Storage and Compute Resource Management via DYRE, 3DcacheGrid, and CompuStore

Ioan Raicu, Ian Foster

1. Overview

Both the industry and academia have an increase demand for good policies and mechanisms to efficiently manage large data sets and large pool of compute resources that ultimately perform computations on the data sets. We define large datasets to contain millions of objects and terabytes of data; these kinds of datasets are common in the Astronomy domain [1, 2, 3], Medical Imaging domain [4], and many other science domains. Large pool of compute resources could be anything from 10s to 1000s of separate physical resources geographically distributed. In order to enable the efficient dynamic analysis of large datasets, we propose three different systems that can interoperate with each other in order to offer a complete storage and resource management solution.

The first system is DYRE, DYnamic Resource pool Engine, which is an AstroPortal specific implementation of dynamic resource provisioning [9]. DYRE essentially handles all the necessary tasks associated with state monitoring, resource allocation based on observed state, resource de-allocation based on observed state, and exposing relevant information to other systems. The main motivations behind dynamic resource provisioning are:

- Allows for finer grained resource management, including the control of priorities and usage policies
- Optimize for the grid user's perspective: reduces delays on per job scheduling by utilizing pre-reserved resources
- Give the Resource Provider the perception that resource utilization is higher than it would normally be
- Opens the possibility to customize the resource scheduler per application basis, including the use of both data resource management and compute resource management information for more efficient scheduling
- Reduced complexity to the application developer as the details of the dynamic resource provisioning are abstracted away

The second system is 3DcacheGrid, Dynamic Distributed Data cache for Grid application, which allows applications to achieve a good separation of concerns between their business logic and the complicated data management task of large data sets. We propose to utilize a tiered hierarchy of storage resources to cache data in faster (i.e. higher throughput, lower latency) and smaller data storage tiers. 3DcacheGrid essentially handles the data indexing necessary for efficient data discovery and access, as well as decides the cache content (the cache eviction policy). The 3DcacheGrid engine offers efficient management for large datasets (in both number of files, size of datasets, number of storage resources used, number of replicas, and the size of the query performed), and at the same time, offer performance improvements to those applications which have data locality in their respective workloads and data access patterns through effective caching strategies. We also expect to improve data availability by giving access to the cached data without the need for the original data.

The third system is CompuStore, a work scheduler that uses both the storage and compute resource management systems 3DcacheGrid and DYRE. The goal of CompuStore is to make the best scheduling decisions given some work, the available compute resources (DYRE), and the available data caches (3DcacheGrid) which are stored on the compute resources.

2. Implementation Details

Two other components (CompuStore and DYRE) are tightly coupled with the 3DcacheGrid engine. The CompuStore scheduler uses the cache information to make better scheduling decisions for the applications that use the 3DcacheGrid engine from the available resource pool maintained by DYRE. Figure 1 shows an overview of the 3DcacheGrid Engine plus its logical relationship to DYRE and CompuStore.

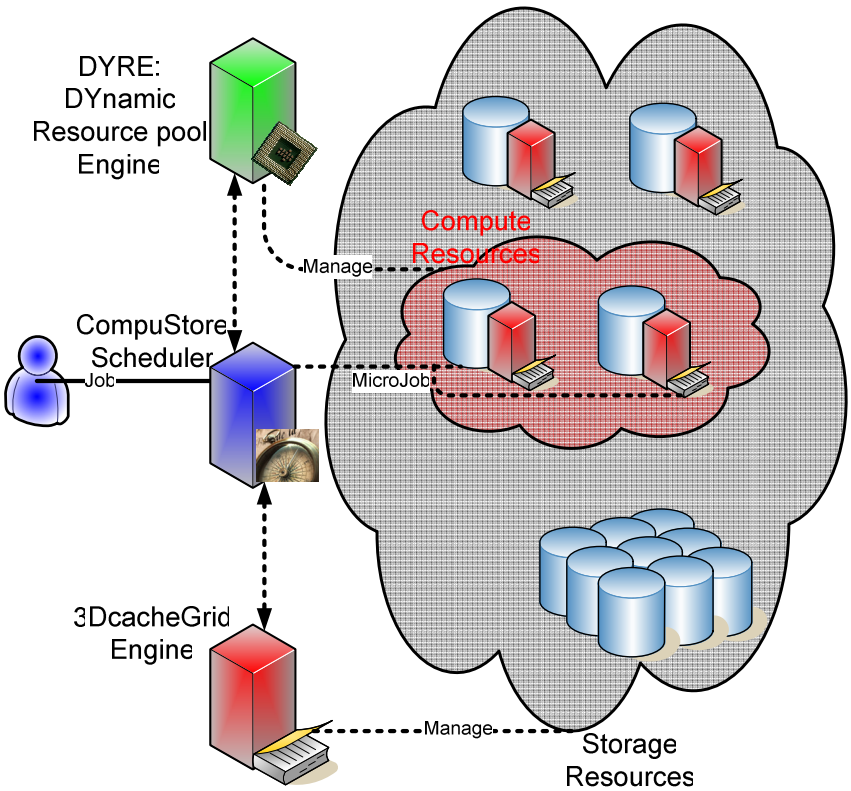

Figure 1: Overview of the 3DcacheGrid Engine, CompuStore, and DYRE

2.1. DYRE: DYnamic Resource pool Engine

We use two basic components to enable the DYRE functionality: web services and GRAM from GT4. We use web services to poll for information regarding the state of the application (in our case, the AstroPortal), and then we use GRAM to provision the necessary resources.

The state information that DYRE polls on the AstroPortal is:

- Stack queue length (qLength): the number of stacks that still need to be performed
- Active stacks (ActiveStacks): the number of stacks that are actively being stacked
- Number of registered workers (RegisteredWorkers): number of compute resources actively participating in stacking operations

The logic behind the resource provisioning is as follows:

```
if (qLength == 0)
{
  lastQLength = qLength;
  //nothing to do...
}
else if (qLength - lastQLength > 0)
{
  int newQLength = qLength - lastQLength;
  int host2Allocate = (int) Math.min(Math.ceil(newQLength*1.0 / 10.0), hostCount - getAllocatedWorkers());
  if (host2Allocate > 0)
  {
        int wallTime = (int) Math.min((int) Math.max(minWallTime, Math.ceil((newQLength / host2Allocate) / 60)), maxWallTime);
        Thread thread[] = new Thread[host2Allocate];
        //allocate resource 1 at a time
        for (int ii = 0;ii<host2Allocate;ii++)
        {
           //start thread to allocate a resource
           thread[ii] = new GramSubmit(this, executable, wallTime, hostType, 1, contact, gramID, ii);
           thread[ii].start();
```

```
            }
            lastQLength = qLength;
  }
  else
            //The maximum allowed resources have already been allocated, nothing to do
}
else
  //qLength has not changed, nothing to do...
```

Basically, the above pseudo code shows the simplistic use of the qLength state information retrieved from the AstroPortal being used to determine the number of resources needed to allocate and for the duration that the resources are needed.

There are some user configurable parameters, that are currently being set to the following values for the AstroPortal:

- minWallTime: minimum amount of time to reserve a resource for (60 minutes)
- hostCount: maximum nuber of resources to reserve (32)
- maxWallTime: maximum amount of time to reserve a resource for (1 day)
- executable: script that starts up the worker with the appropriate arguments (/home/iraicu/java/AstroWorker.1.1/run.worker.sh)
- hostType: type of resources to reserve (ia32-compute)
- contact: address of the GRAM gateway (tg-grid1.uc.teragrid.org)

Furthermore, we currently allocate 1 resource with each GRAM job to ensure that our job submission gets priority in the wait queue since our experience tells us that the smaller the job (lesser resources and lesser time reservation) the faster it makes it through the wait queue that GRAM jobs have to wait in.

Finally, the state information of the resources once they are registered with the AstroPortal is simply an order linked list, which gives us logarithmic cost for the basic insert, find, delete operations, and linear time for retrieving the entire list of registered resources. Having this list sorted was a critical decision to enable the CompuStore to more efficiently make the scheduling decision which would couple the computation and storage resources together.

2.2. CompuStore Scheduler

The CompuStore scheduler is currently designed to be tightly coupled with DYRE and 3DcacheGrid Engine, but the hope is that it can be generalized in the future. The goal of CompuStore is to make the best scheduling decisions given a stacking description, the available compute resources (DYRE), and the available data caches (3DcacheGrid) which are stored on the storage resources.

Given a stacking description, CompuStore first looks up in the 3DcacheGrid to locate any cache that might exist for the stacking in question. At the same time, it constructs a data structure that contains the cache information. This data structure contains two trees that sort its elements by their key. One tree has the key as the location of the storage resource with the value being the number of different caches to be found; the other tree has the key as the number of different caches to be found at the respective storage resource, while the value contains a list of storage resources that have the corresponding number of caches. This data structure allows us to efficiently (logarithmic time) insert, update, and remove elements into both trees, and at the same time, allows us to access the storage resources with the most caches in constant time. We needed a data structure that kept the ordering for both the storage resource and the number of caches so we could efficiently perform the scheduling decision.

The scheduler now can simply use the above data structure to find the storage resource (constant time) that will have the most cache hits, package it together, update the data structure (n log n with the number entries in the package), and send it to the available compute resource that is closest to the storage resource (in the case of the AstroPortal, the compute resource would also contain the storage resource).

2.3. 3DcacheGrid Engine: Dynamic Distributed Data cache for Grid Applications

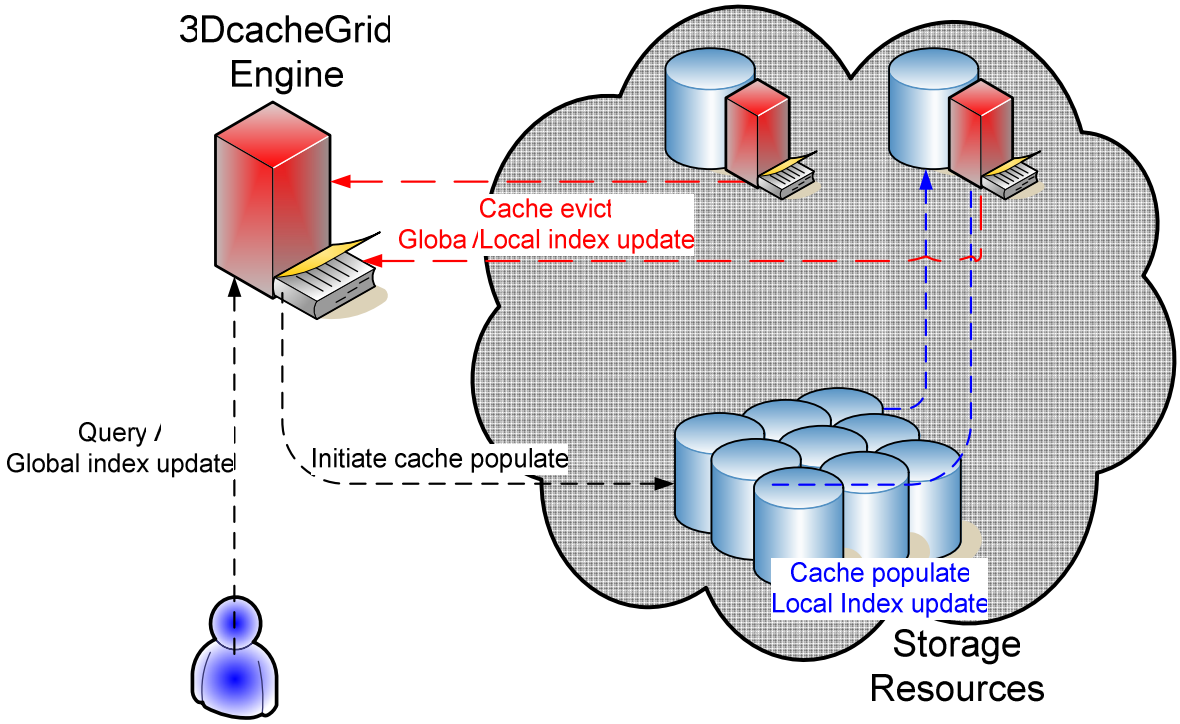

Figure 2 shows more in depth the 3DcacheGrid engine and how it manages the storage resources. We have implemented a hybrid centralized/de-centralized management system for the 3DcacheGrid, but will consider a more distributed management approaches in the event that we face scalability issues in managing large datasets. The hybrid centralized/de-centralized management refers to the fact that there is both a centralized index for ease of finding the needed data, and a decentralized set of indexes (one per storage resource) that manage the actual cache content and cache eviction policies. This hybrid architecture should allow the 3DcacheGrid engine to be both efficient and scalable.

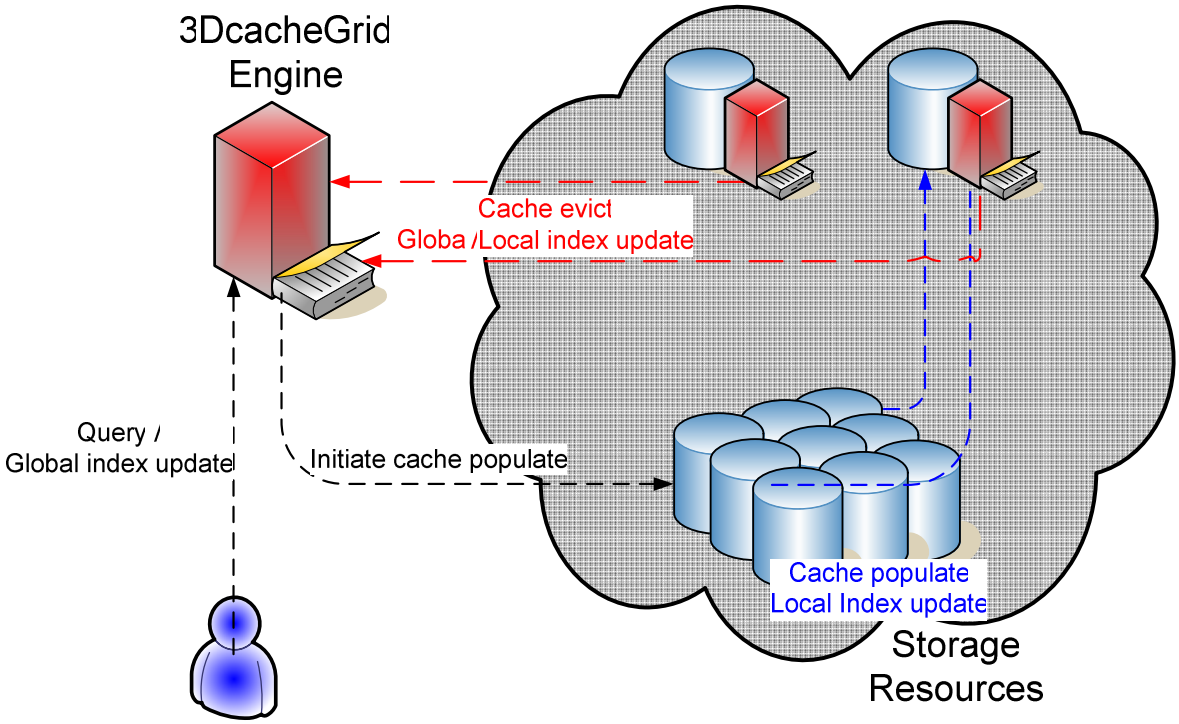

Figure 2: Overview of the 3DcacheGrid Engine

2.4. Usage Scenario

Our first usage scenario will be the AstroPortal [6, 5] and the SDSS DR4 dataset [8]. The AstroPortal is a Web Services-based system that uses grid computing to federate large computing and storage resources for dynamic analysis of large datasets. In essence, the AstroPortal does resource management on both 1) computational resources and on 2) data resources; the 3DcacheGrid can reduce the AstroPortal's data management complexity significantly by off-loading it to the 3DcacheGrid. The AstroPortal is deployed on the TeraGrid distributed infrastructure; it uses the Sloan Digital Sky Survey (SDSS), DR4 dataset, which comprises about 300 million objects dispersed over 1.3 million files, a total of 3 terabytes of compressed data or 8 terabytes of uncompressed data. Although the SDSS DR4 dataset is large, it has the nice property that the data does not change very often. Thus, this property simplifies the data management problem, since the system does not need to provide sophisticated schemes to guarantee replica consistency. Furthermore, the property can also be exploited to design better data staging mechanisms. Some of the key features and expectations of the AstroPortal, SDSS DR4 dataset, and the TeraGrid testbed that make them a good test case for the 3DcacheGrid are: 1) SDSS DR4 dataset is read-only, and when the DR5 dataset will be available, it will be a superset of the DR4 dataset; 2) we expect to find data locality in the data access patterns [7], allowing caching strategies to be the most effective; 3) the typical data access for the "stacking" operation implemented by the AstroPortal will involve the reading and processing of many small pieces of data [5, 6], which means that both low latency to the storage resource as well as parallelization (i.e. having multiple replicas of the data among different caches on different resources) of the data are very important, and 4) The TeraGrid testbed has significant performance and scalability differences among its different storage resources [**Error! Reference source not found.**] giving 3DcacheGrid a good opportunity to optimize the data access through caching from the globally accessible storage resource to the locally accessible storage resource.

3. Performance Results

The experiments that are in this section (Figure 3 - Figure 9) were all done in order to estimate the overall overhead that the 3DcacheGrid Engine and the CompuStore scheduler incurs in addition to the time needed to perform the necessary stacking operation (which was NOT measured in these experiments). There are three set of experiments, each measuring the performance as we varied the Stacking Size on the x-axis (1 – 32K) in relation to one of three parameter spaces plotted on the x-axis: 1) resource pool size, 2) index size, and 3) replication level. Each set of experiments is then broken down into two performance metrics, 1) response time in seconds to complete the data

management and scheduling decision, and 2) the achieved throughput for the data management and scheduling decision in terms of stacks per second.

The parameter space, measured metrics, and assumptions are explained in more details below:

- Measure metrics (z-axis):
  - Time (sec): the time it took to take the corresponding stacking description, lookup the stackings in the index, schedule the stackings to be performed, update the index accordingly with new cache entries, and update the index with cache eviction entries. Essentially, the only thing left is the time needed to transmit the stacking description, perform the actual stacking, and transmit the results back to the user.
  - Throughput (stacks/sec): the number of stacks per second that can be achieved performing all of the same operations that the time metrics captured above.
- Parameter Space (x-axis and y-axis):
  - Stacking size: the size of the stacking in the number of images to be stacked together
  - Resource pool size: the number of storage resources being managed
  - Index Size: the number of entries in the index (this is dependent on the resource pool size an the resource pool capacity)
  - Replication Level: the number of cached copies among different storage resources
- Assumptions:
  - Resource Capacity: the size of the available local disk which will be used for the data cache

Each set of experiments had a certain set of assumptions, which are each clearly described prior to the corresponding figures.

3.1. Stacking Size vs. Resource Pool Size

Assumptions:

- Stacking size: 1 – 32K
- Resource Pool Size: 1 – 32K
- Resource Capacity: 100GB (100,000MB / 6MB ~ 16,667 entries)
- Index Size: maximum of 1.3 million entries, depending on the number of resources used
- Replication Level: 1 cache per resource (unique caches, 1-1 mapping from data archive and cache)

Figure 3 shows the time to complete stackings of size 1 – 32K as we varied the resource pool size from 1 – 32K. All axes are on a logarithmic scale, which shows that a large part of the parameter space is increasing at a relatively linear rate, with the exception of very large stackings and very large resource pool sizes. In practice (the current AstroPortal deployment), we could have stackings as large as we tested in this experiment, but the number of resources would probably be in the range of 10~100, for which the performance seems to be very stable and is not negatively impacted by an increase in the resource pool size. Even for large stackings and less than 100 resources, the time to complete the data management and the scheduling decision is relatively small (within a few seconds).

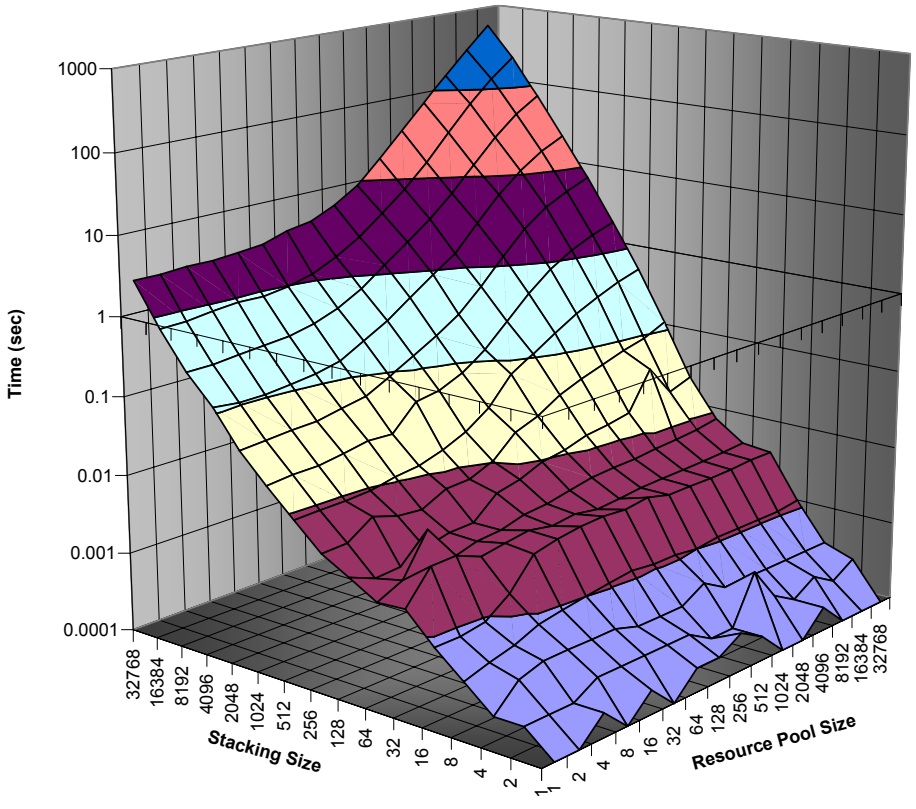

Figure 3: Performance of the data management (3DcacheGrid Engine) and scheduler (CompuStore): the time in seconds to complete as the stacking size and resource pool size is varied

Figure 4 shows a bit of a different story than that of Figure 3 when looking the achieved throughput in terms of stacks per second. Notice that the x and y axis are in reverse order (when compared to the previous figure), and the z axis is not logarithmic; these changes were done to better visualize the throughput metric, and is consistent in all the other throughput figures. There seems to be a relatively low throughput for small stackings, which is not an issue since the AstroPortal uses WS-calls to submit the stackings and retrieve the results, so it is likely that the communication overhead will be far greater than that of the data management and scheduling (i.e. 1 basic WS call takes 10~30 ms while the time per stack for small stacking sizes is 0.1~1 ms). We observe that the throughput increases substantially up to a peak of about 64K stacks per second with medium size stacking sizes and small resource pool sizes. The part of Figure 4 which shows the limitations of the current implementation is when both the stacking size and the resource pool size is large. We literately see the throughput drop to less than 100 stacks/sec at its lowest point with 32K stacking size and 32K resource pool size. Figure 5 has more details to help visualize the area in question.

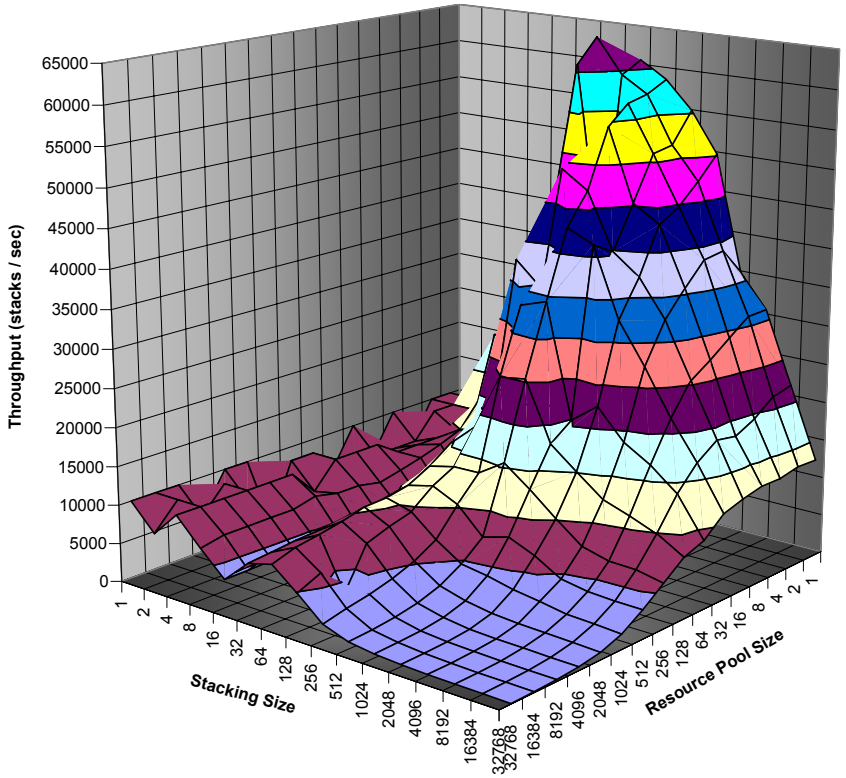

Figure 4: Performance of the data management (3DcacheGrid Engine) and scheduler (CompuStore): the throughput (3D) in stacks/sec as the stacking size and resource pool size is varied

Figure 5 represents the same information as Figure 4 did, but in a 2D grid for better visualization of the key areas that had poor throughput performance. Knowing the performance of the AstroPortal, we can deduce from the grid below that resource pool sizes of 16K and above are not realistic, and that resource pool sizes of 2K probably shouldn't be used. The grid below also shows that with 128 or fewer resources (very realistic for our current AstroPortal deployment), the throughput performance is excellent.

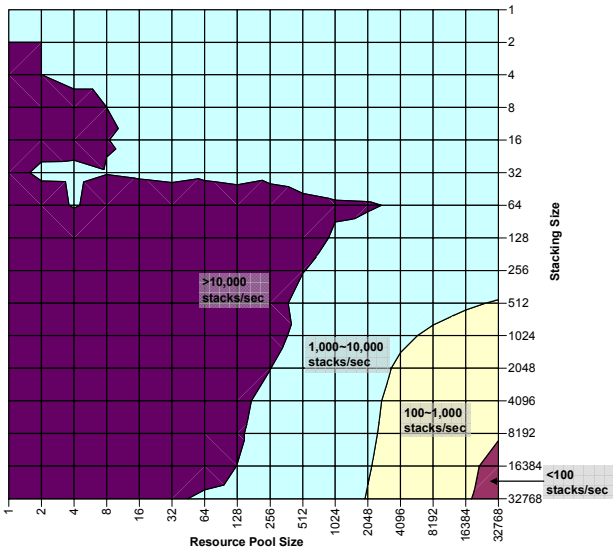

Figure 5: Performance of the data management (3DcacheGrid Engine) and scheduler (CompuStore): the throughput (2D) in stacks/sec as the stacking size and resource pool size is varied

3.2. Stacking Size vs. Index Size

Assumptions:

- Stacking size: 1 – 32K
- Resource Pool Size: 128
- Resource Capacity: 100GB (100,000MB / 6MB ~ 16,667 entries)
- Index Size: 1K ~ 2M entries
- Replication Level: 1 cache per resource (unique caches, 1-1 mapping from data archive and cache)

Figure 6 shows the time to complete stackings of size 1 – 32K as we varied the index size from 1K – 2M. All axes are on a logarithmic scale, which shows that the entire parameter space is increasing at a relatively linear rate. This is a great property to have, however it is of no surprise as the underlying data management is maintained as a hash table where each entry contains a list of replica locations. The drawback of basing our data management implementation on hash tables is that we can only operate on the hash table in-memory, and hence we are limited in the number of entries depending on the amount of memory the particular system has. We tested the limits of the current implementation to be around 2M~10M entries depending on the number of replicas that exist in the entries. One important thing to note is that we have implemented a persistent has table which stores every hash table modifiable operation as well as occasional restore points which can later be used together to reconstruct the hash table contents.

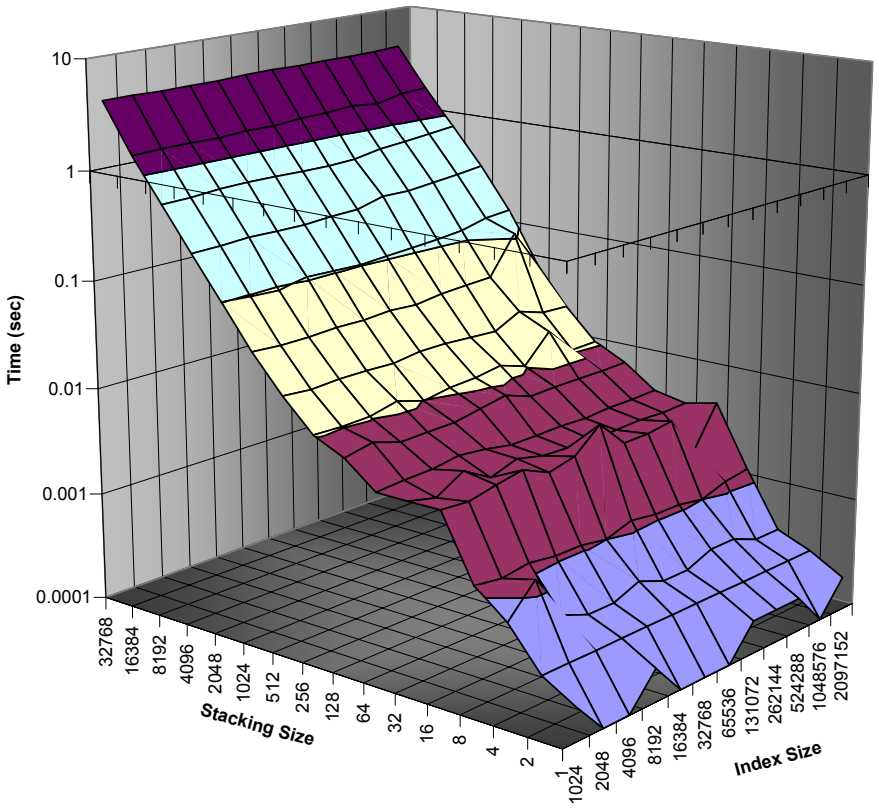

Figure 6: Performance of the data management (3DcacheGrid Engine) and scheduler (CompuStore): the time in seconds to complete as the stacking size and index size is varied

Figure 7 shows the throughput obtained while varying the stacking size ad the index size. We see a similar pattern that the index size does not greatly affect the achieved throughput; similarly, we see that the stacking size does influence the throughput with the peak in the mid values. Keep in mind that the peak throughput achieved here is only about 27K stacks/sec (instead of the 64K stacks per second we saw in Figure 4) since we used a resource pool size of 128 for this experiment, while the peak from Figure 4 was obtained with a significantly smaller resource pool

of size. It is interesting to note that the performance of 5K+ stacks/sec over the entire parameter space is very well suited for not being the bottleneck in the AstroPortal.

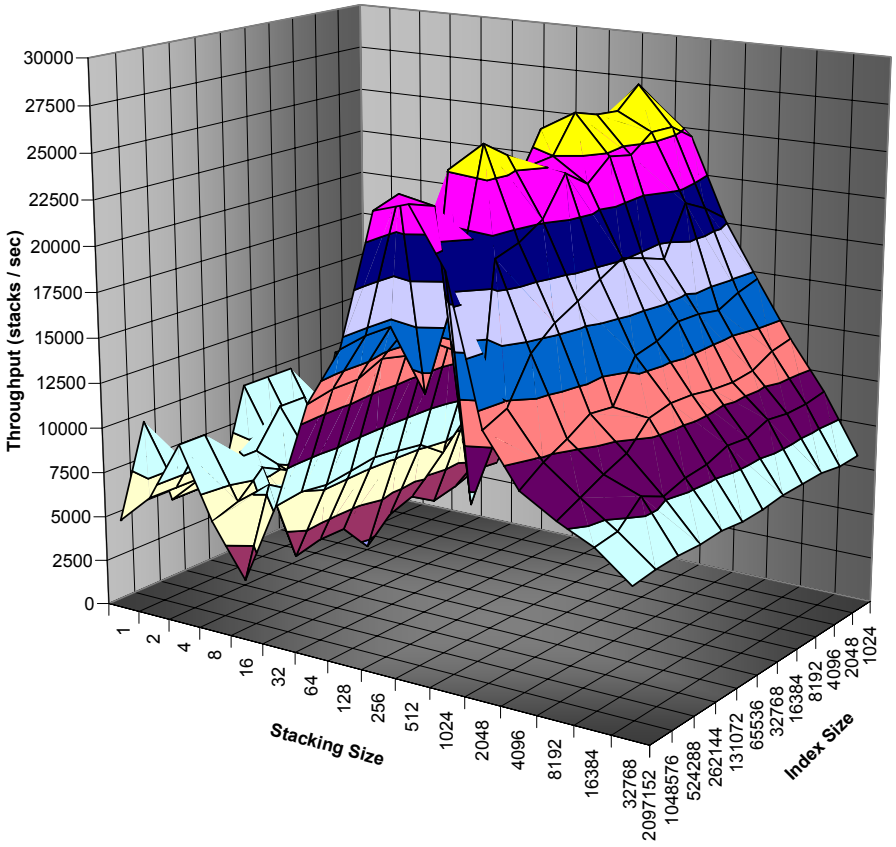

Figure 7: Performance of the data management (3DcacheGrid Engine) and scheduler (CompuStore): the throughput (3D) in stacks/sec as the stacking size and index size is varied

3.3. Stacking Size vs. Replication Level

Assumptions:

- Stacking size: 1 – 32K
- Resource Pool Size: 128
- Resource Capacity: 100GB (100,000MB / 6MB ~ 16,667 entries)
- Index Size: maximum of 1.3 million entries, depending on the number of replication level used
- Replication Level: 1 - 128 caches distributed over all the storage resources

Figure 8 shows the time to complete stackings of size 1 – 32K as we varied the replication level from 1 – 128. All axes are on a logarithmic scale, which shows that the entire parameter space is increasing at a relatively linear rate, with a slight tendency for increased times towards larger replication levels.

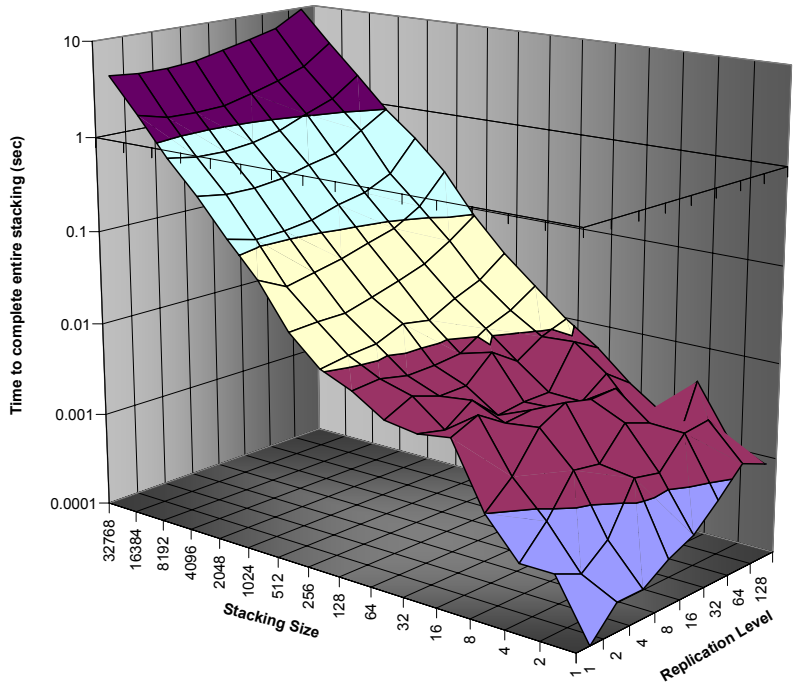

Figure 8: Performance of the data management (3DcacheGrid Engine) and scheduler (CompuStore): the time in seconds to complete as the stacking size and replication level is varied

Figure 9 shows the throughput for varying the stacking size and the replication level. This shows that even with large number of replication, the achieved throughput is still 5K+ for the majority of the parameter space.

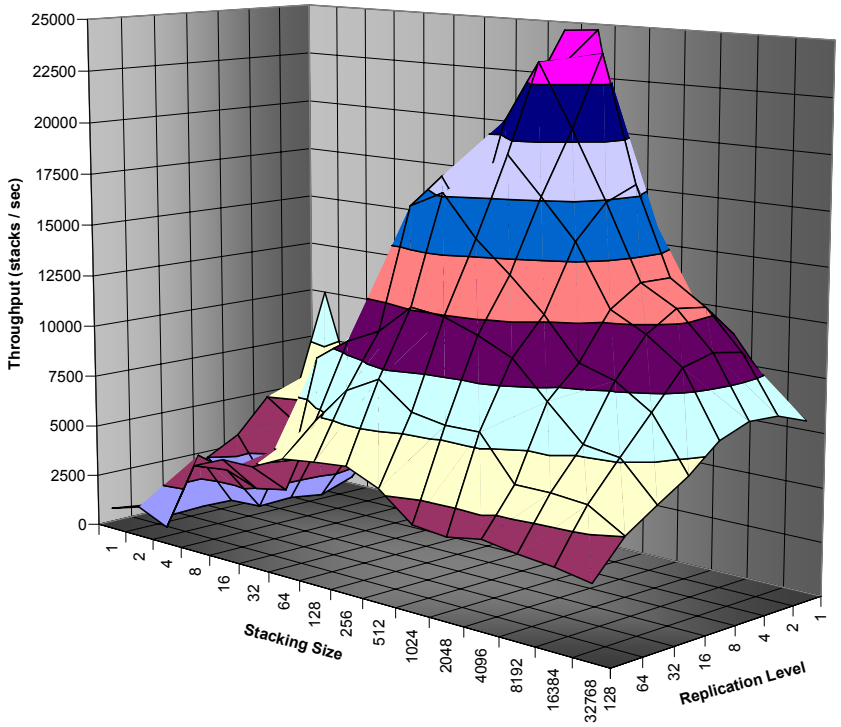

Figure 9: Performance of the data management (3DcacheGrid Engine) and scheduler (CompuStore): the throughput (3D) in stacks/sec as the stacking size and replication level is varied

4. Conclusion

The results from this report show that the 3DcacheGrid Engine and CompuStore will likely not be the bottleneck in the AstroPortal as long as the following is true:

- Stacking size: less than 32K (although another order of magnitude probably won't pose any performance risks)
- Resource pool size: less than 1000 resources might offer decent performance if there is the replication level remains low, but for higher orders of replication, less than 100 resources are recommended
- Index Size: 2M~10M depending on the level of replication using a 1.5GB Java heap; larger index sizes could be supported linearly without sacrificing performance by increasing the Java heap size (needing more physical memory and possibly a 64 bit JVM environment)
- Replication Level: less than 128 replicas (although more could be supported as long as the dataset size remains relatively fixed)
- Resource Capacity: 100GB of local storage per resource (this could be increased, but its unclear what the performance effects would be)

Now that the basic logic of the compute and storage resource management mechanisms have been implemented, we now need to integrate these mechanisms into a running and deployed system, such as the AstroPortal. DYRE has already been integrated, but CompuStore and 3DcacheGrid still needs to be integrated in the AstroPortal. We need to make sure that the integration is done in a modular fashion in such a way that the existing basic round-robin scheduler (which does not take into account any data management information about caches) still functions for comparison purposes.

5. References

[1] GSC-II: Guide Star Catalog II, http://www-gsss.stsci.edu/gsc/GSChome.htm
[2] 2MASS: Two Micron All Sky Survey, http://irsa.ipac.caltech.edu/Missions/2mass.html
[3] POSS-II: Palomar Observatory Sky Survey, http://taltos.pha.jhu.edu/~rrg/science/dposs/dposs.html
[4] Samuel G. Armato III, et al. "Lung Image Database Consortium: Developing a Resource for the Medical Imaging Research Community," Lung Image Database Consortium Research Group, Radiology 2004; 232: 739-748.
[5] I Raicu, I Foster, A Szalay, G Turcu. "AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis", to appear at TeraGrid Conference 2006, June 2006.
[6] I Raicu, I Foster, A Szalay. "Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets", under review at SuperComputing 2006.
[7] A Szalay, J Bunn, J Gray, I Foster, I Raicu. "The Importance of Data Locality in Distributed Computing Applications", NSF Workflow Workshop 2006.
[8] Sloan Digital Sky Survey (SDSS), Release 4 (DR4), http://www.sdss.org/dr4/
[9] Ioan Raicu, Ian Foster. "DRP: Dynamic Resource Provisioning", CEDPS Scalable Services Report, October 2006.