A Dynamic Resource Management System for Network-Attached Accelerator Clusters

Suraj Prabhakaran [1,a], Mohsin Iqbal [2,a], Sebastian Rinke [3,a], Felix Wolf [4,a]
[a]*German Research School for Simulation Sciences, Laboratory for Parallel Programming, 52062 Aachen, Germany*
{[1]s.prabhakaran, [2]m.iqbal, [3]s.rinke, [4]f.wolf}@grs-sim.de

***Abstract*—Over the years, cluster systems have become increasingly heterogeneous by equipping cluster nodes with one or more accelerators such as graphic processing units (GPU). These devices are typically attached to a compute node via PCI Express. As a consequence, batch systems such as TORQUE/Maui and SLURM have been extended to be aware of those additional resources tightly coupled with compute nodes. Recent advances in accelerator technology have given rise to the possibility of using network-attached accelerators in addition to node-attached accelerators. However, current batch systems do not support this new usage scenario of accelerators. This work focuses on the support for batch systems for allocating network-attached accelerators. The most important feature of the proposed batch system is its ability to dynamically allocate network-attached accelerators to jobs at application runtime. We discuss our extensions to the TORQUE and Maui batch system and elaborate on its features in the Dynamic Accelerator-Cluster Architecture, which describes an integration of network-attached accelerators into a cluster system. We also evaluate the dynamic allocation scenarios and show how batch systems can be designed to provide support for more flexible and dynamic cluster systems.**

***Keywords*-dynamic resource management; dynamic scheduling; heterogenous architectures;**

I. Introduction

Accelerators such as graphics processing units (GPUs) have seen successful integration into cluster systems over the recent years. These devices have been able to improve the overall system efficiency by offering increased computational power at minimized energy consumption levels. The Tianhe-1A system at the National Supercomputer Center in Tianjin is a prominent example of an accelerator-equipped supercomputer. These devices are attached locally to the compute nodes through PCI Express and can be programmed with familiar programming models such as CUDA and OpenCL. Typically compute clusters are composed of nodes with and without accelerators attached to them. In such systems, the batch system is responsible for good job distribution such that only jobs requiring accelerators are placed in the nodes that contain them. This ensures efficient use of the cluster resources for any workload that contains a mix of jobs with varying accelerator requirements. Schedulers such as Maui [1] with the TORQUE [2] resource management system and SLURM [3] are examples of batch systems that deliver this functionality.

With recent developments in accelerator hardware, network-attached accelerators, for example, based on the Intel MIC architecture, have become a feasible solution. Unlike a GPU, they can autonomously communicate over the cluster network without requiring a separate host CPU for managing network transfers. State of the art in this field is the ongoing DEEP project [4], which investigates Intel Xeon Phi processors as the basic component of network-attached accelerators. With a novel cluster-booster system architecture, the DEEP project strives to better meet the accelerator requirements of scientific applications with respect to traditional accelerator-equipped clusters. Regarding the efforts of vendors in this area, not only Intel but also Nvidia is working on accelerators capable of initiating network communication. Here, Nvidia's Project Denver [5] is developing a device containing both CPU and GPU on the same chip.

The primary advantage of deploying network-attached accelerators is that compute nodes are not bound to use only the limited number of node-attached accelerators. They may offload computations to as many network-attached accelerators available in the entire cluster. The potential bandwidth penalty between host and accelerator may be hidden using techniques such as double buffering or by longer kernel runtimes. The latter can be achieved by having the main program offload multiple kernels in parallel to a set of network-attached accelerators that communicate directly with each other (e.g., through the well-known MPI). Such MPI kernels can run for an extended period of time without involving the host. Recent works have given rise to high bandwidth cluster interconnects which may well support such architectures. The ongoing DEEP project is an example where such an architecture is under investigation with a high-bandwidth EXTOLL interconnect. Furthermore, such an architecture also enables flexible accelerator usage scenarios where applications have the possibility to adjust the number of accelerators assigned to them at runtime. For example, additional accelerators could be associated with compute nodes running the job when a computational phase of the application demands more accelerators. Since dynamic provisioning of cluster resources can also improve fault tolerance and optimize energy consumption, it is considered as one of the important aspects in reaching exascale [6]. Current batch systems lack both the awareness of network-attached accelerators and dynamic allocation of resources and therefore cannot support such flexible usage scenarios.

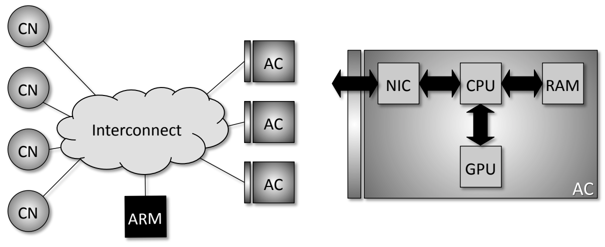
Based on the current trends in accelerator hardware and the above usage scenario of network-attached accelerators, we have developed a dynamic batch system aware of network-attached accelerators, which is based on the TORQUE resource manager and the Maui scheduler. Our batch system assigns accelerators to jobs both statically before the job start or dynamically during the runtime based on the application demands. In our previous work we proposed one of the first architectures to integrate network-attached accelerators into cluster systems called as the Dynamic Accelerator-Cluster (DAC) Architecture. Similar to the DEEP cluster, the DAC Architecture enables computational offloading on network-attached accelerators with static and dynamic allocation of accelerators to compute nodes. While the DEEP cluster is based on Intel's MIC Architecture and the special EXTOLL interconnect, the DAC Architecture focuses on using network-attached accelerators that can be realized using currently available cluster components. Applications have also been able to obtain profitable speedups using multiple accelerators in the DAC Architecture [7]. Our batch system has been successfully used in this architecture and thus, can support an heterogeneous cluster consisting of both node-attached and network-attached accelerators.

In the past (detailed in Section V), dynamic allocation has been primarily studied as scheduling problem. Implementing a dynamic allocation mechanism in a resource management system (RMS) has often been neglected in such projects as it is rather complicated. They lack the functionality to dynamically associate and disassociate nodes to a job due to their inherent design to support only static allocations. In our work, we have extended the TORQUE RMS and the Maui scheduler to support the dynamic allocation of network-attached accelerators. We have extended the TORQUE RMS with functionalities to dynamically request and allocate network-attached accelerators to a job. Also, protocols to dynamically associate and disassociate nodes to a job have been developed. Similarly, the Maui scheduler has been enhanced with features to dynamically allocate resources for an already running job. In this paper, we present our contributions to the TORQUE/Maui batch system for enabling dynamic allocations and evaluate them in a DAC Architecture testbed. By that, we show how batch systems can be designed to be more flexible in accordance with the dynamic usage scenarios of future HPC systems.

The remainder of this article is organized as follows. In Section II, we present an overview of the Dynamic Accelerator-Cluster Architecture and briefly describe its implementation. In Section III we give an overview of our dynamic batch system and demonstrate its usage under the DAC Architecture. In Section IV we discuss the performance of our batch system, with the focus on the dynamic resource allocation scenarios. We review the related work in Section V and finally conclude with an outlook in Section VI.

II. Dynamic Accelerator-Cluster Architecture

In this section, we briefly describe the Dynamic Accelerator-Cluster Architecture which enables the integration of network-attached accelerators into cluster systems.



(a) The DAC Architecture. (b) Accelerator.

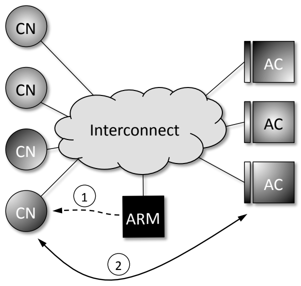
Figure 1. The components of the Dynamic Accelerator-Cluster Architecture.

A. Overview

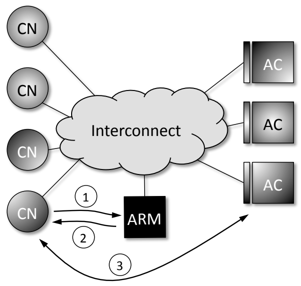
The Dynamic Accelerator-Cluster (DAC) Architecture describes a generic architecture for using network-attached accelerators in a flexible manner. It provides an accelerator independent programming model to offload computations onto network-attached accelerators. Under this architecture, a pool of network-attached accelerators is maintained by an Accelerator Resource Manager (ARM) which allocates the accelerators to compute nodes based on application requirements. Figure 1(a) shows a pictorial representation of the architecture. The compute nodes (CN) represented in the architecture are identical to their counterparts in current cluster environments. The accelerators (AC) represented in this architecture are not bound to be a particular type of accelerator. In its current implementation, the DAC Architecture uses GPUs attached to a host as a network-attached accelerator. The programming model, thereby, enables a transparent mechanism to offload computations onto remote GPUs. Figure 1(b) depicts the network-attached accelerator consisting of a host CPU and a GPU as used in this implementation. The ARM resembles a resource management system. It maintains information about the accelerators and is responsible for handling allocation and deallocation requests from the compute nodes.

B. Execution Model and Accelerator Assignment Strategies

The execution model of this architecture consists of three steps: (i) accelerator allocation, (ii) accelerator usage and (iii) accelerator deallocation. Accelerator allocation can be carried out in two distinct assignment strategies. In the *static assignment strategy*, the required number of accelerators is allocated to compute nodes before job start. These accelerators remain allocated to the compute nodes until job termination. The compute nodes are provided with a computation API similar to CUDA and OpenCL to offload work onto accelerators. In order to identify accelerators, a unique *handle* for each allocated accelerator is used with the API. In the *dynamic assignment strategy*, the accelerators are (de)allocated at job runtime. Compute nodes send a runtime request to the ARM to acquire new accelerators. Note that, it is not guaranteed that a dynamically requested resource will always be



(a) Static assignment.



(b) Dynamic assignment.

Figure 2. Static (a) and dynamic (b) accelerator assignment. Different shadings denote different jobs. Dashed lines denote communication before job start, whereas solid lines denote communication at runtime [7].

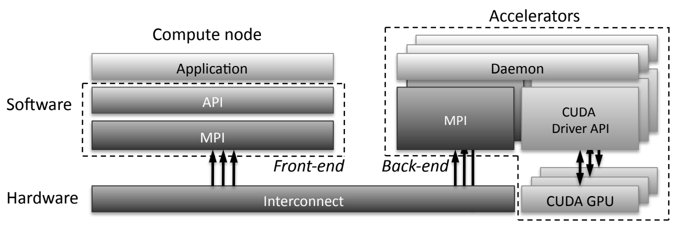


Figure 3. Dynamic accelerator-cluster software architecture with CUDA back-end [7].

available for the application. Subject to availability, the ARM allocates the resources to the requesting compute node. When the requested number of accelerators is not available, the ARM rejects the request. Therefore, users also take into account that the dynamic requests may not always be successful. When a dynamic request is rejected, the application continues its execution with the existing allocated accelerators. When the dynamically allocated accelerators are not needed anymore, the compute nodes can release the accelerators. For dynamic (de)allocation the compute nodes use a resource management API which complements the computation API. A prototypical implementation of the ARM was developed to enable the static and dynamic allocation strategies. Figure 2 illustrates the static and the dynamic assignment scenarios as explained above. Dashed lines between the ARM and the compute nodes represent communication before job start as in the case of a static assignment. Solid lines between the ARM and the compute nodes represent communication at runtime depicting a dynamic assignment.

C. Implementation

As stated earlier, the current version of the DAC Architecture enables computation offloading to the network-attached accelerators consisting of a CUDA-enabled GPU. Computations are CUDA kernels which are executed on the remote GPU. The computation API provides functionality to (i) allocate memory on the accelerators, (ii) copy data to or from accelerators and (iii) launch compute kernels on the accelerators. Figure 3 illustrates the software stack of the DAC Architecture. It consists of a front-end on every compute node and a back-end on every accelerator. The front-end translates API calls into requests which are redirected to the back-end, where a daemon receives those requests and executes them on the CUDA-enabled GPU using the CUDA driver API. The front-end uniquely identifies the back-end through a *handle* and enables a transparent communication between the compute node and the accelerator. Listing 1 illustrates the usage of both the computation and the resource management API with regards to using a remote CUDA-enabled GPU. Here, similar to CUDA, a computation kernel is executed on the accelerator after allocating memory and transferring data to the device. After the kernel execution is complete, data is transferred back and the memory is freed. The *ac_handle* uniquely identifies the accelerator on which the operations are to be performed. The actual communication between the compute nodes and the accelerators is accomplished through a distinct communication protocol based on MPI. Clearly, for the compute nodes to be able to communicate to the daemons running in the accelerator through MPI, they have to reside in the same MPI communicator. A resource management library, which makes use of MPI-2 dynamic process management facilities, is provided to the compute nodes to establish this transparently with the accelerators.

Listing 1. Example program on the Dynamic-Accelerator Cluster Architecture.

```
void main(int argc, char **argv) {

  /* Init the accelerators */
  AC_Init(&ac_handle);
  ...
  /* Allocate memory on device */
  acMemAlloc(cudaMalloc_args,ac_handle);

  /* Transfer memory to device */
  acMemCpy(cudaMemcpy_args,ac_handle);

  /* Execute kernel */
  acKernelCreate(k_name,ac_handle);
  acKernelSetArgs(k_args);
  acKernelRun(k_name,dimGrid,dimBlock);

  /* Transfer memory to host */
  acMemCpy(cudaMemcpy_args,ac_handle);

  /* Free memory on device */
  acMemFree(cudaFree_args,ac_handle);
  ...

  /* Get more accelerators */
  AC_Get(count, &ac_handle_new);
  ...
  /* Free the dynamically obtained accelerators */
  AC_Free(&ac_handle_new);

  /* Finalize */
  AC_Finalize(&ac_handle);
}
```

The resource management API marginally differs with the computation API in its naming conventions. The `AC_Init()` initializes the accelerator usage for the computation API after creating an MPI communicator with the statically allocated accelerators and providing a valid *ac_handle*. The `AC_Get()` call is used to dynamically request additional accelerators from the ARM. Users may use the `AC_Free()` call to release dynamically assigned accelerators. The `AC_Finalize()` routine must be called at the end and releases all the associated accelerators.

In enabling batch system support for the DAC Architecture, the functions of prototypical ARM is completely integrated in the batch system. The resource management library communicates directly with the batch system. Also, note that while using remote GPUs involves additional communication overhead through the cluster interconnect, computationally intensive applications can still benefit from using multiple accelerators. In particular, multiple accelerators can also be used with latency hiding techniques to reduce the visible communication overhead. Furthermore, the implementation also provides an efficient communication protocol which includes pipelining large data transfers, thereby optimizing the overall data transfer. These are described in [7].

III. The Dynamic Batch Scheduler

In this section, we present our extended TORQUE/Maui batch system supporting both static and dynamic allocation of network-attached accelerators. We start by providing an overview of the TORQUE/Maui batch system and proceed to describe our extensions. We demonstrate the functioning of our batch system by means of an example job using one compute node requiring x statically allocated and y dynamically allocated network-attached accelerators.

A. Overview of the TORQUE/Maui Batch System

The TORQUE Resource Manager [2] is a commonly used open-source resource manager for cluster systems. It is based on the PBS project [8] extended to improve scalability and fault tolerance and is currently maintained by Adaptive Computing. TORQUE also contains a basic FIFO scheduler but is generally integrated with a scheduler package like the Maui scheduler [1]. The Maui scheduler provides advanced scheduling features such as job prioritization, fairshare and backfill scheduling which improves the overall system utilization as compared to TORQUE's FIFO scheduler. Considering the large popularity of the TORQUE/Maui batch system in more than hundreds of cluster systems (including GPU clusters) all over the world, we chose to use it for supporting network-attached accelerators. A TORQUE/Maui cluster consists of a headnode and many compute nodes. The headnode runs the `pbs_server` daemon (`server`) and the compute nodes run `pbs_mom` daemon (`mom`). Client commands for submitting and managing jobs can be installed on any host including the headnode. The headnode also runs

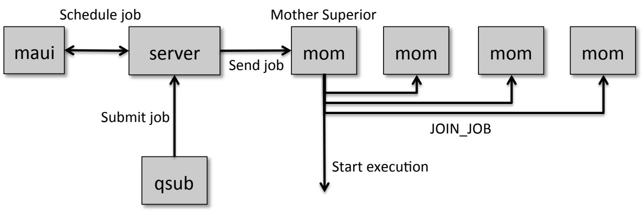


Figure 4. Typical workflow of a job scheduling in the TORQUE/Maui batch system.

the Maui scheduler daemon. The scheduler interacts with the `server` and makes decisions on resource usage and allocation of nodes to jobs. The typical work flow of job scheduling in a TORQUE/Maui cluster is enumerated below and illustrated in Figure 4.

1) Users submit jobs with the `qsub` command indicating the number of nodes (k) and the cores per node (q) required by the job.
   ```
   qsub -l nodes=k:ppn=q jobscript.sh
   ```
2) The `server` takes the request and stores the job information such as job submission time, resources required, etc, internally as *job attributes*. The job is then enqueued for resource allocation.
3) The Maui scheduler retrieves the list of queued jobs and the status of all the resources in the cluster from the `server`. It allocates resources to the jobs based on various site-specific policies and sends the information about the allocated resources for each queued job to the server.
4) The `server` reads the list of allocated hosts for a job and selects the `pbs_mom` running in one of the allocated hosts as the *mother superior* and sends it the complete job information (including the list of allocated hosts).
5) The mother superior, on receiving the instruction to run a job, first connects with each `mom` running on the other allocated hosts. The connection is established by sending what is called a `JOIN_JOB` request along with the job information.
6) After successful connection with all the `mom`s, the mother superior starts the execution of the job script.

Once execution begins, the job can retrieve many information about its TORQUE environment by querying various environment variables set by the `mom`. For example, `PBS_JOBID` contains the global job-ID set by the `pbs_server` for this job. Typically, MPI jobs look into the `PBS_NODEFILE` environment variable which points to a file containing the list of hosts allocated for this job. All the processes of a job can communicate with their local `pbs_mom` through a **TM** API and communicate directly to the server through the Interface Library (IFL) API. For example, user can alter specific job attributes through `pbs_alterjob()` call which is equivalent to `qalter` client command.

B. Extensions for Allocation of Network-Attached Accelerators

To support the use of network-attached accelerators, we have enabled the TORQUE/Maui batch system with an

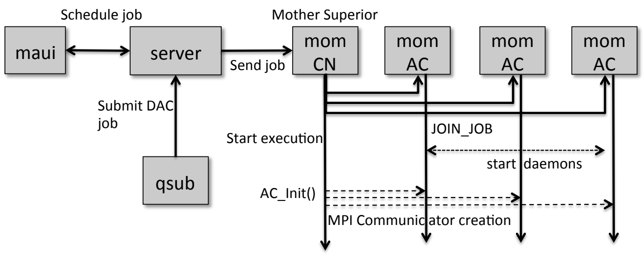


Figure 5. Workflow of a static allocation scenario.

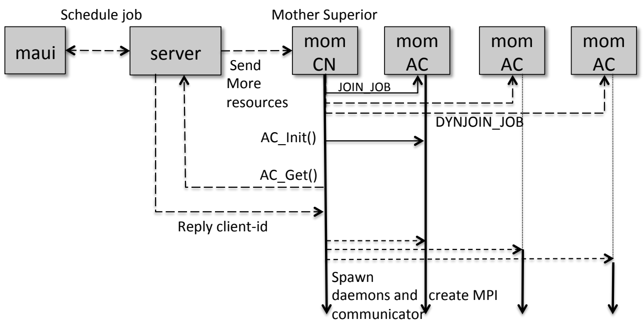


Figure 6. Workflow of a dynamic allocation scenario.

overall awareness of the DAC execution environment and extended it to perform the static and dynamic assignment facilities. For the static assignment, before executing the application on the compute nodes, TORQUE ensures that appropriate daemons are started on the accelerators in order to be used by the compute nodes. We have extended the `server` and the `moms` to perform the above in the DAC environment. The dynamic (de)allocation requires the batch system to be capable of advanced functionalities such as:

- a way to submit dynamic requests to TORQUE,
- dynamic scheduling capabilities in Maui, and
- dynamic (dis)association of resources to a job in TORQUE.

We extended TORQUE's Interface Library with a new call to request additional resources from a job executing in the DAC environment. The resource management library uses the `pbs_dynget()` call to request additional accelerators. The Maui scheduler has been extended to operate on such dynamic requests submitted to the `server` and schedules them with top priority among all the jobs waiting for resources in the queue. Finally, we introduced the capabilities for dynamic addition of resources into a job by the `moms` executing the job. Similarly, to dynamically release accelerators, the `pbs_dynfree()` call has been added to the Interface Library and the `moms` have been extended with capabilities to dynamically disassociate from a job.

C. Static Allocation of Network-Attached Accelerators

In the static allocation scenario, a job requests a particular number of network-attached accelerators during job submission time. The job is not executed until all the required resources are available. During the job start, the resource management library establishes the association with the accelerators. Users may then use the compute node API to offload computations to the accelerators. We consider the example of one compute node requesting x accelerators and describe the functioning of both the batch system and the resource management library, individually, in assigning the accelerators to the compute node. Figure 5 illustrates the scenario with one compute node statically associating with three accelerators.

Batch System: The user requests a DAC execution environment using the extended `qsub` command as shown below.

```
qsub -l nodes=1:acpn=x jobscript.sh
```

The `acpn` job attribute indicates the request of x network-attached accelerators per compute node. For a multi-compute node job with k compute nodes, this request would mean a total of k compute nodes and $k \times x$ accelerators for the job. The `server` then enqueues the job for scheduling. The Maui scheduler allocates the required resources and informs the `server`. The `server` selects the `mother superior` (which is always a compute node) and forwards the job information. Once the `moms` JOIN with each other, the `mother superior` invokes the execution of accelerator daemons the accelerator nodes. The daemons are started such that each set of the accelerators to be associated with a compute node is contained under a single `MPI_COMM_WORLD`. The user application is then started on the compute nodes and the resource management library establishes the connections with the daemons.

Resource Management Library: As stated earlier, the resource management library uses MPI-2 dynamic process management facilities to enable a persistent connection between the compute nodes and the accelerators. Once the daemons are started, the `root` (MPI rank 0) of the accelerator daemons opens an MPI port (using `MPI_Open_port()`). The port information is made available to the compute nodes through a file. When `AC_Init()` call is invoked, the compute nodes retrieve the port information and establish the connection through `MPI_Comm_connect()`/`MPI_Comm_accept()`. The inter-communicator returned through this operation is used to create an intra-communicator through the `MPI_Intercomm_merge()`. In the new MPI intra-communicator, the compute node holds the rank 0 while all the other accelerators have a unique rank ranging from 1 to x. The handle to each accelerator consists of its unique rank in this communicator and is further used by the computation library to transfer data and execute kernels.

While the above scenario exemplifies a job with a single compute node, the process is essentially the same for a multi-compute node job. Each compute node would be associated to x accelerators with a distinct MPI communicator from the other compute nodes. In other words, one compute node cannot access the accelerators associated to the other compute nodes. When the job terminates, all the resources used by the job are released and made available for other jobs.

D. Dynamic Allocation of Network-Attached Accelerators

In the dynamic allocation scenario, compute nodes send a runtime request to the `server` for a definite number of additional accelerators through the `AC_Get()` routine. Upon allocation, the accelerators are associated with the requesting job and made available for use by the compute nodes. As explained for the static case, we consider an example of an application with one compute node and x statically allocated accelerators, which dynamically requests y additional accelerators. Figure 6 illustrates the dynamic allocation scenario of a compute node with one statically allocated accelerator, requesting two additional accelerators. Dashed lines represent communication after the `AC_Get()` call and the solid lines indicate communication before the call during static allocation.

Batch System: The resource management library sends the request for *y* additional accelerators through the `pbs_dynget()` routine which blocks until a response has been received from the server. Upon receiving the request, the `server` enqueues the job again with a special *dynamically queued* state and the Maui scheduler allocates resources for the *dynamically queued* jobs with top priority. The `server` is then informed of the allocated resources, which then forwards the information to the `mother superior` of the job that requested the additional accelerators. Once the information has been successfully forwarded, the `server` responds to the compute node with a *client-id* which uniquely identifies this request and its set of dynamically allocated accelerators. The `mother superior` then sends a `DYNJOIN_JOB` message to the newly allocated accelerators and also updates the existing `moms` with the addition of resources for this job. Preparing the accelerators with the daemons and establishing connection with them is performed by the resource management library. When not enough resources could be allocated for the job, the `server` rejects the request immediately with a negative valued reply. In this case, the application continues to execute with the already allocated accelerators as stated earlier.

Resource Management Library: Once the additional accelerators have been allocated, the resource management library spawns the accelerator daemons on the nodes through the `MPI_Comm_spawn()` call. This call returns an MPI inter-communicator with the accelerator daemons, once they have executed `MPI_Init()`. The compute node, its existing accelerators and the newly active accelerators participate in the `MPI_Intercomm_merge()` call which results in a new intra-communicator with the compute node and all of its associated accelerators. In this intra-communicator, the compute node still holds the MPI rank 0 and the old accelerators hold their old MPI ranks ranging from 1 to x. The newly added accelerators are assigned MPI ranks ranging from $x + 1$ to $x + y$. Updated handles to the statically assigned accelerators and the new handles to the dynamically assigned accelerators are then returned to the user. These can then be used by the computation library.

We use `MPI_Comm_spawn()` instead of starting the daemons through the `moms` as it enables an easier way of creating the MPI communicators as opposed to using MPI Ports and employing `MPI_Comm_connect()`/`MPI_Comm_accept()` which is unavoidable in the case of static assignment.

Due to the fact that MPI is used, a set of dynamically allocated accelerators are started with the accelerator daemons that are encompassed in one `MPI_COMM_WORLD`. Therefore, when the dynamically allocated accelerators are released, they are released as a set identified by the *client-id* through the `AC_Free()` call. The compute nodes first disconnect from the to-be-released accelerators through `MPI_Comm_disconnect()` and send the `server` the *client-id* of the set of dynamically allocated accelerators using the `pbs_dynfree()` call. The server returns a positive reply to the compute node without needing to enqueue the job again and initiates the process of disassociating the accelerator nodes with the job while the user application may continue to execute further. To release the accelerators, the `server` instructs the `mother superior` with the list of hosts that are to be disassociated from the job. The `mother superior` sends a `DISJOIN_JOB` message to the `moms` operating on these hosts resulting in complete disassociation of these `moms` from the job. The `moms` kill all the tasks running on their host (typically any remains of the accelerator daemons) and have its resources free for other jobs. The `mother superior` also sends the information to the other `moms` associated to the job so as to update their database.

In the case of a multi-compute node job, each compute node may use its own `AC_Get()` to obtain additional accelerators. However, the `server`, is able to service only one request at a time per job. This may lead to long waiting time during the application runtime for some compute nodes of the job until their additional accelerators are allocated. This can be avoided using `AC_Get()` collectively over all the compute nodes that request additional accelerators. When requested collectively, one compute node collects the information about the number of accelerators required by each compute node participating in the collective call, and sends a single request to the `server` requesting the total number of required accelerators. However, the disadvantage is that either all the compute nodes get their accelerators allocated or none, since the batch system tries to allocate the total number of accelerators requested. Also, since they all obtain the same *client-id* from the `server`, they may be released only collectively.

E. Scheduling Mechanism

While TORQUE enables the association and disassociation of network-attached accelerators to an existing job, the Maui scheduler is responsible for deciding whether a particular dynamic request is eligible for extending its resource set. We extended the Maui scheduler to schedule and allocate resources for dynamic requests based on the information forwarded by the TORQUE `server`.

Both dynamic and static requests are sent to the Maui scheduler through the same queue. The static requests (`qsub` requests) are in the `queued` state waiting for the resources to be allocated by the scheduler. The dynamic requests hold a special `dynqueued` state. After retrieving the current queue information, the scheduler proceeds to prioritize the jobs based on the system policies. In our current implementation, a basic dynamic priority mechanism places the dynamic requests higher in the queue as compared to the static requests. The dynamic requests are ordered in FIFO manner. While such a policy is not the best for a production system, it serves as a basic mechanism from which more sophisticated dynamic allocation algorithms can be built upon in the Maui scheduler in the future.

In our approach, we do not use advanced reservation techniques for dynamic allocations. This is because, the main objective is to provide flexible usage scenarios to evolving jobs in the DAC architecture. More precisely, at job submission, the application need not know whether or how many dynamic allocations it may require during its execution. The decision to request more resources can be done at runtime without any prior indications to the batch system. Therefore, neither is a dynamic request guaranteed to be satisfied nor will it wait in the queue until the requested resources are available. As stated earlier in Section II-B, when the request cannot be satisfied, Maui rejects the request and the application resumes execution with its existing resource set. Our future version of the Maui scheduler will contain optimal dynamic scheduling strategy which takes both system performance and fairness into consideration.

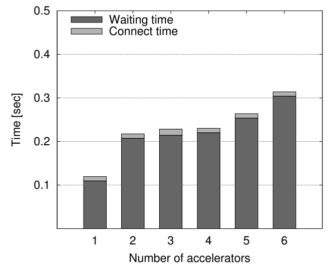
IV. Experimental Evaluation

In this section, we present a quantitative description of the performance of our dynamic batch system in enabling static and dynamic allocation of network-attached accelerators to compute nodes under the DAC environment. Due to the novelty of this usage scenario, real world applications that use network-attached accelerators are still under development in various projects (e.g., the DEEP Project). In our evaluations, we examine the overhead of the dynamic resource allocation under various circumstances in a cluster environment with sample programs and discuss its impact on real world applications, as compared to other works that mainly simulated the dynamic allocations.

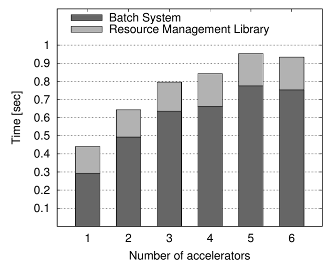
For all our experiments, we used 8 nodes with 2 Intel X5570 processors at 2.93 GHz and with 24 GiB RAM each. All the nodes ran GNU/Linux 2.6.35 (Ubuntu 10.10). As MPI implementation, we used Open MPI 1.6.2. Our experiments to evaluate the batch system's performance did not require the physical presence of an accelerator in these nodes. Out of the 8 nodes, one node was designated as the `server`. It ran the `pbs_server` daemon and the Maui scheduler daemon. The same node was used as the front end. The rest of the 7 nodes were used as both compute nodes and network-attached accelerators in different test scenarios but never at the same time. All the results are an average over 10 trials.

In principle, when submitting a job, if all the required resources are readily available, the time taken to obtain the required nodes and start execution depends only on the rate at which (i) the `server` processes the request, (ii) the resources are allocated by the Maui scheduler and (iii) the `moms` get the job information and join with each other. All of the above involve communication over the network between the batch system components. If all the required resources were not available, the request stays queued at the `server` until these resources become available. Since in the static allocation scenario the required number of accelerators are known prior to the job start, the `server` does not start the job until all the required resources are available. Thus, the static allocation scenario is similar to the traditional way of job submission, and therefore is affected by the same parameters as mentioned above. However, to complete the static allocation, users need to call the `AC_Init()`. Depending upon the number of accelerators requested, the `AC_Init()` call waits until all the accelerator daemons that were started on the accelerator nodes are ready to get connected to the compute node through the `MPI_Comm_connect()`/`MPI_Comm_accept()` routines. Figure 7(a) depicts the time for completion of an `AC_Init()` for various numbers of statically allocated accelerators ranging from 1 to 6. The dark shaded region depicts the amount of time spent by the call in only waiting until all the accelerator daemons were prepared in the remote nodes and were ready to establish a connection with the compute node. The lighter region depicts the time consumed in establishing the MPI communicator with the compute node. We can observe that the waiting time dominates the total time taken and generally increases with increasing number of accelerators. However, statically allocating as many as 6 accelerators requires only around 0.3 seconds.

On the other hand, the dynamic allocation scenario introduces longer waiting time since it includes the time taken by the batch system to allocate additional resources for the job and involves the `moms` to join with each other before the accelerator daemon can be spawned on the host. Figure 7(b) shows the time taken for dynamically obtaining 1 to 6 accelerators by a compute node. The dark shaded part of the graph indicates the waiting time while the lighter regions represent the time spent in performing the MPI operations, i.e., spawning and creating MPI communicators. Naturally, the dynamic allocation of accelerators by the batch system dominates the overall time taken and increases with the increasing number of accelerators. The time spent performing the MPI operations is more or less the same in all the cases. While the time taken for dynamic allocation is much larger compared to the `AC_Init()` in a static allocation, it still ranges only in sub-seconds for obtaining as many as 6 accelerators



(a) Time for completion of AC_Init().



(b) Time for completion of dynamic request.

Figure 7. Time for completion of static and dynamic requests for various number of accelerators.

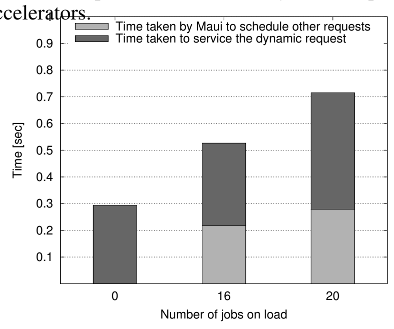


Figure 8. Time taken to dynamcially allocate one accelerator under different load in the batch system.

dynamically. However, the test was made under an ideal scenario where the scheduler and the resource manager are not working on scheduling jobs from a workload.

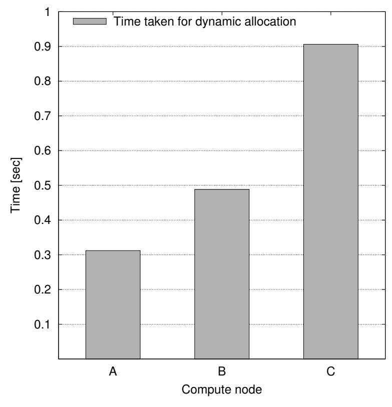


Figure 9. Time taken for completion of consecutive dynamic requests from three distinct compute nodes.

To test the behavior of dynamic allocation in the presence of other workload, we combined the scenario of dynamically requesting one accelerator from a compute node along with large amount of other `qsub` requests. Since the Maui scheduler always treats the dynamic requests from the DAC environment with top priority, when a dynamic request and other `qsub` requests arrive in parallel, the dynamic request will primarily be granted the resources. Therefore, the time taken for the dynamic allocation in such a scenario is similar to the time taken in the absence of any other workload. The workloads may affect the performance of a dynamic allocation request only when the dynamic request arrives at the `server` when the scheduler is already working on allocating resources for the earlier requests. This causes additional waiting time to the dynamic allocation. In Figure 8, we see the effects of the delay introduced in the dynamic scheduling due to the scheduler working on allocating resources for 16 and 20 other `qsub` requests. For this case we also took care that none of the 16 or 20 jobs interfere with the compute node or the accelerator node running under the DAC environment. The lightly shaded region depicts the time spent by the scheduler in servicing the earlier `qsub` requests and darker regions show the time spent on servicing the dynamic request. Clearly, the larger the workload handled by Maui at the time of arrival of the dynamic request, the longer the waiting time for the dynamic request to be serviced.

Finally, Figure 9 compares the time taken for dynamic allocation of one accelerator (excluding the time consumed by the MPI operations) in the case of three compute nodes (*A*, *B* and *C*) from three distinct jobs sending a dynamic request each during the same time. Clearly, due the serial processing of the dynamic requests by the `server`, compute node *C*, as shown in the graph, suffered a longer waiting time.

In general, we can observe from all the scenarios that the time taken for dynamic allocation of accelerators to compute nodes usually lie in the range of sub-seconds. For real-world applications, such an overhead is negligible and may be traded off for an availability of more resources to offload computations.

V. Related Work

Efficient resource management for heterogeneous cluster systems is a field well-investigated due to the rise in the usage of accelerators in the recent years. In our efforts, we focus on integrating network-attached accelerators in cluster systems and providing flexibility in using them. At the time of writing, we are not aware of any other frameworks that use network-attached accelerators and support both static allocation before job start and dynamic allocation during runtime.

With regards to using remote GPUs, Zillians Inc. advertised a solution called V-GPU [9] for dynamic GPU provisioning in clouds which enables clients or compute nodes to offload computations onto remote GPUs. Similar to the DAC Architecture, the number of GPUs associated with the clients can be reconfigured at runtime. However, at the time of writing, neither the software itself nor any information on the dynamic resource handling mechanisms have been published. Other GPU virtualization frameworks, such as rCuda [10], vCuda [11], and MGP [12] enable computation offloading on remote GPUs. However they do not support dynamic allocation of remote GPUs to compute nodes.

Other efforts to introduce flexibility in heterogeneous clusters mainly proposed solutions for effective scheduling for dynamic allocations in [13], [14] and [15]. Even in homogeneous clusters, dynamic resource allocation was mainly studied as a scheduling problem for evolving and malleable applications in [16] [17] [18]. This is mainly due to the technical challenges involved in enabling these features in a resource management system. However, the focus of our work was on enabling the dynamic resource allocation facilities in the resource manager through which we understand the practical advantages and challenges. While we do not provide the best scheduling policy for dynamic requests by giving them top priority, we plan to implement a better policy in the future.

In the similar lines, Cera et al. [19] developed support for malleable MPI applications in the OAR resource manager. MPI applications can acquire resources through `MPI_Comm_spawn()` and new processes can be started on the newly available resources. While we provide the same for the network-attached accelerators in the DAC environment, with little extensions to our modified TORQUE resource manager, any malleable application could be supported. Our extensions to TORQUE is not restricted to be used only with the Maui scheduler. Any scheduler capable of dynamic scheduling and allocation can be integrated with our version of TORQUE. Since TORQUE is already widely used in many cluster systems around the world, such a portable solution comes only as an advantage.

VI. Conclusion and Outlook

During the last couple of years, accelerators have gained increased importance and already play a vital role in today's heterogeneous cluster systems. Given the recent advances in accelerator technology, network-attached accelerators seem to be one of the next logical steps. Integrating these devices in cluster systems requires the support of the batch system to fully leverage those new resources and their usage scenarios. In this paper, we presented a dynamic batch system based on TORQUE and Maui which integrates the use of network-attached accelerators. Independent of the architecture of the accelerators in the system, we showed how our batch system can be seamlessly used for flexible accelerator usage scenarios, as demonstrated with the Dynamic Accelerator-Cluster Architecture. Such a dynamic allocation and deallocation facility contributes to optimized utilization of cluster resources. The experimental evaluations clearly show that the overhead of dynamic allocation of accelerators is negligible for real-world applications as the time consumed for dynamic allocations lie in the range of subseconds. Our batch system can be successfully installed on heterogeneous clusters employing both node-attached and network-attached accelerators. In the future, we plan to improve our batch system with fault tolerance and better scheduling policies which includes allocating less number of accelerators in the case where enough accelerators were not available during a dynamic request. As scheduling dynamic requests with top priority may lead to unfair usage scenarios, we plan to build better scheduling policies taking fairshare into account. Along with it, we also plan to extend our batch system to support evolving and malleable jobs under any execution environment in cluster systems. Considering the wide acceptability of TORQUE/-Maui batch system, we believe that our improved batch system could be easily deployed in production systems.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under Grant Agreement n° 287530.

References

[1] D. B. Jackson, Q. Snell, and M. J. Clement, "Core algorithms of the maui scheduler," in *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP '01. Springer-Verlag, 2001, pp. 87–102.

[2] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. ACM, 2006.

[3] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[4] D. A. Mallon, N. Eicker, M. E. Innocenti, G. Lapenta, T. Lippert, and E. Suarez, "On the scalability of the clusters-booster concept: a critical assessment of the deep architecture," in *Proceedings of the Future HPC Systems: the Challenges of Power-Constrained Performance*, ser. FutureHPC '12. ACM, 2012, pp. 3:1–3:10.

[5] Nvidia. Project denver. http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/.

[6] J. Dongarra and et al., "The international exascale software project roadmap," *International Journal on High Performance Computing Applications*, pp. 3–60, 2011.

[7] S. Rinke, D. Becker, T. Lippert, S. Prabhakaran, L. Westphal, and F. Wolf, "A dynamic accelerator-cluster architecture," in *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ser. ICPPW '12. IEEE Computer Society, 2012, pp. 357–366.

[8] R. L. Henderson, "Job scheduling under the portable batch system," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, ser. IPPS '95. Springer-Verlag, 1995, pp. 279–294.

[9] Zillians. (2011) V-GPU. http://www.zillians.com/vgpu/. [Online]. Available: http://www.zillians.com/vgpu/

[10] J. Duato, A. J. Pena, F. Silla, J. C. Fernandez, R. Mayo, and E. S. Quintana-Orti, "Enabling cuda acceleration within virtual machines using rcuda," in *Proceedings of the 2011 18th International Conference on High Performance Computing*, ser. HIPC '11. IEEE Computer Society, 2011, pp. 1–10.

[11] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU accelerated high performance computing in virtual machines," in *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2009.

[12] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *Proc. of the International Conference on Cluster Computing (PPAAC Workshop)*. IEEE, Sep. 2010.

[13] D. Kumar, Z.-Y. Shae, and H. Jamjoom, "Scheduling batch and heterogeneous jobs with runtime elasticity in a parallel processing environment," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, may 2012, pp. 65 –78.

[14] S.-S. Boutammine, D. Millot, and C. Parrot, "A runtime scheduling method for dynamic and heterogeneous platforms," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, 2006, pp. 8 pp. –282.

[15] L. Barsanti and A. C. Sodan, "Adaptive job scheduling via predictive job resource allocation," in *Scheduling Strategies for Parallel Processing*. Springer Verlag, 2006, pp. 115–140.

[16] C. Klein and C. Pérez, "Towards scheduling evolving applications," in *Proceedings of the 2011 international conference on Parallel Processing*, ser. Euro-Par'11. Springer-Verlag, 2012, pp. 117–127.

[17] B.-P. Gan and S.-Y. Huang, "Scheduling dynamically evolving parallel programs using the genetic approach," in *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, vol. 1, 2000.

[18] J. Hungershofer, "On the combined scheduling of malleable and rigid jobs," in *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, 2004, pp. 206–213.

[19] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux, "Supporting malleability in parallel architectures with dynamic cpusets mapping and dynamic mpi," in *Proceedings of the 11th international conference on Distributed computing and networking*, ser. ICDCN'10. Springer-Verlag, 2010, pp. 242–257.