The Quest for Scalable Support of Data-Intensive Workloads in Distributed Systems

Ioan Raicu,[1] Ian T. Foster,[1,2,3] Yong Zhao[4]
Philip Little,[5] Christopher M. Moretti,[5] Amitabh Chaudhary,[5] Douglas Thain[5]
[1]Department of Computer Science, University of Chicago, Chicago, IL, USA
[2] Computation Institute, University of Chicago, Chicago, IL, USA
[3] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
[4] Microsoft Corporation, Redmond, WA, USA
[5] Department of Computer Science & Engineering, University of Notre Dame, Notre Dame, IN, USA
iraicu@cs.uchicago.edu, foster@mcs.anl.gov, yozha@microsoft.com,
plittle1@nd.edu, cmoretti@nd.edu, achaudha@nd.edu, dthain@nd.edu

ABSTRACT
Data-intensive applications involving the analysis of large datasets often require large amounts of compute and storage resources, for which data locality can be crucial to high throughput and performance. We propose a "data diffusion" approach that acquires compute and storage resources dynamically, replicates data in response to demand, and schedules computations close to data. As demand increases, more resources are acquired, thus allowing faster response to subsequent requests that refer to the same data; when demand drops, resources are released. This approach can provide the benefits of dedicated hardware without the associated high costs, depending on workload and resource characteristics. To explore the feasibility of data diffusion, we offer both a theoretical and an empirical analysis. We define an abstract model for data diffusion, introduce new scheduling policies with heuristics to optimize real-world performance, and develop a competitive online cache eviction policy. We also offer many empirical experiments to explore the benefits of dynamically expanding and contracting resources based on load, to improve system responsiveness while keeping wasted resources small. We show performance improvements of one to two orders of magnitude across three diverse workloads when compared to the performance of parallel file systems with throughputs approaching 80 Gb/s on a modest cluster of 200 processors. We also compare data diffusion with a best model for active storage, contrasting the difference between a pull-model found in data diffusion and a push-model found in active storage.

Categories and Subject Descriptors
Copyright 2009 ACM 978-1-60558- 587-1/09/06...$5.00.D.4.2 [**Operating Systems**]: Storage and Management – *storage hierarchies*

General Terms
Algorithms, Management, Measurement, Performance, Design.

Keywords
Data diffusion, data management, data-aware scheduling, Falkon

HPDC'09, June 11–13, 2009, Munich, Germany.

1. INTRODUCTION
The ability to analyze large quantities of data has become increasingly important in many fields. To achieve rapid turnaround, data may be distributed over hundreds to thousands of computers. Traditional techniques commonly found in scientific computing (i.e., the reliance on parallel file systems with static configurations) do not scale to today's largest systems for data-intensive applications, as the rate of increase in the number of processors outpaces parallel file system performance.

For example, a cluster we used in our experiment (with 316 processors) has a parallel file system rated at 1 GB/s, yielding 3.2 MB/s per processor of bandwidth. The second largest open science supercomputer, the IBM Blue Gene/P at Argonne National Laboratory, has 160K processors and a parallel file system rated at 65 GB/s, yielding a mere 0.4 MB/s per processor. That is an 8X reduction in bandwidth per processor between a cluster from 2002 and one from 2009. This trend will likely continue, with advances in many-core processors expected to increase the number of cores two orders of magnitude over the next decade.

We argue that in such circumstances, data locality is critical to the successful and efficient use of large distributed systems for data-intensive applications [1, 2]. One approach to achieving data locality—adopted by Google [3, 4]—is to build large compute-storage farms dedicated to storing data and responding to user requests for processing. However, such approaches can lead to idle resources if load varies over time and the data of interest.

We propose an alternative *data diffusion* approach [5], in which resources required for data analysis are acquired dynamically from a local resource manager (LRM), in response to demand. Resources may be acquired either "locally" or "remotely"; their location matters only in terms of associated cost tradeoffs. Both data and applications "diffuse" to newly acquired resources for processing. Acquired resources and the data that they hold can be cached for some time, allowing more rapid responses to subsequent requests. Data diffuses over an increasing number of processors as demand increases, and then contracts as load reduces, releasing processors back to the LRM for other uses.

Data diffusion involves a combination of dynamic resource provisioning, data caching, and data-aware scheduling. The approach is reminiscent of cooperative caching [6], cooperative web-caching [7], and peer-to-peer storage systems [8]. Other data-

aware scheduling approaches tend to assume static resources [9, 10], in which a system configuration dedicates nodes with roles (i.e., clients, servers) at startup and no support is provided to increase or decrease the ratio between client and servers based on load. In our approach, however, we need to dynamically acquire not only storage resources but also computing resources. In addition, datasets may be terabytes in size, and data access is for analysis (not retrieval). Further complicating the situation is our limited knowledge of workloads, which may involve many different applications. In principle, data diffusion can provide the benefits of dedicated hardware without the associated high costs.

The performance achieved with data diffusion depends crucially on the characteristics of application workloads and the underlying infrastructure. As a first step toward quantifying these dependences, we conducted experiments [5] with both micro-benchmarks and a large-scale astronomy application and showed that data diffusion improves performance relative to other approaches, as well as provides improved scalability as aggregated I/O bandwidth scaled linearly up to 64 nodes.

This paper is an evolution in both breadth and depth of data diffusion as presented in [5]. In search for a deeper understanding, we have defined a *data diffusion abstract model* (Section 3.1). We also discuss the data-aware scheduler (Section 2.2) and improved scheduling policies with heuristics to optimize real-world performance (Section 2.2). Moreover, as an initial provably sound algorithm we offer 2Mark, an *O(NM)*-competitive caching eviction policy (Section 3.2), for a constrained problem on *N* stores each holding at most *M* pages. This is the best possible such algorithm with matching upper and lower bounds (barring a constant factor).

In broadening the scope of the original work, we have explored the benefits of dynamic resource provisioning (our previous work investigated only static resource provisioning), which allows the set of both compute and storage resources to expand and contract based on load, to improve system responsiveness while keeping wasted resources under control. We explored this space with two workloads, the monotonically increasing workload (Section 4.1) and the sin-wave workload (Section 4.2). We also explored the all-pairs workload [11] (Section 4.3), which allows us to compare data diffusion with a best model for active storage [12]. Experiments are performed on a subset of a 316-processor cluster.

The contributions of this paper lie in the deeper analysis of data diffusion at both the theoretical and practical levels. We present an O(NM)-competitive algorithm for the scheduler as well as a proof of its competitive ratio, define new heuristics to improve scheduling decisions, explore varying arrival rate workloads to stress the dynamic resource provisioning, and compare data diffusion with the best-case model of active storage.

2. DATA DIFFUSION ARCHITECTURE

We implement data diffusion [5] in the Falkon task dispatch framework [13]. This section describes Falkon and data diffusion.

2.1 Falkon and Data Diffusion

To enable the rapid execution of many tasks on distributed resources, Falkon combines (1) multilevel scheduling [14] to separate resource acquisition (via requests to batch schedulers) from task dispatch and (2) a streamlined dispatcher to achieve several orders of magnitude higher throughput (487 tasks/s) and scalability (54K executors, 2M queued tasks) than other resource managers [13]. Recent work has achieved throughputs in excess of 3750 tasks/s and scalability up to 160K processors [15].

Falkon is structured as a set of (dynamically allocated) *executors* that cache and analyze data; a *dynamic resource provisioner* (DRP) that manages the creation and deletion of executors; and a *dispatcher* that dispatches each incoming task to an executor. The provisioner uses tunable allocation and deallocation policies to provision resources adaptively. Individual executors manage their own caches, using local eviction policies, and communicate changes in cache content to the dispatcher. The dispatcher sends tasks to nodes that have cached the most needed data, along with the information on how to locate needed data; executors access needed data from local disk, peer executors, or persistent storage.

To support data-aware scheduling, we implement a centralized index within the dispatcher that records the location of every cached data object; this is similar to the centralized NameNode in Hadoop's HDFS [16]. This index is maintained as a loosely coherent entity with the contents of the executor's caches via periodic update messages generated by the executors. Each executor maintains a local index to record the location of its cached data objects. This hybrid architecture provides a good balance between latency to the data and good scalability. A prior study [5] showed that a centralized index can often perform better than a distributed index at modest scales (up to thousands of processors).

Falkon supports the queuing of incoming tasks, whose length triggers the dynamic resource provisioning to allocate resources via GRAM4 [17] from the available set of resources, which in turn allocates the resources and bootstraps the executors on the remote machines. The scheduler sends tasks to compute nodes, along with the necessary information about where to find related input data. Initially, each executor fetches needed data from remote persistent storage. Subsequent accesses to the same data results in executors fetching data from other peer executors if the data is already cached elsewhere. The current implementation runs a GridFTP server [18] at each executor, which allows other executors to read data from peer caches. If a data item is not found at any of the known locations, it attempts to retrieve the item from persistent storage; if this also fails, the respective task fails.

In our experiments, we assume data follows the normal pattern found in scientific computing, namely, write-once/read-many (the same assumption HDFS makes in the Hadoop system [16]). Thus, we avoid complicated and expensive cache coherence schemes other parallel file systems enforce. We implement four cache eviction policies: *Random*, *FIFO*, *LRU*, and *LFU* [6]. Our empirical experiments all use LRU; we will study the other policies, including additional ones such as LRV [19], in future work.

2.2 Data-Aware Scheduler

Data-aware scheduling is central to data diffusion, since harnessing data locality in application access patterns is critical to performance and scalability. We implement four dispatch policies.

The **first-available** (FA) policy ignores data location information when selecting an executor for a task; it simply chooses the first available executor and provides the executor with no information concerning the location of cached data objects. The executor must fetch all data needed by a task from persistent storage. This policy is used for experiments not using data diffusion.

The **max-compute-util** (MCU) policy leverages data location information, maximizing resource utilization even at the potentially higher cost of data movement. It sends a task to an available executor, preferring ones with the most local data.

The **max-cache-hit** (MCH) policy uses information about data location to dispatch each task to the executor with the largest amount of data needed by that task. If that executor is busy, task dispatch is delayed until the executor becomes available. This strategy reduces data movement operations compared to FA and MCU but may lead to poor processor utilization.

The **good-cache-compute** (GCC) policy is a hybrid MCH/MCU policy. The GCC policy sets a threshold on the minimum processor utilization to decide when to use MCH or MCU. We define processor utilization to be the number of processors with active tasks divided by the total number of processors allocated. MCU used a threshold of 100%, trying to keep all allocated processors in use. We find that relaxing this threshold (e.g., to 90%) works well in practice, since it keeps processor utilization high and it gives the scheduler flexibility to improve cache hit rates significantly when compared to MCU alone.

The scheduler is window-based. It takes the scheduling window W size (i.e., $|W|$ as the number of tasks to consider from the wait queue when making the scheduling decision), and it starts to build a per task scoring cache hit function. If at any time a best task is found (i.e., achieves a 100% hit rate to the local cache), the scheduler removes this task from the wait queue and adds it to the list of tasks to dispatch to this executor. This process is repeated until the maximum number of tasks are retrieved and prepared to be sent to the executor. If the entire scheduling window is exhausted and no best task is found, the m tasks with the highest cache hit local rates are dispatched. In the case of MCU, if no tasks are found that would yield any cache hit rates, then the top m tasks are taken from the wait queue and dispatched to the executor. For MCH, if no tasks are returned, the executor returns to the free pool of executors. For GCC, the aggregate CPU utilization at the time of scheduling decision determines which action to take. Prebinding of tasks to nodes can negatively impact cache-hit performance if multiple tasks are assigned to the same node, and each task requires the entire cache size, effectively thrashing the cache contents at each task invocation. In practice, we find that per task working sets are small (megabytes to gigabytes) while cache sizes are bigger (tens of gigabytes to hundreds of gigabytes), making the worst case not common.

The scheduler's complexity varies with the policy used. For FA, the cost is constant, as it simply takes the first available executor and dispatches the first task in the queue. MCH, MCU, and GCC are more complex, with a complexity of $O(|T_i| + \min(|Q|, W))$, where T_i is the task at position i in the wait queue and Q is the wait queue. This could equate to many operations for a single scheduling decision, depending on the maximum size of the scheduling window and queue length. Since all data structures used to keep track of executors and since files use in-memory hash maps and sorted sets, operations are efficient. In another study [20], we have shown that the data-aware scheduler can perform thousands of scheduling decisions per second, effectively netting scheduling costs on the order of milliseconds per decision.

3. THEORETICAL EVALUATION

We define an abstract model that captures the principal elements of data diffusion in a manner that allows analysis. We first define the model and then analyze the computational time per task, caching performance, workload execution times, arrival rates, and node utilization. We also present an O(NM)-competitive algorithm for the scheduler and give a proof of its competitive ratio.

3.1 Abstract Model

Our abstract model includes computational resources on which tasks execute and storage resources on which data needed by the tasks is stored. Simplistically, we have two regimes: the working data set fits in cache, S≥W, where S is the aggregate allocated storage and W is the working data set size; and the working set does not fit in cache, S<W. We can express the time T required for a computation associated with a single data access as follows, both depending on H_l (data found on local disk), H_c (remote disks), or H_s (centralized persistent storage).

$S \geq W \quad : (R_l+C) \leq T \leq (R_c+C)$
$S < W \quad : (R_c+C) \leq T < (R_s+C)$

Here R_l, R_c, R_s are the average cost of accessing local data (l), cached data (c), or persistent storage (s), and C is the average amount of computing per data access. The relationship between cache hit performance and T can be expressed as follows.

$S \geq W \quad : T = (R_l+C)*HR_l + (R_c+C)*HR_c$
$S < W \quad : T = (R_c+C)*HR_c + (R_s+C)*HR_s$

Here HR_l is the cache hit local disk ratio, HR_c is the remote cache ratio, and HR_s is the cache miss ratio; $HR_{l/c/s} = H_{L/C/S}/(H_L + H_C + H_S)$. We can merge the two cases such that the time to complete task i is $TK_i = C + R_l*HR_l + R_c*HR_c + R_s*HR_s$.

The time needed to complete an entire workload D with K tasks on N processors is

$$T_N(D) = \sum_{i=1}^{K} TK_i$$

where D is a function of K, W, A, C, and L.

We define speedup to be $SP = T_1(D) / T_N(D)$. Efficiency is defined as $EF = SP / N$.

The maximum task arrival rate (A) that can be sustained is

$S \geq W \quad : N*P/(R_l+C) \leq A_{max} \leq N*P/(R_c+C)$
$S < W \quad : N*P/(R_c+C) \leq A_{max} < N*P/(R_s+C)$

where P is the execution speed of a single node. These regimes can be collapsed into a single formula: $A = (N*P/T)*K$.

We can express a formula to evaluate tradeoffs between node utilization (U) and arrival rate; counting data movement time in node utilization, we have $U = A*T/(N*P)$.

Although the presented model is simplistic, it accurately reflects the time to complete various workloads [5] for an astronomy application [21], with an average of 6% model error and a standard deviation of 5%. Because of space constraints, we do not present the details of this validation.

3.2 *O(NM)*-Competitive Caching

Among known algorithms with provable performance for minimizing data access costs, none can be applied to data diffusion, even if restricted to the caching problem. For instance, LRU maximizes the local store performance but is oblivious of the aggregate cached data and persistent storage. Towards developing an algorithm with provable performance, we show that

the difficulty lies not only in there being multiple stores, but also in the possibility of there being multiple copies of the same object in different stores. For the case where there cannot be such multiple copies, we give an O(NM) competitive ratio online algorithm [22]. An online algorithm solves a problem without knowledge of the future, while an offline optimal [22] is a hypothetical algorithm that has knowledge of the future. The competitive ratio is the worst-case ratio of their performance and is a measure of the quality of the online algorithm, independent of a specific request sequence or workload characteristics.

In the constrained version of the problem there are N stores, each capable of holding M objects of uniform size. Requests are made sequentially to the system, each specifying a particular object and a particular store. If the store does not have the object at that time, it must load the object to satisfy the request. If the store is full, it must evict one object to make room for the new object. If the object is present on another store in the system, it can be loaded for a cost of R_c, which we normalize to *1*. If it is not present in another store, it must be loaded from persistent storage for a cost of R_s, which we normalize to $s = R_s / R_c$. Note that if $R_s < R_c$, we can use LRU at each node instead of 2Mark to maintain competitive performance. We assume R_l is negligible.

All stores in the system are allowed to cooperate (or be managed by a single algorithm with complete state information). This approach allows objects to be transferred between stores in ways not directly required to satisfy a request (e.g., to back up an object that would otherwise be evicted). Specifically, two stores may exchange a pair of objects for a cost of *1* without using extra memory space. Further, executors may write to an object in their store. The system is not allowed to keep multiple copies of an object simultaneously on different stores.

We propose an online algorithm 2Mark (using the well-known marking algorithm [22] at two levels) for data diffusion. Let the corresponding optimum offline algorithm be OPT. For a sequence σ, let $\text{2Mark}(\sigma)$ be the cost 2Mark incurs to handle the sequence, and define $\text{OPT}(\sigma)$ similarly. 2Mark may mark and unmark objects in two ways, designated *local-marking* an object and *global-marking* an object. An object may be local-marked with respect to a particular store (a bit corresponding to the object is set only at that store) or global-marked with respect to the entire system. 2Mark interprets the request sequence as being composed of two kinds of phases, *local-phases* and *global-phases*. A local-phase for a given store is a contiguous set of requests received by the store for M distinct objects, starting with the first request the store receives. A global-phase is a contiguous set of requests received by the entire system for NM distinct objects, starting with the first request the system receives. We prove that $\text{2Mark}(\sigma) \le (NM + 2M/s + NM/(s+v)) \cdot \text{OPT}(\sigma)$ for all sequences σ, which establishes that is O(NM)-competitive. From the lower bound on the competitive ratio for simple paging [22], this is the best possible deterministic online algorithm for this problem, barring a constant factor.

2Mark essentially uses an M-competitive marking algorithm to manage the objects on individual stores and the same algorithm on a larger scale to determine which objects to keep in the system as a whole. When a store faults on a request for an object that is on another store, it exchanges the object it evicts for the object requested (see Figure 1). We establish a bound on the competitive ratio by showing that every cost incurred by 2Mark can be correlated to one incurred by OPT. These costs may be *s-faults* (in which an object is loaded from persistent storage for a cost of *s*), or they may be *1-faults* (in which an object is loaded from another cache for a cost of *1*). The number of 1-faults and s-faults incurred by 2Mark can be bounded by the number of 1-faults and s-faults incurred by OPT in sequence σ.

```
Input: Request for object p at store X from sequence σ
1  if p is not on X then
2     if X is not full then /* No eviction required */
3        if p is on some store Y then
4           Transfer p from Y to X
5        else
6           Load p to X from persistent storage
7        end
8     else /* Eviction required to make space in X */
9        if all objects on X are local-marked then
10          local-unmark all /*Begins new local phase */
11       end
12       if p is on some store Y then
13          Select an arbitrary local-unmarked object q on X
14          Exchange q and p on X and Y
            /* X now has p and Y has q */
15          if p was local-marked on Y then
16             local-mark q on Y
17          end
18       else /* p must be loaded from persistent storage */
19          if all objects in system are global-marked then
20             global-unmark and local-unmark all objects
         /*Begins new global phase & local phases at each store */
21          end
22          if all objects on X are global-marked then
23             Select an arbitrary local-unmarked object q on X
24             Select an arbitrary store Y with at least one global-unmarked
               object or empty space
25             Transfer q to Y, replacing an arbitrary global-unmarked
               object or empty space
26          else
27             Evict an arbitrary global-unmarked object q on X
28          end
29          Load p to X from persistent storage
30       end
31    end
32 end
33 global-mark and local-mark p
```

Figure 1: Algorithm 2Mark

Because of our restricted file access patterns (write-once, read-many), we do not worry about having multiple copies of the same object in different caches and keeping these caches synchronized.

Consider the *i*th global phase. During this global phase, let OPT load objects from persistent storage *u* times, and exchange a pair of objects between stores *v* times, incurring a total cost of *su*+*v*. Every object loaded from persistent storage by 2Mark is globallymarked and not evicted from the system until the end of the global phase. Since the system can hold at most *NM* objects, the number of objects loaded by 2Mark in the *i*th global phase is at most *NM*. We claim OPT loads at least one object from persistent storage during this global phase. This is true if this is the first global phase as all the objects loaded by 2Mark have to be loaded by OPT as well. If this is not the first global phase, OPT must satisfy each of the requests for the distinct *NM* objects in the previous global phase by objects from the system and thus must *s*-fault at least once to satisfy requests in this global phase.

Within the *i*th global phase consider the *j*th local phase at some store *X*. The renaming of objects ensures that any object *p* removed from *X* because of a request for *p* at some other store *Y* is never requested again at *X*. Thus, the first time an object is

requested at X in this local phase, it is locally marked and remains in X for all future requests in this local phase. Hence, X can 1-fault for an object only once during this local phase. Since X can hold at most M objects, it incurs at most M 1-faults in the jth local phase. We claim that when $j \neq 1$, OPT incurs at least one 1-fault in this local phase. The reasoning is similar to that for the ith global phase: since OPT satisfies each of the requests for M distinct objects in the previous local phase from cache, it must 1-fault at least once in this local phase. When $j=1$, however, it may be that the previous local phase did not contain requests for M distinct objects. There are, however, at most NM 1-faults by 2Mark in all the local phases in which $j=1$, for the N stores each holding M objects, in the ith global phase.

Since OPT has the benefit of foresight, it may be able to service a pair of 1-faults through a single exchange. In this both the stores in the exchange get objects that are useful to them, instead of just one store benefiting from the exchange. Thus, since OPT has v exchanges in the ith global phase, it may satisfy at most $2v$ 1-faults and 2Mark correspondingly has at most $2vM + NM$ 1-faults. The second term is due to 1-faults in the first local phase for each store in this global phase. Thus the total cost in the ith global phase by 2Mark is at most $sNM + 2vM + NM$, while that of OPT is at least $s+v$, since $u \geq 1$ in every global phase.

4. EMPIRICAL EVALUATION

We measured the performance of the data-aware scheduler on various workloads, with both static (SRP) and dynamic (DRP) resource provisioning, and ran experiments on the Argonne/University of Chicago TeraGrid [23] (up to 100 nodes, 200 processors). The Falkon service ran on an 8-core Xeon 2.33 GHz, 2 GB RAM, Java 1.5, 100 Mb/s network, and 2 ms latency to the executors. Each node had a local disk with at least 50 GB free. The persistent storage was GPFS [24] with <1 ms latency to executors and had enough storage capacity to store the entire working set per workload.

The three subsections that follow cover three diverse workloads: monotonically increasing (MI), sine-wave (SI), and all-pairs (AP). We use workloads MI and SI to explore the dynamic resource provisioning support in data diffusion and the various scheduling policies (e.g., FA, GCC, MCH, MCU) and cache sizes (e.g., 1 GB, 1.5 GB, 2 GB, 4 GB, and 50 GB); the smaller cache sizes are artificially made smaller to explore the relationship between aggregate cache size and workload working set. We use the AP workload to compare data diffusion with active storage [11].

4.1 Monotonically Increasing Workload

The MI workload has a high I/O to compute ratio (10MB:10ms). The dataset is 100 GB (10K x 10 MB files). Each task reads one file chosen at random (uniform distribution) from the dataset, and computes for 10 ms. The arrival rate is initially 1 task/s and is increased by a factor of 1.3 every 60 seconds to a maximum of 1000 tasks/s. The increasing function is $A_i = \min(\lceil ceiling(A_{i-1} * 1.3), 1000 \rceil, 0 \leq i < 24$, which varies arrival rate A from 1 to 1000 in 24 distinct intervals, makes up 250K tasks and spans 1415 seconds. This workload aims to explore a varying arrival rate under a systematic increase in task arrival rate, to evaluate the scheduler's ability to optimize data locality.

We investigated the performance of the FA, MCH, MCU, and GCC policies, while analyzing cache size effects by varying node cache size (1 GB to 4 GB). We define several metrics:

Demand (Gb/s): throughput needed to satisfy arrival rate
Throughput (Gb/s): measured aggregate transfer rates
Wait Queue Length: number of tasks ready to run
Cache Hit Global: file access from a peer executor cache
Cache Hit Local: file access from local cache
Cache Miss: file accesses from the parallel file system
Speedup (SP): SP = $T_N(FA) / T_N(GCC|MCH|MCU)$
CPU Time (CPU_T): amount of processor time used
Performance Index (PI): PI=SP/CPU_T, normalized [0…1]
Average Response Time (AR_i): time to complete task i, including queue time, execution time, and communication costs

The baseline experiment (FA policy) ran each task directly from GPFS, using dynamic resource provisioning. Aggregate throughput matches demand for arrival rates up to 59 tasks/s but remains flat at an average of 4.4 Gb/s beyond that. At the transition point when the arrival rate increased beyond 59, the wait queue length also started growing to an eventual maximum of 198K tasks. The workload execution time was 5011 seconds, yielding 28% efficiency (ideal being 1415 seconds).

We ran the same workload with data diffusion with varying cache sizes per node (1 GB to 4 GB) using the GCC policy, optimizing cache hits while keeping processor utilization high (90%). The dataset was diffused from GPFS to local disk caches with every cache miss (the red area in the graphs); global cache hits are in yellow and local cache hits in green. The working set was 100 GB, and with a per node cache size of 1 GB, 1.5 GB, 2 GB, and 4 GB caches, we get aggregate cache sizes of 64 GB, 96 GB, 128 GB, and 256 GB. The 1 GB and 1.5 GB caches cannot fit the working set in cache, while the 2 GB and 4 GB cache can.

We first analyze the 1 GB cache size experiment (see Figure 2). Throughput keeps up with demand better than the FA policy, up to 101 tasks/s arrival rates (up from 59), at which point the throughput stabilizes at an average of 5.2 Gb/s. Within 800 seconds, working set caching reaches a steady state with a throughput of 6.9 Gb/s. The overall cache hit rate was 31%, resulting in a 57% higher throughput than GPFS. The workload execution time is reduced to 3762 seconds, down from 5011 seconds for the FA policy, with 38% efficiency.

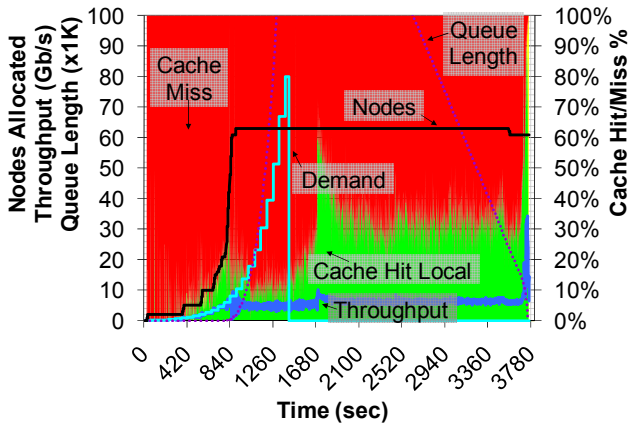

Figure 2: MI workload, 250K tasks, 10 MB:10 ms ratio, up to 64 nodes using DRP, GCC policy, 1 GB caches/node

The same experiment with 1.5 GB caches improved efficiency to 89%, as a result of improved cache hit rates of 78%. Both the 1 GB and 1.5 GB cache sizes achieve reasonable cache hit rates,

despite the fact that the cache sizes are too small to fit the working set in cache; the reason is that the data-aware scheduler looks deep (i.e., window size set to 2500) in the wait queue to find tasks that will improve the cache hit performance.

Figure 3 shows results with 2 GB local caches (128 GB aggregate). Aggregate throughput is close to demand (up to the peak of 80 Gb/s) for the entire experiment. We attribute this good performance to the ability to cache the entire working set and then schedule tasks to the nodes that have required data to achieve cache hit rates approaching 98%. Note that the queue length never grew beyond 7K tasks, significantly less than for the other experiments (91K to 198K tasks long). With an execution time of 1436 seconds, efficiency was 99%.

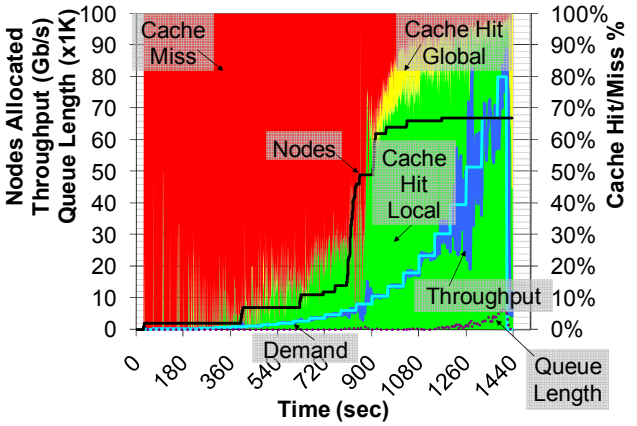

Figure 3: MI workload, 250K tasks, 10 MB:10 ms ratio, up to 64 nodes using DRP, GCC policy, 2 GB caches/node

Figure 4 summarizes the aggregate I/O throughput measured in each of the seven experiments we conducted. The solid bars are the average throughput achieved from start to finish, partitioned among local cache, remote cache, and GPFS; the thin black line is the "peak" (99th percentile sample) throughput achieved. The peak is interesting because of the progressive increase in job submission rate and may be viewed as a measure of how far a particular method can go in keeping up with application demands.

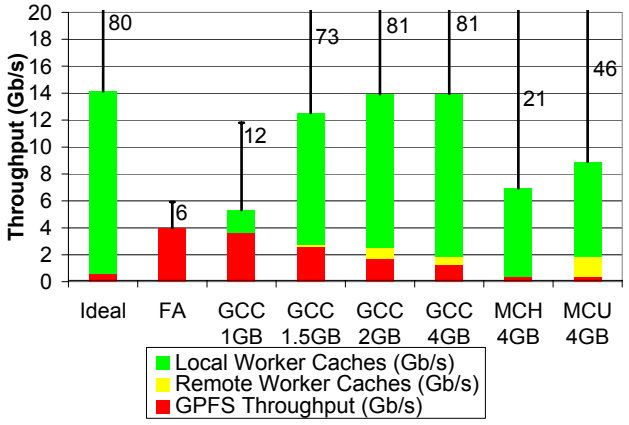

Figure 4: MI average and peak (99th percentile) throughput

We see that the FA policy had the lowest average throughput of 4 Gb/s, compared to between 5.3 Gb/s and 13.9 Gb/s for data diffusion (GCC, MCH, and MCU with various cache sizes), and 14.1 Gb/s for the ideal case. In addition to having higher average throughputs, data diffusion also achieved significantly higher throughputs toward the end of the experiment (the black bar) when the arrival rates are highest, as high as 81 Gb/s (compared to 6 Gb/s for the FA policy. Note also that GPFS file system load (the red portion of the bars) is significantly lower with data diffusion than for the GPFS-only experiments (FA), ranging from 0.4 Gb/s to 3.6 Gb/s depending on the size of the caches. Remote caches showed a lower network load, with most policies being under 1 Gb/s with the exception of the MCU policy at 1.5 G/s.

The performance index attempts to capture the speedup per processor time achieved (see Figure 5). Notice that while GCC with 2 GB and 4 GB caches each achieve the highest speedup of 3.5X, the 4 GB case achieves a higher performance index of 1 as opposed to 0.7 for the 2 GB case. The reason is that fewer processor resources were used throughout the 4 GB experiment (17 CPU-hours instead of 24 CPU-hours). This reduction in resource usage was due to the larger caches, which in turn allowed the system to perform better with fewer resources for longer durations; hence, the wait queue did not grow as fast, thereby resulting in less aggressive resource allocation. Notice the performance index of the FA policy, which uses GPFS solely; although the speedup gains with data diffusion compared to the FA policy are modest (1.3X to 3.5X), the performance index of data diffusion is significantly more (2X to 34X).

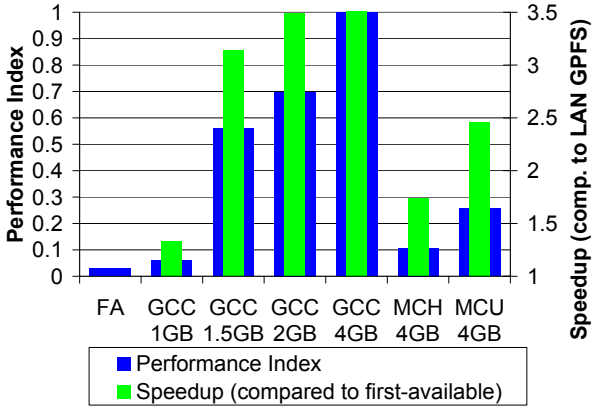

Figure 5: MI workload PI and speedup comparison

The response time is critical for interactive applications. We see a significant difference (500X) between the best response time (GCC, 3.1 seconds per task) and the worst response time (FA, 1569 seconds). A principal factor influencing the average response time is the time tasks spend in the wait queue. In the worst (FA) case, the queue length grew to 198K tasks as the allocated resources could not keep up with the arrival rate. In contrast, the best (GCC) case queued up only 7K tasks at its peak.

The experiments presented in this subsection show that large enough aggregate caches to hold the entire working set is important to achieve near-optimal performance, although smaller caches can still be effective as long as the scheduler inspects tasks deep in the wait queue. Furthermore, the ability to keep up with higher demands and keep wait queues short allows data diffusion to be a good candidate for data-intensive interactive applications.

4.2 Sine-Wave Workload

The previous subsection explored a workload with monotonically increasing arrival rates. To explore how well data diffusion deals

with decreasing arrival rates, we define a sine-wave (SW) workload that follows the function (where time is elapsed minutes from the beginning of the experiment).

$$A = \left\lfloor (\sin(sqrt (time + 0.11) * 2.859678) + 1) * (time + 0.11) * 5.705 \right\rfloor$$

This workload aims to explore the data-aware scheduler's ability to optimize data locality in the face of frequent joins and leaves of resources caused by variability in demand. This function is essentially a sine-wave pattern, in which the arrival rate increases in increasingly stronger waves, increasing up to 1000 tasks/s arrival rates. The working set is 1 TB large (100K files of 10 MB each), and the I/O to compute ratio is 10 MB:10 ms. The workload is composed of 2M tasks, where each task accesses a random file (uniform distribution) and takes 6505 seconds to complete in the ideal case. The testbed includes up to 100 nodes, with local disks of at least 50 GB free; we therefore set the cache size to 50 GB per node for these experiments (instead of the 1 GB to 4 GB in Section 4.1), since our aim here was to investigate the dynamic resource provisioning effectiveness on a variable arrival rate workload.

Our first experiment consisted of running the SW workload with all computations running directly from the parallel file system and using 100 nodes with static resource provisioning. We see the measured throughput keep up with the demand up to the point when the demand exceeds the parallel file system peak performance of 8 Gb/s; beyond this point, the wait queue grew to 1.4M tasks, and the workload needed 20491 seconds to complete (instead of the ideal case of 6505 seconds), yielding an efficiency of 32%. Note that although we are using the same cluster as in the only MI workload (Section 4.1), GPFS's peak throughput is higher (8 Gb/s vs. 4 Gb/s) because of a major upgrade to both hardware and software in the cluster between running these experiments.

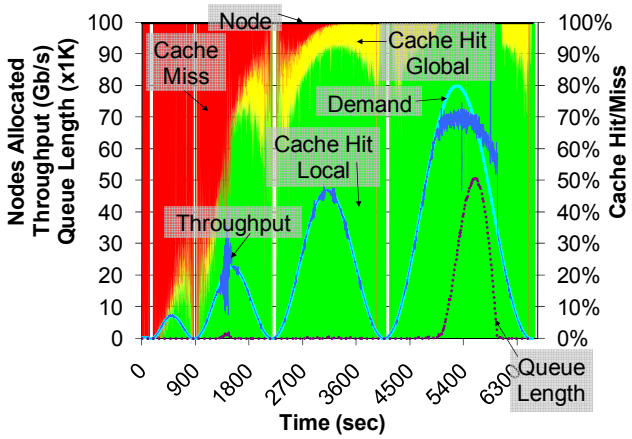

Figure 6: SW workload, 2M tasks, 10MB:10ms ratio, 100 nodes, GCC policy, 50GB caches/node

Enabling data diffusion with the GCC policy, setting the cache size to 50GB, the scheduling window size to 2500, and the processor utilization threshold to 90%, we get a run that took 6505 seconds to complete (see Figure 6), yielding an efficiency of 100%. We see the cache misses (red) decrease from 100% to 0% over the course of the experiment, while local cache hits (green) frequently make up 90%+ of the cache hits. Note that the data diffusion mechanism was able to keep up with the arrival rates throughout, with the exception of the peak of the last wave, when it was able to achieve only 72 Gb/s (instead of the ideal 80 Gb/s), at which point the wait queue grew to its longest length of 50K tasks. The global cache hits (yellow) is stable at about 10% throughout, which reflects the fact that the GCC policy is oscillating between optimizing cache hit performance and processor utilization around the configured 90% threshold.

Enabling dynamic resource provisioning, Figure 7 shows the workload still manages to complete in 6697 seconds, yielding 97% efficiency. To minimize wasted processor time, we set each worker to release its resource after 30 seconds of idleness. Note that upon releasing a resource, its cache is reset; thus, after every wave, cache performance is again poor until caches are rebuilt. The measured throughput does not fit the demand line as well as the static resource provisioning did, but it increases steadily in each wave and achieves the same peak throughput of 72 Gb/s after enough of the working set is cached.

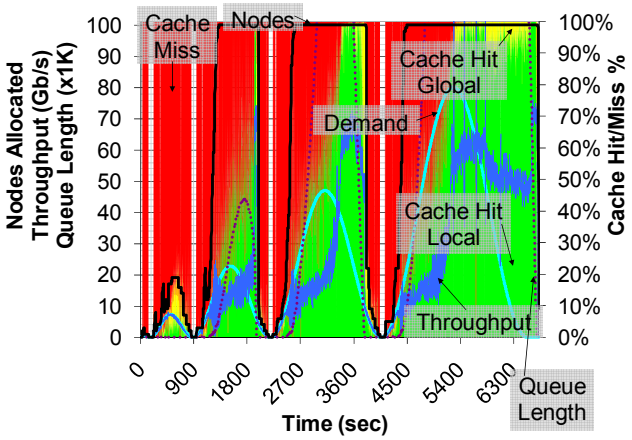

Figure 7: SW workload, 2M tasks, 10MB:10ms ratio, up to 100 nodes with DRP, GCC policy, 50GB caches/node

In summary, we see data diffusion make a significant impact. Using the dynamic provisioning where the number of processors is varied based on load does not hinder data diffusion's performance significantly (achieves 97% efficiency) and yields less processor time consumed (253 CPU hours as opposed to 361 CPU hours for SRP and GCC and 1138 CPU hours for FA).

4.3 All-Pairs Workload

In previous work, several of us addressed large-scale data-intensive problems with the Chirp [12] distributed file system. Chirp has several advantages, such as delivering an implementation that behaves like a file system and maintains most of the semantics of a shared file system. Moreover, Chirp offers efficient distribution of datasets via a spanning tree, making Chirp ideal in scenarios with a slow and high-latency data source. However, Chirp does not address data-aware scheduling. Therefore, when used by All-Pairs [11], it typically distributes an entire application working data set to each compute node local disk prior to the application running. We call the All-Pairs use of Chirp *active storage*. This requirement hinders active storage from scaling as well as data diffusion, making large working sets that do not fit on each compute node local disk difficult to handle, and producing potentially unnecessary transfers of data. Data diffusion transfers only the minimum data needed per job.

To understand the comparison between data diffusion and the best model of active storage, we first define a common benchmark for data-intensive computing, namely, All-Pairs (AP). Variations of the AP problem occur in many applications, for example when we want to understand the behavior of a new function F on sets A and B or to learn the covariance of sets A and B on a standard inner product F [11]. The AP problem is easy to express in terms of two nested for loops over some parameter space. This regular structure also makes it easy to optimize its data access operations. Nevertheless, AP is a challenging benchmark for data diffusion, because of its on-demand, pull-mode data access strategy.

In previous work [11], we conducted experiments with biometrics and data mining workloads using Chirp. The most data-intensive workload was where each function executed for 1 second to compare two 12 MB items, for an I/O to compute ratio of 24 MB:1000 ms. At the largest scale (50 nodes and 500x500 problem size), we measured an efficiency of 60% for the active storage implementation, and 3% for the demand paging (to be compared to the GPFS performance we cite). These experiments were conducted in a campuswide heterogeneous cluster with nodes at risk for suspension, network connectivity of 100 Mb/s between nodes, and a shared file system rated at 100 Mb/s from which the data set needed to be transferred to the compute nodes.

Because of differences in our testing environments, a direct comparison is difficult, but we compute the best case for active storage as defined in [11] and compare measured data diffusion performance against this best case. Our environment has 100 nodes (200 processors) that are dedicated for the duration of the allocation, with 1 Gb/s network connectivity between nodes, and a parallel file system (GPFS) rated at 8 Gb/s. For the 500x500 workload, data diffusion achieves a throughput that is 80% of the best case of all data accesses occurring to local disk (see Figure 8). We computed the best case for active storage to be 96%. In practice, however, based on the efficiency of the 50-node case from previous work [11] that achieved 60% efficiency, we believe the 100-node case will not perform significantly better than the 80% efficiency of data diffusion. Running the same workload through Falkon directly against a parallel file system achieves only 26% of the throughput of the purely local solution.

To push data diffusion harder, we made the workload 10X more data-intensive by reducing the compute time from 1 second to 0.1 seconds, yielding an I/O-to-compute ratio of 24 MB:100 ms (see Figure 9). For this workload, the throughput steadily increased to about 55 Gb/s as more local cache hits occurred. We found extremely few cache misses, thus indicating the high data locality of the AP workload. Data diffusion achieved 75% efficiency. Active storage and data diffusion transferred similar amounts of data over the network (1536 GB for active storage and 1528 GB for data diffusion with 0.1 s compute time and 1698 GB with 1 s compute time workload) and to and from the shared file system (12 GB for active storage and 62 GB and 34 GB for data diffusion for 0.1 s and 1 s compute time workloads, respectively). With such similar bandwidth usage, similar efficiencies were expected.

Our comparison between data diffusion and active storage essentially involves a comparison of pushing versus pulling data. The active storage implementation pushes all the needed data for a workload to all nodes via a spanning tree. With data diffusion, nodes pull only the files immediately needed for a task, creating an incremental spanning forest (analogous to a spanning tree, but one that supports cycles) at runtime that has links both to the parent node and to any other arbitrary node or persistent storage. We measured data diffusion to perform comparably to active storage on our 200-processor cluster, but differences exist between the two approaches. Data diffusion depends more on having a well-balanced persistent storage for the amount of computing power, but it can scale to larger number of nodes because of the more selective nature of data distribution [20]. Furthermore, data diffusion needs to fit only the per task working set in local caches, rather than an entire workload working set as is the case for active storage.

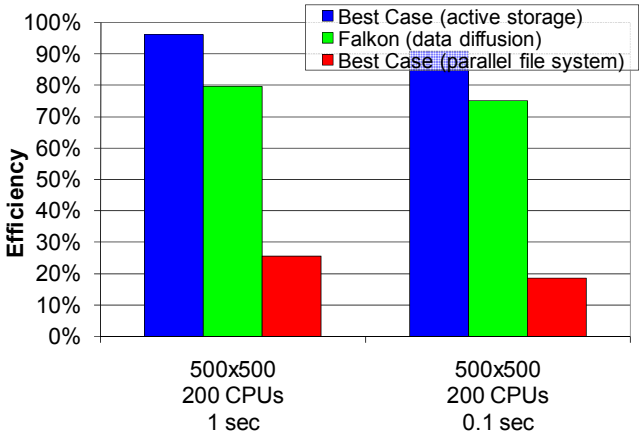

Figure 8: AP workload efficiency

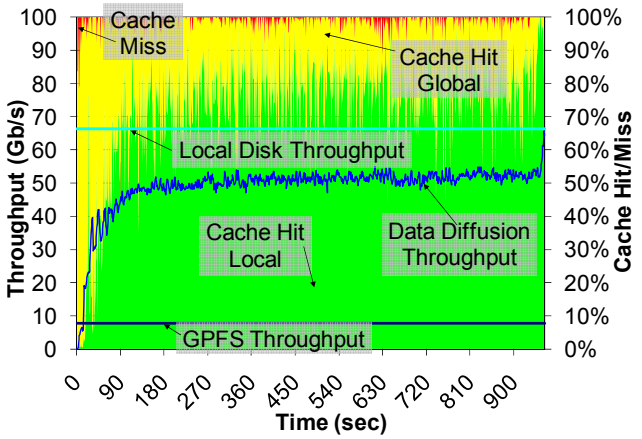

Figure 9: AP workload, 500x500=250K tasks, 24 MB:100 ms, 100 nodes, GCC policy, 50 GB caches/node

5. RELATED WORK

Over the past decade, considerable work has been done on data management of distributed systems. We believe our discussion in the preceding sections has provided readers the necessary background to understand the sometimes-subtle details we now describe between data diffusion and other systems.

The Stork [25] scheduler seeks to improve performance and reliability when batch scheduling by explicitly scheduling data placement operations. While Stork can be used with other system components to co-schedule CPU and storage resources, no attempt is made to retain nodes and harness data locality in data access patterns between tasks.

The GFarm team implemented a data-aware scheduler in Gfarm using an LSF scheduler plug-in [9, 26]. Their performance results

are for a small system in comparison to our own results and offer relatively slow performance (6 nodes, 300 jobs, 900 MB input files, 0.1–0.2 jobs/s, and 90 MB/s to 180 MB/s data rates); furthermore, the papers present no evidence that their system scales. In contrast, we have tested our proposed data diffusion with 200 processors, 2M jobs, input data ranging from 1byte to 1 GB per job, working sets of up to 1 TB, workflows exceeding 1000 jobs/sec, and data rates exceeding 9 GB/s.

BigTable [27], Google File System (GFS) [3], MapReduce [4], and Hadoop [16] couple data and computing resources to accelerate data-intensive applications. However, these systems all assume a dedicated set of resources, in which a system configuration dictates nodes with roles (i.e., clients, servers) at startup, and there is no support to increase or decrease the ratio between client and servers based on load; note that upon failures, nodes can be dynamically removed from these systems, but this is done for system maintenance, not to optimize performance or costs. This is a critical difference, as these systems are typically installed by a system administrator and operate on dedicated clusters. Falkon and data diffusion work on batch-scheduled distributed resources (such as those found in clusters and Grids used by the scientific community), which are shared by many users. Although MapReduce/Hadoop systems can also be shared by many users, nodes are shared by all users and data can be stored or retrieved from any node in the cluster at any time. In batch scheduled systems, sharing is done through abstraction called jobs which are bound to some number of dedicated nodes at provisioning time. Users can access only those nodes that are provisioned to them; and when nodes are released, there are no assumptions on the preservation of node local state (i.e., local disk and RAM). The tight coupling of execution engine (MapReduce, Hadoop) and file system (GFS, HDFS) means that scientific applications must be modified to use these underlying non-POSIX-compliant file systems to read and write files. Data diffusion coupled with the Swift parallel programming system [28, 29] can enable the use of data diffusion without any modifications to scientific applications, which typically rely on POSIX-compliant file systems. Furthermore, through the use of Swift's check-pointing at a per task level, failed application runs (synonymous with a job for MapReduce/Hadoop) can be restarted from the point at which they previously failed; although tasks can be retried in MapReduce/Hadoop, a failed task can render the entire MapReduce job failed. We also note that data replication in data diffusion occurs implicitly as a result of demand (e.g., popularity of a data item), while in Hadoop an explicit parameter must be tuned per application and typically incurs unnecessary performance hindering overheads. We believe Swift and data diffusion are more generic for scientific applications and better suited for batch-scheduled clusters and Grids.

Two systems often compared with MapReduce and GFS are Sphere [30] and Sector [31]. Sphere is designed to be used with the Sector Storage Cloud and implements certain specialized, but commonly occurring, distributed computing operations. For example, the MapReduce programming model is a subset of the Sphere programming model, as the Map and Reduce functions could be any arbitrary functions in Sphere. Sector is the underlying storage cloud that provides persistent storage for the data required by Sphere and manages the data for Sphere operations. Sphere is analogous to Swift, and Sector is analogous to data diffusion, although they each differ considerably. For example, Swift is a general-purpose parallel programming system, and the programming model of both MapReduce and Sphere is a subset of the Swift programming model. Data diffusion and Sector are similar in function, both providing the underlying data management for Falkon and Sphere, respectively. However, Falkon and data diffusion have been tested mostly in LANs, while Sector targets WANs. Data diffusion has been architected to run in nondedicated environments, where the resource pool (both storage and compute) varies based on load, provisioning resources on-demand and releasing them when they are idle. Sector runs on dedicated resources and focuses on decreasing the resource pool as a result of failures. Another important difference between Swift running over Falkon and data diffusion, as opposed to Sphere running over Sector, is the ability to run "black box" applications on distributed resources without any need to modify legacy applications; access to files are done over POSIX read and write operations. Sphere and Sector take the approach of MapReduce, in which applications are modified to support the read and write operations of applications.

With respect to provable performance results, several online competitive algorithms handle problems in scheduling (see [32] for a survey) and others problems in caching (see [22] for a survey), but none, to the best of our knowledge, combine the two. The closest problem in caching is the two-weight paging problem [33]; it allows for different page costs but assumes a single cache.

6. CONCLUSION AND FUTURE WORK

Dynamic analysis of large data sets is becoming increasingly important in many domains. When building systems to perform such analyses, we face difficult tradeoffs. Do we dedicate computing and storage resources to analysis tasks, enabling rapid data access but wasting idle resources? Or do we move data to compute resources, incurring potentially expensive transfer costs?

This paper studied data diffusion, which seeks to combine elements of both dedicated and on-demand approaches. We envision data diffusion as a process in which data is stochastically moving around in the system, through which different applications can reach their dynamic equilibrium. One can think of a thermodynamic analogy of an optimizing strategy, in terms of energy required to move data around ("potential wells") and a "temperature" representing random external perturbations ("job submissions") and system failures. This paper proposes exactly such a stochastic optimizer.

The key idea in data diffusion is that we respond to demands for data analysis by allocating data or compute systems and by migrating code or data to those systems. We retain these dynamically allocated resources for some time, so that workloads with data locality can obtain the performance benefits of dedicated resources. To explore this approach, we have extended the Falkon framework to cache data at executors and incorporated a data-aware scheduler in the dispatcher.

Our work is significant because of the support that data-intensive applications require, with the growing gap between parallel file system performance and the increase in the number of processors per system. The contributions of this paper lie in the deeper analysis of data diffusion at both the theoretical and the practical levels. We present an O(NM)-competitive algorithm for the scheduler, as well as a proof of its competitive ratio; define new heuristics to improve scheduling decisions; explore the effectiveness of data diffusion under varying arrival rate workloads; and compare data diffusion with active storage.

We plan to explore more sophisticated algorithms that address what happens when an executor is released. Should we discard cached data? Should it be moved to another executor, or should it be moved to persistent storage? Do cache eviction policies affect cache hit ratio performance? Answers to these and other related questions will presumably depend on workload and system characteristics. We also have preliminary work that addresses data-intensive applications on petascale systems with our file-based collective I/O primitives for loosely coupled applications [34]. We will explore methods of supporting data-intensive science, aiming for the largest scales (e.g., hundreds of thousands of processors) available to the open science community.

ACKNOWLEDGMENTS

This work was supported in part by the NASA Ames Research Center GSRP Grant Number NNA06CB89H and by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. This research was also supported in part by the National Science Foundation through TeraGrid resources provided by UC/ANL. We also thank Alex Szalay for the contributions and ideas on the inception of data diffusion.

REFERENCES

[1] A. Szalay, J. Bunn, J. Gray, I. Foster, I. Raicu. "The Importance of Data Locality in Distributed Computing Applications," NSF Workflow Workshop 2006

[2] J. Gray. "Distributed Computing Economics," Technical Report MSR-TR-2003-24, Microsoft Research, 2003

[3] S. Ghemawat, H. Gobioff, S.T. Leung. "The Google File System," ACM SOSP 2003, pp. 29-43

[4] J. Dean, S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters," OSDI 2004

[5] I. Raicu, Y. Zhao, I. Foster, A. Szalay. "Accelerating Large-scale Data Exploration through Data Diffusion," ACM Workshop on Data-Aware Distributed Comp. 2008

[6] S. Podlipnig, et al. "A Survey of Web Cache Replacement Strategies," ACM Computing Surveys, 2003

[7] R. Lancellotti, et al. "A Scalable Architecture for Cooperative Web Caching," Web Engineering Workshop 2002

[8] R. Hasan, et al. "A Survey of Peer-to-Peer Storage Techniques for Distributed File Systems," ITCC 2005

[9] W. Xiaohui, et al. "Implementing Data Aware Scheduling in Gfarm Using LSF Scheduler Plugin Mechanism," GCA05, 2005

[10] P. Fuhrmann. "dCache, the Commodity Cache," MSST 2004

[11] C. Moretti, et al. "All-Pairs: An Abstraction for Data-Intensive Cloud Computing," IPDPS 2008

[12] D. Thain, et al. "Chirp: A Practical Global Filesystem for Cluster and Grid Computing," JGC, Springer, 2008

[13] I. Raicu, et a. "Falkon: A Fast and Light-weight tasK executiON Framework," IEEE/ACM SC 2007

[14] G. Banga, et al. "Resource Containers: A New Facility for Resource Management in Server Systems," OSDI 1999

[15] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. "Toward Loosely Coupled Programming on Petascale Systems," IEEE SC 2008

[16] A. Bialecki, et al. "Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware," http://lucene.apache.org/hadoop/, 2005

[17] M. Feller, et al. "GT4 GRAM: A Functionality and Performance Study," TeraGrid Conference 2007

[18] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster. "The Globus Striped GridFTP Framework and Server," ACM/IEEE SC, 2005

[19] P. Cao, et al. "Cost-Aware WWW Proxy Caching Algorithms," USENIX Symposium on Internet Technologies and Systems, 1997

[20] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. Moretti, A. Chaudhary, D. Thain. "Towards Data Intensive Many-Task Computing," under review at Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management, 2009

[21] I. Raicu, I. Foster, A. Szalay, G. Turcu. "AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis," TeraGrid Conf. 2006

[22] E. Torng. "A Unified Analysis of Paging and Caching," Algorithmica 20, 175–200, 1998

[23] ANL/UC TeraGrid Site Details, http://www.uc.teragrid.org/tg-docs/tg-tech-sum.html, 2007

[24] F. Schmuck, R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," FAST 2002

[25] T. Kosar. "A New Paradigm in Data Intensive Computing: Stork and the Data-Aware Schedulers," IEEE CLADE 2006

[26] X. Wei, et al. "Integrating Local Job Scheduler – LSF with Gfarm," ISPA05, vol. 3758/2005, 2005

[27] F. Chang, et al. "Bigtable: A Distributed Storage System for Structured Data," USENIX OSDI 2006

[28] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, M. Wilde. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," IEEE Workshop on Scientific Workflows 2007

[29] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, M. Wilde. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments," Grid Computing Research Progress, Nova Pub. 2008

[30] R. Grossman, Y. Gu. "Data Mining Using High Performance Clouds: Experimental Studies Using Sector and Sphere," ACM KDD 2008

[31] Y. Gu, et al. "Distributing the Sloan Digital Sky Survey Using UDT and Sector," e-Science 2006

[32] K. Pruhs, et al. "Online Sscheduling," Handbook of Scheduling: Algorithms, Models, and Performance Analysis, 2004

[33] S. Irani. "Randomized Weighted Caching with Two Page Weights," Algorithmica, 32:4, 624-640, 2002

[34] X. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, M. Wilde. "Design and Evaluation of a Collective I/O Model for Loosely-coupled PetascaleProgramming," IEEE MTAGS 2008