This is a preliminary version. Final version is to appear in Proceedings of IEEE ICAC 2010.

Predictable Time-Sharing for DryadLINQ Cluster

Sang-Min Park and Marty Humphrey
Department of Computer Science, University of Virginia
Charlottesville, VA 22904, USA
{sp2kn | humphrey}@cs.virginia.edu

ABSTRACT

This paper addresses the scheduling problem that popular data parallel programming systems such as DryadLINQ and MapReduce are facing today. Designing a cluster system in a multi-user environment is challenging because cluster schedulers must satisfy multiple, possibly conflicting, enterprise goals and policies. Particularly for these new types of data-intensive applications, it continues to be a challenge to simultaneously achieve both high throughput and predictable end-to-end performance for jobs (e.g., predictable start/end times). The conventional approach of scheduling these types of jobs is to attempt to determine a best mapping between a task and a node before the job executes, and the scheduling system ceases to be involved for a given job once the job starts executing. Instead, as described in this paper, we define a reactive *containment and control* mechanism for scheduling and executing distributed tasks, schedule the jobs, and then continually monitor and adjust resources as the job executes. More specifically, a DryadLINQ task in our system is contained in virtual machine and distributed controllers regulate progress of the task at runtime. Using online, feedback-controlled VM CPU scheduling, our system provides a job a capability to speed-up or slow-down progress of concurrent sub-tasks so that the job can make predictable progress while sharing system resources with other jobs. The new capability allows an enterprise to enforce flexible scheduling policies such as fair-share and/or prioritizing jobs. Our evaluation results using five well-known DryadLINQ applications show the implemented distributed controllers achieve high throughput as well as predictable end-to-end performance.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design – *distributed systems.*

General Terms

Algorithms, Design, and Experimentation

1. INTRODUCTION

Recently, data parallel cluster frameworks such as MapReduce/Hadoop [1][2], and DryadLINQ [3] have gained great popularity. Because they offer scalable performance on commodity clusters, increasing numbers of enterprise applications such as web-log mining and machine learning, as well as scientific analysis, have been developed. Beyond the cluster scale, many believe that MapReduce and DryadLINQ are promising abstractions in the Cloud. As an example, Amazon.com recently started to offer a Hadoop implementation on EC2 instances [4].

While the SQL-like operations of MapReduce and DryadLINQ greatly simplify programming a cluster, scheduling data-parallel jobs on a shared cluster remains a hard problem. In general, an enterprise wants to design a cluster system that is optimized for multiple performance metrics including high throughput, fairness, and low response time for interactive jobs. However, optimization on one metric often conflicts with the others. For example, achieving high throughput via concurrent job execution causes issues of fairness and increased latency for interactive jobs. Therefore, advanced algorithms take multiple metrics into account. A good example is Quincy, Microsoft's internal Dryad scheduler that achieves fair-share as well as high throughput [5]. However, even these sophisticated algorithms have limitations when considering long-running computations. A scheduler can kill a task if running another task on the occupied node is a more immediate concern. Killing a task wastes the work and becomes a serious issue when workloads are mostly long-running jobs. In addition, fairness is in terms of the number of allocated nodes regardless of how much resources a job actually consumes at run-time (e.g., switch bandwidth). We believe this is a fundamental limitation of traditional task scheduling in which a task is *"mapped"* to a node according to a scheduler's policy. Once a mapping is complete, it is hard to reflect changes in the schedule.

In this paper, we approach the scheduling problem from a different direction by leveraging our earlier research [6][7]. We present the design and implementation of distributed controllers for DryadLINQ cluster using our reactive containment and control mechanism. The distributed controllers regulate the throughput of DryadLINQ tasks so that a job's end-to-end performance can be predictably manipulated even in the presence of concurrent jobs. Instead of killing a task, the scheduler forces the task to run at low clock speed and make a reduced rate of progress, so that *both* activities can predictably be run on the shared resource. Using the controllers, we implement enterprise scheduling goals and policies while achieving high throughput.

We summarize contributions of this paper as follows:

1) We present a mapping from classic control theory to the problem of scheduling a data parallel cluster. We show the use of standard, model-based control is a viable approach to implement a policy-based cluster scheduler.
2) We show that the *containment and control* mechanism is a valuable addition to the conventional cluster scheduling.
3) We extend our earlier work to consider for the first time cluster scheduling and data-intensive application-specific frameworks.

The rest of this paper is organized as follows: In section 2, we present related work, followed by the overview of data intensive computing and DryadLINQ in section 3. Section 4 presents the control-theoretic design of distributed controllers for DryadLINQ cluster. Section 5 presents the integration of distributed controllers with a queue-based cluster scheduler. Section 6 presents evaluation results on our local cluster testbed. In Section 7, we conclude with discussions.

2. RELATED WORK

Loosely coupled data parallel programs on commodity cluster have gained great popularity after Google introduced their MapReduce paper in 2004 [1]. Apache Hadoop [2] is a popular open-source implementation of MapReduce and has been used for data mining in enterprise (e.g., Yahoo!, Facebook) [8], as well as for academic research [9][10][11]. Dryad is another data parallel execution engine from Microsoft Research [12]. Dryad uses directed acyclic graph (DAG) as a flexible expression of distributed computation compared to the stricter pipeline of Map-Partition-Reduce. DryadLINQ is an implementation of LINQ, a .NET-based, SQL-like data query language, for Dryad cluster [3]. Compared to MapReduce, DryadLINQ offers an extended set of data operations to simplify writing a complex algorithm. It is also shown that the extended operators allow better optimization of distributed computations [13]. To achieve ease-of-programming on MapReduce, there are domain-specific languages on top of Hadoop, such as Sawzall [14] and Pig [15], which translate user-friendly interfaces to multiple phases of Map and Reduce.

There are in general two approaches for scheduling data parallel jobs in a shared cluster. In a coarse-grained resource-sharing model, a cluster is managed by batch queuing systems such as Torque and Microsoft HPC cluster. Once a job is allocated on compute nodes, it runs exclusively until termination or the queue's time limit. Hadoop-On-Demand (HOD) is an example of this model [16]. HOD uses Torque to provision a virtual Hadoop cluster over a large physical cluster. The academic release of DryadLINQ uses a queue-based Windows HPC cluster. There are two significant issues with the coarse-grained sharing. The first problem is low throughput. Following Amdahl's law, in general, a job with parallel tasks cannot fully use the allocated nodes. Underutilization of a cluster leads to low throughput in long term. Secondly, it is difficult to express and enforce enterprise policy with a queuing scheduler. Although there are limited tools to implement policy including node partitioning, reservation and priority-based preemption, they often require an administrator's intervention. Furthermore, it is difficult to enforce policy on a running job, if the system has no support for checkpointing.

Another major approach to the cluster scheduling is fine-grained resource-sharing in which multiple jobs are run concurrently. Hadoop's built-in scheduler is based on this model. Fine-grained sharing achieves high throughput because a job's execution is interleaved with others. The immediate problem of the approach, however, is that a job's performance is significantly affected by concurrent jobs. Once a *"big"* job, in terms of numbers of tasks and running time, is scheduled, small jobs suffer from delayed response time. Furthermore, if a job is highly data intensive and saturates a bottleneck resource, such as a switch, the jobs sharing the same resource suffer from degraded performance. The Hadoop fair scheduler addresses the problem by considering a job's fair-share of cluster nodes [17]. For N concurrent jobs on K cluster nodes, a job cannot use more than K/N nodes at a time. However, this simple heuristics conflicts with data-locality optimization because, in the worst case, all K/N nodes may access data across a bottlenecked switch. Quincy, Microsoft's internal Dryad scheduler, addresses the fair-share and data-locality in a single framework [5]. It addresses the limitation of basic fair-share heuristics by mapping the scheduling problem to min-cost flow, a graph matching problem, which generates globally optimal schedule in terms of data locality and fair-share. Zaharia et al. presents heuristics to implement fair-share and data-locality for a Hadoop cluster [18]. In our opinion, there are two significant limitations in the fine-grained schedulers. The first is the impact of long-running jobs. Quincy has shown good fairness for a workload of many short-running jobs and few long-running jobs. However, it often *kills* a task to enforce a new, globally optimal schedule. Preemption by killing is a serious issue if the workload has a large fraction of long-running jobs. Secondly, the existing schedulers do not care about fairness in job's runtime. While allocating K/N nodes to a job seems a fair decision, the job's performance can be significantly affected by the characteristics of concurrent jobs. For example, a job's excessive use of switch bandwidth is not fair to concurrent jobs. Our approach, based on containment and control, is a good solution to these limitations.

Control theory is at the core of our modeling and scheduling DryadLINQ cluster [19][20]. The theory has been used in a variety of software services including real-time scheduling [21][22], QoS for web servers [23], storage systems [24], and QoS control in virtualized environments [25][26][27]. Our earlier works showed the basic control-theoretic methodology to solve unpredictability of HPC applications [6][7]. We showed that we can precisely regulate progresses of concurrent, single-threaded HPC jobs that are running inside VMs in a VMM server. Using the precise control, we can successfully meet deadlines of concurrent jobs. This paper extends the techniques presented in the previous works by implementing cluster-level distributed controllers and integrating them with DryadLINQ runtime. To the best of our knowledge, we are one of the first to present the control-theoretic scheduler for a data-parallel cluster.

3. DATA INTENSIVE COMPUTING

3.1 Data Intensive Computing - Overview

Loosely-coupled distributed computing has been a popular programming model across scientific domains and enterprise computing. In the model, a job consists of a large number of worker tasks that are managed by a root task, which performs task scheduling, data distribution, and remote execution. Communications are typically limited between the root and worker tasks that exchange control information. Condor is the well-known framework that implements the model [28]. While the loose-coupling with flat control structures has found many applications such as a bag-of-task model and parameter sweep experiments, more recent frameworks such as DryadLINQ and MapReduce organize distributed computations in a more structured way. By enforcing a particular structure of distributed computations, the frameworks provide users with simple programming interfaces and semantics, with which a framework map users' programs onto distributed resources efficiently.

In MapReduce and DraydLINQ, streamlined computation, inspired by functional programming, is one of most unique features that are relevant to our work. The streamlined computation is in contrast to conventional, imperative programming by which a programmer writes a complex, domain-specific algorithm that process data set as a whole. Distributed tasks of an imperative program typically performs the following steps: (1) copying input files from remote storage to a local disk, (2) reading whole or parts of the file's contents into memory, (3) processing in-memory data, and (4) finally writing processed results to a local storage. All these steps are done in isolation in sequence. However, in streamlined computation, whole steps are carried out concurrently as a pipeline. For example, executing

write() statement in step (4) initiates execution of the DryadLINQ pipeline and triggers *read()* in step (1), reading the first element (e.g., a string line) from the remote storage. Then the element is processed throughout the pipeline. At one instant, all four steps – copying, reading, processing, and writing – can be carried out concurrently (depending on an operator's type, there is an exception to writing, because some operators should process all contents before producing the computation's results; sorting is an example). Using the streamline model in DryadLINQ, we can control whole data-intensive pipeline using a knob implemented by virtualized CPU (i.e., I/O workloads can be controlled using CPU). In our experiments with representative DryadLINQ operators, we observed that there is a good linear fit from VCPU to operators' throughput. In addition, progress at one point of a pipeline largely corresponds to progress of the entire task, encompassing remote I/O and computation. Therefore, we can "*sense*" progress of the complex, distributed task by placing a *sensing* operator into one position of a pipeline. Also, it is easy to implement the sensing function because the frameworks provide similar sensor libraries for monitoring and debugging purposes.

3.2 DryadLINQ - Overview

```
IEnumerable<string> GetTopKWord(IEnumerable<string> input, int k)
{
        return input
        .SelectMany(x => x.Split(' '))
        .GroupBy(x => x)
        .OrderByDescending(x => x.Count())
        .Take(k)
        .Select(x => x.Key);
}
```

Figure 1(a): An example LINQ program that returns top *k* words in text. The program splits string lines into words that are grouped with the same word. Then sorting is done to the grouped words with the number of word's occurrence as a key. Finally, the first *k* grouped words are projected into string-type results.

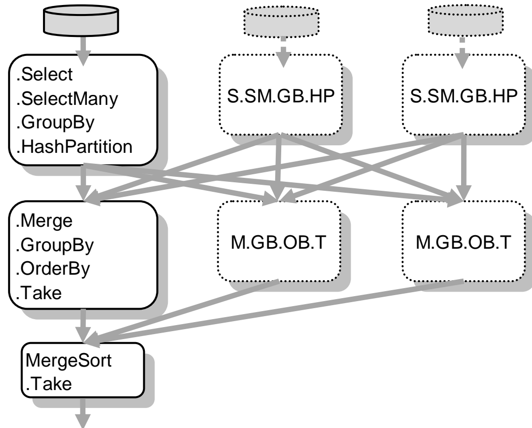


Figure 1(b): Dryad execution plan for *GetTopKword()*. Grouped words are distributed over cluster nodes via hash partitioning, followed by grouping and sorting in parallel.

DryadLINQ is a modern example of data intensive computing. The DryadLINQ programming environment consists of *LinqToDryad*, a compiler for translating LINQ expressions into a Dryad execution plan, and the Dryad runtime system.

LINQ and LINQ to Dryad

Language Integrated Query (LINQ) is a set of C# language constructs and .NET runtime that provide type-safe queries over enumerable objects [29]. The LINQ system offers a hybrid of declarative and imperative programming by which a programmer writes type-safe computation over data sets. The various runtimes provide storage and platform-specific implementations, for example LINQ-in-memory, LINQ to SQL, and PLINQ. The notable LINQ operations include *Select*, *Where*, *Join*, *GroupBy*, and *OrderBy*. Each operation is functionally similar to SQL counterparts. DryadLINQ is an implementation of standard LINQ expressions for cluster computers. Figure 1(a) presents an example LINQ program that is similar to *WordCount*.

LinqToDryad is an open-source library that compiles a LINQ program into a Dryad execution plan. A DryadLINQ programmer writes a program in much the same way as writing a desktop LINQ program. When the program's flow invoke derived LINQ methods in DryadLINQ library, the *LinqToDryad* walks through LINQ expression tree (that is analogous of abstract syntax tree in compilers), emits translated C# source codes which are compiled into a DLL. Another output of *LinqToDryad* is a Dryad execution plan in DAG format. Both the library and DAG files are shipped to a cluster along with a submission of the Dryad master task. Note that the compiled methods implement a chain of standard LINQ operators with some extended operators to support Dryad-specific functions including input merging, output partitioning, and file I/O. In other words, distributed tasks of original LINQ program are extended LINQ programs. Figure 1(b) illustrates the Dryad execution plan for the program in Figure 1(a).

Dryad Runtime

The Dryad runtime manages execution of distributed tasks expressed in DAG.

Inter-Process Communication

The Dryad runtime relies on a networked file system (i.e., NTFS share) as a default IPC mechanism. When a Dryad worker executes a task (extended LINQ expressions), the master task gives locations of input files to the worker who reads and de-serializes raw data into .NET objects (i.e., DryadLINQ supports strong type system). Note that LINQ's streamlined computation is implemented with Dryad's file-based IPC. When *write()* is called in the last operation of pipeline, *read()*is executed in the first operation, reading the elements from the remote storage.

Scheduling

The Dryad master task dispatches worker tasks to idle compute nodes in the precedence order denoted in a DAG. The Dryad scheduler implements data-locality optimization, fault-tolerance via deterministically re-executing a failed task, and duplicated execution to avoid performance losses due to "straggler".

The DryadLINQ academic release uses coarse-grained sharing model using a queue-based Windows HPC cluster. Quincy is Microsoft's internal Dryad scheduler based on fine-grained sharing model [5]. Because we have no knowledge about inner-workings of the latter, we limit our discussion on the academic release. When a DryadLINQ program submits a Dryad master task to the head node of a HPC cluster, the task is placed into a queue of the head node, which schedules incoming jobs in FCFS order. When executed, a Dryad job is given exclusive access to entire compute nodes of a cluster. The DryadLINQ program running on a client desktop is pending on the cluster job and resumes execution after the cluster job completes successfully. More details of DryadLINQ system are found in the literature from Microsoft Research [3][12]

4. DryadLINQ TASK CONTROL

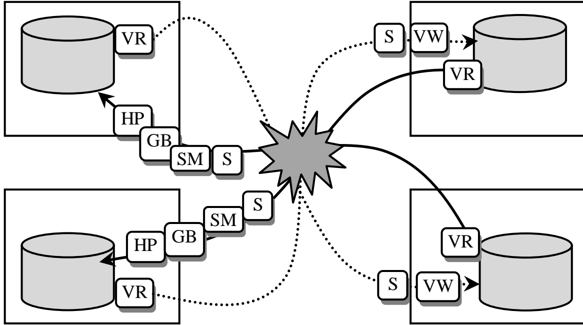


Figure 2: Distributed, streamlined execution of two DryadLINQ jobs, *GetTopKWord* and *Grep*. The acronyms are as follows: VR – VertexRead, HP – HashPartition, GB-GroupBy, SM-SelectMany, S-Select, VW-VertexWrite.

A DryadLINQ job consists of many streamlined tasks that are executed in parallel. To increase a cluster's throughput, we run multiple such jobs concurrently, as illustrated in Figure 2. When multiple jobs are run concurrently, unconstrained competition in data paths results in highly unpredictable execution of each job. The streamlined tasks of each job compete for limited bandwidths in disks, NICs, and switches. To achieve high throughput with predictable end-to-end performance, we need a mechanism to precisely "*control*" throughput of each DryadLINQ task. To ensure good performance of a task, we need to ***throttle up*** the task's throughput. To ensure good performance of other tasks, we need to ***throttle down*** the throughput and release shared resources. This observation leads us to design a feedback loop for controlling a task's byte-level throughput.

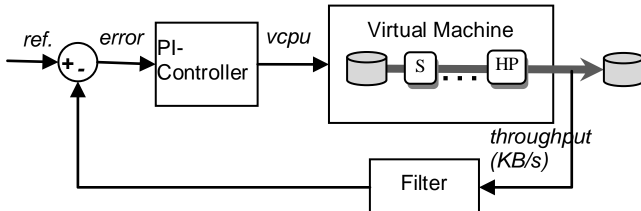


Figure 3: Feedback control loop for DryadLINQ task

Instead of an ad hoc feedback loop, we design a loop systematically using the standard, model-based control architecture with PI controller. Figure 3 presents the feedback loop with the sensor, the actuation, and the controller. One possible approach to create a model-controller is to design an application-specific feedback loop. We do not believe this is a scalable approach because a DryadLINQ programmer must undertake the profiling and controller design, which would be too overwhelming to be practical. Because LINQ defines a manageable set of standard operators and DryadLINQ introduces only few extensions, we can create models and controller parameters for few characteristic LINQ operators that are compute and data intensive. Such operators include *Select*, *OrderBy*, *GroupBy*, and *Join*. A complex LINQ algorithm is implemented with those well-defined operators. Thus, our run-time system looks up operators in a compiled library and choose the model and controller parameters for the tasks automatically. When more than two characteristic operators appear in a LINQ pipeline, we pick the operator that is more compute and data intensive. For example, if a pipeline consists of *Select* and *OrderBy*, we use the model and controller of *OrderBy* because its steady-state throughput at full CPU cycle is lower (see Table 1).

4.1 Sensing and Actuation

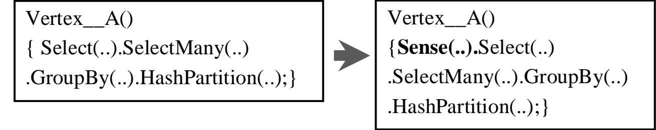


Figure 4: The modified Dryad vertex with sensor

A sensor measures throughput of a DryadLINQ pipeline in the unit of KB/s. As we discussed previously, the sensed throughput encompasses the throughput of *read()* at source storage, CPU's operation on in-memory objects, and optionally the *write()* into destination storage. Although, any of the three discrete points can be a bottleneck, the most frequent source is the *read()* from source storage. The in-memory LINQ operations become hardly a bottleneck especially when PLINQ, a multi-core-version of LINQ, is enabled at compute nodes. We implement the sensor as a C# function that returns objects of type *IEnumerable<T>* and modifies *LinqToDryad* to insert the sensor function when the library compiles the original LINQ expression. As shown in Figure 4, the translated, distributed version of LINQ programs include the sensor function with signature *IEnumerable<T> sense(IEnumerable<T>)*. The sensor enumerates objects with lazy evaluation (*yield* keyword in C#) so that when an operator in LINQ pipeline starts enumerating objects, the enumeration in the sensor is triggered and the throughput is measured. The measured instant throughput is periodically reported to a controller.

We use fine-grained VM CPU scheduler as an actuation mechanism. In modern virtual machine managers such as Hyper-V and Xen, the VMM's credit-based scheduler allows one to control CPU usage per VM accurately and at fine granularity, as if the VM is run at configurable clock speed. We run a DryadLINQ task in a Hyper-V VM and use credit cap interface, via a library call, to dynamically adjust a task's throughput. The credit cap sets an upper limit of the VM's CPU usage (we call it VCPU hereafter). When the VCPU is reduced, the task pipeline running inside the VM does not get enough CPU cycles and its throughput is reduced almost linearly with respect to reduced VCPU.

4.2 Modeling via System Identification

Our hypothesis is that a simple liner model can capture the relationship between VCPU and a throughput of DryadLINQ operators. We use a first-order linear model to represent the relationship:

$$T(k) = aT(k-1) + bC(k-1) \quad (1)$$

In the model, $T(k)$ and $C(k)$ represent the throughput and VCPU cap at time k, respectively. The model tells that the measured throughput at time k is determined by the measurement at time *(k-1)* and the VCPU at time *(k-1)*. Note the throughput in the previous cycle affects the measurement at the current cycle because changes in the actuation value in the previous cycle may not be fully reflected to the current measurement. In our previous works on controlling data-intensive scientific workloads, we observed that there is some non-trivial latency between the actuation time and the changes in the output [6]. In the work, we showed the simple first-order model successfully captures the relationship in the data intensive workloads.

We use System Identification, a black-box modeling approach to derive the model parameters *a, b* of equation (1). For each LINQ operator to profile (e.g., *Select, OrderBy*), we run the

Dryad-version of the operator that reads data from a remote storage and write results to a local storage. The operator benchmark is run inside the VM while we vary VCPU following a discrete sine wave. We run the benchmark on one of our cluster compute nodes (Intel 8-core CPU). The parameters of the system identification can be summarized as follows:

- Sampling period: 2 seconds
- Period of sine wave: 14 seconds (7 throttling in a cycle)
- Duration of experiments: 112 seconds (8 sine cycles)
- Amplitude of VCPU: 0 – 100 %

Note the choice of the parameters is a result of our trial-and-error process to find the good linear fit. With the input (*C(k)*) and output (*T(k)*) traces obtained from the profiling, we run a least-square estimator to choose the right values of *a* and *b*. For some operators, the estimator preprocesses the traces to filter out a few cycles that contain obvious noise in the output. Table 1 lists the first-order model parameters for different LINQ operators. The table includes R^2 values for validation and the model's prediction about steady-state throughput at 100% VCPU.

Table 1. First-order model parameters for important LINQ operators

	Parameters	R^2	Steady-state output at 100% vcpu
Select	a=0.26, b=101.73	0.85	52.2 MB/s
GroupBy	a=0.05, b=47.31	0.74	2.5 MB/s
OrderBy	a=0.13, b=57.13	0.87	3.3 MB/s
Join(HashJoin)	a=0.32, b=187.54	0.93	13.8 MB/s

Table 1 shows good linear fit for a broad range of VCPU from 0 to its maximum (amplitude of system identification). This may sound strange because it implies CPU cycle is the factor that limits I/O throughput. The reason is a streamlined computation of DraydLINQ (and MapReduce as well). In one sampling period, a DryadLINQ task repeatedly performs four steps – copying, reading, processing, and writing – of a pipeline. When VCPU is throttled down, the processing step is slowed down, which in turn slows down the I/O operation on remote storage. In fact, the table shows that the most data-intensive operator, *Select*, exhibits better linear fit than the most compute-intensive operator, *GroupBy*. This linear fit would not be possible with imperative applications with isolated sequences of I/O and CPU phases. In that case, it would be necessary to devise more than one actuation knob (e.g., VCPU and token bucket rate limiting) and perform modeling separately for each phase.

4.3 Feedback Controller Design

We use Proportional-Integral (PI) control as control logic in the feedback loop. We chose the PI controller because it has been shown to work well for data-intensive HPC workloads [6][7]. The PI control law has the mathematical form:

$$C(k) = C(k-1) + (K_P + K_I)E(k) - K_P E(k-1) \qquad (2)$$

With a given error, *E(k)*, (reference– measured throughput) of the feedback loop, control parameters K_P and K_I determine the controller output, *C(k)*, which is the Hyper-V's VCPU. The PI control law (2) achieves two important control goals: zero steady-state error and controller speed. The integral term in the control law ensures that the controller compensates all past errors thereby achieving accuracy of the loop. The proportional term allows the controller to react rapidly because the controller output is proportional to a control error in a previous cycle.

Controller design involves analyzing the properties of target system through profiling; setting up control goals in terms of stability, accuracy, and speed; and finally choosing the controller parameters that satisfy the control goals. Z-transformation is often used to encode time-domain linear equations to transfer functions which can be conveniently combined via algebraic manipulations. Because the space does not permit us to present every detail of control-theoretic design, we refer readers to control text books [19][20] and our earlier works [6][7].

The goal of controller design is to ensure that control parameters satisfy the stability, accuracy, and settling time requirements. In our system, the controller design has been mostly automated by custom MATLAB scripts. When the model parameters for a specific DraydLINQ operator are given, the controller design heuristics find the PI parameters, K_P and K_I, that makes the loop stable and fast. The heuristics uses the well-known control theorems when it searches for near-optimal control parameters. More details of the control-tuning heuristics can be found in [7]. We discuss on important controller properties in both the theoretical and empirical stand point.

Stability

Stability is one of the most important considerations in controller design. If a controller is unstable, the controller's output, VCPU, oscillates between the upper and lower extreme values. Theoretically, the PI control law can satisfy stability requirements (Bounded Input, Bounded Output stability, more precisely), by finding K_P and K_I that results in a closed-loop transfer function with poles of the loop less than magnitude 1. However, when controlling a DryadLINQ task (data intensive jobs in general), this theoretic result should be used as a guideline in practice, not as an absolute proof. When multiple data-intensive tasks compete for limited I/O bandwidths, they may cause huge disturbances to each other, and in some rare occasions, the unmanageable disturbance causes the controller's output to oscillate between two extreme values. Furthermore, we have distributed feedback-controllers that run autonomously with no explicit coordination. It is known that guaranteeing stability of distributed controllers is difficult. Nevertheless, the formal proof is valuable because it significantly reduces the chances of unstable controller. An ad hoc control is more vulnerable to unstable condition.

Accuracy

As we discussed earlier, the PI control law ensures the loop achieves 100% accuracy as it accounts for all previous errors. In our work, this means, by changing Hyper-V's VCPU dynamically, the task's throughput eventually equals the assigned target throughput. As shown in the evaluation, we observed that as long as a loop is stable (VCPU is not saturated for long periods), the controller achieves almost 100% accuracy in reality.

Controller Speed

It is desirable that a controller quickly settle to a steady-state value in the event of reference changes or disturbances. For instance, a task's throughput should be rapidly reduced, if there is an urgent task scheduled on a shared resource. There is a theorem that approximates the settling time of the PI-control loop [19][20] and our controller design heuristics use the theorem. The heuristics iteratively search for the K_P and K_I that is stable and has the shortest settling time, until the heuristic's running time limit. Therefore, the controller parameters chosen by the heuristics are near-optimal in terms of settling time. In the empirical evaluation, we found the controller speed is mostly comparable to the off-line prediction by the heuristics. Thus, the control theorem about settling time serves as a valuable guideline.

5. TIME-SHARING DryadLINQ CLUSTER

So far, we presented the feedback control of a DryadLINQ task to achieve a desired throughput. A DryadLINQ job consists of a large number of such tasks that are under control by distributed controllers. We run multiple jobs concurrently on a cluster to increase cluster utilization and aggregate throughput. We assume in this paper that a cluster's throughput capacity is known in advance. Our experimental, 4-nodes cluster has approximately 100 MB/s aggregate throughput (a disk has 25MB/s aggregate, sustainable bandwidth). A job is allocated a fraction of a cluster's capacity according to cluster policy. For example, if fair-share is the policy, the job would be given 50 MB/s as the job's throughput capacity. It is a role of the job's global scheduler to split the assigned capacity to the job's sub-tasks in a way to reduce *makespan*. This section presents the integration of distributed controllers with a queue-based cluster scheduler.

5.1 Hybrid Scheduler with Cluster Queue

While cluster management with queues is convenient and cost-effective, low utilization and low aggregate throughput are serious problems. Furthermore, because jobs are exclusively allocated and run for unknown execution time, it is difficult to implement flexible multi-job policies such as fair-share or immediate execution of interactive jobs. We address the problems by integrating the distributed controllers with a queue-based cluster scheduler (Microsoft HPC cluster). Figure 5 presents the hybrid architecture with distributed controllers. In the architecture, there are more than one cluster queues that manage disjoint set of VMs in the cluster. For example, two VMs of a VMM compute node are managed by two different queues. Thus, the jobs submitted to each queue can be executed concurrently on a physical node. On each VMM layer, there is a controller daemon that runs feedback loops for each task running inside one of its VM.

There is a Global DryadLINQ Scheduler that is tightly integrated with a queue. The scheduler monitors DryadLINQ jobs submitted to each queue and dynamically determines throughput of a job's sub-tasks using the heuristics presented below.

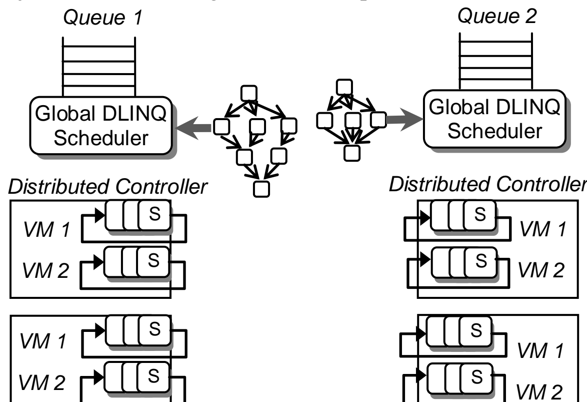


Figure 5: Hybrid cluster architecture with distributed controllers

5.2 Scheduling Distributed Controllers

The global scheduler schedules tasks' throughput, within a job's throughput capacity assigned by cluster policy. The scheduler's goal is to use the limited capacity as efficient as possible to reduce completion time of a job. The global scheduler sets throughput references on distributed feedback controllers. In our current prototype, we implement simple heuristics by which the capacity is evenly distributed to tasks that are ready to run. A typical DryadLINQ job consists of iterative phases of parallel worker and serial merger (e.g., In MapReduce terms, *K* map and 1 reduce for *K* node cluster). Therefore, for a throughput capacity *T* and *K* cluster nodes, the parallel workers and serial mergers are typically allocated *T/K* and *T* respectively as a reference throughput.

Note that setting a target throughput for a task does not necessarily mean that the controller of the task will sustain it. The controller would not be able to keep the reference even at 100% VCPU, if the reference is too high. For instance, when the serial merger is allocated the full capacity of a job (because there's no other available tasks), the bandwidth of a local disk cannot sustain such high throughput. Also setting a target throughput at too high a value can negatively affect the tasks of other jobs sharing a resource. In fact, we may need task-level throughput limits in addition to a job-level capacity. However, we do not enforce task-level limits in our current prototype because the heuristics do not in general saturate capacity of a resource for long periods. The execution time of the serial merger is typically a few seconds. The long-running parallel workers are allocated T/*K* most times. When *N* jobs run concurrently, T_i/K for job $i<=N$ is relatively small and does not in general saturate capacity of a shared resource.

Although the simple heuristics have worked well for our cluster prototype, we believe there can be other useful heuristics for different applications and clusters of different scale. For example, if a frequent failure of a task is expected, letting few tasks run at high throughput would be a better idea than even distribution, because fast completion of some tasks allows early re-execution of the failed task. Also, an explicit coordination between global schedulers would be necessary to avoid unstable conditions due to the isolated operation of distributed controllers. We will explore these ideas in our future work.

6. EVALUATION

We implemented the DryadLINQ distributed controllers and its integration with a queue-based Microsoft HPC scheduler. This section evaluates our implementation in two parts. The first evaluation report traces of a job's throughput to show the controller's behavior in isolation and in aggregation. The second part evaluates an end-to-end performance of the jobs with two scheduling policies, fair-share and differentiated priority.

6.1 Experimental Setup

Our prototype cluster has 4 compute nodes with 8-cores on each node. Because the cluster has been built incrementally, the compute nodes of the cluster are heterogeneous. Table 2 summarizes the configuration.

Table 2. Prototype Cluster Configuration

	CPU	RAM	Disk	O/S
Head Node	AMD Athlon Dual-core at 2.8Ghz	4 GB	1 HDD – 7200 RPM	Microsoft Windows Server 2008 R2 / Microsft HPC Pack 2008
Node 01-02	Intel Xeon (Nehalem) 2 quad cores at 2.26Ghz	12 GB	1 HDD - 7200 RPM	
Node 03-04	AMD Opteron 2 quad cores at 1.7Ghz	16 GB	1 HDD - 7200 RPM	

While we have 8 cores on each compute node, Hyper-V limits 4 VCPUs on one VM. To allow concurrent execution of tasks on a shared physical CPU, we run 4 VMs with 4 VCPUs. Therefore, concurrent jobs running on a node see total 16 cores although there are only 8 physical cores on the node. 16 virtualized compute nodes (4 physical nodes * 4 VMs) are evenly distributed by two queues. When a job is submitted to either queue, there are

8 compute nodes available for a job. The PLINQ, LINQ's multi-core version, is enabled on each compute node.

Our evaluation uses five DryadLINQ applications:

- **Grep** searches for a keyword in distributed text files. Two LINQ operators, *Select* and *Where* are used.
- **GrepByJoin** is similar to *Grep* except that there are multiple keywords to search for. It uses *Join* operator to join the list of keywords with the words in distributed text files.
- **GetTopKWord** counts the number of words' occurrence in large text files and return the top *K* most frequent words. *GroupBy* and *OrderBy* are the two important operators.
- **TeraSort** sorts a set of 100 byte records that is distributed in a cluster. *OrderBy* is the only operator used in this application.
- **SkyServer** is the most complex application in our set. It implements Q18 of SkyServer [30]. We have SkyServer database stored as distributed flat files. The entries in two tables (*photoObjAll* and *neighbor*) are joined to find the celestial object that has the attributes of interest. The query consists of *Select*, *Join*, and *Where*. The Dryad plan for this program consists of 11 different vertices most of which are run in parallel. In a cluster of 8 compute nodes, there are 72 tasks to execute.

6.2 Evaluation of Distributed Controllers

In this subsection, we evaluate correctness of the distributed controllers. Note the goal of distributed controllers is to achieve a desired aggregate throughput (job's capacity), which is fulfilled by correct operation of each feedback controller for a distributed task. As we presented in 5.2., the global scheduler sets throughput references on distributed tasks. The reference throughput for a task is an even distribution of a job's assigned capacity. We run five applications with three increasing capacities. For example, we run *Grep* iteratively with 30 MB/s, 50 MB/s, and 70 MB/s of throughput capacity. Each job is continuously submitted through one of the cluster queues. To evaluate the system with non-trivial disturbances, we run a background job that is continuously submitted through the other queue. The background job in this experiment is *Grep*, which is the most data intensive application in our benchmark set. To keep the aggregate throughput of the jobs (test job + background job) under a cluster's capacity, the background job is under control by distributed controllers and allocated 30 MB/s as its capacity. Figure 6 and 7 report traces of the sensed throughput in the level of task and job, respectively.

Figure 6 illustrates a controller's operation on a DryadLINQ task. Among sub-tasks of a job, we report traces of one, the longest-running task, executed on one of compute nodes. Note a DryadLINQ task is a pipeline of extended LINQ operators. The subtitles of each graph list the operators in each pipeline. A characteristic operator (bold font in the subtitles) on each pipeline is chosen for mapping the control parameters for the task. The graph also shows dynamically changing VCPU (actuation value), which is scaled down by .1, to be presented in conjunction with throughput. In the beginning of each task, the VCPU is set at a constant value (50 out of max 100) and the feedback controller starts to change VCPU while tracking the reference.

Overall, the graphs in Figure 6 show that the distributed controllers track the reference correctly. The task of a *Grep* shows the best controllability among the applications. We summarize our findings regarding the controller's performance:

- In general, the controllers achieve good accuracy. Although there are some periods that measured throughput is above or below the target, especially in the beginning of execution, the controllers eventually compensate the previous errors due to the PI controller's integral term.
- Some operators show unstable actuation despite the good accuracy. While the changes in VCPU in *Grep*, *GrepByJoin*, and *SkyServer* are small, *TeraSort* and *GetTopKWord* show rapid fluctuations in the actuation. We found that *OrderBy* and *GroupBy* operators show irregular instant throughput which causes PI controller to change VCPU to a larger extent.
- It takes about 30-50 seconds to track the new reference (settling time). Because we have 2 seconds as a sample period, this means, 15-25 cycles are necessary to settle to a new reference. This result is comparable to our offline prediction. When control parameters for each DryadLINQ operator is chosen, the model-based predictions for settling time were in the range of 15-20 cycles which is the near-optimal results.

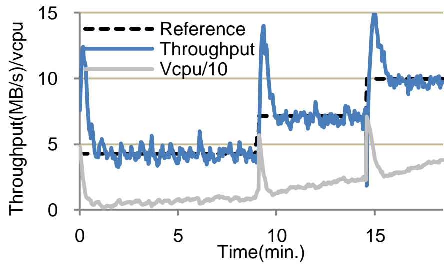
Figure 7 presents the traces of job-level, aggregate throughput. The graph contains throughput sample in every instant (2 sec) and the average of the sampled throughput up to the time in x-axis. We also added instant throughput when there is no control. The unconstrained throughput is measured without a background job. First of all, in all graphs, we can clearly see that a job's aggregate throughput is controlled consistently. Compared to the unconstrained throughput (the high fluctuating lines in the left), the controlled case shows constant throughput that is close to the capacity limit. In addition to a controller's correct behavior in isolation, this confirms that distributed controllers work well in the presence of concurrent controllers. However, there are some periods with noticeable errors. We summarize the reasons:

- While the controllers show good performance in the beginning, the throughput cannot keep up with the capacity in the end (See the low tails of the average throughput). This is because there are not enough concurrent tasks to sustain the throughput. In general, a job's parallel tasks begin execution simultaneously and some tasks completes earlier than the others. Our heterogeneous cluster is the main reason. When only few tasks run, they cannot meet the high reference.
- While the instant throughput shows fast change, the average of instant throughput takes some time to meet the capacity. This is due to the high error in the beginning of a controller's operation (note we have 30-50 seconds settling time).

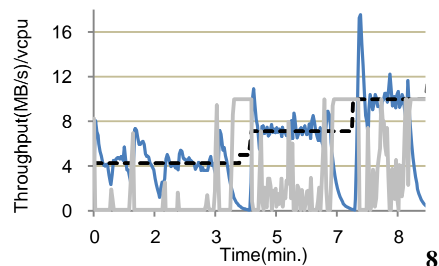
6.3 Evaluating End-to-End Performance

In the previous subsection, we evaluated the controllability of DryadLINQ throughput. This subsection examines how the added capability, throughput control, actually affects a job's end-to-end performance. We evaluate with two scheduling policies, fair-share and differentiated priority. The fair-share policy ensures that concurrent jobs get even share of a cluster's capacity. The differentiated priority assigns an arbitrary fraction of a cluster's capacity to concurrent jobs so that some jobs are given higher capacity than others. In the experiment, we setup two cluster queues to which job instances are continuously submitted. The metric of interest is a job's *makespan* which we define as a job's completion time without accounting for queuing delay. SkyServer is chosen as the job instance because it is the most complex application in our benchmark set. Furthermore, it is a well-known real-life E-Science application. As we explained earlier, there are 72 tasks in a single execution of SkyServer. We run 50 instances of SkyServer job repeatedly. Including the two baselines to compare, there are four experimental results.

1) **Fair-Share policy**: each queue is given 50% of the cluster's capacity. 50 job instances are submitted to either queue.



7.



8.

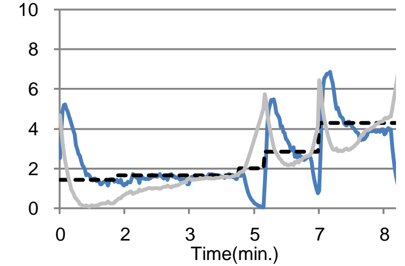


Figure 6(a) Grep: ***Select***, *Where*

Figure 6(b) TeraSort: *Merge*, ***OrderBy***

Figure 6(c) GrepByJoin: ***Join***

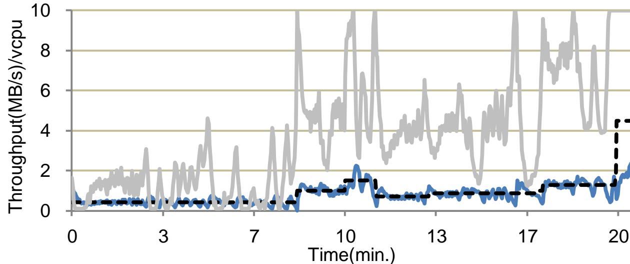


Figure 6(d) GetTopKWord: *Select, SelectMany,* ***GroupBy***, *HashPartition*

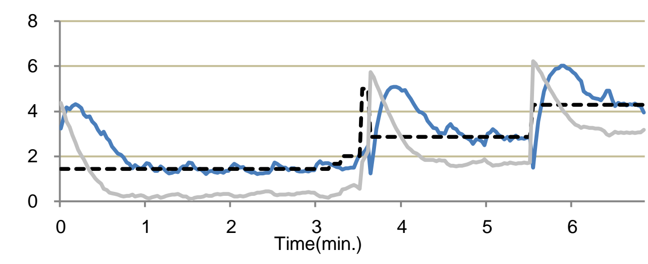


Figure 6(e) SkyServer: ***Join***, *Where, Select, Distinct, HashPartition*

Figure 6: Controlled throughput of a DryadLINQ task. The graph contains three lines: the dotted line is the reference throughput set by global scheduler; the solid line is the actual, controlled throughput; the grey line is the VCPU (actuation) that is scaled down (by .1) to fit in the graph.

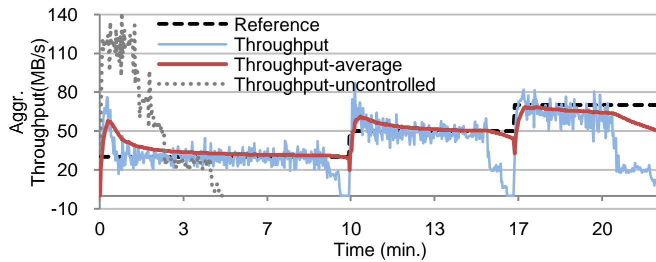


Figure 7(a) Grep

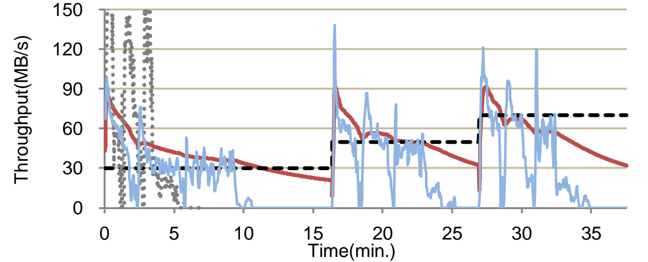


Figure 7(b) TeraSort

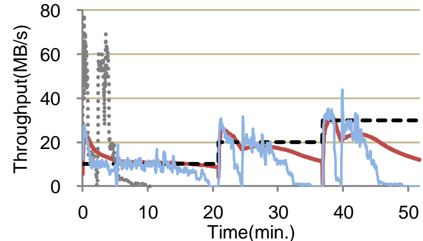


Figure 7(c) GrepByJoin

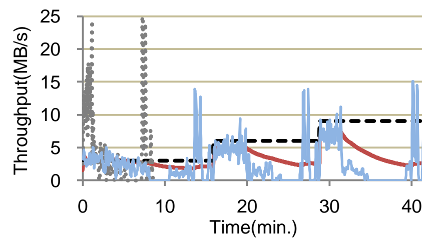


Figure 7(d) GetTopKWord

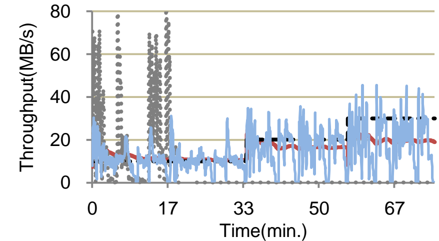


Figure 7(e) SkyServer

Figure 7: Aggregate throughput under control by distributed controllers. Each graph draws four lines: the dotted line is the job's capacity; the fluctuating solid line is the measured throughput at every sample instant (2 sec); the smooth solid line is the average throughput up to the time in x-axis; finally the fluctuating, dotted line in the left is the instant throughput when there is no control.

2) **Differentiated priority policy**: two queues are given 2/3 and 1/3 of the cluster's capacity, respectively. We call them high priority and low priority queue. 50 job instances are submitted to either queue repeatedly. Because high priority jobs finish early, more instances are processed via the high priority queue.
3) **Physical, exclusive cluster**: in this configuration, we setup a cluster of 4 physical compute nodes with a single queue. PLINQ uses 8-cores of each compute node. Windows Server 2008 R2 is the operating system on each compute node. 50 job instances were executed exclusively.
4) **Unmanaged, time-shared virtual cluster**: we setup a VM-based cluster without our control mechanism. The setup includes multiple VMs to share physical CPUs on each compute node. Each VM is given 100% of VCPU. This configuration is identical to our policy use-case, except for the lack of distributed controllers.

Another interesting baseline that we did not investigate is concurrent executions on a physical node. The queue-based HPC cluster does not allow this configuration because a compute node cannot belong to more than one queue (in contrast, two VMs of a node can be members of two queues, respectively). Running 50 job instances took about 4-5 hours. Figure 8 and Table 3 presents the results. In the result, high/low priority jobs correspond to the jobs submitted through corresponding queues in the differentiated priority policy. We discuss each case in turn.

First, the baseline of physical, exclusive cluster shows the fastest and consistent *makespan*. However, there is only one job instance that is running on a cluster at a time, compared to two

concurrent jobs in three other cases. Therefore, the throughput of this baseline is the lowest among the three cases. While running 50 instances took 5h 37 min., the fair-share policy took only 3h 49 min. (i.e., 47% lower throughput).

The baseline of unmanaged, virtualized cluster shows different results. Although the average *makespan* is short (585 sec) and the overall throughput is good, each *makespan* is highly inconsistent. The standard deviation of *makespan* is very high (270 sec). Before a submission of SkyServer, we cannot tell whether the job will finish earlier than 10 minutes. It is clear that concurrency without control causes highly unpredictable end-to-end performance.

We can clearly see the distributed controllers for fair-share policy achieve its goal. 50 job instances consistently show similar *makespan*. 12.8 seconds of standard deviation is even better than exclusive cluster (16 sec.). Also the total execution time representing the cluster's throughput is the best of all four cases.

The differentiated priority policy shows that it achieves its goal as well. On average, a high priority job completes 1.54 times faster than a low priority job (777/503 sec). Note this difference does not exactly corresponds to our capacity assignment for the two priorities (2/3 and 1/3). This is largely due to the fact that some SkyServer tasks do not achieve 100% accuracy in tracking the reference throughput. Some tasks finish too early for a feedback controller to sustain the reference. Still, however, the large difference of *makespan* using throughput control shows the value of this policy. Furthermore, the results show highly consistent *makespan* of all 50 instances.

In summary, the result of this experiment confirms that distributed control of time-shared, virtualized cluster achieves both good throughput and predictable end-to-end performance that is crucial to implement cluster policy. Unmanaged time-sharing shows good throughput but a job's run-time behavior is highly unpredictable. Physical, exclusive cluster guarantees a job's predictable behavior, but it causes low cluster throughput.

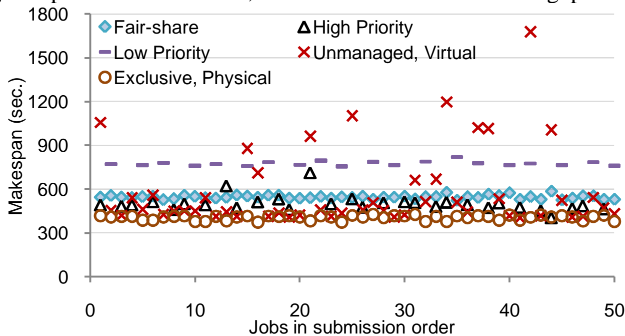


Figure 8: *Makespan* of 50 identical SkyServer jobs

Table 3. Running time statistics of four different cases

	Fair-share	Differentiated Priority	Unmanaged, Virtual	Physical, exclusive
Avg.makespan (sec)	545.7	High: 503.7 Low: 777.6	585.0	405.0
Stdev.makespan (sec)	12.8	High: 59.5 Low: 17.2	270.8	16.0
Sum of makespan	3h 49m	4h 18m	4h 4m	5h 37m

9. DISCUSSION / CONCLUSION

Table 1 lists system identification of four most frequent LINQ operators. In our experiments, system identification of those four operators was sufficient to cover the five applications. This implies that it would be possible to control a wide variety of existing and future DryadLINQ applications by modeling only a handful number of LINQ operators. The cluster administrator would run system identification for only the relevant LINQ operators, and a user's DryadLINQ applications are automatically bound to controller parameters of their LINQ operators. This eliminates application-specific system identification and controller design, and significantly reduces the overhead of control-theoretic scheduling. This is a viable approach because users develop DryadLINQ applications by chaining a set of standard LINQ operators with a user-defined function given as an argument to each operator. The user-defined function is often very simple and has constant time-complexity. For example, in Figure 1(a), the statement, "*OrderByDescending(x => x.Count())*" is given a user-defined function (Lambda expression) that counts the number of elements in a string array. From performance perspectives, the constant-time *Array.Count()* is negligible compared to complex sorting implementation in DryadLINQ run-time. Thus system identification of a LINQ operator can cover a wide variety of DryadLINQ task whose performance is largely determined by the implementation of the operator, not a user-defined function. We use this approach in our evaluation with five applications. Note, however, that some applications would still require application-specific system identification (and control design), if their user-defined functions significantly affect a task's performance. As we increase our set of DryadLINQ applications, we expect to test this idea more thoroughly.

The distributed controllers presented in this paper have no explicit coordination. Therefore there is a possible concern that a controller running in isolation with a given reference can conflict with the other controllers. Insufficient disk or switch bandwidth can cause conflicts among the controllers that share the bottlenecked resource. Currently the coordination is implicitly implemented by distributing a cluster's static capacity to concurrent jobs, followed by an even distribution of the allocated capacity to parallel tasks. Many research attempts to coordinate distributed controllers, especially in autonomous computing area where the focus is to coordinate controllers with conflicting objectives (e.g., power vs. performance [31], platform vs. virtualization management [32]), and in real-time systems area [22] where the goal is to control distributed real-time tasks with a well-defined task model (e.g., task periodicity, deadline, and system capacity). However, to the best of our knowledge, no earlier work addresses distributed control of data intensive applications, such as MapReduce and DryadLINQ, which exhibit very different workload patterns from web transactions and real-time tasks. Our recent work on admission control [33] is perhaps the most relevant research regarding data-intensive workloads. In the work, we created an admission controller that explicitly accounts for both a HPC server's capacity and utilization demand by concurrent computational and data-intensive jobs such that the sum of claimed utilization by concurrent jobs can be kept below the server's capacity. We found, however, that controllers running independently in a HPC server frequently oversubscribe the server and result in unpredictability of many jobs. A feedback was in fact a source of unpredictability because a low-performing job keeps demanding more VCPU from the server. Our solution to the problem was 1) to create a per-server adaptation service that mediates interactions between controllers, 2) by switching a job's performance model depending on the server's current load (pessimistic model when heavily loaded), and 3) by enforcing

priority among controllers such that, when overloaded, some controllers are forced to give up their VCPU allocation. We envision extending the approach to a cluster scale so that a single global controller oversees an admission of a task (and its associated controller) into a cluster, and mediates the interactions between tasks. This one-to-many coordination would be simple and less expensive than many-to-many coordination between distributed controllers.

We integrated the distributed controllers with a queue-based cluster scheduler. It would be very interesting research if we similarly extend our approach – containment and control – to the schedulers based on fine-grained sharing model. Quincy and Hadoop Fair-scheduler are examples of such schedulers. In particular, the schedulers already implement a feedback mechanism in which the number of dispatched tasks (e.g., mappers) per user is dynamically adjusted to accommodate a job's arrival and termination. However, their control for fair-share is limited to only balancing the number of running tasks (or allocated machines) without accounting for a job's actual data throughput that spans shared disks and switch. Fair-share in terms of allocated machines may not be proportional to fair-share in terms of a job's end-to-end performance. While our byte-level throughput control can address policy regarding a job's end-to-end performance, admittedly it could be more complex and expensive to operate during a job's entire lifecycle. Therefore, we can envision a hybrid approach by which a proactive scheduler throttle up/down a job's dispatched tasks and our reactive, byte-level throughput controller begins to operate when the system discovers that a particular job is consuming too much shared bandwidth and affects other jobs negative way. Also, our time-shared approach would be used to preempt a task instead of killing it, as used in the current proactive schedulers. We will explore this direction in our future work.

In conclusion, we present the *containment and control(C&C)* as an alternative approach and a valuable addition to conventional cluster scheduling. The distributed controllers for DryadLINQ cluster shows the *C&C* is a promising approach for achieving computation's predictable end-to-end performance, as well as high cluster throughput. The design and implementation based on standard model-based control architecture is presented as a systematic approach for building distributed controllers.

10. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and the paper shepherd, Renato Figueiredo, for their valuable feedbacks. We also thank Zach Hill whose feedback helped us to improve our experiments. This material is based upon work supported by the National Science Foundation under Grant No: CSR-0916905.

References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Sixth Symposium on Operating System Design and Implementation (OSDI'04), San Francisco, CA, December, 2004.

[2] Apache Hadoop: http://hadoop.apache.org/

[3] Y. Yu, *et. al.* DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. Symposium on Operating System Design and Implementation (OSDI), December, 2008.

[4] Amazon Elastic MapReduce: http://aws.amazon.com/elasticmapreduce/

[5] M. Isard, *et. al.* Quincy: Fair Scheduling for Distributed Computing Clusters. Proceedings of 22nd ACM Symposium on Operating Systems Principles (SOSP), October 2009.

[6] Sang-Min Park and Marty Humphrey. Feedback-Controlled Resource Sharing for Predictable eScience. IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC08), Nov 15-21, 2008, Austin, Texas.

[7] Sang-Min Park and Marty Humphrey. Self-Tuning Virtual Machines for Predictable eScience. Proceedings of IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'09), May 18-21, 2009, Shanghai, China.

[8] Applications and organizations using Hadoop: http://wiki.apache.org/hadoop/PoweredBy

[9] M45 Supercomputing Project: http://research.yahoo.com/node/1884

[10] S. Kavulya, *et. al.* An Analysis of Traces from a Production MapReduce Cluster. Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-09-107. December 2009.

[11] The Web Lab: A Joint Project of Cornell University and the Internet Archive: http://www.weblab.infosci.cornell.edu/index.html

[12] M. Isard, *et. al.* Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. Eurosys Conference, March 2007.

[13] Y. Yu, *et. al.* Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. ACM Symposium on Operating Systems Principles (SOSP), October 2009.

[14] R. Pike, *et. al.* Interpreting the Data: Parallel Analysis with Sawzall. Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure 13:4, pp. 227-298.

[15] Hadoop Pig: http://hadoop.apache.org/pig/

[16] Hadoop On Demand: http://hadoop.apache.org/common/docs/r0.17.1/hod.html

[17] The Hadoop Fair scheduler: http://hadoop.apache.org/common/docs/r0.20.1/fair_scheduler.html

[18] M. Zaharia, *et. al.* Job Scheduling for Multi-User MapReduce Clusters, UC Berkeley Technical Report EECS-2009-55, April 2009.

[19] J.L. Hellerstein, *et. al.* Feedback Control of Computing Systems. Wiley-IEEE Press, August 2004.

[20] K. Astrom, B. Wittenmark. Adaptive Control. Addison-Wesley, 1994.

[21] C. Lu, *et. al.* Feedback Control Real-Time Scheduling: Frame-work, Modeling, and Algorithms, Real-Time Systems, Special Issue on Control-theoretical Approaches to Real-Time Computing, 23(1/2): 85-126, July/September 2002.

[22] C. Lu, *et. al.* Feedback Utilization Control in Distributed Real-Time Systems with End-to-End Tasks. IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 6, June 2005.

[23] C. Lu, *et. al.* Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers. IEEE Transactions on Parallel and Distributed Systems, September 2006.

[24] M. Karlsson, *et. al.* Triage: Performance differentiation for storage systems using adaptive control. ACM Transactions on Storage. volume 1, issue 4 , November 2005.

[25] Y. Zhang, *et. al.* Friendly Virtual Machines Leveraging a Feedback-Control Model for Application Adaptation. ACM/Usenix Int. Conf. on Virtual Execution Environments (VEE), 2005.

[26] P. Padala, *et. al.* Adaptive control of virtualized resources in utility computing environments. EuroSys 2007: 289-302

[27] J. Xu, *et. al.* On the Use of Fuzzy Modeling in Virtualized Data Center Management. International Conference on Autonomic Computing (ICAC), 2007.

[28] D. Thain, *et. al.* Distributed computing in practice: the Condor experience. Concurrency - Practice and Experience, February 2005.

[29] Language Integrated Query (LINQ): http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

[30] SkyServer Queries: http://www.sdss.jhu.edu/SQL/SQLQueries.html

[31] J. Kephart, *et. al.* Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Tradeoffs. IEEE International Conference on Autonomic Computing, June 2007.

[32] S. Kumar, *et. al.* vManage: Loosely Coupled Platform and Virtualization Management in Data Centers. IEEE International Conference on Autonomic Computing, June 2009.

[33] Sang-Min Park and Marty Humphrey. Predictable High Performance Computing using Feedback Control and Admission Control. To appear at IEEE Transactions on Parallel and Distributed Systems, 2010.