University of Hamburg

project report

Moab Evaluation

Author:
Florian Ehmke

Supervisor:
Timo Minartz

December 9, 2011

Contents

1 Introduction

Often the energy costs are a large amount of the total costs of ownership (TCO) of a cluster. The longer the cluster is used the larger this slice of the cake grows. Different approaches exist to alleviate this problem. CPUs lower their clock speed when they are idle (which often already decreases the power consumption by 50 %), hard disks spin down and GPUs lower the clock speed of both core and memory. All of these methods are applied very fast and on demand. However the idle power consumption of today's systems remains high. If large idle times are likely to happen the best option could be to completely shutdown a node. Most resource managers and schedulers don't support this. As soon as one node is turned off it is reported as offline and no longer eligible to run jobs. Moab provides this functionality which is the topic of this report.

We will investigate Moab's behaviour during several different workloads und measure how much energy can be conserved.

1.1 Moab

Moab Workload Manager is a powerful resource management and scheduling system for clusters and grids. Moab is able to work with many other resource management and monitoring tools such as IPMI (Intelligent Platform Management Interface). Moab comes with features to reduce energy consumption of a cluster by shutting down nodes that are not utilized.

1.2 eeClust

The evaluation of Moab was performed on the eeClust[1] in Germany.

1.2.1 Hardware

The eeClust (energy efficient cluster) consists of 10 nodes. 5 of these nodes are powered by an AMD CPU (Opteron 6168 @ 1.900 MHz), the other 5 nodes by an Intel CPU (Xeon Nehalem X5560 @ 2.800 MHz). 2 switches are used for networking. An Allnet 4806W takes care of the service network (IPMI) while a D-Link DGS-1210-48 is used for all the other networking tasks. The power consumption of every node is measured through a LMG 450 Power Meter and stored in a database every 100 ms.

1.2.2 Software

Both Torque and Maui are installed on the eeClust as resource manager and job scheduler. During the moab evaluation phase maui and Moab were running parallel.

[1]http://www.eeclust.de/

2 Installation

We installed Moab adaptive hpc suite version 5.4.3. As a resource manager we used Torque which was already installed and used together with Maui.

The installation process consists of two steps:

1. `./configure`
2. `./make install`

Configure accepts several options. For instance we had to specify which resource manager moab should work with (`-with-pbs`).

We didn't use the standard locations for binaries to avoid conflicts with Maui, which should remain the primary scheduler on this cluster. The tools and binary directories were installed to `/sw/moab` and the moab homedir has been set to `/opt/moab`.

Before starting Moab we had to place the license file in the `$MOABHOMEDIR`. To enable Moab on the nodes we had to create a moab.cfg file in each nodes `/etc` folder that consists of only one line on which we had to specify the port Moab uses at the server.

```
SCHEDCFG[Moab]  SERVER=eeclust:42600
```

The default port was already used by Maui. We also had to copy the binaries of the client commands that we were going to use on the nodes.

3 Setup

Most of the configuration takes place inside of the `moab.cfg` which lies inside the `$MOABHOMEDIR`.

3.1 General

To enable collaboration with Torque we had to add the following lines to the configuration file.

```
RMCFG[Moab]  TYPE=PBS
RMCFG[Moab]  SUBMITCMD=/usr/bin/qsub
RMCFG[Moab]  SBINDIR=/usr/sbin
```

3.2 Green Computing

Green computing requires 2 scripts by which moab can monitor and control the power state of each node. The scripts need to be configured on per-resource manager basis. At first we tried to utilize the IPMI-interface that is installed in the Moab tools directory. That way we were able to both monitor and change the power state of each node. However after a second look (green computing wasn't working "right") we saw that further changes were needed. As stated in the documentation the state (not to be confused with the power state) needs to be reported as `Unknown`. This is necessary because a node is still eligible to run jobs when it was powered down by the green resource manager. The actual state of the node is in that case `idle`. Furthermore the IPMI-interface initiates a cold shutdown which is not what we intend.

3.2.1 Scripts

We decided to implement both scripts on our own in python. The cluster query script directly reports the current power states and the node power script initiates a soft shutdown.

Example output of cluster.query.ARCH.py

```
ehmke@eeclust:~/scripts$ ./cluster.query.amd.py
amd1 POWER=OFF STATE=Unknown
amd2 POWER=OFF STATE=Unknown
amd3 POWER=OFF STATE=Unknown
amd4 POWER=OFF STATE=Unknown
amd5 POWER=OFF STATE=Unknown
ehmke@eeclust:~/scripts$ ./cluster.query.intel.py
intel1 POWER=OFF STATE=Unknown
intel2 POWER=OFF STATE=Unknown
intel3 POWER=ON STATE=Unknown
intel4 POWER=ON STATE=Unknown
intel5 POWER=ON STATE=Unknown
```

See appendix A.1 for a listing of these scripts.

3.2.2 Parameters

Green functionality isn't enabled by default which means moab won't use the native resource manager to power down nodes as long as we don't set the `POWERPOLICY` to `OnDemand` in `moab.cfg`:

```
NODECFG[DEFAULT]        POWERPOLICY=OnDemand
```

To reduce the delay between a job submission and a job start when all or most of the nodes are idle and therefore shutdown it is a possible to specify that a subset of the available nodes won't be shutdown although they are idle. `MAXGREENSTANDBYPOOLSIZE 5` means that at any time at least 5 nodes are powered on. We set that parameter to 0 to maximize the energy saving.

```
MAXGREENSTANDBYPOOLSIZE 0
```

Since our cluster consists of 5 AMD and 5 Intel nodes we need to specify different times for the boot and shutdown process. An Intel node for example is shut down 2 times faster then an AMD node. For that reason we created two partitions each having their own native resource manager. We then were able to specify the `NODEPOWERONDURATION` and `NODEPOWEROFFDURATION` on a per-resource manager basis.

Setting up partitions:

```
NODECFG[intel1] PROVRM=intel PARTITION=intel
NODECFG[intel2] PROVRM=intel PARTITION=intel
NODECFG[intel3] PROVRM=intel PARTITION=intel
NODECFG[intel4] PROVRM=intel PARTITION=intel
NODECFG[intel5] PROVRM=intel PARTITION=intel

NODECFG[amd1]   PROVRM=amd PARTITION=amd
NODECFG[amd2]   PROVRM=amd PARTITION=amd
NODECFG[amd3]   PROVRM=amd PARTITION=amd
NODECFG[amd4]   PROVRM=amd PARTITION=amd
NODECFG[amd5]   PROVRM=amd PARTITION=amd
```

Configuring native resource managers:

```
RMCFG[intel]    TYPE=NATIVE RESOURCETYPE=PROV
RMCFG[intel]    CLUSTERQUERYURL=exec:///scripts/cluster.query.intel.py
RMCFG[intel]    NODEPOWERURL=exec:///scripts/node.power.py
RMCFG[intel]    PROVDURATION=80

RMCFG[amd]      TYPE=NATIVE RESOURCETYPE=PROV
RMCFG[amd]      CLUSTERQUERYURL=exec:///scripts/cluster.query.amd.py
RMCFG[amd]      NODEPOWERURL=exec:///scripts/node.power.py
RMCFG[amd]      PROVDURATION=100
```

Specifying the measured poweron- and poweroffduration (see section 4.1):

```
PARCFG[amd]     NODEPOWERONDURATION=1:10
PARCFG[amd]     NODEPOWEROFFDURATION=0:30

PARCFG[intel]   NODEPOWERONDURATION=1:10
PARCFG[intel]   NODEPOWEROFFDURATION=0:10
```

Moab decides whether or not to shutdown a node depending on how long it has been idle. That has the advantage that for example wrong wall-clocktimes don't negatively affect the energy savings. The downside of that is (if the wallclocktime would be correct) although the scheduler has the knowledge that a node **will** be idle for a certain amount of time that knowledge won't be used to shutdown a node. The node has to be idle for the given amount of time first. The corresponding parameter `NODEIDLEPOWERTHRESHOLD` has to be specified in seconds.

```
NODEIDLEPOWERTHRESHOLD   150
```

By default Moab doesn't log power-related events. To enable logging these events we added the following line to `moab.cfg`.

```
RECORDEVENTLIST          +NODEMODIFY
```

4 Measurements

4.1 Energy Saving Potential

Even though the nodes consume more power during the boot or shutdown process (except when shutting down an Intel node – then it's 120 W average power consumption during the shutdown process versus 133 W idle power consumption) it is not significant enough to justify a long idle time. During the 70 second boot time an Intel node consumes an average of 150 W – only 17 W more than during idle time. AMD nodes consume 175 W average during the 70 second boot process as opposed to the 105 W during idle time. When shutting down either node the average consumption is only 120 W.

Table 1 presents the average of 5 complete boot and shutdown procedures. The $T_{boot}, E_{boot}, T_{shutdown}$ and $E_{shutdown}$ columns show the duration and energy consumption of the corresponding procedures. P_{idle} and P_{off} show the average power consumption when a node is idle or off. T_{min} stands for the minimum time a node has to be idle until the energy consumption is higher than it would have been if the node would be off (which includes one boot and one shutdown procedure).

	T_{boot}	E_{boot}	$T_{shutdown}$	$E_{shutdown}$	P_{idle}	P_{off}	T_{min}
intel	70 s	10,5 kJ	10 s	1,2 kJ	133 W	8 W	88.48 s
amd	70 s	12,25 kJ	30 s	3,6 kJ	105 W	8 W	155.15 s

Table 1: Duration, power and energy values for the different node states

How to calculate T_{min} (Break-even point):

$$T_{min} = \frac{P_{off} \times T_{boot} - E_{boot} + P_{off} \times T_{shutdown} - E_{shutdown}}{P_{off} - P_{idle}}$$

Assumed an Intel node is idle for ca. 90 seconds (The exact time to reach the break-even point would be 88.48 s) and will be shut down to save energy the consumed energy consists of: 10 s shutting down the node, 10 s being off and 70 s booting the node = 1,2 kJ + 0,08 kJ + 10,5 kJ = 11,78 kJ. Otherwise the consumed energy would have been 90 s * 133 W = 11970 J (11,97 kJ). That means even though the node was only shutdown for 10 seconds it was still enough time to save some energy.

Since the AMD nodes not only consume more energy during both the shutdown and boot process but also consume 28 W less during idle compared to an Intel node they need to be idle for a longer time before it pays out to shut them down.

If an AMD node is idle for ca. 160 seconds (exact time would be 155.15 s) and not shutdown the consumed energy is 160 s * 105 W = 16800 J (16,8 kJ). If shutdown and therefore powered off for 60 seconds the energy consumption would consist of 3,6 kJ (shutdown) + 0,48 kJ (off) + 12,25 kJ (boot) = 16,33 kJ.

4.2 Scenario

All jobs solve partial differential equations using a parallel application called `partdiff-par`. The program is started with 1000 interlines and between 1000 and 4000 iterations depending on how many nodes are used. With 1000 interlines a Matrix with the dimension 8008 will be calculated which uses 0.513 gigabytes memory. The jobs run between 5 and 20 minutes while the wallclocktime is set to 30 minutes. In total 8 different jobs were used. Both for Intel and AMD 4 jobs that require 1, 2, 3 and 4 nodes. Each of these jobs got queued 4 times but not all at once. The time between the first and the second submission of the 8 jobs was 60 seconds followed by 10 minutes, 200 seconds and 400 seconds.

Since the AMD nodes have more CPUs than the Intel nodes the jobs should finish faster on them resulting in some idle time towards the end of one run. That's because `partdiff-par` scales well with more CPUs and in this case the 12 AMD cores running at 1.9 GHz perform better than the Intel Xeon X5560 running at 2,8 GHz.

To produce different workloads we changed the backfilling algorithms used by the Moab scheduler. We used these 3 modes:

`FIRSTFIT` The first job that fits into the current backfill window will be started.

`BESTFIT` For each job that fits into the current backfill window a *degree of fit* will be calculated. The Job with the best *degree of fit* will be started.

`GREEDY` A *degree of fit* for all possible combinations of jobs that fit into the backfill window will be calculated. The best combination will be started.

Each algorithm was tested once with and without green enabled. Our main interest is the overall consumed energy for each run but also if the activation of green computing negatively effects the scheduling regarding runtime of the accumulated jobs. To evaluate this we used the Energy-Delay-Product (EDP or $E \cdot T$) which is defined as follows:

$$EDP = Joule \cdot seconds$$

It is not always desirable to shutdown nodes to save energy. If doing so, it results in much longer execution time because of the produced overhead. Most of the time fast scheduling and execution time is the most important criterion. If the overhead due to the `OnDemand` `POWERPOLICY` becomes too large it will result in a bad EDP score.

4.3 Variations

Unfortunately there occured variations in the runtime of `partdiff-par` during the execution of the configurations with `BACKFILLPOLICY` set to `BESTFIT` and `GREEDY`. The jobs needed considerably longer to finish, especially when the `POWERPOLICY` was set to `STATIC`. With `POWERPOLICY` set to `OnDemand` the jobs needed about 10 % longer (in total) and with `POWERPOLICY` set to `STATIC` it was about 30 %.

4.4 Comparisons

The run with static powerpolicy was as expected the fastest. 7061 seconds (117 minutes) after the first job was submitted the last job finished. During that time all nodes together consumed 14959,764 kJ.

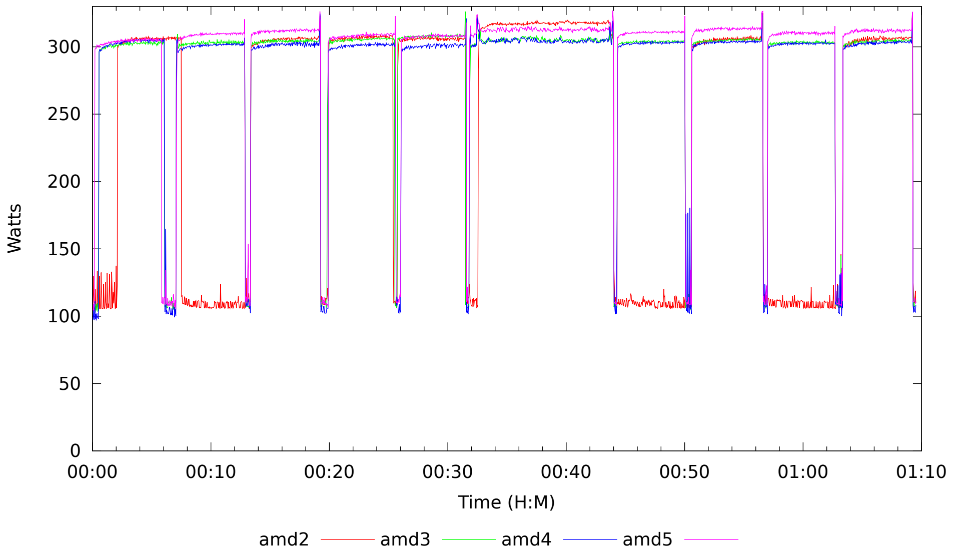

Figure 1: power consumption over time for **AMD** (`POWERPOLICY STATIC`)

With `POWERPOLICY` set to `OnDemand` the overall runtime (7587 seconds) was increased by 7.5 % compared to the `STATIC` run with. However during that time the 8 nodes together only consumed 13736.383 kJ which is a decrease of 8.2 %. Although that sounds not too much it has to keep in mind that the workload does not include large idle times which are, depending on the application field, likely to happen in real environments. In this case the last AMD job which used all 4 nodes finished 44 minutes before the last Intel job finished. That means the AMD nodes were idle 35 % of the time.

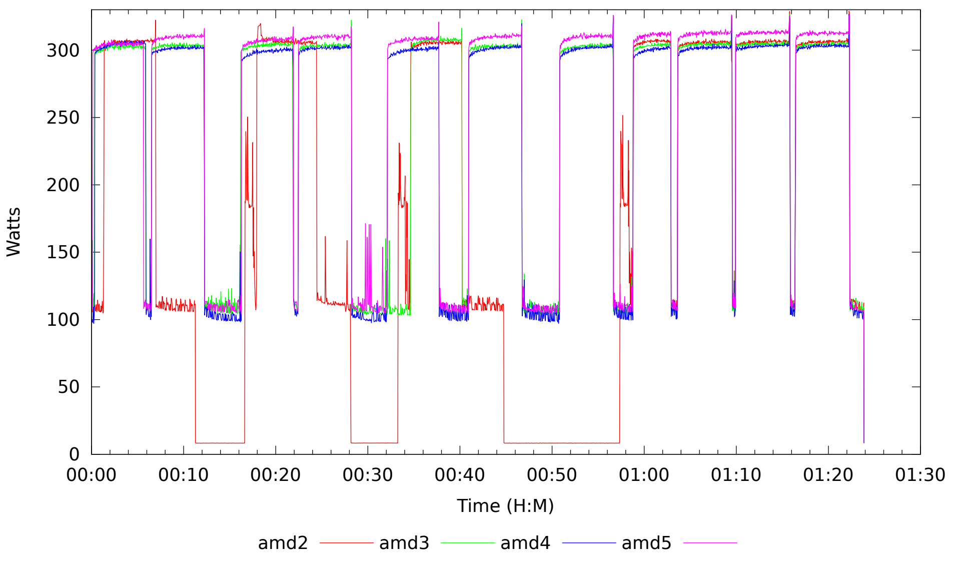

Figure 2: power consumption over time for **AMD** (`POWERPOLICY OnDemand`)

Looking at the graph representing the power consumption over time (see figure 4) one can see that shutting down nodes produces a certain overhead. In this scenario it is easily visible (see e.g. minute 30) because the jobs run between 5min and 20min whereas a complete reboot already takes 2min.

This is even better visible looking at figure 5 where the total power consumption over time accumulated of all nodes is compared between a test run using `POWERPOLICY OnDemand` and one using `POWERPOLICY STATIC`. At the first marked point in the chart one can see that the test run using `OnDemand POWERPOLICY` needs noticeable longer to get to a point where the `STATIC` run has been before (all 8 nodes active). At the second marked point it is visible at a first glance that the different `POWERPOLICY` has affected the scheduling. While during the `STATIC` run the cluster is for almost 30 minutes at full capacity during minute 15 and 45 the `OnDemand` run shows a different picture. From that point on the graphs continue to be very different not only because the `OnDemand` run consumes less energy but also because the scheduling has changed. Most of the time the `OnDemand` graph remains below the `STATIC` graph which is why the overall consumed energy is lower although the overall runtime was longer.

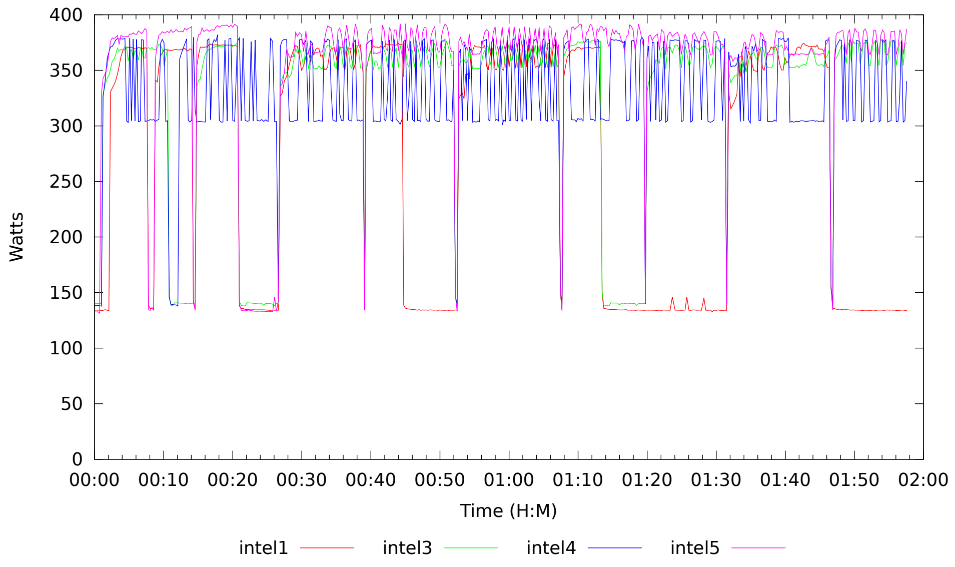

Figure 3: power consumption over time for **Intel** (`POWERPOLICY STATIC`)

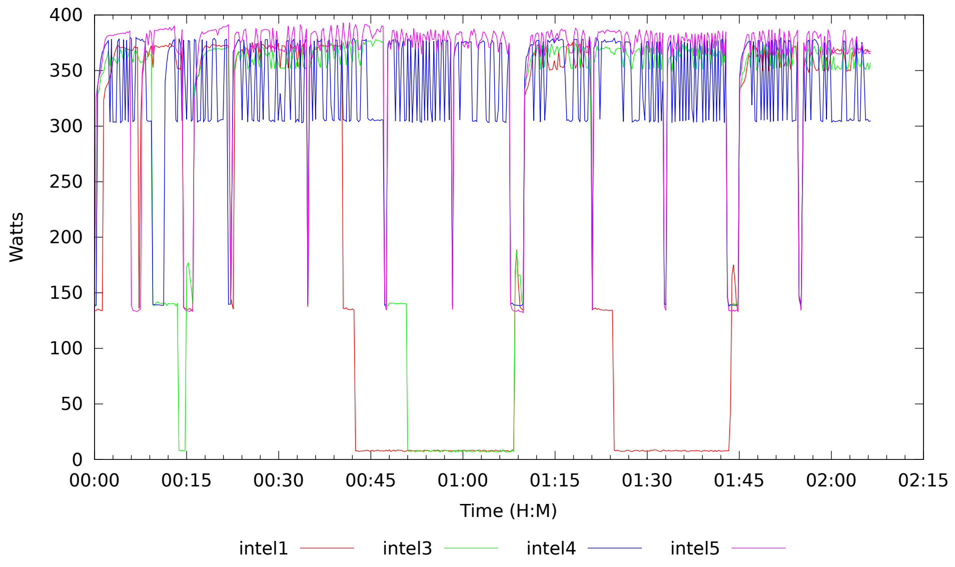

Figure 4: power consumption over time for **Intel** (`POWERPOLICY OnDemand`)

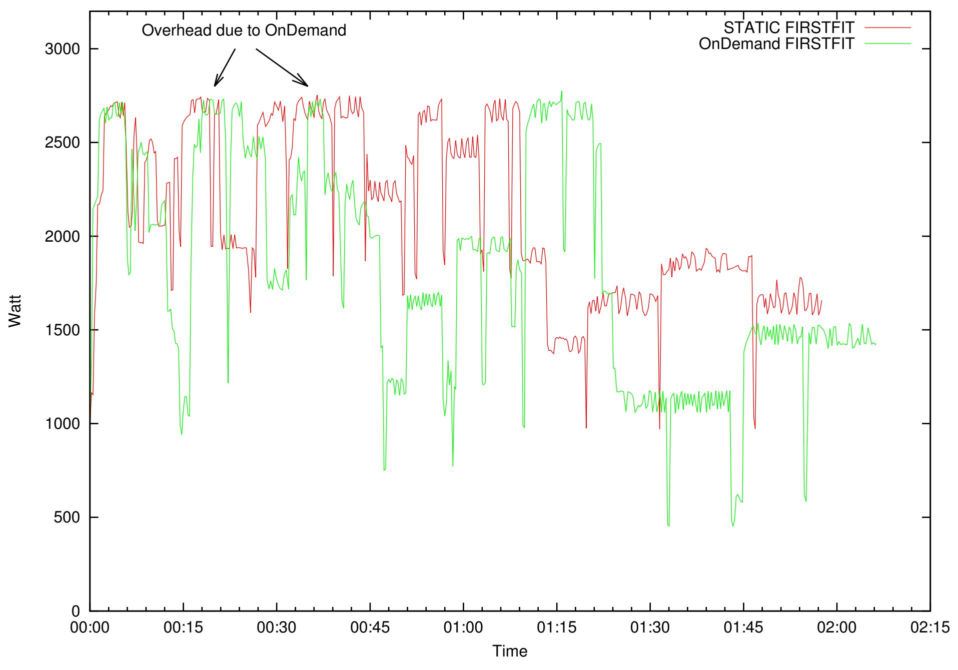

Figure 5: Process of the power consumption (`STATIC` vs. `OnDemand`)

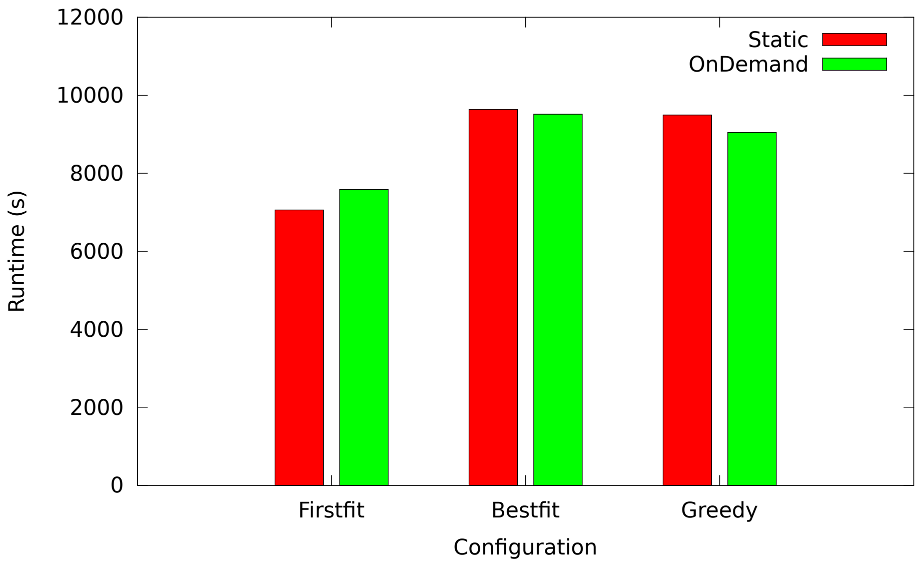

Figure 6: Total time elapsed in seconds during the 6 different test runs

The EDP of the `OnDemand` run is slightly better than the `STATIC` run (104217 vs. 105630, see figure 8) which is a result of the difference between the consumed energy being higher than the difference between the runtimes. The other test runs resulted in similar results as figure 6, 7 and 8 show.

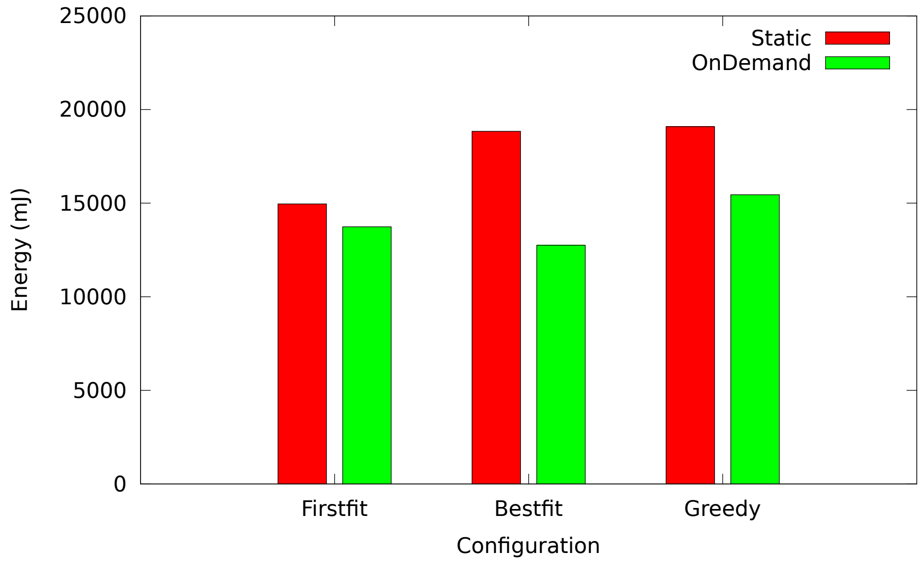

Figure 7: Total energy consumption in mJ of each different configuration

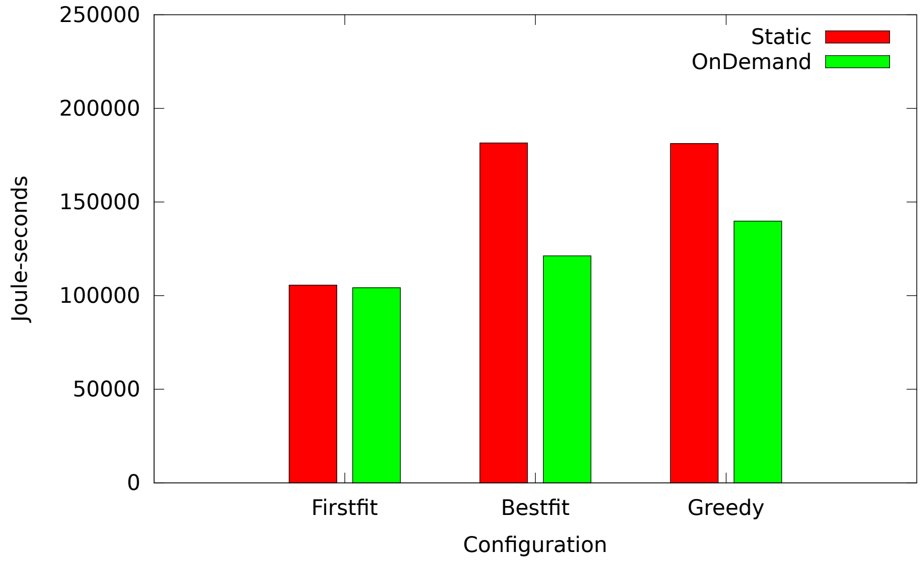

Figure 8: Energy-Delay-Product of the 6 different configurations (smaller is better)

4.5 Blizzard

All our measurements so far took place on the eeClust which uses “normal” hardware. The effect of that is that for example T_{min} is extremely low. The hardware ist almost predestinated to make use of Moab’s green features. Large-Scale clusters often have special hardware and operating systems. These systems usually take much longer to boot or shutdown. Furthermore the workload on these clusters is different to out artificially created workload. In this section we will take a closer look at these differences and try to estimate the potential savings.

The supercomputer Blizzard of the DKRZ consists of 264 IBM Power6 nodes. Its peak performance is 158 TeraFlop/s. Each of the 264 nodes has 16 Dual-core CPUs (8448 cores total). The cluster has more than 20 TeraByte main memory and uses an Infiniband network.

4.5.1 Energy Saving Potential (Blizzard)

Table 2 shows some facts about a typical IBM Power6 node installed in the Blizzard supercomputer. The variations in both boot and shutdown time depend on how many nodes at the same time are booted or shutdown. The power consumption when a node is powered off is just assumed – we had no real values for that node state.

state	duration	power consumption
boot	15,5 - 30 min	2550 W - 4250 W
shutdown	5 - 6 min	2550 W - 4250 W
idle	-	2550 W - 3083 W
off	-	ca. 100 W

Table 2: Duration and power consumption of the node states

Since the values in Table 2 are not as accurate as for the eeClust and the boot and shutdown times vary we calculated the worst and best case scenario for T_{min}.

T_{min} (Break-even point) for the Blizzard (see section 4.1):

worst case: 3659 s (ca. 61 min)
best case: 2083 s (ca. 35 min)

4.5.2 Load investigation

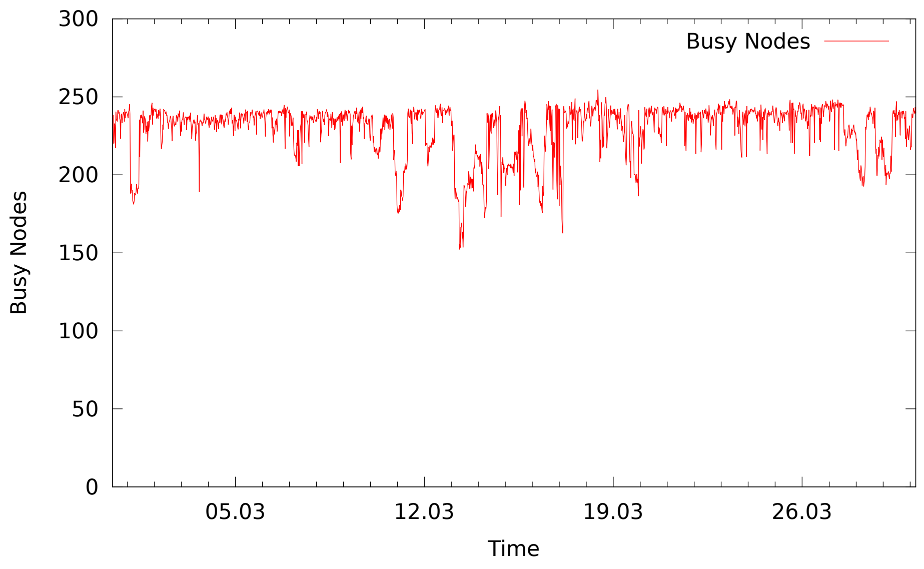

Figure 9: Load over one month (March 2011) on the DKRZ Blizzard cluster

To investige how much potential for savings is available we monitored the load of the DKRZ Blizzard supercomputer over 1 month. See appendix A.3 for the script that generated the charts (figure 9, 10 and 11). This chart differentiates only idle and busy nodes. That means if *node A* is idle for 30 minutes and *node B* is idle for 60 minutes right after *node A* has continued to be busy it appears in the chart like one idle node for 90 minutes. In Figure 9 one can see that there is definately potential. For example around the middle of that month the load drops towards 180 busy nodes several times. Since these windows are relatively long (several hours - one day) it would most likely pay out to shutdown some of the nodes.

To get an idea how long the timespans are when some nodes could be shutdown we took a closer look at some thresholds. The idea behind this is that one node can be idle for a very long time in total but if that time consists of many very short times it may not be worthwhile to shutdown that node. Figure 10 shows that when there are between 1 and 10 nodes idle the length of that idle time is about 6 hours. That average idle time is much shorter for 10 to 15 idle nodes. Thus if there are 15 nodes idle on an average 10 of them will stay idle for 6 hours whereas the other 5 will compute again in 2 hours. These interrupts of the often very long total idle time (see figure 11) make it harder to save energy in terms of shutting down one node. In that case good scheduling is needed to maximize the length of the average idle times without increasing the total idle time.

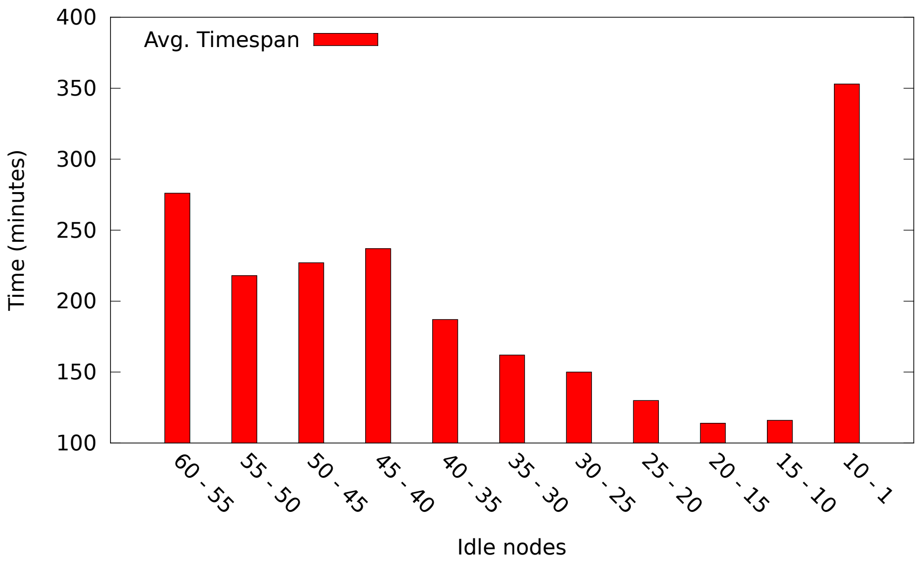

Figure 10: Average length of the timespans the nodes between the thresholds spent idle

If we assume that the *perfect* scheduler is used we can use the data of figure 9 together with the T_{min} (see section 4.5.1) to calculate the possible savings. During the timespan shown by figure 9 5668949 CPU hours were available and 636310 of these were spent idle which is 5.986%. Not all of these *idle* CPU hours could have been avoided by shutting down certain nodes. We had to subtract the boot and shutdown times and skip timespans that were shorter than T_{min}. That left us with 536297 CPU hours which could have been spent shutdown (5.045%) which equals 22793 kilowatt hours (kWh). If the energy costs are about 0,13 € per kilowatt hour 2963 € could have been saved during this particular timespan.

5 Summary

Shutting down one node to save the energy is a very drastic action. On the eeClust it wasn't much of a big deal since it is a very small cluster with ordinary operating systems. Other systems may have much longer boot and shutdown times which makes it harder to profit from shutting down nodes. If the node is not shutdown long enough to get to a break-even point wherefrom the energy consumption is lower than if turned on it has 2 negative effects: No energy was saved although the node was shutdown and the scheduling was affected negatively. Moab allows the user to specify how long a node may be idle before it will be shutdown. The big problem that there is no

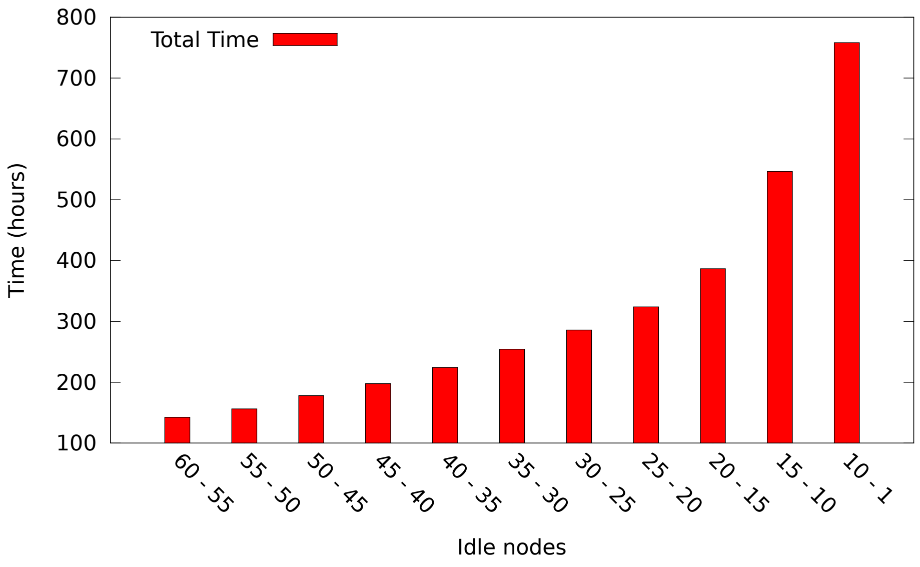

Figure 11: Total amount of time the nodes between the certain thresholds spent idle

way to ensure that in the near future there will be no eligible jobs that cause a node to boot up again remains. Especially in environments with a near 100% workload it is difficult to use the remaining time frames for savings due to shutting down nodes. Shutting down idle nodes is therefore no general purpose-solution to save energy. It must be evaluated if it's a feasible option or if it would slow down the scheduling in a way that the overall energy consumption becomes worse. Finally one can say that this works best at specific workloads. For example if there are continually very large jobs which require many smaller jobs to finish before enough resources are available to start these jobs. Another example would be a workload that exhibits seasonal variations. It shouldn't be a problem to adapt to these variations.

I would like to thank Carsten Beyer and Ernst-Gunar Ortlepp for their datailed replies to my questions and also the Moab team for their support and the evaluation license.

References

[1] *Adaptive computing*. Mar. 2011. URL: `http://www.adaptivecomputing.com/`.

A Scripts

A.1 Clusterquery

The only difference between `cluster.query.amd.py` and `cluster.query.intel.py` is the array of hostnames in line 18.

```
#! /usr/bin/python
# used by moab
# script to report cluster query data

import subprocess
import sys

def bash(cmd,cwd=None):
        retVal = subprocess.Popen(cmd, shell=True, \
                stdout=subprocess.PIPE, cwd=cwd). \
                stdout.read().strip('\n').split('\n')
        if retVal==['']:
                return 0
        else:
                return retVal

def main():
        nodes = ['intel1','intel2','intel3','intel4','intel5']
        for n in nodes:
                ic = 'ipmitool -H %s-ipmi -U X -P X -L USER ' % (n,) +\
                          'power status | cut -d " " -f4'
                print n + ' POWER=' + str(bash(ic)[0]).upper() + \
                          ' STATE=Unknown'
        return 0

if __name__ == "__main__":
        sys.exit(main())
```

A.2 Node power

```
#! /usr/bin/python
# used by moab
# script to power on or off nodes

import subprocess
import sys

def bash(cmd,cwd=None):
        retVal = subprocess.Popen(cmd, shell=True, \
                stdout=subprocess.PIPE, cwd=cwd). \
                stdout.read().strip('\n').split('\n')
        if retVal==['']:
                return 0
        else:
                return retVal

def main(argv=None):
        if argv is None:
                argv = sys.argv

        if len(argv) != 3:
                print 'usage: ' + str(argv[0]) +
                        ' <node>[,<node>] <ON | OFF>'
                return 1

        nodes = argv[1].split(',')
        mode = str(argv[len(argv) - 1]).lower()

        for node in nodes:
                ipmicmd = "ipmitool -U X -P X -H" + \
                        " %s-ipmi power %s" % (node, mode)
                if ipmicmd == 0:
                        return 1
                bash(ipmicmd)

        return 0

if __name__ == "__main__":
        sys.exit(main())
```

A.3 Blizzard accounting

```
#!/usr/bin/python
import sys
import time
import math
import os

ncpu = 32 # CPU cores per node
mintime = 3659 # The calculated T_min

def open_accounting(filename,path):
  try:
    f = open(path + filename)
  except IOError:
    print IOError
    return 1
  return f

def close_accounting(f):
  f.close()
  return 0

def get_ts(date):
  return int(time.mktime(time.strptime(date, "%y%m%d %H%M%S")))

def get_start_end(l):
  start = 0
  end = 0
  selist = []
  for e in l:
    if start == 0 or e[0] < start:
      start = e[0]
    if end == 0 or e[1] > end:
      end = e[1]
  selist.append(start)
  selist.append(end)
  return selist

def load_accounting(f,joblist):
  global ncpu
  for l in f:
    p = l.split()
    hc = 0
    start = 0
    end = 0
    for i,e in enumerate(p):
      if e == 'EB':
        start = get_ts(str(p[i + 1] + ' ' + p[i + 2]))
      if e == 'EE':
        end = get_ts(str(p[i + 1] + ' ' + p[i + 2]))
      if e == 'HC':
        if int(p[i + 1]) > 0: # normal, hosts
          hc = int(p[i + 1])
        elif int(p[i + 1]) == 0: # serial job
          hc = float(1 / ncpu)
        elif int(p[i + 1]) < 0: # special cases
          if int(p[i + 1]) == -99:
            hc = 0
          elif int(p[i + 1]) == -100:
            hc = 0
          else:
            hc = float(1 / \
            math.fabs(float(p[i + 1])))

    joblist.append([start, end, hc])

  return joblist

# sort list by timestamp (joblist[0][0])
def sort_list(joblist):
  return sorted(joblist, key = lambda element : element[0])


def get_active_hosts(j,l):
  hosts = 0.0
  for e in l:
    if j > e[0] and j < e[1]:
      hosts += e[2]
  return hosts

def process_accounting(l, step):
  n_host_sum = 0
  n_host_sum_inner = 0
  n_entries = 0
  n_entries_inner = 0

  inner_start = 0
  inner_end = 0

  selist = get_start_end(l)
  start = selist[0]
  end = selist[1]

  try:
    f = open('timeline','w')
  except IOError:
    print IOError
    return 1
```

```
boundaries = []
for i in range(248):
  boundaries.append([i + 1,0,[],0])

for i in range(start, end + 1, int(step)):
  hc = get_active_hosts(i,1)
  print str(i) + ' ' + str(hc)

  n_host_sum += hc
  n_entries += 1

  # Calculate timespans, total idl CPUh etc

  inner_start = 1298889239
  inner_end = 1301461239
  # Exclude some time at the beginning and the end of the month
  if i > inner_start and i < inner_end:
    n_entries_inner += 1
    n_host_sum_inner += hc

    # For each boundary
    for b in boundaries:
      # if the current hostcount is lower than this boundary
      if float(hc) < float(b[0]):
        # if it is the first time the hostcount
        # deceedes the boundary, store the time
        if b[1] == 0:
          b[1] = int(i)

      # if the current hostcount is higher then this boundary
      if float(hc) > float(b[0]):
        # if the boundary has been deceeded before
        # save the elapsed time
        if b[1] > 0:
          b[2].append(int(i) - int(b[1]))
          b[3] += 1
        b[1] = 0

# special case, boundary was deceeded but never exceeded
for b in boundaries:
  if b[1] > 0:
    if b[3] == 0:
      b[2].append(end - b[1])
      b[3] = 1


print 'AVG ACTIVE HOSTS: ' + str(float(n_host_sum) / \
          (float(n_entries)))
print 'AVG ACTIVE INNER HOSTS: ' + str(float(n_host_sum_inner) / \
          (float(n_entries_inner)))
print 'AVG IDLE TIME SPANS: '

# Generate gnuplot data
for i,b in enumerate(boundaries):
  print str(boundaries[i][0]) + ' ',

  if b[2] > 0 and b[3] > 0:
    print str(float(sum(b[2])) / float(b[3])),
    print str(sum(b[2])) + ' count: ' + str(b[3])
  else:
    print '0',
      print str(sum(b[2])) + ' count: ' + str(b[3])

  idletime = 0
  realidletime = 0
  for b in boundaries:
    # for each idle timespan of this node
    for t in b[2]:
      if t > mintime:
        realidletime += t
        idletime += t - mintime

  print 'IDLETIME: ' + str(realidletime)
  print 'SHUTDOWNTIME: ' + str(idletime) + ' during: ' + \
    str(inner_start) + ' - ' + str(inner_end)
  return 0


def main(args):
  if len(args) != 3:
    print "Provide folder with accounting files as argument!"
    print "Resolution in seconds."
    print args[0] + " <accounting folder> <resolution>"
    return 1

  path = args[1]
  step = args[2]

  if path[len(path) - 1] != '/':
    path += '/'

  joblist = []
  listing = os.listdir(path)
  for infile in listing:
    print "Loaded: " + infile
    f = open_accounting(infile,path)
    load_accounting(f, joblist)
    close_accounting(f)

  sjoblist = sort_list(joblist)
  return process_accounting(sjoblist, step)

if __name__ == '__main__':
  sys.exit(main(sys.argv))
```