A Power State Transition Algorithm for Power Controller of Energy Efficient Computer Clusters

Amgalan Ganbat[1,a)] Kenjiro Taura[1,b)]

Abstract: With the increase in both performance and scale of computer clusters, energy consumption has attracted much attention because of their impact on the environment and inevitably increasing operational cost. While various different approaches are being proposed to reduce power consumption of clusters, one promising approach is to make the idle nodes into low-power consuming state (i.e., turn-off or sleep) when the system's utilization level is low. However, complex dependencies among physical resources and services make it difficult to create common power control tool for such clusters. In this paper we propose an algorithm which solves complex dependencies among cluster nodes including both physical and service nodes. In addition, since different facilities offer various kinds of services and each has unique structure, they have different system management policies and constraints. This situation makes it hard to create a common power control tool that works on different kinds of clusters. Therefore, by exploiting our algorithm, we propose a versatile, scalable power control tool, ClusterNap, with which users can define their own policies and constraints that explicitly suits for the physical resources they have and the services they provide.

Keywords: State transition algorithm, ClusterNap, Energy-proportional computing, Cluster management

1. Introduction

Until recently recource management of computer clusters and datacenters were mainly about scheduling workloads or monitoring resources. However, as the performance and size of such facilities increase, energy consumption has increased dramatically and energy consumption cost alone now takes the largest part of total operational cost of most clusters. Therefore energy awareness tends to become indispencable part of resource management of computer clusters.

In 2007 Barroso et al [1] introduced a concept called *Energy-Proportional Computing* where they insisted that resources should consume energy proportional to the utilization of the system. It is because power consumption tends to be almost constant in many of current clusters and datacenters, while utilization level of computer clusters change through some periods. In other words, even when there is very low utilization, the cluster system consumes almost same as or only little bit lower energy than high-utilization period.

There are various approaches towards energy-proportionality. One of the most promising approach is to turn-off or make low-power comsuming state for some parts of the resources (nodes) when utilization level is low. Fig 1 shows CPU utilization sample of Google's 5000 servers collected during 6 months of period. From this figure, we can see that most of the time, server utilization is below 40 percent. That is, since most of the time servers have low utilization, we can turn off possible part of the servers so that electricity consumption can be reduced. Even though many research have shown the efficiency of this *node power state change method* in some specific situations, to our best knowledge, there is no such common tool which can be implemented in different kind of clusters as a part of their resource managers. One of the biggest factor making it difficult to create such common tool is the diversity of clusters. Each computer cluster has different physical structure and the services they provide also vary significantly. Moreover, when there are complex dependencies among the nodes, both physical resources and services, it is very hard to find the *optimal low power state* of cluster which is enough for keeping the requested services work gracefully. Finding the optimal power state can be reduced to Pseudo-Boolean constraint problem and it is a NP-hard problem.

The first contribution of this paper is that we propose a *Power State Transition Algorithm*, part of which also solves near optimal power state by *greedy way*. The main goal of this algorithm is to find the state transition way to reach the cluster's optimal power state when there are some certain nodes (a node can be both physical resource or a service) requested. Power State Transition Algorithm is explained in Section 3.

We mentioned that node power state changing method is a promising way to reduce a cluster's energy consumption while still continuing the services gracefully. However, since different cluster systems provide various kinds of services and have different policies and global constraints (examples are introduced in Section 2), it is hard to have common cluster power control tool that is adaptable in those different clusters.

The second contribution of our paper is that we are proposing a prototype of cluster power control tool, ClusterNap, which can be adapted to many different kinds of clusters. ClusterNap uses Power State Transition Algorithm and is introduced in Section 4.

[1] The University of Tokyo
[a)] amgaa@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

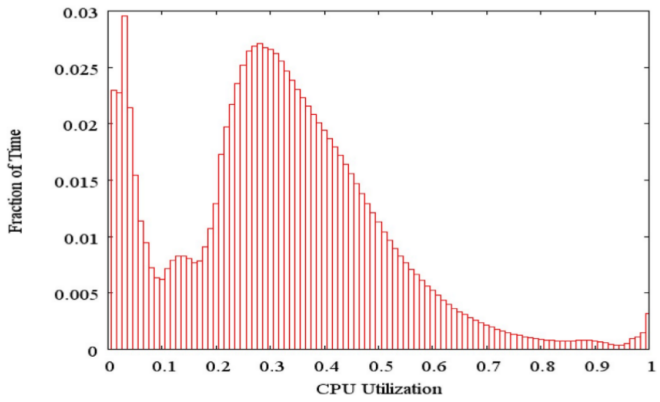

Fig. 1 CPU utilization sample of Google's 5000 servers in 6 months [2]

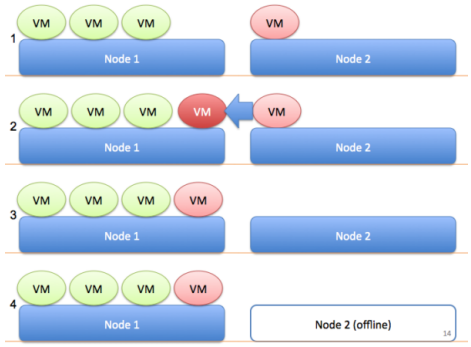

Fig. 2 VM consolidation

2. Related works

Clusters using VMs

One of the technology development that made node power down method useful is Virtual Machine (VM) consolidation. Hardware technology development in multi-core processing capabilities of servers has enabled a single server to hold multiple VM on itself without any problem and this is called VM consolidation. Consequentially, many clouds employ resource virtualization technique to abstract system resources (both hardware and software) from its host OS to provide their services such as IaaS or PaaS. Moreover, live migration technology of VMs let us move a VM from one physical host to another by imposing only scale of milliseconds of delay to the user. It means we can freely choose the VM scheduling policy. Younge et al [3], i.e., proposed an efficient resource management way for cloud computing environments. By introducing power aware VM shceduling technique, VM migration and other techniques they improved overall system efficiency with little overhead.

While VM consolidation (fig 2) enables us to use node power down method in clusters, our algorithm and proposal prototype tool can cooperate these resource management techniques and can acts as power controlling interface to deploy these techniques easily.

Clusters with DFS

As mentioned in previous section, cluster management must be aware of constraints under which the service it provides is available. Another example to emphasize the importance of being aware of cluster constraints is clusters which use Distributed File Systems (DFS) such as Hadoop Distributed File System (HDFS). Such clusters convert big data into multiple numbers of smaller sized data blocks and distribute them among cluster nodes to enable data parallelism. To provide data availability and fault tolerance, most common file systems replicate data blocks and keep each replica on different nodes. Therefore, to keep the fault tolerance and data availability, even the idle nodes have to be in active state even when the utilization level is very low. This policy certainly wastes a lot of energy. In this case, to leverage node power state change method, one could determine the group of nodes that together contain at least one of the replicas of all data blocks so at least we can provide data availability constraints. Once we find the group of nodes which has at least one replica of all data blocks, now rest of the nodes can be turned into low power mode. To maximize the energy reduction, one should find the minimum number of such group nodes to be kept active. For example, let's suppose we have nodes $\{node1, node2, node3, node4, node5\}$ and data blocks $\{b1, b2, b3, b4, b5, b6\}$. Now let's assume that data blocks are distributed among the nodes as follows:

$$node_1 = \{b_1, b_4\}$$
$$node_2 = \{b_2, b_5\}$$
$$node_3 = \{b_3, b_6\}$$
$$node_4 = \{b_1, b_2, b_3\}$$
$$node_5 = \{b_4, b_5, b_6\}$$

From above distribution of blocks, among many other possible solutions, we can easily find that node4 and node5 suffice data availability. However, finding this minimum number of sets (nodes) which covers all of elements (datablocks) is known as Minimum Covering Set Problem and is known as NP-Complete. Kim et al [4] proposed an approximate greedy algorithm to solve this problem. Kim's study showed that node power state change method can reduce energy consumption significantly. By configuring data blocks properly, our algorithm also solves this problem and it is shown in section *Experiment*. However, the main goal of our algorithm is to solve the dependencies between all kind of nodes (physical nodes and services).

Maheshwari et al [5] proposed a dynamic reconfiguring algorithm for clusters running MapReduce jobs. This dynamic reconfiguring technique enables the system to turn-off or on nodes in the cluster according to the systems utilization.

Other works

Since 2009, Dustin Kirkland et al [6] has developed a simple configurable power control tool, PowerNap, which works on a single standalone Ubuntu server. By giving the freedom to configure system policies and constraints to the user, PowerNap tool has become very flexible that it can be used in many different kind of environments. While PowerNap works on a standalone node, our proposal tool works on computer clusters (multiple nodes which have many dependencies on each other).

Even though all these approaches mentioned in this section is trying to save electric consumption by using node power down method, none of them considered the dependencies between physical resources. Most of them only concentrated on clusters

$$min : k_a \times node_a + k_b \times node_b + ...k_y \times node_y + k_z \times node_z$$
$$s.t : Dependency(D) \wedge requested\_nodeA \wedge requested\_nodeB...$$

Fig. 4 Minimum Power State finding problem can be reduced to *Pseudo-Boolean problem*. It is considered NP-Hard

which provide specific services and proposes solution for these specific services. On the other hand, our proposal algorithm and power control tool can be applied to any kind of clusters and acts as basic power controlling interface to realize these approaches.

Yokoyama et al [7] proposed a power API on a cluster that also controls clusters components inlcuding servers, switches and storages. At the present, this API does not consider the dependencies between physical nodes and services.

3. Power State transition algorithm

3.1 Main problem

Turning node's power status into low power consuming state, when the cluster system utilization level is low, is much more difficult than the case of services provided by a single, standalone node.

First, the Quality of Services (QoS) must be guaranteed. Services the cluster system provides have to continue and be in certain quality level regardless of the power states of the nodes in that system. In addition, the system as a whole must be stable regardless of the power states of the nodes. In other words, not only enough nodes must be active, but also right nodes must be alive. Cluster power controller must select the right nodes to be active to keep the system stability and it is not a simple task for most of clusters since there are most probably very complex dependencies among the both physical resources and services. Finding the minimum power state of a cluster when certain nodes are requested can be reduced to *Pseudo-Boolean* constraints (fig 4).

When a cluster's system dependency *Dependency(D)*, current power state and requested node *requested_nodeA, requested_nodeB* are given, we can generate pseudo-boolean constraints shown in fig 4, where k_i $(i = a, b, c, ...)$ can be power consuming coefficient of each nodes. *Nodes* can take value of *TRUE* (ON) or *FALSE* (OFF). Finding the satisfying assignment, which results in minimum power consumption, is thus, in above case, considered as NP-Hard problem. We use simple greedy algorithm to find approximate optimal solution and it is described in section 3.2.2.

Second, even if we know the minimum power state, finding the correct path to reach that state is also important. To reach the minimum power state gracefully, we need to turn-off and turn-on right nodes in right orders. Our Power State Transition algorithm solves this problem and is described in section 3.2

3.1.1 Node dependencies

In this paper, a *node* is either physical resource (i.e. server, management node, storage device, network switch) or a service (i.e. state of a Virtual Machine running, state of being able to connect to a server or access data). A node can have 3 different power states; **RUNNING** (or **ON**), **SHUT-OFF** (or **OFF**) and **Unknown**. There are 3 different dependencies we consider; **RUN-dependency**, **ON-dependency** and **OFF-dependency**. All dependencies are written in *Conjunctive Normal Form* of Boolean logic.

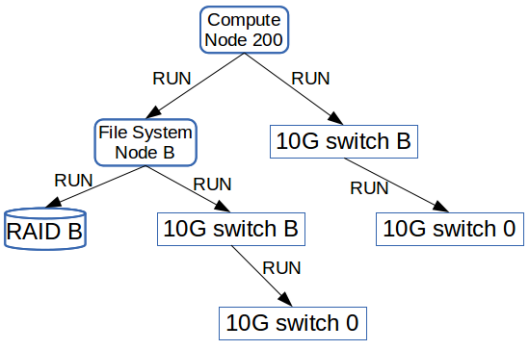

Fig. 5 Example of a simple RUN-dependency

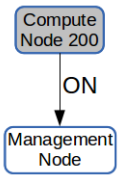

Fig. 6 Example of a simple ON-dependency

RUN-dependency

If "*nodeB*" has to be ON to keep "*nodeA*" ON, we say there is RUN-dependency between "*nodeA*" and "*nodeB*". Alternatively, we say "*nodeA*" is RUN-dependent on "*nodeB*" and is represented as $nodeA \xrightarrow{\text{RUN}} nodeB$. For example, in fig 3, a very typical HPC cluster is depicted. In this cluster, "*Compute Node 200*" might be RUN-dependent on "*Filesystem Node B*" and "*10G switch B*" as shown in fig 5. Moreover, "*Filesystem Node B*" can be RUN-dependent on "*RAID B*" and "*10G switch B*".

ON-dependency

Likewise, if "*nodeB*" has to be ON to make "*nodeA*" ON from OFF state, we say there is ON-dependency between "*nodeA*" and "*nodeB*". Moreover, we say "*nodeA*" is ON-dependent on "*nodeB*" and is represented as $nodeA \xrightarrow{\text{ON}} nodeB$. For example, to turn-on "*Compute Node 200*" in fig 3, we might need "*management Node*" to be ON and that management node might not need to be ON once our requested server becomes ON. Therefore we represent this situation as ON-dependency fig 6.

OFF-dependency

If "*nodeB*" has to be ON to make "*nodeA*" OFF from ON state, we say there is OFF-dependency between "*nodeA*" and "*nodeB*". We say "*nodeA*" is OFF-dependent on "*nodeB*" and is represented as $nodeA \xrightarrow{\text{OFF}} nodeB$. A similar example as ON-dependency can be applied to OFF-dependency.

AND operation in dependencies

When a node is dependent on several nodes at the same time, we can use Boolean AND operation. For example, in fig 5, "*Compute Node 200*" is RUN-dependent on $FilesystemNodeB$ AND $10GswitchB$.

OR operation in dependencies

A node can be dependent on one of several nodes such as

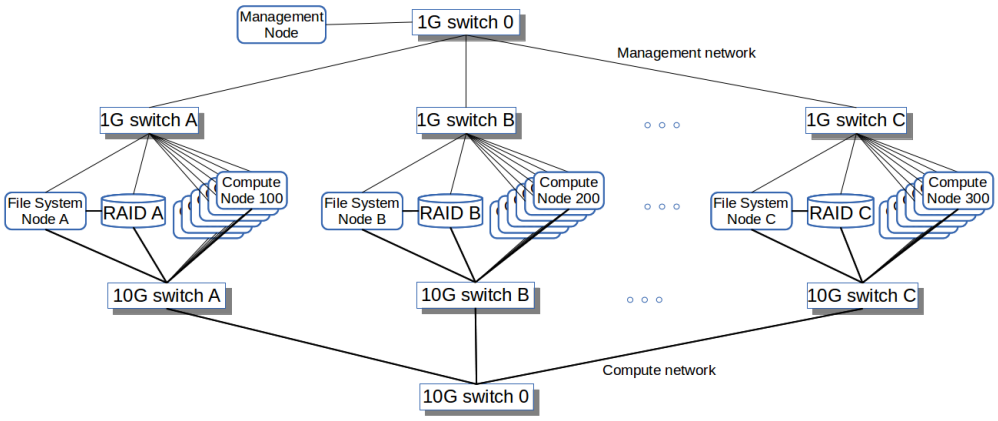

Fig. 3 Example of a typical HPC cluster

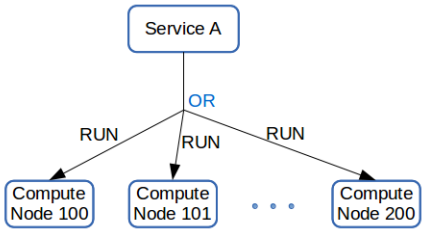

Fig. 7 Example of OR operation case in RUN-dependency

$(node_1$ **AND** $node_2)$ **OR** $(node_3$ **AND** $node_4$ **AND** $node_5)$ **OR** $(node_6)$

Fig. 8 An example of nodes written in CNF

$NodeA \xrightarrow{\text{ON}}$ ($NodeB$ OR $NodeC$ OR $NodeD$). For example, if a service can run when only one of $ComputeNode100$ or $ComputeNode101$ or $ComputeNode200$ is ON, we can use OR operation in our dependency (fig 7).

The dependents of a node should be *Conjunctive Normal Form* (CNF) of Boolean logic. It means Dependents are written as conjunction (OR operation) of *clauses*, where a clause is a disjunction(AND operation) of nodes. An example is shown in fig 8.

3.2 State Transition Algorithm

In section 3.1, the main problems of power state changing method are discussed. In addition, the main terminologies those are used in State Transition Algorithm are also described.

The main flow of State Transition Algorithm is shown in fig 9. First State Transition Algorithm gets information of Current power state of system $\mathbf{State_{current}}$, requested nodes (nodes that are requested by user) **R**, and dependency of system **D**.

With these three values, we need to calculate the minimum power state $\mathbf{State_{min}}$ to which our system need to reach. How we determine $State_{min}$ is explained in section 3.2.2.

Once the $State_{min}$ is determined, we need to take action to approach that state. Therefore, based on D, R and $State_{current}$, our algorithm determines nodes which we can turn-off $\mathbf{Nodes_{to-OFF}}$

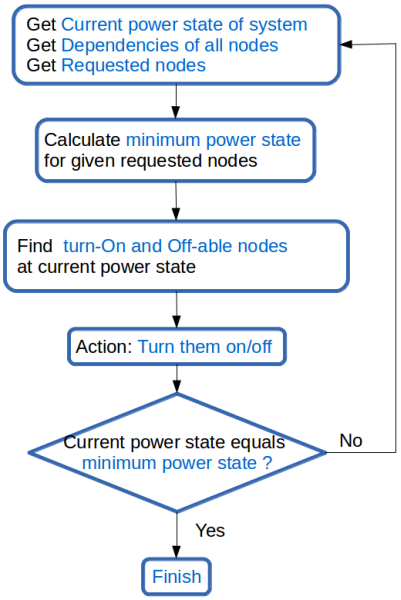

Fig. 9 General flow of the Power State Transition Algorithm

and nodes which we can turn-on $\mathbf{Nodes_{to-ON}}$ to approach the minimum power state. In section 3.2.3, we explain how $Nodes_{to-ON}$ and $Nodes_{to-OFF}$ are determined.

After determing $Nodes_{to-OFF}$ and $Nodes_{to-ON}$, power contoller of the system tries to turn off $Nodes_{to-OFF}$ and turn-on $Nodes_{to-ON}$.

Since the power controller changed power states of some nodes, now $State_{current}$ is changed. If $State_{current}$ equals $State_{min}$, algorithm finishes. Else, algorithm starts from the beginning until it reaches $State_{min}$

3.2.1 Current power state, requested nodes, dependencies

System's current power state $State_{current}$ can be a vector. Each element of $State_{current}$ corresponds to each node of the cluster and takes value of either one of ON, OFF, or $Unknown$.

Requested nodes R is a list of node names which are requested by user.

Dependency D contains the names of all nodes and their dependent nodes described in section 3.1.1. Since there are three

```
1: procedure MINIMUM POWER STATE(D, R)
2:     State_min = R
3:     while R ≠ 0 do
4:         N = choose a node from R
5:         for node in D[N]'s clauses                ▷ D[N] is in CNF do
6:             if node is in State_min then
7:                 remove node from D[N]'s all clauses
8:             end if
9:         end for
10:        State_min += Clause with the least elements from D[N]
11:        remove N from R
12:    end while
13:    return State_min
14: end procedure
```

Fig. 10 Minimum power state algorithm

different dependencies (RUN, ON, OFF), we can further divide D into $\mathbf{D_{RUN}}$, $\mathbf{D_{ON}}$, and $\mathbf{D_{OFF}}$ respectively.

3.2.2 Minimum Power State

Minimum power state is a power state of a cluster that comsumes the least power possible while still keeping the requested nodes running. When dependency $\mathbf{D_{RUN}}$ and requested nodes **R** are given, we can construct a Pseudo-Boolean constraint shown in fig 4 . In this case, solution of this problem is a group of nodes which *satisfies* the boolean function on the second line while keeping the equation in the first line *minimum*. As mentioned in section 3.1, it is a NP-hard problem. Fortunately, there are many approximate Pseudo-Boolean constraint solvers proposed. A similar problem is Linux's *Minimum install problem*. Since, in Linux OS, packages have their dependencies, there are multiple ways to install a package. Finding the way to install minimum size (or number) of packages can also be reduced to Pseudo-Boolean problem. One of the available solutions is proposed by Tucker et al [8]. In our case, we are using a simple greedy way to determine the group of nodes for minimum power state. The algorithm is shown in fig 10.

This algorithm is somewhat similar to *Greedy set-covering algorithm* introduced in Leiserson et al's book [9]. In our algorithm, we choose the least number of dependency nodes of a requested node and put them into $State_{min}$ node list. Of course, dependency nodes those are already in $State_{min}$ is excluded. This algorithm tries to get the least number of nodes that satisfies the seconds line boolean equation in fig 4 by greedy manner.

3.2.3 Turn-On and Off-able nodes

Now we know which power state we are trying to reach (section 3.2.2). However, to reach the destination state $State_{min}$, we cannot just turn-off ON nodes that are not in $State_{min}$, and turn-on OFF nodes that are included in $State_{min}$ since there are ON, OFF, and RUN-dependencies. Therefore, we need to know exactly which nodes we should turn-off and which ones we should turn-on at the moment without causing any inconvenience to the system.

In fig 11, the procedure of determining turn-On and Off-able nodes ($Nodes_{to-ON}$ and $Nodes_{to-OFF}$) is described. First, in line 2-6 in fig 11, we are determining what other nodes are required to be ON to make OFF nodes ON that are in $State_{min}$. Moreover, nodes that are required to be ON to make ON nodes OFF

```
1: procedure NODES TO ON/OFF(D_RUN, D_ON, D_OFF, State_min, State_current)
2:     Nodes_to-ON = Nodes_OFF ∩ State_min
3:     Nodes_to-OFF = Nodes_ON \ Nodes_to-ON
4:     Nodes_to-ON += MinimumPowerState(D_RUN * D_ON, Nodes_to-ON)
5:     Nodes_to-ON += MinimumPowerState(D_RUN * D_OFF, Nodes_to-OFF)
6:     Nodes_to-ON = Nodes_to-ON \ Nodes_to-OFF
7:
8:     # Exclude not Turn-On-able nodes
9:     for Node in Nodes_to-ON do
10:        if not (at least one clause of Node's RUN * ON − dependency
   CNF has all members ON) then
11:            remove Node from Nodes_to-ON
12:        end if
13:    end for
14:
15:    # Exclude not Turn-Off-able nodes
16:    for Node in Nodes_to-OFF do
17:        if not (at least one clause of Node's OFF − dependency CNF
   has all members ON) then
18:            remove Node from Nodes_to-OFF
19:        end if
20:    end for
21:
22:    for Node in Nodes_to-OFF do
23:        if Another "ON node" is RUN-dependent on Node then
24:            remove Node from Nodes_to-OFF
25:        end if
26:    end for
27:
28:    return Nodes_to-ON , Nodes_to-OFF
29: end procedure
```

Fig. 11 Algorithm to determine On and Off-able nodes

that are not in $State_{min}$ is also determined here. Each of them are now on the list of $Nodes_{to-ON}$ and $Nodes_{to-OFF}$ respectively. In line 4, $*$ is a operation where *Cartesian product* of each node's RUN-dependents and ON-dependents are taken (dependents are in CNF) and returned in CNF. Same logic applies to the operation in line 5.

However, from $Nodes_{to-ON}$ we cannot make all of them ON at the same time. For example, in fig 3, let us assume all "File System Node A", "RAID A", and "Compute Node100" are OFF. When the node "Compute Node 100" is requested, all three nodes are in $State_{min}$ list according to the dependency. However, we cannot just turn-on all three of them at the same time. It is becaue according to the RUN-dependency, we should first turn-on "RAID A", then "Filesystem A" and finally turn-on "Compute Node 100". Therefore, only turn-On-able node is "RAID A" and other two nodes should not be included in $Nodes_{to-ON}$ list (line 8-12 in fig 11).

Similar logic applieas to Turn-Off able nodes $Nodes_{to-OFF}$. For example, let us assume that all three of "File System Node A", "RAID A", and "Compute Node100" are ON and we do not need them anymore so we should turn-off them. However, we cannot just turn-off them at the same time. We should turn-off them in right order; i.e first "Compute Node 100", then "Filesystem Node A" and finally "RAID A" (line 15-19 in fig 11).

Moreover, we should check that if another ON node is dependent on any of these three nodes. In that case, we also cannot

turn-off that node (line 21-25 in fig 11).

3.2.4 Action

Once we determine $Nodes_{to-ON}$ and $Nodes_{to-OFF}$, now power controller does its job - to change their power states. Since each kind of node can be turned-on/off by different ways, this part should be configured by system's owner. We propose our solution to this state change way in ClusterNap in section 4.1.2.

By repeating the steps explained through section 3.2.1 - 3.2.4, cluster's power state $State_{current}$ approaches to $State_{min}$. In real clusters, some nodes have *Unknown* state especially during power state change. Our algorithm also considers this situation and takes *actions* based on the state of nodes which are only ON or OFF.

4. ClusterNap

As mentioned in Section 1, ClusterNap is a simple, configurable power control tool for computer clusters. Only by configuring dependencies among the nodes and services in computer cluster, ClusterNap determines which nodes should be ON and which can be off (or be in a low-power mode). Moreover, it can change the power state of cluster in right order while still keeping the quality of services being provided at good level by exploiting our proposal algorithm introduced in Section 3.

ClusterNap is designed to work easily with other resource managing and monitoring softwares. For example, ClusterNap can get information of requested nodes or services from Torque resource manager [10]. We are also working on to make it possible to use other resource monitoring tools, such as Ganglia and Nagios [11] [12], as its nodes' state monitoring module. In the following subsections we introduce main design of ClusterNap.

4.1 Software design

As mentioned before, ClusterNap should be easily and freely configurable for any kind of cluster. We tried to make it as easily configurable as possible.

4.1.1 Dependencies

We mentioned that there are three kind of dependencies our Power State Transition Algorithm considers; RUN, ON, and OFF dependency. Each of these dependencies are written in corresponding folder with the extention of ***.dep***. Dependency of each node can be written in a single *.dep* file or multiple nodes' dependencies can be written in a single file. A dependency file can have arbitrary name, but should have *.dep* extention. An example of a dependency file is shown in fig 12.

In fig 12, dependencies of four nodes (RAID1, RAID2, VM1, Data1) are written in a single file. We can easily see that RAID2 is dependent on "*(guppy-minnie **AND** guppyfs) **OR** guppy3*". In other words dependency is written in *Conjunctive Normal Form* (CNF) of Boolean logic.

4.1.2 State change script

ClusterNap changes power state of any given node in cluster by executing user-defined script on some user-chosen server in the system. For example, to turn-on serverA, a user might execute a script which contains IPMI command on it as root user on serverB. In other words, if the user defines a script (i.e., a script which contains command like "*ipmitool -I lanplus -H serverA_IP serverA -U root power on*"), *a node name* which contains this script (for example, serverB), and a *username* (for example *root*), we can change power state of any node in the cluster.

```
# Storage dependencies
node:RAID1
depends: serverA | serverB
node:RAID2
depends: serverC, serverD | serverE

# VMs
node:VM1
depends: serverF

# Data
node:Data1
depends: RAID1 | RAID2
...
```

Fig. 12 Configuration of node dependencies

```
Name: fs1
Host/user/path: serverA, amgaa, /path/to/script.sh

Name: fs2
Host/user/path: serverB, root, /path/to/script.sh
Host/user/path: serverC, root, /path/to/script.sh
...
```

Fig. 13 Configuration of node state-change scripts

In fig 13, an example of configuration file of power state change scripts shown. There can be multiple options to change power state of a single node. For example, in fig 13, node fs2 can be turned-on by running a script on either of node guppy2 or guppy3. ClusterNap chooses either guppy2 or guppy3 since one of them might be already *OFF*.

4.1.3 Requested nodes

ClusterNap takes requested nodes from a folder named "*requested/*". Therefore, all that a user should do is just to create a file which has the same name as requested node in the folder *requested*. Since only a single command "*touch requested/nodename*" can create such file, it is relatively simple for other resource managing tools to cooperate with ClusterNap.

4.1.4 Node states

ClusterNap takes current power states if all nodes from a folder named "*state/*". Similar to the folder "*requested/*", in folder "*states/*", there are files which have same names as all nodes in the system. Inside each of these files, one of **1**,**0**, or **-1** is written. If 1 is written inside of a file named serverA, it means that the node serverA is *ON*. If the value was 0 or -1, it implies that node is *OFF* or *Unknown* respectively.

For now, we have not completed creating the module which updates power states of all nodes in the system constantly. However, since the configuration is as simple as mentioned here, making use of resource monitoring tools such as Ganglia and Nagios should not be difficult.

Once the user has defined all necessary configurations mentioned in section 4.1.1 - 4.1.4, ClusterNap runs the Power State Transition Algorithm and takes necessary power state change actions until the system reaches possible minimum power consuming state.

4.2 Other application of ClusterNap

Until now, we have been emphasizing the fact that how ClusterNap would be useful for physical cluster systems. There is another potential application of ClusterNap. One of the key opportunities cloud technology giving us is its ability to dynamically provision virtual resources on fine-grained way. In other words, cloud service users can get necessary resources as much as or as little as they want according to their needs in near real-time. Since cloud users have to pay as they use resources, getting the optimized amount of resources is very important economic task. Therefore, when the user is using cluster like virtual resources on cloud, they also can make use of ClusterNap in their virtual cluster. They can define their own policies and constraints on it so that their virtual cluster scales up or scales down dynamically preventing the user from buying excess amount of virtual resources on cloud.

5. Experiment

We conducted a simple simulation on a virtual cluster with similar structure as a cluster shown in fig 3. We used KVM virtual machines as physical resources (compute and manage nodes, switches, RAIDs, and filesystem servers). With VMs, we can define complicated dependencies comparatively easily and test them. Following two cases were tested.

Case 1: HPC cluster shown in fig 3

We created a model of HPC cluster shown in 3. This cluster has 24 nodes: 1 management node, 3 switches, 5 filesystem servers, 5 RAIDs, and 10 compute nodes. From the state of all nodes being OFF, we tried to make random nodes ON. In all trials, our tool succeeded to make requested nodes ON gracefully. Moreover, after making requested nodes ON, we also made requested nodes not-requested, and executed our algorithm. All trials also succeeded.

Case 2: Data replication placement

To check how our tool solves complicated dependencies among services we conducted the same example shown in section 2. Data blocks $\{b1, b2, b3, b4, b5, b6\}$ are distributed among the nodes as follows:

$$node_1 = \{b_1, b_4\}$$
$$node_2 = \{b_2, b_5\}$$
$$node_3 = \{b_3, b_6\}$$
$$node_4 = \{b_1, b_2, b_3\}$$
$$node_5 = \{b_4, b_5, b_6\}$$

We can configure the dependencies of above data blocks as shown in fig 14.

In above case our tool returns $node_1, node_2, node_3$ while the optimal solution was $node_4$ and $node_5$. This situation is understandable because our Minimum Power State algorithm (fig 10) solves Pseudo-Boolean problem in greedy way. However, since our State Transition Algorithm consists of several blocks as shown in fig 9, the Minimum Power State algorithm's part can be further optimized and we can use other sophisticated Pseudo-Boolean constraint solvers. If we configure the dependencies among services correctly, this case shows that our tool can also solve other resource allocation problems either.

```
# Data block dependencies
node:b1
depends: node_1 | node_4

node:b2
depends: node_2 | node_4

node:b3
depends: node_3 | node_4

node:b4
depends: node_1 | node_5

node:b5
depends: node_2 | node_5

node:b6
depends: node_3 | node_5
```

Fig. 14 Configuration of data block dependencies

5.1 Further works

For the algorithm part, as mentioned in previous section, further optimization of Minimum Power State Algorithm is considerable. It is because, in our tool, we use very simple greedy algorithm for Pseudo-Boolean constraint solver and even in some simple cases, our tool does not return the quite optimal solutions.

For ClusterNap, integration of ClusterNap with other resource managing tools is important. For any kind of clusters, working with workload schedulers (i.e., Torque resource manager) is vital. However, to work with such tools, users should define their policies and constraints. Furthermore, getting each node's power state correctly is also important for ClusterNap to work gracefully. Therefore, cooperating with resource monitoring tools, such as Ganglia and Nagios, is our next direction.

6. Conclusion

In this paper, we proposed a Power State Transition Algorithm for power controller of computer clusters. This algorithm takes current power state, requested nodes and dependency information of cluster and solves a way to reach near minimum power consuming state of that cluster.

In addition, we proposed prototype of a flexible and versatile computer cluster power control tool, ClusterNap, which can be applied to various kind of cluster environments. This tool can work alone or with other resource managing and monitoring tools easily. This tool also can potentially be used as a resource manager of virtual resources in cloud environment. ClusterNap uses Power State Transition Algorithm and is tested on virtual cluster we built. It showed ClusterNap can make cluster reach to minimum power consuming state gracefully and eventually reduce power consumption.

References

[1] Barroso, L. and Holzle, U.: The Case for Energy-Proportional Computing, *Computer*, Vol. 40, No. 12, pp. 33–37 (2007).
[2] Barroso, L. and Hölzle, U.: The datacenter as a computer: An introduction to the design of warehouse-scale machines, *Synthesis Lectures on Computer Architecture* (2009).
[3] Younge, A. J., von Laszewski, G., Wang, L., Lopez-Alarcon, S. and Carithers, W.: Efficient resource management for Cloud computing environments, *International Conference on Green Computing*, pp. 357–364 (2010).
[4] Kim, J. and Rotem, D.: Energy proportionality for disk storage using replication, pp. 81–92 (2011).
[5] Maheshwari, N., Nanduri, R. and Varma, V.: Dynamic energy efficient data placement and cluster reconfiguration algorithm for MapReduce framework, *Future Generation Computer Systems*, Vol. 28, No. 1, pp. 119–127 (2012).
[6] Kirkland, D.: PowerNap configurable daemon for Ubuntu, https://launchpad.net/powernap (2012). Accessed: 2013-07-03.
[7] Yokoyama Daisaku, T. K. and Masaru, K.: Towards Energy Efficient Programming — An Implementation of a Power Control Interface on a Cluster, *IPSJ Special Interest Group on Programming*, No. 2, pp. 1–7.
[8] Tucker, C., Shuffelton, D., Jhala, R. and Lerner, S.: Opium: Optimal package install/uninstall manager, *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, IEEE, pp. 178–188 (2007).
[9] Leiserson, C. E., Rivest, R. L., Stein, C. and Cormen, T. H.: *Introduction to algorithms*, The MIT press (2009).
[10] Staples, G.: TORQUE resource manager (2006).
[11] Massie, M. L., Chun, B. N. and Culler, D. E.: The ganglia distributed monitoring system: design, implementation, and experience, *Parallel Computing*, Vol. 30, No. 7, pp. 817–840 (2004).
[12] Josephsen, D.: *Building a monitoring infrastructure with Nagios*, Prentice Hall PTR (2007).