the Entropy sensors that a VM has changed state, from active to inactive or vice versa, 2) tries to compute a reconfiguration plan starting from the current configuration that requires the fewest possible migrations and leaves the cluster in a viable, optimal configuration, and 3) if successful, initiates migration of the VMs, if the new configuration uses fewer nodes than the current one, or if the current configuration is not viable. The reconfiguration engine then waits 5 seconds before repeating the iteration, to accumulate new information about resource usage. In this process, the Entropy sensors periodically send requests to the HTTP interface of the Xen hypervisor on the current node to obtain the CPU usage of the local VMs, and infer state changes from this information. An Entropy sensor also receives a message from the reconfiguration engine when a VM should be migrated, and sends requests to the Xen hypervisor HTTP interface to inform it which VM should be migrated and to which node.

Previous approaches to achieving a viable, configuration have used heuristics in which a locally optimal placement is chosen for each VM according to some strategy [3, 7, 11, 17]. However, local optimization does not always lead to a globally optimal solution, and may fail to produce any solution at all. Entropy instead uses *Constraint Programming* (CP), which is able to determine a globally optimal solution, if one exists, by using a more exhaustive search, based

```
// Instantiating a new problem
Problem pb = new Problem();

// Declaration of the variables and their associated domains
IntDomainVar x = pb.makeEnumIntVar("x", 0, 10);
IntDomainVar y = pb.makeEnumIntVar("y", 0, 10);
IntDomainVar z = pb.makeEnumIntVar("z", 0, 10);

// Declaration of the constraint
IntExp exp = pb.plus(x,y);
Constraint c = pb.eq(exp, z);

// The constraint is plugged into the problem
pb.post(c);

// We start solving.
pb.solve();
```

Figure 1: Java code using the Choco library for finding values of variables x, y, and z in the range 0 to 10, such that $x + y = z$

on a depth first search. The idea of CP is to define a problem by stating constraints (logical relations) that must be satisfied by the solution. A *Constraint Satisfaction Problem* (CSP) is defined as a set of variables, a set of domains that represent the set of possible values that each variable can take on and a set of constraints that represent required relations between the values of the variables. A *solution* for a CSP is a variable assignment (a value for each variable) that simultaneously satisfies the constraints. To solve CSPs, Entropy uses the Choco library [10], which can solve a CSP where the goal is to minimize or maximize the value of a single variable. Figure 1 shows an example of Choco code, which solves the problem of finding values of variables x, y, and z in the range 0 to 10, such that $x + y = z$.

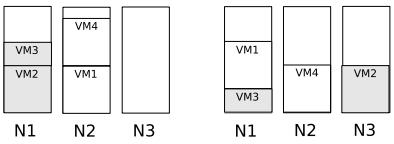
Because Choco can only solve optimization problems of a single variable, the reconfiguration algorithm proceeds in two phases. The first phase finds the minimum number n of nodes that are necessary to host all VMs. We refer to this problem as the *Virtual Machine Packing Problem* (VMPP). The second phase minimizes the reconfiguration time, given the chosen number of nodes n. We refer to this problem as the *Virtual Machine Replacement Problem* (VMRP). Solving these problems may be time-consuming. While the reconfiguration engine runs on the cluster resource management node, and thus does not compete with VMs for CPU and memory, it is important to produce a new configuration quickly to maximize the benefit of consolidation. Thus, we limit the total computation time for both problems to 1 minute, of which the first phase has at most 15 seconds, and the second phase has the remaining time. These durations are sufficient to give a nontrivial improvement in the solution, as compared to the FFD heuristic, as shown in Section 5. Furthermore, the constraint solver is implemented such that if the computation times out without the solver having found a solution that has been proved to be optimal, then the best solution found so far is returned.

3 The Virtual Machine Packing Problem

The objective of the VMPP is to determine the minimum number of nodes that can host the VMs, given their current processing unit and memory requirements. We first present several examples that illustrate the constraints on the assignment of VMs to nodes, then consider how to express the VMPP as a constraint satisfaction problem, and finally describe some optimizations that we use in implementing a solver for this problem using Choco.

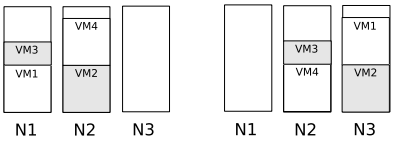
3.1 Constraints on the assignment of VMs to nodes

Each node in a cluster provides a certain amount of memory and number of processing units, and each VM requires a certain amount of memory, and, if active, a processing unit. These constraints must be satisfied by a viable configuration. For example, if every node is a uniprocessor, then the configuration in Figure 2(a) is not viable because it includes two active VMs on node N_1. On the other hand, the configuration in Figure 2(b) is viable because each VM has access to sufficient memory and each node hosts at most one active VM.



(a) Non-viable configuration (b) Viable configuration

Figure 2: Non-viable and viable configurations. $\texttt{VM}_2$ and $\texttt{VM}_3$ are active



(a) A minimal viable configuration (b) Another minimal viable configuration

Figure 3: Viable configurations. $\texttt{VM}_2$ and $\texttt{VM}_3$ are active

To achieve consolidation, we must find a viable configuration that uses the minimum number of nodes. For example, the configuration shown in Figure 2(b) is viable, but it is not minimal, because, as shown in Figure 3(a), $\texttt{VM}_2$ could be hosted on node N_2, using one fewer node. The problem of finding a minimal, viable configuration is reductible to the NP-Hard *2-Dimensional Bin Packing Problem* [15], where the dimensions correspond to the amount of memory and number of processing units.

The VMPP may have multiple solutions, as illustrated by Figures 3(a) and 3(b), which both use two nodes. These solutions, however, may not all entail the same number of migrations. For example, if we perform consolidation

with Figure 2(b) as the initial configuration, we observe that only 1 migration is necessary to reach the configuration shown in Figure 3(a) (moving $\texttt{VM}_2$ onto N_2), but 2 are necessary to reach the configuration shown in Figure 3(b) (moving $\texttt{VM}_3$ onto N_2 and $\texttt{VM}_1$ onto N_3).

3.2 Expressing the VMPP as a constraint satisfaction problem

To express the VMPP as a CSP, we consider a set of nodes $\mathcal{N}$ and a set of VMs $\mathcal{V}$. The goal is to find a viable configuration that minimizes the number of nodes used. The notation H_i, defined below, is used to describe a configuration.

Definition 3.1 *For each node $n_i \in \mathcal{N}$, the bit vector $H_i = \langle h_{i1}, \ldots, h_{ij}, \ldots, h_{ik} \rangle$ denotes the set of VMs assigned to node n_i (i.e., $h_{ij} = 1$ iff the node n_i is hosting the VM v_j).*

We express the constraints that a viable configuration must respect each VM's processing unit and memory requirements as follows. Let $\mathcal{R}_p$ be the vector of processing unit demand of each VM, $\mathcal{C}_p$ be the vector of processing unit capacity associated with each node, $\mathcal{R}_m$ be the vector of memory demand of each VM, and $\mathcal{C}_m$ be the vector of memory capacity associated with each node. Then, the following inequalities express the processing unit and memory constraints:

$$\begin{array}{ll} \mathcal{R}_p \cdot H_i \leq \mathcal{C}_p(i) & \forall n_i \in \mathcal{N} \\ \mathcal{R}_m \cdot H_i \leq \mathcal{C}_m(i) & \forall n_i \in \mathcal{N} \end{array}$$

Given these constraints, our goal is to minimize the value of the variable X, defined as follows, where the variable u_i is 1 if the node i hosts at least one VM, and 0 otherwise.

$$X = \sum_{i \in \mathcal{N}} u_i, \text{ where } u_i = \begin{cases} 1, \exists v_j \in \mathcal{V} \mid h_{ij} = 1 \\ 0, \text{otherwise} \end{cases} \quad (1)$$

We let x_{vmpp} denote this solution.

The solver dynamically evaluates the remaining free place (in terms of both processing unit and memory availability) on each node during the search for a minimum value of X. This is done by solving *Multiple Knapsack* problems using a dynamic programming approach [16].

3.3 Optimizations

In principle, the constraint solver must enumerate each possible configuration, check whether it is viable, and compare the number of nodes to the minimum found so far. In practice, this approach is unnecessarily expensive. Our implementation reduces the computation cost using a number of optimizations.

Choco incrementally checks the viability and minimality of a configuration as it is being constructed and discards a partial configuration as soon as it is found to be non-viable or to use more than the minimum number of nodes found so far. This strategy reduces the number of configurations that must be considered.

It furthermore tries to detect non-viable configurations as early as possible, by using a *first fail* approach [8] in which VMs that are active and have greater memory requirements are treated earlier than VMs with lesser requirements. This strategy reduces the chance of computing an almost complete configuration and then finding that the remaining VMs cannot be placed within the current minimum number of nodes.

In principle, the domain of the variable X is the entire set of non-negative integers. We can, however, significantly reduce the search space and improve the performance of the solver by identifying lower and upper bounds that are close to the optimal value and are easy to compute. As a lower bound, we take the number of active VMs divided by number of processing units available per node (Equation 2). If we find a solution using this number of VMs, then it is known to be optimal with no further tests. As an upper bound, we take the value computed by the First Fit Decreasing (FFD) heuristic, which has been used in other work on consolidation [3, 17, 18] (Equation 3). The FFD heuristic assigns each VM to the first node it finds satisfying the VM's processing unit and memory requirements, starting with the VMs that require the biggest amount of memory. This heuristic tends to provide a good value, in a very short time (less than a second) but the result is not guaranteed to be optimal and the heuristic may indeed not find any solution. In the latter case, the upper bound is the minimum of the number of nodes and the number of VMs.

$$X \geq min \left\lceil \frac{\sum_{v_i \in \mathcal{V}} \mathcal{R}_p(i)}{\mathcal{C}_p(j)} \right\rceil, n_j \in \mathcal{N} \tag{2}$$

$$X \leq \begin{cases} x_{ffd} \\ min(|\mathcal{N}|, |\mathcal{V}|), \text{otherwise} \end{cases} \tag{3}$$

Furthermore, we observe that some nodes or VMs may be equivalent, in terms of their processing unit and memory capacity or demand, and try to exploit this information to improve the pruning of the search tree. If the resources offered by a node n_i are not sufficient to host a VM v_i, then they are also not sufficient to host any VM v_j with the same requirements. Furthermore, the VM v_i cannot be hosted by any other node n_j with the same characteristics as n_i. These equivalences are defined as follows:

$$\forall n_i, n_j \in \mathcal{N} \mid n_i \equiv n_j \Leftrightarrow \mathcal{C}_p(i) = \mathcal{C}_p(j) \wedge \mathcal{C}_m(i) = \mathcal{C}_m(j) \tag{4}$$

$$\forall v_i, v_j \in \mathcal{V} \mid v_i \equiv v_j \Leftrightarrow \mathcal{R}_p(i) = \mathcal{R}_p(j) \wedge \mathcal{R}_m(i) = \mathcal{R}_m(j) \tag{5}$$

4 The Virtual Machine Replacement Problem

The solution to the VMPP provides the minimum number of nodes required to host the VMs. However, as illustrated in Section 3.1, for a given collection of

VMs, there can be multiple configurations that minimize the number of used nodes and the number of migrations required to reach these configurations can vary. The objective of the *Virtual Machine Replacement Problem* (VMRP) is to construct a reconfiguration plan for each possible configuration that uses the number of nodes determined by the VMPP, and to choose the one with the lowest estimated reconfiguration cost. In the rest of this section, we consider how to construct a reconfiguration plan, how to estimate its cost, and how to combine these steps into a solution for the VMRP.

4.1 Constructing a reconfiguration plan

The constraint of viability has to be taken into account both in the final configuration and also during migration. A migration is *feasible* if the destination node has a sufficient amount of free memory and, when the migrated VM is active, if the destination node has a free processing unit. However, to obtain an optimal solution it is often necessary to consider a configuration in which some migrations are not immediately feasible. We identify two kinds of constraints on migrations: *sequential constraints* and *cyclic constraints*.

A sequential constraint occurs when one migration can only begin when another one has completed. As an example, consider the migrations represented by the reconfiguration graph shown in Figure 4. A reconfiguration graph is an oriented multigraph where each edge denotes the migration of a VM between two nodes. Each edge specifies the virtual machine to migrate, the amount of memory $\mathcal{R}_m$ required to host it and its state A (active) or I (inactive). Each node denotes a node of the cluster, with its current amount of free memory $\mathcal{C}_m$ and its current free capacity for hosting active virtual machines $\mathcal{C}_p$. In the example in Figure 4, it is possible to consolidate the VMs onto only two nodes, by moving $\texttt{VM}_1$ from N_1 to N_2 and moving $\texttt{VM}_2$ from N_2 to N_3. But these migrations cannot happen in parallel, because as long as $\texttt{VM}_2$ is on N_2, it consumes all of the available memory. Thus, the migration of $\texttt{VM}_1$ from N_1 to N_2 can only begin once the migration of $\texttt{VM}_2$ from N_2 to N_3 has completed.

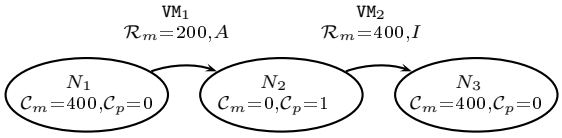


Figure 4: A sequence of migration

A cyclic constraint occurs when a set of infeasible migrations forms a cycle. An example is shown in Figure 5(a), where, due to memory constraints, $\texttt{VM}_1$ can only migrate from node N_1 to node N_2 when $\texttt{VM}_2$ has migrated from node N_2, and $\texttt{VM}_2$ can only migrate from node N_2 to node N_1 when $\texttt{VM}_1$ has migrated from node N_1. We can break such a cycle by inserting an additional migration. A *pivot* node outside the cycle is chosen to temporarily host one or more of the VMs. For example, in Figure 5(b), the cycle between $\texttt{VM}_1$ and $\texttt{VM}_1$ is broken by migrating $\texttt{VM}_1$ to the node N_3, which is used as a pivot. After breaking all cycles of infeasible migrations in this way, an order can be chosen for the migrations as in the previous example. These migrations include moving the VMs on the pivot nodes to their original destinations.

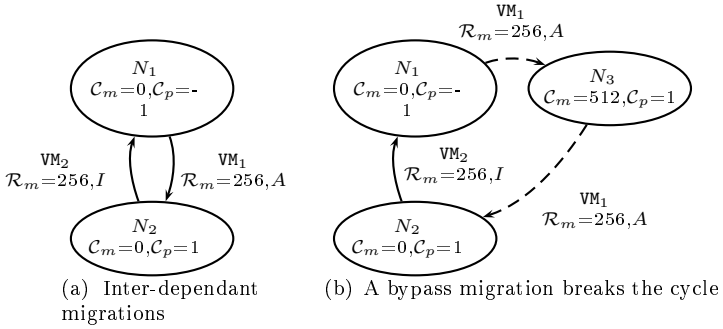


Figure 5: Cycle of non-feasible migrations

Taking the above issues into account, the algorithm for constructing a reconfiguration plan is as follows. Starting with a reconfiguration graph, the first step is to identify each cycle of infeasible migrations, identify a node in each such cycle where the VMs to migrate have the smallest total memory requirement, and select a pivot node that can accomodate these VMs' processing unit and memory requirements. The result is an extended reconfiguration graph in which for each such chosen VM, the migration from the current node to the destination node in the desired configuration is replaced by a migration to the pivot followed by a migration to the destination node. Subsequently, the goal is to try to do as many migrations in parallel as possible, so that each migration will take place with the minimum possible delay. Thus, the migration plan is composed of a sequence of *steps*, executed sequentially, where the first step consists of all of the migrations that are initially feasible, and each subsequent step consists of all of the migrations that have been made feasible by the preceding steps. As an example, Figure 6 shows a reconfiguration graph that has been extended with a migration of VM$_5$ first to node N_2 and then to node N_3 to break a cycle of infeasible migrations. From this reconfiguration graph, we obtain a three-step reconfiguration plan. The first step migrates VM$_1$, VM$_3$, VM$_4$ and VM$_5$ (to the pivot N_2). Then the second step migrates VM$_2$ and VM$_7$. Finally, the third step migrates VM$_5$ to its final destination.

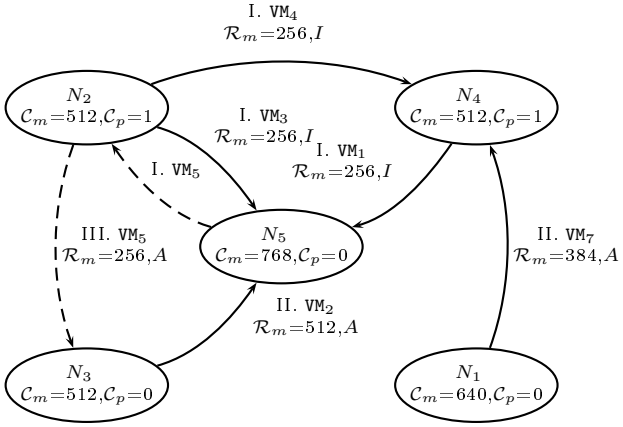


Figure 6: A reconfiguration plan

4.2 Estimating the cost of a reconfiguration plan

The cost of performing a reconfiguration includes both the overhead incurred by the migrations themselves and the degradation in performance that occurs when multiple active VMs share a processing unit, as occurs when a migration is delayed due to sequential or cyclic constraints. The latter is determined by the duration of preceding migrations. In this section, we first measure the cost and duration of a single migration, and then propose a cost model for comparing the costs of possible reconfiguration plans.

Migration cost Migrating a VM from one node to another requires some CPU and memory bandwidth on both the source and destination nodes. When there is an active VM on either the source or destination node, it will have reduced access to these resources, and thus will take longer to complete its task. In this section, we examine these costs in the context of a homogeneous cluster.

Figure 7 shows the set of possible contexts in which a migration can occur, depending on the state of the affected VMs, in the case where each node is a uniprocessor. Because a migration only has an impact on the active and migrated VMs, we ignore the presence of inactive, non-migrated VMs in this analysis. An inactive VM can move from an inactive node to a node hosting an active VM (Inactive To Active, or ITA), from a node hosting an active VM to an inactive node (Inactive From Active, or IFA), or from one node hosting an active VM to another (Inactive From Active To Active, or IFATA). Similarly, an active VM can move to an inactive node (Active To Inactive, or ATI) or to an active node (Active To Active, or ATA), although the latter is never interesting in a uniprocessor setting as a uniprocessor node should not host multiple active VMs at one time.

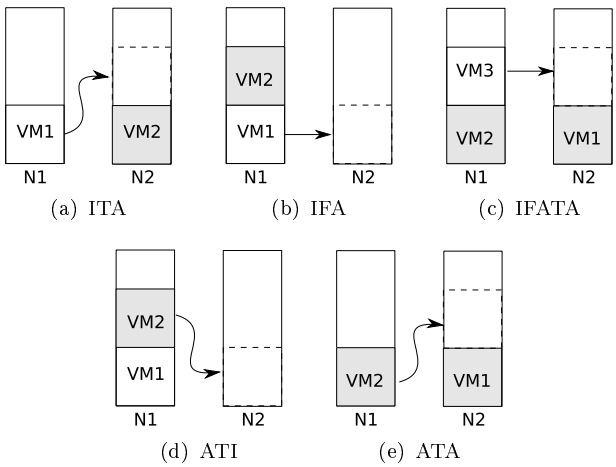


Figure 7: Different contexts for a migration. $\texttt{VM}_2$ is active

In order to evaluate the impact of a migration for each context, we measure both the duration of the migration and the performance loss on active VMs. Tests are performed on two identical nodes, each with a single AMD Opteron 2.4GHz CPU and 4Gb of RAM interconnected through a 1Gb link. We use three

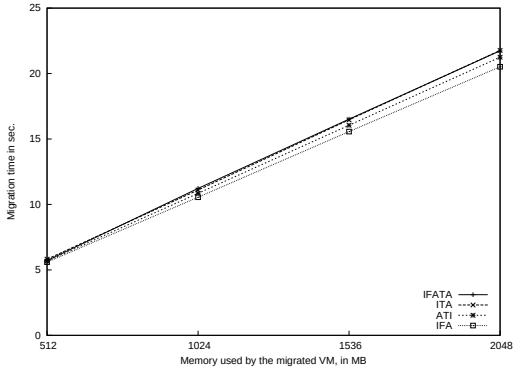


Figure 8: Duration of VM migration

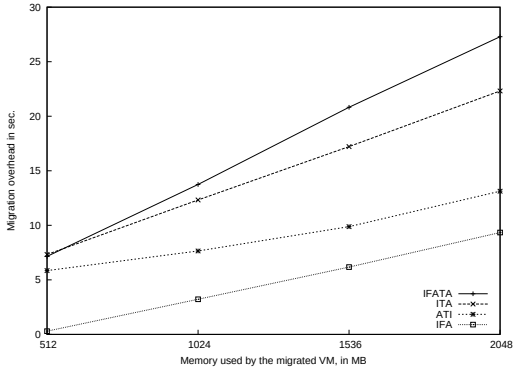


Figure 9: Impact of migration on VM performance

VMs: $\texttt{VM}_1$, which is inactive, and $\texttt{VM}_2$ and $\texttt{VM}_3$, which are active and execute a BT.W task embedded in a NASGRID ED benchmark [6]. The VMs are placed on the nodes according to the IFATA, ITA, ATI, and IFA configurations. We vary the amount of memory allocated to the migrated VM from 512 to 2048 MB. Figure 8 shows the average duration of the migration in terms of the amount of memory allocated to the migrated VM. Figure 9 shows the increase of the duration of the benchmark due to the migration of a VM using a given amount of memory.

We observe first that the duration of the migration mostly depends on the amount of memory used by the migrated VM. Second, the performance loss varies significantly according to the context of the migration. For the context IFA, the only overhead comes from reading the memory pages on node N_1, as writing the pages on the inactive node N_2 does not have any impact on an active VM. For the context ATI, it is the active VM that migrates; in this situation, the migration is a little more expensive: because Xen uses an incremental copy-on-write mechanism to migrate the memory pages of a VM [5], multiple passes are needed to recopy memory pages that are updated by the activity of the VM during the migration process. The context ITA incurs an even higher overhead, as writing the memory pages of $\texttt{VM}_1$ on node N_2 uses up most of the CPU resources on that node, which are then not available to $\texttt{VM}_2$. Finally, the

context IFATA incurs the highest overhead as the migrations act on both the source and the destination node. This overhead is comparable to the sum of the overhead of contexts IFA and ITA.

This evaluation of the cost of migrations shows that migrating a VM has an impact on both the source and destination nodes. The migration reduces the performance of co-hosted active virtual machines for a duration that depends on the context of the migration. In the worst case, the performance loss of a computational task is about the same as the duration of the migration. Although the overhead can be heavy during the migration time, the migration time is fairly short, and thus has little impact on the overall performance. Nevertheless, these numbers suggest that the number of migrations should be kept to a minimum.

Migration cost model Figures 8 and 9 show that the overhead for a single migration and the delay incurred for preceding migrations both vary principally in terms of the amount of memory allocated to the migrated VMs. Thus, we base the cost model on this quantity.

The cost function f is defined as follows. The estimated cost $f(p)$ of a reconfiguration plan p is the sum of the costs of the migrations of each migrated VM v (Equation 6). The estimated cost $f(v)$ of the migration of a VM v is the sum of the estimated costs of the preceding steps, plus the amount of memory allocated to v (Equation 7). Finally, the estimated cost $f(s)$ of a step s is equal to the largest amount of memory allocated to any VM that is migrated in step s. This estimated cost conservatively assumes that one step can only begin when all of the migrations of the previous step have completed. For the reconfiguration plan shown in Figure 6, the estimated cost of step II is 512, the estimated cost of the migration of $\texttt{VM}_2$ is 768, and the estimated cost of the whole reconfiguration plan is 4224.

$$f(p) = \sum_{v \in p} f(v) \tag{6}$$

$$f(v) = \mathcal{R}_m(v) + \sum_{s \in prevs(v)} f(s) \tag{7}$$

$$f(s) = max(\mathcal{R}_m(v)), v \in s \tag{8}$$

4.3 Implementing and optimizing the VMRP

To express the VMRP as a CSP, we again use the constraints that a configuration must be viable, as described in Section 3.2, and additionally specify that the number of nodes used in a configuration is equal to the solution of the VMPP (Equation 9):

$$\sum_{i \in \mathcal{N}} u_i = x_{vmpp} \tag{9}$$

For each configuration that satisfies these constraints, the solver constructs a reconfiguration plan p, if possible. The optimal solution k is the one that minimizes the variable K, defined as follows (Equation 10):

$$K = f(p) \tag{10}$$

Minimizing the cost of a reconfiguration provides a plan with fewer migrations and steps, and a maximum degree of parallelism, thus reducing the duration and the impact of a reconfiguration.

The lower bound for K is the sum of the cost of migrating each VM that must migrate *i.e.* when multiple active VMs are hosted on the same node. The upper bound corresponds to the cost of the reconfiguration plan p_{vmpp} associated with the configuration previously computed by VMPP:

$$(\sum_{v \in \mathcal{V}_{migrate}} \mathcal{R}_m(v)) \leq K \leq f(p_{vmpp}) \tag{11}$$

Like the VMPP, the VMRP uses equivalences to reduce the time required to find viable configurations. For the VMRP, however, the equivalence relation between VMs has to be more restrictive to take into account the impact of their migration. Indeed, migration of equivalent VMs must have the same impact on the reconfiguration process. Thus, equivalent VMs must have the same resource demands and must be hosted on the same nodes. In this situation, the equivalence relation between two VMs is formalized by Equation 12.

$$\begin{aligned} \exists v_i, v_j \in \mathcal{V} \mid v_i \equiv v_j \Leftrightarrow & \mathcal{R}_p(i) = \mathcal{R}_p(j) \wedge \\ & \mathcal{R}_m(i) = \mathcal{R}_m(j) \wedge \\ & host(v_i) = host(v_j) \end{aligned} \tag{12}$$

Entropy dynamically estimates the cost of the plan associated with the configuration being constructed based on information about the VMs that have already been assigned to a node. Then, Entropy estimates a minimum cost for the complete future reconfiguration plan. For each VM that has not yet been assigned to a node, the solver looks at VMs that can not be hosted by their current node and increases the cost with these future migrations. Finally, the solver determines whether the future configuration based on this partial assignment might improve the solution or will necessarily be worse. In the latter situation, the solver abandons the configuration currently being constructed and searches for another assignment.

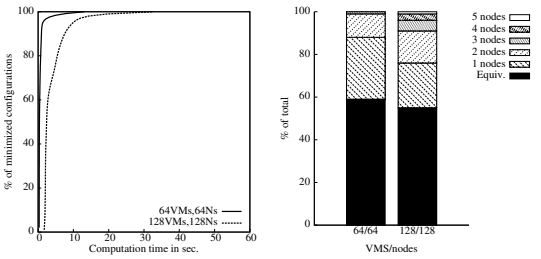
5 Evaluations

Entropy uses constraint programming in order to find a better reconfiguration plan than that found using locally optimal heuristics. Nevertheless, the more exhaustive search performed by constraint programming is only justified if it leads to a better solution within a reasonable amount of time. In this section, we first evaluate the two phases of the reconfiguration algorithm of Entropy on simulation data, to illustrate the range of benefit that Entropy can provide. We then use Entropy on a cluster in the Grid'5000 experimental testbed on a collection of programs from the NASGrid benchmark suite [6].

5.1 Evaluation of the VMPP and VMRP

The VMPP includes the number of nodes in the configuration identified by the FFD heuristic as an initial upper bound, and thus neither its solution nor that of the VMRP will ever use more nodes than the FFD solution. In this section, we measure the time required for our constraint-based reconfiguration engine to significantly reduce both the number of nodes and the cost of the reconfiguration plan, as compared to the solution proposed by the FFD heuristic, on a range of simulated data. We have used these results as the basis of the timeouts chosen in Entropy, as described in Section 2. In our evaluation, we consider solving the VMPP and the VMRP using either FFD or Entropy. The FFD solution to the VMPP is the number of nodes in the configuration chosen by the FFD heuristic, and the FFD solution to the VMRP is the minimal reconfiguration plan that produces this configuration.

We consider two classes of problem sizes, each using 64 or 128 nodes and an equal number of VMs. For each class, we have randomly generated 100 configurations with the following properties: Each VM needs zero or one processing units, depending on its state, and 1 or 2 GB of memory. Nodes each have one processing unit and 3GB of memory. The same configurations are used for evaluating the solutions of both the FFD and Entropy implementations of the VMPP and the VMRP. The dedicated node that executes the reconfiguration algorithm has an AMD Opteron 2.0GHz CPU and 2GB of RAM. The reconfiguration algorithm is implemented in Java and runs on the standard Sun Java 1.5 virtual machine.



(a) Minimization of X (b) Improvement wrt. FFD

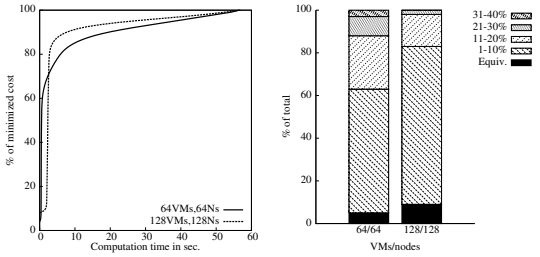
Figure 10: Properties of the solution of the VMPP for various problem sizes

Evaluation of the VMPP Figure 10(a) shows the percentage of problems in each class for which the minimum number of nodes has been determined within the given amount of time. The computation time for solving the VMPP is principally determined by the total number of VMs and nodes and by the number of equivalence classes, as identified in Section 3.3. For the two classes, the solver needs fewer than 5 seconds to compute the minimum number of nodes for 90% of the configurations.

As shown in Figure 10(b), Entropy finds a better packing by up to 5 fewer nodes for 47% of the configurations. Contrary to the heuristic that stop after the first complete assignment of the VMs, Entropy continues to compute a better solution until it times out or proves the optimality of the current one.

Evaluation of the VMRP Figure 11(a) shows the progression in finding a configuration with minimum cost, K. Because of the high cost of creating and evaluating the reconfiguration plans, the solver is never able to prove that a configuration has the smallest reconfiguration plan in the time allotted. Thus, we consider a solution to be minimal until one with a 10% lower reconfiguration cost is computed.[1] The graph denotes the percentage of solutions where the reconfiguration cost associated with the computed configuration is minimal, over time. The necessary time for computing a configuration with a minimal reconfiguration cost is principally determined by the number of VMs and nodes. After 10 seconds, 90% of the configurations with 64 nodes are minimal. Configurations with 128 nodes require a computation time of 20 seconds.

Figure 11(b) shows the effectiveness of the reduction of K by comparing the reconfiguration cost of the original solution computed by Entropy for the VMPP with the cost of the final configuration. The solution produced for the VMRP uses the same number of nodes as the solution produced for the VMPP but has a reconfiguration cost that is up to 40% lower. Entropy reduces the reconfiguration cost for 93% of the configurations.



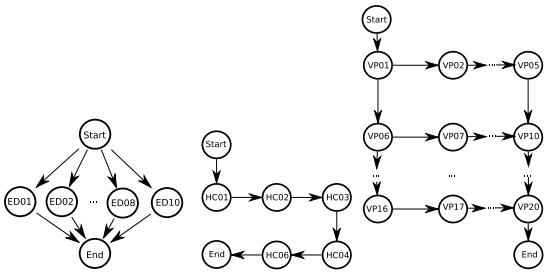
(a) Minimization of K (b) Improvement wrt. VMPP

Figure 11: Properties of the solution of the VMRP for various problem sizes

5.2 Experiments on a cluster

We now apply Entropy on a real cluster composed of 39 nodes, each with a AMD Opteron 2.0 GHz CPU and 2GB of RAM. One node is dedicated to the reconfiguration engine and three nodes are used as file servers that provide the disk images for the VMs. The remaining 35 nodes run the Xen Virtual Machine Monitor with 200MB of RAM dedicated to Xen's Domain-0. These nodes host a total of 35 VMs that run benchmarks of the NASGrid benchmark suite [6]. This benchmark suite is a collection of synthetic distributed applications designed to rate the performance and functionalities of computation grids. Each benchmark is organized as a graph of tasks where each task corresponds to a scientific computation that is executed on a single VM. Edges in the graph represent the task ordering. This ordering implies that the number of active VMs varies during the experiment; there are typically from 10 to 15 active VMs. Entropy, however, is unaware of these task graphs, instead relying on the instantaneous

[1] We use the threshold of 10% in this figure to account for the fact that the reconfiguration cost function only provides an estimate.



(a) ED (b) HC (c) VP

Figure 12: Computation graphs of NASGrid Benchmarks

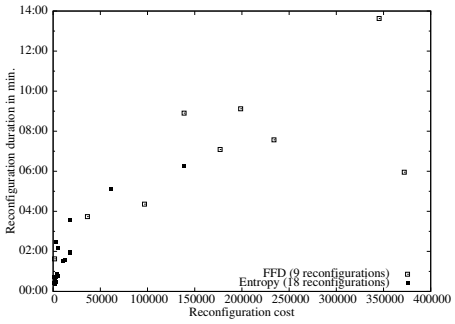


Figure 13: Reconfiguration plans computed by FFD and Entropy

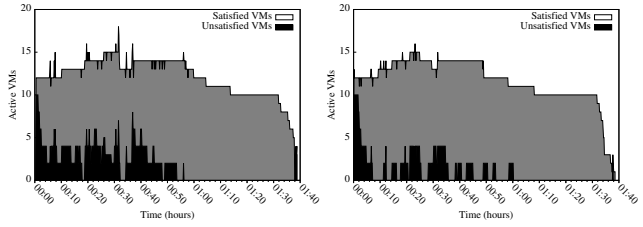
descriptions provided by the sensors to determine which VMs are active and inactive.

The 35 VMs are assigned to the various tasks of the NASGrid benchmarks ED, HC, and VP, whose computation graphs are shown in Figure 12. Each set of VMs associated with a given benchmark has its own NFS file server that contains the VMs' disk image. The ED benchmark uses 10 VMs with 512 MB of RAM each. It has one phase of computation that concerns all of its VMs. The HC benchmark uses 5 VMs with 764 MB of RAM each. This benchmark is fully sequential and has only one active task at a time. Finally, the VP benchmark uses 20 VMs, with 512MB of RAM each. This benchmark has several phases where the number of active VMs varies. Before starting the experiment, each VM is started in an inactive state, in an initial configuration computed using Entropy. This configuration uses 13 nodes and corresponds to a maximum packing. All three benchmarks are started at the same time. We test the benchmarks using FFD and Entropy as the reconfiguration algorithm.

Figure 13 shows the estimated cost of each reconfiguration plan selected using FFD and Entropy and the duration of its execution. The relationship between the cost and the execution time is roughly linear, and thus the cost function f is a reasonable indicator of performance for plans created using both FFD and Entropy. Furthermore, we observe that reconfiguration based on Entropy plans typically completes much faster than reconfiguration based on FFD plans. Indeed, the average execution time for plans computed with FFD is about 413

seconds while the average execution time for plans computed with Entropy is only 107 seconds. With short reconfiguration plans, Entropy is able to quickly react to the frequent changes in the activity of VMs, and thus quickly detects and corrects non-viable configurations. Entropy performs 18 short reconfigurations over the duration of the experiment, while the FFD-based algorithm performs 9 longer ones.

Figures 14(a) and 14(b) show the activity of VMs while running the benchmarks with FFD and Entropy, in terms of the number of active VMs that are *satisfied* and *unsatisfied*. Satisfied VMs are active VMs that have their own processing unit. Unsatisfied VMs are active VMs that share a processing unit. The average number of unsatisfied VMs is 1.75 for FFD and 1.05 for Entropy. The number of unsatisfied VMs is a significant criterion to rate the benefit of a reconfiguration algorithm. An unsatisfied VM indicates a non-viable configuration, and thus a performance loss.



(a) FFD (b) Entropy

Figure 14: Activity of VMs

When the benchmarks start, 12 VMs become active at the same time. Entropy quickly remaps the VMs and obtains a viable configuration by minute 6. FFD, on the other hand, does not reach a viable configuration until much later. The total number of active VMs increases at minute 10, thus increasing the number of unsatisfied VMs. As Entropy is not in a reconfiguration state at that time, it computes a new configuration and migrates the VMs accordingly, to obtain a viable configuration by minute 11. FFD, on the other hand, is in the midst of migrating VMs at the point of the first peak of activity, according to a previously computed, and now outdated, reconfiguration plan. FFD only reaches a viable configuration in minute 18. In this situation, we consider that an iteration of the reconfiguration process using FFD takes too much time as compared to the activity of the VMs.

The average response time of a reconfiguration process measures the average duration between detecting the presence of unsatisfied VMs and the next viable configuration. It indicates the capacity of the reconfiguration process to scale with the activity of VMs. For this experiment, the average response time for FFD is 248 seconds. For Entropy, the average response time is 142 seconds.

Figure 14(b) shows that number of unsatisfied VMs is always zero after 1:00. This is due to the unequal duration of the benchmarks. At minute 50, the benchmark HC ends its computation. Then the activity of VP changes at minutes 54 and 58 and requires a reconfiguration. For the remaining time, there is no new phase that makes unsatisfied VMs: The end of the last phase of VP

at 1:10 does not require a reconfiguration and the activity of the last running benchmark, ED, is constant.

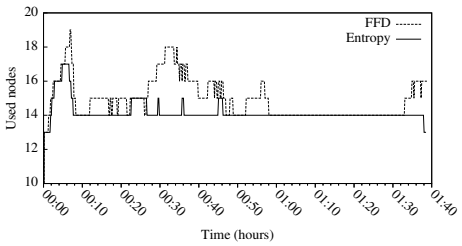


Figure 15: Number of nodes used with FFD and Entropy

Figure 15 shows the number of nodes used to host VMs. Reconfiguration plans computed with FFD require more migrations and thus tend to require more pivot nodes. For this experiment, the reconfiguration process based on FFD requires up to 4 additional pivot nodes. This situation is particularly unfortunate when consolidation is used to save energy, by powering down unused nodes, as nodes have to be turned on just to perform some migrations. Entropy, which creates smaller plans, requires at most one additional pivot nodes, and thus provides a environment favorable to the shutting down of unused nodes.

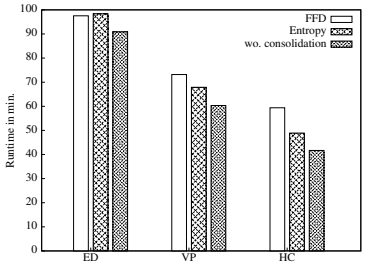


Figure 16: Runtime Comparison

By minimizing the duration of non-viable configurations, Entropy reduces the performance loss due to consolidation. Figure 16 shows the runtime of each benchmark for FFD, Entropy and for an environment without any consolidation. In the latter situation, each VM is definitively assigned to its own node to avoid performance loss due to the sharing of processing units. In this context, 35 nodes are required. The global overhead for all benchmarks compared to a execution without consolidation is 19.2% for FFD. Entropy reduces this overhead to 11.5%.

We can summarize the resource usage of the various benchmarks in terms of the number of nodes used per hour. Without any consolidation, running the benchmarks consumes 53.01 nodes per hour. Consolidation using FFD reduces this consumption to 24.53 nodes per hour. Consolidation using Entropy further reduces this consumption to 23.21 nodes per hour. However, these numbers are affected by the duration of each benchmark. When all benchmarks are running, the consolidation only comes from the reconfiguration engine that dynamically mixes inactive VMs with active VMs in the different phases of the applications. When a benchmark stops, it creates zombie VMs that still require memory

resources but should be turned off. Thus, to estimate the consumption that only results from mixing inactive and active non-zombie VMs, we consider the consumption until the end of the first benchmark to complete, HC. In this situation, running the three benchmarks without consolidation consumes 24.31 nodes per hour, with FFD consumes 15.34 nodes per hour, and with Entropy consumes only 11.72 nodes per hour.

6 Related work

Power-Aware VM replacement Nathuji *et al.* [12] present power efficient mechanisms to control and coordinate the effects of various power management policies. This includes the packing of VMs through live migration. They later extended their work to focus on the tradeoff between the Service Level Agreements of the applications embedded in the VMs and the necessity to satisfy hardware power constraints [13]. Entropy addresses the reconfiguration issues brought by the live migration of VMs in a cluster and provides a solution to pack VMs in terms of their requirements for processing units and memory, while minimizing the duration of the reconfiguration process and its impact on performance.

Verma *et al.* [17] propose an algorithm that packs VMs according to their CPU needs while minimizing the number of migrations. This algorithm is an extension of the FFD heuristic and migrates VMs located on overloaded nodes to under-exploited nodes. Restricting migrations to only those from overloaded nodes to underloaded nodes has the effect that all selected migrations are directly feasible; the sequential and cyclic constraints that we have identified in Section 4 cannot arise. Nevertheless, this implies that the approach may miss opportunities for savings, in cases where rearranging the VMs within the underloaded nodes would enable other, even more beneficial migrations. In this situation, this approach fails, potentially violating any Service Level Agreements, even if there is a possible solution. Entropy exploits a larger set of possible VM migrations by addressing sequential and cyclic constraints, and thus can be used to solve the more complex reconfiguration problems that can occur in a highly loaded environment.

Performance Management through replacement Khanna *et al.* [11] propose a reconfiguration algorithm that assigns each VM to a node in order to minimize the unused portion of resources. VMs with high resource requirements are migrated first. Bobroff *et al.* [3] base their replacement engine on a forecast service that predicts, for the next forecast interval, the resource demands of VMs, according to their history. Then the replacement algorithm, which is based on an FFD heuristic, selects a node than can host the VM during this time interval. To ensure efficiency, the forecast window takes into account the duration of the reconfiguration process. However, this assignment does not consider sequential and cyclic constraints, which impact the feasibility of the reconfiguration process and its duration.

VMs replacement issues Grit *et al.* [7] consider some VMs replacement issues for resource management policies in the context of Shirako [9], a system for on-demand leasing of shared networked resources in federated clusters. When a

migration is not directly feasible, due to sequence issues, the VM is paused using suspend-to-disk. Once the destination node is available for migration, the VM is resumed on it. Entropy only uses live migrations in order to prevent failures in the user environment due to suspending part of a distributed application.

Sandpiper [18] is a reconfiguration engine, based on an FFD heuristic, to relocate VMs from overloaded to under-utilized nodes. When a migration between two nodes is not directly feasible, the system identifies a set of VMs to swap in order to free a sufficient amount of resources on the destination node. Then the sequence of migrations is executed. This approach is able to solve simple replacement issues but requires some space for temporarily hosting VMs on either the source or the destination node. By identifying pivot nodes and bypass migrations, Entropy can resolve cycles without performing multiple swap operations that increase the number of migrations thus the duration of the reconfiguration process.

7 Conclusion and Future Work

Previous work has rejected the use of constraints in implementing consolidation as being too expensive. In this paper, we have shown that the overhead of consolidation is determined not only the time required to choose a new configuration, but also by the time required to migrate VMs to that configuration. Our constraint-programming based approach, which explicitly takes into account the cost of the migration plan, can indeed reduce the number of nodes and the migration time significantly, as compared to results obtained with the previously used FFD heuristic. We have implemented this approach in our consolidation manager Entropy, and shown that it can reduce the consumption of cluster nodes per hour for a collection of NASGrid benchmarks by over 50% as compared to static allocation and by almost 25% as compared to consolidation using FFD.

The configurations considered in this paper are fairly simple, because in the clusters available in the Grid'5000 experimental testbed, every node has only a single processor and all nodes have the same amount of memory. Our approach, however, is directly applicable to clusters providing multiprocessors and nodes with non-homogeneous memory availability, because the number of processors and the amount of memory available are simply parameters of the VMPP and VMRP problems. We will extend our results to such clusters when they become available to us.

In future work, we plan to consider the problem of admission control for clusters providing consolidation. We expect that simulation results, like those described in Section 5.1, can help to identify the number of tasks that a cluster providing consolidation can accept. We also plan to consider the applicability of the approach to other kinds of software than scientific computations, such as e-commerce.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed [4], an initiative from the French Ministry of Research through

the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners.

Availability

The prototype Entropy is available on our webpage:
http://www.emn.fr/x-info/entropy/

References

[1] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, Oct. 2003. ACM Press.

[2] F. Benhamou, N. Jussien, and B. O'Sullivan, editors. *Trends in Constraint Programming*. ISTE, London, UK, May 2007.

[3] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 119–128, May 2007.

[4] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Iréa. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, Nov. 2006.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pages 273–286, Boston, MA, USA, May 2005.

[6] M. Frumkin and R. F. V. der Wijngaart. NAS grid benchmarks: A tool for grid space exploration. *Cluster Computing*, 5(3):247–255, 2002.

[7] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual machine hosting for networked clusters: Building the foundations for "autonomic" orchestration. In *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on*, pages 1–8, Nov. 2006.

[8] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, October 1980.

[9] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing networked resources with brokered leases. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[10] N. Jussien, G. Rochart, and X. Lorca. The CHOCO constraint programming solver. In *CPAIOR'08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, Paris, France, June 2008.

[11] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 373–381, 2006.

[12] R. Nathuji and K. Schwan. VirtualPower: Coordinated power management in virtualizaed entreprise systems. In *21st Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[13] R. Nathuji and K. Schwan. VPM tokens: virtual machine-aware power budgeting in datacenters. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 119–128, New York, NY, USA, 2008. ACM.

[14] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[15] P. Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 648–662. Springer, 2004.

[16] M. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. In *Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-01)*, pages 113–124, 2001.

[17] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of HPC applications. In P. Zhou, editor, *ICS*, pages 175–184. ACM, 2008.

[18] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399