Guaranteeing High Availability Goals
for Virtual Machine Placement

Eyal Bin, Ofer Biran, Odellia Boni, Erez Hadad, Eliot K. Kolodner, Yosef Moatti
IBM Haifa Research Lab
{bin,biran,odellia,erezh,kolodner,moatti}@il.ibm.com

Dean H. Lorenz∗
lorenz.dean@gmail.com

Abstract—The placement of virtual machines (VMs) on a cluster of hosts under multiple constraints, including administrative (security, regulations) resource-oriented (capacity, energy), and QoS-oriented (performance) is a highly complex task. We define a new high-availability property for a VM; when a VM is marked as k-resilient, as long as there are up to k host failures, it should be guaranteed that it can be relocated to a non-failed host without relocating other VMs. Together with Hardware Predictive Failure Analysis and live migration, which enable VMs to be evacuated from a host before it fails, this property allows the continuous running of VMs on the cluster despite host failures. The complexity of the constraints associated with k-resiliency, which are naturally expressed by Second Order logic statements, prevented their integration into the placement computation until now. We present a novel algorithm which enables this integration by transforming the k-resiliency constraints to rules consumable by a generic Constraint Programming engine, prove that it guarantees the required resiliency and describe the implementation. We provide some preliminary results and compare our high availability support with naive solutions.

I. INTRODUCTION

With the growing use of server virtualization in IaaS/PaaS cloud infrastructures [1], the problem of optimizing the placement of virtual machines (VMs) on a cluster of hosts is gaining importance and a good solution to it is critical. Such a placement must satisfy constraints from various management domains ([2]), such as meeting the resource demands of VMs (e.g., for CPU and memory), complying with security / isolation policies (e.g., assuring that VMs of certain customers will not reside on the same host) and with compatibility / location policies (e.g., restrict the placement of certain VMs to certain hosts) and at the same time optimizing the placement towards goals such as performance (e.g., [2], [3]) and energy conservation (e.g., [4]). The ability to relocate a VM from one host to another (e.g., using live migration [5], [6], [7]) facilitates an iterative process of managing placement in a cloud where each iteration consists of computing an optimized placement of VMs on the hosts of the cluster, and then realizing the computed placement by performing relocations.

A commonly-used method for computing an optimized placement is using a Constraint Programming (CP) engine ([8], [3]), given that the placement problem is an NP-hard variant of the N-dimensional Bin-Packing problem ([2]). CP is an emerging operations research and artificial intelligence field in both theory and application ([9], [10]). In its classic definition it assigns values to variables having finite domains such that given constraints defined over the variables are satisfied. Employing CP to optimize VM placement is a convenient technique of directly defining the placement constraints over variables representing the assignments of VMs to hosts and the allocations of resources for the placed VMs at each host. Resource constraints commonly require that the sum of the allocations for placed VMs at a given host does not exceed the host capacity, for each type of resource. Anti-colocation constraints forbid placing two VMs together on the same host. Anti-location constraints disallow placing specific VMs on specific hosts. As a simple example consider anti-colocation: The variable $p_x$ denotes the placement of VM $x$, that is if VM $x$ is placed on host $h$ then $p_x = h$. So to express anti-colocation between VMs $x$ and $y$ we simply add the constraint rule $p_x \neq p_y$.

One important management domain for placement is high availability (HA) of the services provided by running VMs. In general, the HA property ensures the continued operation of applications running in VMs, despite failures of the hosts on which the VMs are running, or failures of the VMs themselves. A detailed survey of various techniques of exploiting redundancy of resources and VMs for service HA can be found in [11], discussing replication patterns such as active/active, active/passive and cold standby. In this paper, we propose a novel cold standby technique with unique properties that make it useful for HA while not affecting the VM performance when facing increased service load. Additionally, our proposed technique elegantly integrates into an existing set of constraints representing a placement problem that can be input to a CP engine. Last, the HA guarantees can be differentiated on a per-VM basis.

We base our new HA technique on the observation that Hardware Predicted Failure Analysis alerts (HwPFA) can be combined with live migration to enable the continued operation of VMs. HwPFA alerts, supported by many platforms, provide an advanced warning of expected hardware failures [12], [13]. Using our approach an HwPFA alert triggers the cluster management system to move the VMs on a failing host to other hosts. If there is enough time, this can be done by live relocation, which enables the continuous operation of the applications on the VMs. Otherwise, cold relocation is possible, which will start the VM on another host with a

∗Work done while at IBM Haifa Research Lab.
small interruption. Thus, our approach offers an HA property at the VM level (termed resiliency) without maintaining actual replicas of the VM, which results in better utilization of the physical environment. Using HwPFA and live relocation helps to significantly reduce the overall downtime typically incurred by host failures for cold-standby techniques (e.g., [14]).

Our technique features a novel combination of three important placement properties. First, it precomputes a backup location for each resilient VM, such that when the VM’s host fails, the VM can be relocated to a non-failed host without relocating any other VM. We call this one-step evacuation. Second, our technique allows spreading the backup locations for resilient VMs across the hosts, such that the backup space can be used as extra resource space by the running VMs. Thus, the backup space, when not used for failures, can be used by running VMs to accommodate extra load. Third, our technique overlaps the backup locations while at the same time guaranteeing that each resilient VM can be assigned an exclusive backup location for each failure scenario. This allows significantly increased resource utilization compared to a naive approach that dedicates multiple distinct locations to a resilient VM in order to withstand multiple host failures.

More precisely, we define a k-resilient VM as a VM that should be relocated to an alternative running host when its current host fails and there are up to \((k - 1)\) additional host failures. As providing the resiliency guarantees is expensive in terms of resource reservation, in real scenarios the resiliency level of a VM will vary according to its importance. In particular, a VM specified as 0-resilient does not enjoy any HA guarantees. Furthermore, we define a given placement as resilient if it satisfies the resiliency requirements of all its VMs: for any subset of the hosts of size \(k\) that fails, all of the \(k\)-resilient VMs on the failed hosts can be relocated to other non-failed hosts, without relocating any VMs already running on the non-failed hosts. In addition, all of the other placement constraints must still hold.

\(k\)-resiliency constraints are naturally expressed over subsets. In particular, “find a placement \(P\) such that for any subset \(S\) of up to \(k\) hosts, there exists a valid placement \(P'\) that is identical to \(P\) except for the VMs on hosts in \(S\) with \(k\) or higher resiliency level, that on \(P'\) are placed on hosts NOT in \(S\)”. Notice that the above constraint is a Second Order logic statement with universal - existential quantification over sets. Thus, solving \(k\)-resiliency requires solving an exponential number of sub-problems even when considering only the number of host subsets of size \(k\) out of \(n\) hosts - \(\binom{n}{k}\). Hence, a direct implementation of a \(k\)-resiliency problem as CP constraints would be too large for a generic engine to produce a solution in a timely fashion.

Therefore, another most important and novel aspect of our paper is that it shows how to transform a complex placement problem with \(k\)-resiliency constraints into a simpler constraint satisfaction problem that can be input to a generic engine and produce results in reasonable time. We provide an algorithm for this transformation, and thus provide a practical way for guaranteeing high availability goals for the placement of virtual machines using a CP engine. We are not aware of other solutions for this problem, except for designating \(k\) hosts as dedicated stand-bys, devoid of running VMs. This host redundancy technique is known also as \(N+M\) clustering ([15]).

A drawback of this standby technique is that it does not allow the resources of the standby hosts to be exploited when there are no failures. Also, with anti-location constraints it is not always trivial to find a standby solution, if it at all exists.

The rest of this paper is organized as follows: in Section II we provide an intuitive high level description of our solution and its techniques; in Section III we give a formal definition of the resilient placement problem; in Section IV we give the detailed algorithm and correctness proof; in Section V we present simulation results and discuss their meaning; in Section VI we survey related works; and in Section VII we give a summary and conclusions.

II. OUR SOLUTION IN A HIGH LEVEL

Our solution is based on creating a transformed placement problem that includes shadow virtual machines, and the solution to this problem (that can be solved by a CP engine) determines the resilient placement.

Intuitively, shadows are place-holders for VMs; that is, a VM can be evacuated to a location of its shadow if its current host fails. The constraints on the shadow placement guarantee that the VMs may actually relocate to their shadow locations, without violating resource constraints or other placement constraints. Multiple shadows provide resiliency to multiple host failures. Thus, in order to make a VM resilient to up to \(k\) host failures, at least \(k\) shadows of the VM need to be placed (in addition to the VM itself). As a further restriction, these shadows must not share the same host with one another or with their VM; in other words, all the shadows of a VM and the VM itself are anti-colocated. This is required to guarantee that at least one shadow will survive on a working host following up to \(k\) host failures.

The challenge is to construct the right shadow placement constraints. For example, a trivial solution that meets all the resiliency requirements and feasibility constraints is treating all shadow VMs as regular VMs. However, this trivial approach would make the overall resource requirements for the solution up to \((k + 1)\) times larger than the solution without High Availability properties. Such high resource requirements severely limit the solution space and thus reduce the chance of the engine finding a suitable placement solution. Moreover, the additional location constraints (e.g., anti-colocation) must be applied to every combination of shadows and VMs, further limiting the solution space. The extra resource space consumed by shadows can be considerably reduced through the realization that unlike actual VMs, shadow machines can overlap, that is, share the same resources, if it can be guaranteed that, regardless of which hosts fail, their original VMs will never be evacuated together to occupy those shadows’ spaces.

In order to enable such shadow overlaps, our algorithm employs an intelligent scheme of numbering shadows and failures, in a way that clearly identifies the possible overlaps.
of actual VM evacuations. In particular, each failing host is assigned a unique index (1 through \( k \)) and each shadow of a VM is assigned a unique index. Upon failure of a host indexed \( i \) the VMs on that host are evacuated to the location of their \( i \)-th shadow.\(^1\) Consequently, we can define focused placement constraints that apply only to specifically numbered shadows and VMs that may overlap following host failures. For example, VMs that are placed on different hosts cannot be evacuated together to shadows with the same index (as each host would be assigned a different failure index); therefore, their shadows with same index can overlap. An extreme example of this overlap is the the fall-back to a standby solution that our shadow numbering scheme enables.

III. The Problem

In this section we provide a formal definition of our problem. First we define the (non-resilient) Virtual Machine Placement Problem VMP, which satisfies various placement constraints (e.g., resource feasibility); however, does not provide high availability properties. Then, we add resiliency constraints to derive the Resilient VM Placement problem, Problem RVP. A feasible solution to Problem RVP provides a placement that satisfies individual VM high-availability goals, without compromising the “standard” placement constraints, given by Problem VMP.

The rest of this section is organized as follows. First we introduce some notations and formally define Problem VMP. Next, we formally define the availability constraints; namely, the properties that make a placement resilient to host failures. Finally, we give the full formal definition of Problem RVP, which we solve in the next section.

A. Preliminaries and Notations

Let \( V \) denote the set of VMs, let \( H \) denote the set of hosts, and let \( Q \) denote the set of resource types. A placement \( \mathcal{P}(V, H) = \{p_v\}_{v \in V} \) is a mapping of the VM set \( V \) on to the host set \( H \), where \( p_v \in H \), and \( p_v = h \) if and only if \( v \) is placed on \( h \).

We say that a placement, \( \mathcal{P} \), is feasible if it satisfies the resource constraints as well as the (optional) location constraints, as follows.

Resource constraints The amount of resources consumed by all the VMs that are placed on a host must not exceed the host’s capacity. Formally, given resource requirements for all VMs, \( \{r_v\}_{v \in V, q \in Q} \)\(^2\) and resource capacities for all hosts, \( \{c_h\}_{h \in H, q \in Q} \), \( \mathcal{P} \) must satisfy

\[
\sum_{v \in V, s.t. p_v = h} r_{vq} \leq c_{hq}, \quad \forall h, q \in Q. \tag{1}
\]

Anti-location constraints An anti-location constraint requires that a VM \( v \) is never located on a host \( h \). When such a constraint exist, we say that \( v \) and \( h \) are anti-located. Formally, given a group of anti-location host-VM pairs \( AL \equiv \{(h_i, v_i)\}_{i=1,2,\ldots} \), the placement \( \mathcal{P} \) must satisfy

\[
p_v \neq h \quad \forall (h, v) \in AL. \tag{2}
\]

Anti-colocation constraints An anti-colocation constraint requires that two VMs, \( v \) and \( u \), are never located on the same host. When such a constraint exists, we say that VMs \( v \) and \( u \) are anti-colocated. Formally, given a group of anti-colocation VM pairs \( ACL \equiv \{(u_i, v_i)\}_{i=1,2,\ldots} \), the placement \( \mathcal{P} \) must satisfy

\[
p_u \neq p_v \quad \forall (u, v) \in ACL. \tag{3}
\]

The above constraints formally define the (non-resilient) VM placement problem, Problem VMP, as shown in Figure 1. A solution to Problem VMP can be found through several methods, as described in previous work. For example, it is straightforward to describe these constraints to a CP engine, as they translate into a First Order logic satisfaction problem. On the other hand, a solution to Problem VMP does not guarantee any resiliency properties.

B. Availability constraints

We now add constraints to achieve a placement that is not only feasible, but also resilient to host failures. Each VM is associated with a resiliency level that defines the number of host failures that the VM must be able to overcome. A VM can overcome a host failure if, upon detection of imminent failure of its current host, it can be relocated (via live or cold VM migration) to another, suitable, non-failed host. If the number of host failures exceeds the resiliency level of a VM then the existence of a suitable evacuation target for the VM is not

\footnotetext[1]{Our full solution, as described later, also takes into account the situation in which a VM cannot be evacuated to its \( i \)-th shadow, because the shadow is also placed on a failed host.}

\footnotetext[2]{We assume that each resource type is required independently; namely, a VM \( v \in V \) will always require \( r_{vq} \) of resource \( q \), regardless of the current load on any resource. We also assume that the requirements are linearly additive; namely if virtual machine \( v, u \in V \) require \( r_{vq} \) and \( r_{uq} \) of resource \( q \), then both of them together require \( r_{vq} + r_{uq} \), placed on the same host.}
guaranteed. We say that a placement is resilient if it can meet the availability levels of all VMs.

Note that any placement that is the result of such VM evacuation, regardless of which hosts fail, must still be feasible, as defined by Problem VMP. Furthermore, we do not allow reshuffling; namely, only VMs that were placed on the failed host may be evacuated to a non-failed host; all other VMs must stay where they are (this is critical for a quick response to HwPFA or returning to running state in actual failures).

Next, we formally define resiliency.
Let \( L = \{ l_v \}_{v \in V} \) denote the resiliency levels of all VMs, where \( 0 \leq l_v < |H| \) for all \( v \in V \), and \( l_v = 0 \) means that \( v \) is not guaranteed to overcome even a single failure. A VM \( v \in V \) must be \( l_v \)-resilient; namely, able to overcome up to \( l_v \) host failures through migration. This means that, for every resilient VM, there must exist a suitable alternative host that can become an evacuation target for the VM, if its current host fails. To that end, we define an evacuation function \( \text{evac}(\cdot) : V \rightarrow H \) that provides an evacuation target for all VMs; namely, a machine \( v \) that is running on host \( f \) is moved to host \( \text{evac}(v) \) upon failure of \( f \). We say the \( \text{evac}(\cdot) \) is a proper evacuation function, if

\[
\text{evac}(v) \neq p_v \quad \text{for every} \quad v \in V, \quad \text{such that}, \quad l_v > 0. \tag{4}
\]

The evacuation function implies a new placement upon each host failure; that is, for a particular placement \( P(V, H) \), and a particular failed host \( f \), the \( \text{evac}(\cdot) \) function induces a new placement \( P(V, H \setminus \{ f \}) \) that is the outcome of the evacuation. With slight abuse of notations we write \( P(V, H \setminus \{ f \}) \leftarrow \text{evac}(P(V, H), f) \), where

\[
p_v^f = \begin{cases} 
\text{evac}(v) & \text{if} \quad p_v = f, \\
p_v & \text{otherwise}. \tag{5}
\end{cases}
\]

In order for a placement \( P(V, H) \) to be resilient, the induced new placement must provide a feasible placement for all VMs with resiliency requirements (i.e., with \( l_v > 0 \)). In other words, there must exist a proper evacuation function \( \text{evac}(\cdot) \), such that, for every \( f \in H \), \( \text{evac}(P(V, H), f) \) is feasible. Furthermore, since there may be additional host failures, the new, induced, placement must also be resilient (with all VM resiliency levels decreased by 1).

We now extend our evacuation process to support multiple host failures. A resilient placement must provide not only an initial feasible placement and an initial evacuation function, but also a method to update the evacuation function after each host failure. Let \( F = \{ f_1, f_2, \ldots, f_k \} \subseteq H \) denote the set of failed hosts, where \( f_1 \) denotes the first host to fail, \( f_2 \) denotes the next failed host, and so on, until \( f_k \), where \( k = \max_{v \in V} (l_v) \). For every \( F \subset H, |F| \leq k \), our evacuation process must provide a corresponding set of evacuation functions \( \{ \text{evac}(\cdot) : V \rightarrow H \setminus \{ f_1, f_2, \ldots, f_i \} \}_{0 \leq i < k} \), where \( \text{evac}(0) \) is the initial evacuation function, \( \text{evac}(i) \) is the updated evacuation function, after host \( f_1 \) fails, and so on.

Denoting the initial placement by \( P^0(V, H) \), these evacuation functions induce (for a particular failure set \( F \)) a sequence of intermediate placements \( \{ P^0, P^1, \ldots, P^k \} \), as defined by the following transitions

\[
\begin{align*}
&P^0(V, H), \text{evac}^0() \\
&\xrightarrow{f_1} P^1(V, H \setminus \{ f_1 \}), \text{evac}^1() \\
&\xrightarrow{f_2} P^2(V, H \setminus \{ f_1, f_2 \}), \text{evac}^2() \\
&\xrightarrow{f_3} \cdots \\
&\xrightarrow{f_k} P^k(V, H \setminus F), \text{nil}.
\end{align*}
\tag{6}
\]

Each host failure induces a new placement through the transition \( P_i \rightarrow P_{i+1} = \text{evac}(P_i, f_{i+1}) \); namely

\[
p_{v}^{i+1} = \begin{cases} 
\text{evac}(v) & \text{if} \quad p_v^i = f_{i+1}, \\
p_v^i & \text{otherwise}. \tag{7}
\end{cases}
\]

Such a solution is feasible only if each intermediate placement, \( P_i \), is feasible for every \( 0 \leq i \leq k \). Each \( P_i \) must provide placement (on non-failed hosts) for all VMs that did not yet exceed their resiliency level; namely,

\[
p_v^i \notin \{ f_1, f_2, \ldots, f_i \} \quad \forall v \in V, \quad \text{s.t.,} \quad l_v > i.
\]

Put differently, each evacuation function, \( \text{evac}(v) \), must be proper w.r.t. \( i \); namely,

\[
\text{evac}(v) \notin \{ p_v^i \} \cup \{ f_1, f_2, \ldots, f_{i+1} \} \quad \forall v \in V, \quad \text{s.t.,} \quad l_v > i. \tag{8}
\]

C. Formal problem definition.

We can now formally define Problem RVP, as shown in Figure 2. Problem RVP is much harder to solve than the non-resilient Problem VMP. Constraints that are expressed as “for every sequence of host failures” introduce Second Order logic statements with universal - existential quantification over
sets which makes it practically impossible for a CP engine to handle as is. In the following section, we utilize the concept of shadow machines to transform Problem RVP into a much simpler problem, Problem VSP, that is very similar to Problem VMP and hence practically solvable.

IV. OUR SOLUTION

In this section we present a solution to Problem RVP. The main difficulty in solving Problem RVP is the Second Order logic constraint “for every sequence of host failures...”. In order to overcome this difficulty we transform constraints of Problem RVP into simpler constraints that are independent of the failure sequence.

As described earlier, the core of our solution is adding shadow VMs to the placement computation. The idea is that the initial placement of the shadow machines determines the evacuation functions (and eventual placement) for every possible host failure sequence. Thus, the complex high order constraints of Problem RVP can be transformed into lower order constraints on the initial placement of the VMs and their shadows.

Our solution can be described through the following steps

Our Solution.

Given An instance of Problem RVP on \( H \) and \( V \)

1) Add shadow machines to \( V \) to create a larger set of machines \( V^* = V \cup \{ \text{shadow machines} \} \)
2) Create an instance of Problem SVP on \( H, V^* \) by transforming the resiliency, resource, and location constraints of Problem RVP into new resource and location constraints (without resiliency)
3) Find a solution to the created instance of the non-resilient shadow placement Problem SVP (e.g., using a CP engine)
4) From the solution of Problem SVP construct a solution to the original Problem RVP

The rest of this section is organized as follows. First, we formally define shadows and the method by which the evacuation functions are derived from their placement (i.e., the method of using a shadow placement solution to construct a solution of Problem RVP). Then, we develop the transformed feasibility constraints and define the transformed Problem SVP. Finally, we provide a correctness proof.

A. Shadows and evacuation

Our algorithm employs an intelligent scheme of numbering shadows and failures, in a way that clearly identifies the possible overlaps of actual VM evacuations. Consequently, we can define focused placement constraints that apply only to specifically numbered shadows and VMs that may overlap following host failures.

For each VM \( v \in V \), we create \( l_v \) shadow machines and denote these by \( v^1, v^2, \ldots, v^{l_v} \); we also use \( v^0 \) to indicate the actual VM \( v \). Let \( V^* \) denote the set of all VMs and their shadows; namely

\[
V^* \equiv \{ v^i \}_{v \in V, 0 \leq i \leq l_v}. \tag{9}
\]

We produce a placement of all VMs and all their shadows,

\[
p^0(V^*, H) = \{ p^{0,v}_v \}_{v \in V, 0 \leq i \leq l_v};
\]

where \( p^{0,v}_v = h \) if and only if machine \( v \) is placed on host \( h \) and \( p^{0,v}_v = h \) if and only if shadow \( v^i \) is placed on host \( h \). In the following, when clear from context, we omit the superscript 0, and use \( p_v \) instead of \( p^{0,v}_v \).

The evacuation functions are determined by the initial placement of the shadow machines. A VM that resides on a failed host (i.e., \( p_v = f^j \in F \)) is evacuated to the location of its shadow \( v^i \); namely, \( \text{evac}^{-1}(v) = p_v \). It is possible that \( v^i \) also resides on a failed host (i.e., \( p_{v^i} = f^j \in F \)). In this case, \( v \) is evacuated to the location of \( v^j \); namely \( \text{evac}^{-1}(v) = p_{v^i} \).

We continue this process, until we find a shadow of \( v \) that is located on a non-failed host. More formally, \( \text{evac}^{-1}(v) \) is defined through the following procedure:

\[
\text{evac}^{-1}(v)
\]

1) if \( l_v < i \)
2) then return \( p^{i-1}_{v} \)
3) result \( \leftarrow p_{v^0} \)
4) while result \( \in \{ f^1, f^2, \ldots, f^l \} \)
5) do result \( \leftarrow p_{v^i} \)
6) return result

The procedure starts from the initial location of \( v \) and systematically examines all the locations of its shadows, until it finds a non-failed host. The order in which these locations are examined is derived from the failure order; specifically, the failure number of each host. Note that the only dependency on \( i \) is through the current subset of failed hosts. In particular, \( \text{evac}^{-1}(v) \) is called only if \( v \) is currently on \( f^i \) (i.e., \( p_{v^i} = f^j \)) and \( p_{v^i} \) is the first host examined. An example of shadow placement and the evacuation process is shown in Figure 3.

The above shadow numbering scheme provides step 1 of our solution. The evacuation procedure and Equation (6) provide step 4 of our solution; namely a method to convert a placement of shadows and VMs into a solution of Problem RVP. In the next section we develop placement constraints on the numbered shadows and VMs (step 2 of our solution) that ensure feasibility of the converted solution.

B. Constructing the feasibility constraints

We now derive the shadow and VM placement constraints that ensure both proper evacuation functions (as defined by Equation (8)) and feasibility (as defined by Equations (1), (2), and (3)). As explained above, all these constraints must hold for the initial placement and evacuation functions, as well as for any intermediate placement and evacuation function that is created following host failures.

Proper evacuation functions. As defined by (8), \( \text{evac}^i() \) is a proper evacuation function if it evacuates any \( v \in V \) with \( l_v > i \) to an alternate non-failed host. We must provide conditions that would ensure that Procedure \( \text{evac}^i() \) always terminates and that it returns a non-failed host if \( l_v > i \).
The high level idea is that there are \( l_v \) shadows for each VM, \( v \); therefore, if the initial placement places all shadows of \( v \) on different hosts (i.e., \( v \) and all its shadows are anti-collocated) then \( v \) can always be evacuated to a non-failed host if there are no more than \( l_v \) failures. Formally, the anti-colocation constraints on the initial placement to ensure proper evacuation functions are

\[
p_{v,i} \neq p_v^i \quad \forall 0 \leq i < j \leq l_v, v \in V. \tag{10}\]

**Anti-location constraints** An anti-location constraint requires that a VMs \( v \) is never located on a host \( h \) if \((h, v) \in AL\); namely,

\[
p_{v,i}^j \neq h \quad \forall (h, v) \in AL, 0 \leq i \leq l_v.
\]

Since a VM may be evacuated to the location of any of its shadows, we must add anti-location constraints between \( h \) and any of \( v \)'s shadows. Formally, the transformed anti-location constraints are given by

\[
p_{v,i} \neq h \quad \forall (h, v) \in AL, 0 \leq i \leq l_v. \tag{11}
\]

**Anti-colocation constraints** An anti-colocation constraint requires that two VMs \((u, v) \in ACL\) are never collocated on the same host; namely

\[
p_{v,i}^j \neq p_u^j \quad \forall (u, v) \in ACL, 0 \leq i \leq \min\{l_u, l_v\}. \tag{12}
\]

An obvious constraint would be to require that \( v \) and any shadow of \( v \) is anti-collocated with \( u \) and any shadow of \( u \); however, this constraint is very restrictive as it requires at least \( 2 + l_v + l_u \) hosts. We overcome this by utilizing the shadow numbering. The idea is that, since \( u \) and \( v \) are anti-collocated, their hosts cannot have the same failure number, hence their matching shadows do not have to be anti-collocated. The transformed anti-colocation constraint states that \( v \) and \( u \) are initially anti-collocated, and all non-matching shadows of \( v \) and \( u \) are initially anti-collocated with each other and with the virtual machines. Formally, it is given by

\[
\forall (u, v) \in ACL,
\]

\[
p_{v,0}^0 \neq p_u^0 \quad \text{and} \quad p_{v,i} \neq p_{u,j} \quad \forall 0 \leq i \neq j, 0 \leq i \leq l_u, 0 \leq j \leq l_v.
\]

**Resource constraints** The resource constraints are the most difficult to transform and have a different form than the resource constraints of Problem VMP. The amount of resources consumed by all the VMs that are placed on a host must not exceed the host’s capacity throughout the evacuation process; namely

\[
\forall 0 \leq i \leq k, h \in H \setminus \{f^1, f^2, \ldots, f^i\}, q \in Q,
\]

\[
\sum_{v \in V, \text{s.t. } p_v^i = h} r_{vq} \leq c_{hq}.
\]

This implies that the computation for each host must include the resources consumed by all VMs that are placed on \( h \), as well as any VM that may be evacuated to \( h \). Consider a shadow \( v^i \) that resides on a host \( h \); namely \( p_{v,i} = h \). The VM \( v \) may be evacuated to \( h \) from the \( i \)-th failed host \( f^i \), in several cases. One case is when \( f^i \) is the initial placement of \( v \) (i.e., \( p_{v,0} = f^i \)); in this case, \( v \) would directly be evacuated to \( h \) upon the failure of \( f^i \). A second case is when another shadow of \( v, v^j \), is placed on \( f^j \) (i.e., \( p_{v,j} = f^j \)) and \( v \) resides on some (other) failed host \( f^j, j < i \); in this case, the evacuation procedure would first evacuate \( v \) to \( f^i \) (through a call to \( \text{evac}^j(v) \)) and then evacuated \( v \) to \( h \) upon failure of \( f^i \) (through a call to \( \text{evac}^i(v) \)). A third case is when another shadow of \( v, v^j \), is placed on \( f^i \) (i.e., \( p_{v,j} = f^i \)) and \( v \)

---

**Fig. 3.** An example of shadow placement and the evacuation process.
resides on some (other) failed host \(f^j, j > i\); in this case, the evacuation process would attempt to evacuate \(v\) to \(f^i\) upon failure of \(f^j\); however, since \(f^i\) is a failed host, it will eventually evacuate \(v\) to \(h\).

It is clear from this analysis that a VM with any shadow \(v^j, j \neq i\) on \(f^i\) might be evacuated to \(h\). Let \(E(n \rightarrow h)\) denote the set of VMs that may be evacuated from \(n\) to \(h\) if \(n\) fails and is marked as \(f^i\). That is

\[
E(n \rightarrow h) = \{v \in V \text{ s.t., } h = p_{v^i} \text{ and } n = p_{v^j} , \text{ for some } j \neq i, 0 \leq j \leq l_v.\}
\]  

(13)

Using \(E(n \rightarrow h)\), we can derive an upper bounds on the amount of resources of type \(q\) that is required to support all VMs that might be placed on a host \(h\) as result of the evacuation process. Equation (14) provides an initial resource feasibility constraint.

\[
c_{hq} \geq \sum_{v \in V; s.t. \ p_v = h} r_{vq} + \sum_{1 \leq t \leq k} \max_{n \in H; n \neq h} \sum_{u \in V; u \in E(n \rightarrow h)} r_{uq}. \quad (14)
\]

Another, simpler, upper bound is implied by the fact that all the VMs in \(E(n \rightarrow h)\) might reside, at some point, on \(n\) (e.g., if \(n\) does not fail, and some other host is marked as \(f^i\)). Therefore, the resource requirement of all the VMs in \(E(n \rightarrow h)\) cannot exceed the total capacity of \(n\); namely

\[
\sum_{u \in V; u \in E(n \rightarrow h)} r_{uq} \leq c_{hq}.
\]

This is used to derive an improved resource feasibility constraint,

\[
c_{hq} \geq \sum_{v \in V; s.t. \ p_v = h} r_{vq} + \sum_{1 \leq t \leq k} \min\{c_{nq}, \max_{u \in V; u \in E(n \rightarrow h)} r_{uq}\}. \quad (15)
\]

We proceed with a formal definition of the transformed shadow placement problem.

C. The transformed problem

We now summarize the discussion of the previous sections. Given an instance of Problem RVP, we transform the constraints on \(H, V\) into constraints on \(H, V^*\), as follows.

**Resource constraints** Using (15), the transformed resource constraints are

\[
c_{hq} \geq \sum_{v \in V; s.t. \ p_v = h} r_{vq} + \sum_{1 \leq t \leq k} \min\{c_{nq}, \sum_{u \in V; u \in E(n \rightarrow h)} r_{uq}\} \quad \forall h \in H, q \in Q. \quad (16)
\]

**Anti-location constraints** Using (11), the transformed anti-location constraints are

\[
p_{v^i} \neq h \quad \forall (h, v^i) \in AL^i \subseteq H \times V^* \quad (17)
\]

where \(AL^i\) is given by

\[
(h, v^i) \in AL^i \quad \forall (h, v) \in AL, 0 \leq i \leq l_v.
\]

**Anti-colocation constraints** Using (10) and (12), the transformed anti-colocation constraints are

\[
p_{a^i} \neq p_{a^j} \quad \forall (u^i, v^j) \in ACL' \subseteq V^* \times V^*, \quad (18)
\]

where \(ACL'\) is given by

\[
(v^i, v^j) \in ACL' \quad \forall 0 \leq j < i \leq l_v, v \in V \quad \text{ and }
\]

\[
(u^i, v^j) \in ACL' \quad \forall (u, v) \in ACL \quad \text{ and } \quad (u^i, v^j) \in ACL' \quad \forall i \neq j, 0 \leq i \leq l_u, 0 \leq j \leq l_v, \quad (u, v) \in ACL.
\]

These transformed constraints are used to construct a new, simplified Problem SVP as shown in Figure 4.

**Problem SVP (Shadow and VM Placement).**

**Given** the following input:

1) A set of VMs and shadows \(V^*\), as defined by (9), and resource requirements \(\{r_{vq}\}_{v \in V, q \in Q}\).
2) A set of hosts \(H\) and resource capacities \(\{c_{hq}\}_{h \in H, q \in Q}\).
3) Transformed anti-location constraints \(AL' \subseteq H \times V^*\).
4) Transformed anti-colocation constraints \(ACL' \subseteq V^* \times V^*\).

**Find** a feasible placement, \(P^*(V^*, H)\), of all VMs and all their shadows, as defined by Equations (16), (17), and (18).

![Fig. 4. Problem SVP (Shadow and VM Placement).](image)

Problem SVP is very similar to Problem VMP; hence readily solvable by existing methods, such as utilizing a CP engine. The difference between the problems is that Problem SVP is defined over a much larger VM population (consisting of all VMs and all their shadows) and that the resource constraints of Problem SVP are much more complex. Note that these resource constraints are still very conservative (i.e., loose). Finding tighter resource bounds is left for future work.

D. Correctness proofs

In this section we formally prove the correctness of our solution.

Lemma 1 provides a correctness proof of (10).

**Lemma 1.** If \(p_{v^i} \neq p_{v^j}\) for every \(0 \leq j < i \leq l_v\) and \(i < l_v\), then \(\text{evac}^i(v)\) always terminates and returns a non-failed host.

**Proof:** Clearly, since \(i < l_v\), the procedure enters the while loop; furthermore, by the condition of the while loop, if the procedure returns then the result is a non-failed host. We must show that it is not an infinite loop. Consider the sequence of host \((h_1, h_2, \ldots)\) that are examined by the procedure. Each host in this sequence is an initial location of \(v\) or one of its shadows, as follows. \(h_1 = p_{v^i}; h_1 = f^j_i\), for some \(f^j_i \in F\) then \(h_2 = p_{v^j_i}; \) if \(h_2 = f^j_2 \in F\) then \(h_3 = p_{v^j_2}; \) and so on.

We show that this sequence does not contain repetitions and that it terminates with a non-failed host. Since there are exactly \(i\) failed hosts when \(\text{evac}^i(v)\) is called and since there are \(l_v > i\) shadows, the sequence always has a next candidate to examine. Since the host set is finite, we only need to show that there are no repetitions. We prove through contradiction.
Let $h_a$ be the first repeating host in this sequence; namely, there exists $h_b$ in the sequence with $b < a$ and $h_a = h_b$. If $b = 1$ then, by definition, $h_a = p_{v, h_a} = h_1 = p_{q, 0}$. Since $j_a - 1$ is a failed host number we must have $j_a - 1 \neq 0$, i.e., $v^{j_a - 1}$ is a shadow machine. By the lemma condition, it cannot be co-located with its VM $v^{j_a}$; namely, $p_{v, h_a} \neq p_{q, 0}$; a contradiction.

Similarly, if $b > 1$ then by definition, $h_a = p_{v, h_a} = h_b = p_{v, h_b}$. Since the shadows $v^{j_a - 1}$ and $v^{j_b - 1}$ cannot be co-located unless they are equal, we must have $j_a - 1 = j_b - 1$. This implies $h_a - 1 = h_b - 1$, as $h_a - 1 = f^{j_a - 1}$ and $h_b - 1 = f^{j_b - 1}$. Hence, the first repeating host in the sequence must be $h_a - 1$ and not $h_a$; a contradiction.

Lemma 2 provides a correctness proof of (11).

**Lemma 2.** If $p_{v, i} \neq h$ for every $0 \leq i \leq l_v$ then VM $v$ is never evacuated to host $h$, regardless of host failures.

**Proof:** The result returned by evac$(v)$ is $p_{v, i}$ for some $0 \leq i \leq l_j$ (i.e., the initial placement of $v$ or one of its shadows). By the lemma condition, $p_{v, i} \neq h$; therefore, the result of an evacuation cannot be $h$. Since the initial placement is not $h$ either, $v$ is never placed on $h$.

Lemma 3 provides a correctness proof of (12).

**Lemma 3.** If $p_{v, i} \neq p_{q, 0}$ and $p_{u, i} \neq p_{q, 0}$ for every $i \neq j$ ($0 \leq i \leq l_v, 0 \leq j \leq l_u$) then VM $u$ is never located on the same host as VM $v$, regardless of host failures.

**Proof:** By the lemma condition, $p_{v, i} \neq p_{q, 0}$, so $v$ and $u$ are initially not co-located; thus, the only possibility for $v$ and $u$ to reside on the same host is via evacuation. As shown above (see proof of Lemma 1), calls to evac$(v)$ follow a sequence of hosts $(x_1, x_2, \ldots)$, where $x_1 = p_{v, i} = f^{i_1}$, $x_2 = p_{v, i-j} = f^{i_2}$, and so on. Similarly, all the calls of evac$(u)$ follow a sequence of hosts $(y_1, y_2, \ldots)$, where $y_1 = p_{q, 0} = f^{i_1}$, $x_2 = p_{v, i-j} = f^{i_2}$, and so on. By the lemma assumption, $x_1 = p_{v, i} \neq p_{q, 0}$ for all $0 \leq i \leq l_v$; therefore, $x_1 = p_{v, i} \notin \{y_1, y_2, \ldots\}$; thus $u$ cannot be evacuated at any point to $x_1$. Similarly, $v$ cannot be evacuated at any point to $y_1$.

Assume, by contradiction, that at some point, through evacuation, $v$ and $u$ are placed on the same host. This implies that for some $1 < a \leq l_v$, $1 < b \leq l_u$, we have $x_a = y_b$; hence $p_{v, i} = p_{q, 0}$. By the lemma assumption, we must have $j_a - 1 = j_b - 1$, implying $f^{j_a - 1} = f^{j_b - 1}$, which leads to $x_a - 1 = y_b - 1$. Continuing this reasoning, we get either $x_1 \in \{y_1, y_2, \ldots\}$, implying $v$ might be evacuated to $p_{q, 0}$ or $y_1 \in \{x_1, x_2, \ldots\}$, implying $u$ might be evacuated to $p_{q, 0}$. Both these cases are a contradiction as, by the lemma condition, all the shadows of $u$ are anti-co-located with $v$ and all shadows of $v$ are anti-co-located with $u$.

Lemma 4 provides a correctness proof of Equation (16).

**Lemma 4.** If, for all $h \in H, q \in Q$

$$c_{h,q} \geq \sum_{v \in V} \sum_{s.t. p_v = h} r_{vq} + \sum_{n \in H} \max_{1 \leq i \leq k} \min_{n \notin H} \left\{c_{nq} \sum_{u \in V} \sum_{n \notin H} r_{uq} \right\}$$

4The case of $i > l_v$ does not affect this as $v$ stays in its current location.

then the total resource requirements of all VMs that may end up residing on a (non-failed) host, following other host failures, would never exceed the host’s capacity.

**Proof:** For a given host $h$ and resource type $q$, the sum can be stated as

$$c_{h,q} \geq A(v, h) + \sum_{1 \leq i \leq k} \max_{n \notin h} B(q, n, i)$$

The term $A(v, h)$ equals the total resource requirements of type $q$ for all VMs that are initially placed on $h$. Obviously, these requirements stay unchanged so long as $h$ does not fail.

The term $B(q, n, i)$ is an upper bound of the total resource requirements of type $q$ for all VMs that are in $E(n \rightarrow h)$ (see discussion preceding Equation (15)). The summation checks which other host $n \in H$ would, if it fails and marked $f^1$, presents the worst case, in terms of resource requirements on $h$. Thus, $B(q, n, i)$ is an upper bound, regardless of the eventual actual failure numbering.

Using Lemmas 1, 2, 3, and 4, we have established the following theorem.

**Theorem 1.** A feasible, resilient solution to the Problem RVP can be found by transforming it into Problem SVP, solving Problem SVP and combining the solution with the evacuation process defined through Procedure evac$(v)$ and Equations (6) and (7).

**V. Simulation Results**

To evaluate the practicality of our method, we used a general-purpose CP engine developed at IBM Haifa Research Lab [16]. We built a constraint model for the SVP problem, and tested it with three sets of tests called “RD_1”, “RD_2” and “RD_3”, respectively. In all tests, the average CPU load (that is the ratio of the total CPU demand of the VMs to the total CPU capacities of the hosts) was fixed at 60%, which is typical of a highly loaded cluster. The number of VMs is 4 times the number of hosts. Host capacity is uniformly distributed over three possible values of 500, 750 and 1000. VM demand is uniformly distributed over 5 values of 90, 105, 120, 135, and 150. Last, distributions of the resiliency level $k$ for each of the three sets is given in Table I.

<table>
<thead>
<tr>
<th>Test Set</th>
<th>% of $k = 0$</th>
<th>% of $k = 1$</th>
<th>% of $k = 2$</th>
<th>% of $k = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD_1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RD_2</td>
<td>40</td>
<td>40</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>RD_3</td>
<td>65</td>
<td>20</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

For each of the three test sets, we compute the resilient placement using our SVP constraint model, which we refer to as Shadow-Based Solution (SBS), on four cluster sizes of: 8, 16, 24 and 32 hosts. For each cluster size, we do 60 runs and compute the average result. The graphs below show the averages. The tests were run on an IBM HS-21 blade with
Figure 5 shows the average running times and standard deviations for the three test sets with an increasing number of hosts in the pool. The results show that our method can be indeed used for practical cluster sizes. For example, consider the rightmost points - on a cluster size of 32 hosts (comparable to the maximal VMware DRS cluster of 32 hosts [17]) with 128 VMs. The computation time is less than 300 seconds for RD_1, where all VMs are 1-resilient. This is reasonable for guaranteeing resiliency in a cloud management framework that performs a periodic optimization loop to address load changes. Note that this computation is not done on the critical failure handling path, and that considering the elapsed times for VM migrations (can be several minutes) the periodic optimization does not need to be very frequent. We expect that further effort invested in the SVP constraint model could improve the current results significantly\(^5\).

SBS yields an average extensibility more than 5 times better than that of DSH. This advantage is due to SBS that allows the spreading of the backup space across all hosts to accommodate extra demand. In contrast, DSH keeps several hosts empty on purpose, thereby losing the ability to leverage their capacity for extensibility.

Figure 6 shows the average running times and standard deviations for the three test sets with an increasing number of hosts in the pool. The results show that our method can be indeed used for practical cluster sizes. For example, consider the rightmost points - on a cluster size of 32 hosts (comparable to the maximal VMware DRS cluster of 32 hosts [17]) with 128 VMs. The computation time is less than 300 seconds for RD_1, where all VMs are 1-resilient. This is reasonable for guaranteeing resiliency in a cloud management framework that performs a periodic optimization loop to address load changes. Note that this computation is not done on the critical failure handling path, and that considering the elapsed times for VM migrations (can be several minutes) the periodic optimization does not need to be very frequent. We expect that further effort invested in the SVP constraint model could improve the current results significantly\(^5\).

SBS yields an average extensibility more than 5 times better than that of DSH. This advantage is due to SBS that allows the spreading of the backup space across all hosts to accommodate extra demand. In contrast, DSH keeps several hosts empty on purpose, thereby losing the ability to leverage their capacity for extensibility.

VI. RELATED WORK

As already noted in Section I and in [18], [11], the area of combining high-availability and virtualization seems to be an emerging direction. In this section we provide a survey of known work that is related to ours both directly and indirectly.

The N+M technique of adding M standby hosts to as spares for N working hosts to accommodate up to M failures [15] is probably the best-known solution to the problem we discussed. Dedicating the k biggest hosts out of N hosts to serve as empty spares is indeed a valid solution to k-resiliency, as shown in Section II, and is also the fall-back solution to be used when the CP engine is unable to produce a solution in a predefined time-limit. However, it suffers from drawbacks compared to other solutions that spread the backup space over the running hosts. First, as already indicated in Section I, the resources of the standby hosts are not available for running VMs when there is no failure, to help handling excess load. Second, if the running hosts have different capacities, all the standby hosts need to be as large as the biggest host for all resource types - so if one host has the biggest CPU capacity, and another has the biggest memory capacity, all the standby hosts need to have both the biggest CPU capacity and the biggest memory capacity. Last, having anti-location constraints further worsen the requirements of standby hosts, since specific hosts that are compatible with specific VMs need to be added as standbys.

[18] assumes that an application service is provided by a group of identical VMs. Their approach is to ensure that there will always be at least a minimum number of VMs in the group providing the service (where this number is derived from the required response time). This is done by employing a redundant configuration, where additional running VMs of the application service are provisioned, such that for any set
of up to \( k \) failing hosts the minimal number of VMs for each application service will be maintained. While this approach may allow in principle for quicker recovery from host failure than this work, it also consumes significantly more resources, since maintaining actual redundant replicas of VMs instead of shadows prevents the overlap of the extra space, even at times of no failure - unless the replicas are used for additional value, such as increased service capacity.

DeCoRAM [19] features a technique that is somewhat similar to ours, but in a different context – using replication to ensure the resiliency of real-time tasks for up to \( k \) processor failures. For each primary task, their algorithm places \( k \) additional anti-colocated backup tasks which may overlap. Unlike our solution, they examine all possible sets of \( k \) failures, and resort to a greedy heuristic approach. DeCoRAM places tasks sequentially and if any task cannot be resiliently placed on the current set of processors, another processor is added. This is different from our setting, where adding more hosts (processors) is not a valid option. Also, we define a broader problem, for example, by allowing anti-colocation constraints.

VMware provides VMware HA [14], a cold standby technique where a VM is restarted automatically on another host after a failure of the host on which it resides. This results in the interruption of the applications running on those VMs. The technique presented in this paper is also a cold standby, but its incurred downtimes should be smaller compared to VMware HA since it uses HwPFA to initiate live relocation before the failure. Thus, if the failure detection is sufficiently accurate and early, the service of the VM will suffer from little to no downtime.

Utilizing CP for placement computation provides an elegant and flexible approach that can easily be extended to take into account additional constraints, thus allowing a clear separation of the problem domain from the solution domain. A constraint solving approach is used by Entropy [8] for placing VMs on hosts while minimizing the number of active servers and the number of migrations required to reach a new configuration. A later work [3] extends the Entropy work with the dynamic provisioning of VMs directed by high-level SLA goals. However, placement computations in neither of these works consider availability of the VMs.

VII. CONCLUSIONS

In this paper we introduced a novel technique for incorporating high availability goals into the optimization of the placement of virtual machines on a cluster. This enables achieving both optimized placement (according to any optimization goal) and guaranteeing a property of cluster resiliency with per-VM resiliency levels. The theoretical problem as defined belongs to a high complexity class and we provide a heuristic algorithm, which is not guaranteed to find a solution in all cases. However, an optimized and resilient solution, when found, can be very beneficial. We showed how our transformed rules enable a CP engine to compute a solution in reasonable time on practical cases. The simulation results showed the advantage of our technique in achieving a load balancing optimization goal compared to a simple standby host solution (which also may not always be possible).

Other than extending a placement problem with resiliency constraints, our solution features reuse of the backup space in production hosts, thus avoiding reservation of dedicated backup hosts. Also, it allows setting of differentiated resiliency levels is on a per-VM basis, thus incurring overhead according to the importance of a VM. Finally, our solution enables preemptive evacuation of VMs from failing hosts without “reshuffling” other VMs first in order to free resources.

REFERENCES