Dynamic Load Balancing for Distributed Search

L. Huston[1] A. Nizhner[2] P. Pillai[1] R. Sukthankar[1,2] P. Steenkiste[2] J. Zhang[3]
[1] Intel Research Pittsburgh; [2] Carnegie Mellon University; [3] University of Michigan

Abstract

This paper examines how computation can be mapped across the nodes of a distributed search system to effectively utilize available resources. We specifically address computationally intensive search of complex data, such as content-based retrieval of digital images or sounds, where sophisticated algorithms must be evaluated on the objects of interest. Since these problems require significant computation, we distribute the search over a collection of compute nodes, such as active storage devices, intermediate processors and host computers. A key challenge with mapping the desired computation to the available resources is that the most efficient distribution depends on several factors: relative power and number of compute nodes; network bandwidth between the compute nodes; the cost of evaluating query predicates; and the selectivity of the given query. This wide range of variables renders manual partitioning of the computation infeasible, particularly since some of the parameters (*e.g.*, available network bandwidth) can change during the course of a search. This paper proposes several techniques for dynamic partitioning of computation, and demonstrates that they can significantly improve efficiency for distributed search applications.

1. Introduction

Rapid advances in storage technology and digital media acquisition has led to an explosive growth in the volume of large, rich datasets. Acquiring the data is only one part of the problem and many obstacles impede the efficient use of this data. This paper focuses on one of these problems: search — the ability to efficiently extract a set of data items that meet some user-specified constraints.

For text and numerical data, the standard approach has been to build indexes and employ these for efficient retrieval. Unfortunately, this approach requires that the data items be reduced to a small number of numeric values. For many rich data items (*e.g.*, brain MRIs), it is impossible to distill the interesting features into a small set of numbers. Instead, a search may require performing expensive computation over the entire dataset to find the items that match the desired characteristics.

Performing computation on this large body of data introduces several challenges. First, such searches are computationally expensive, typically demanding a distributed approach. Second, unlike many distributed computing tasks, the data objects can be very large. Naively transferring objects over the network to a centralized compute server or idle machines can be prohibitively expensive in terms of the system I/O cost, and may erase the benefits of distributing the computation. A practical approach to the problem requires balancing the benefits of additional computational resources against the cost of shipping data.

We have developed a system, Diamond [9], that addresses this problem by distributing the search over a collection of machines. A key element of the Diamond architecture is the use of active storage [1, 10, 13], storage devices with local processing, to eliminate irrelevant data from the dataset before shipping it over the network (see Figure 1). Application-specific filtering code, termed a *searchlet*, is distributed across the active storage devices and the user's host machine (and optionally, over several intermediate compute nodes).

We summarize several salient aspects of our system design. First, while discarding all of the irrelevant data at the storage device would be ideal in terms of network resource utilization, performing the necessary computation at the storage nodes is typically infeasible given their limited processing power. Diamond must resolve the tension between fully utilizing the available computational resources while ensuring that irrelevant data is not unnecessarily propagated through the system. Second, the optimal distribution of computation across the system may change over time: as hardware upgrades affect the balance of processing power between the nodes, as changes in the network affect the cost of transferring large quantities of data, and as concurrent searches add load to different parts of the system. Furthermore, the correct distribution depends on the selectivity and computational requirements of the queries, and on the distribution of data on the storage devices. Our approach is to dynamically distribute the load across the components of the system. Performing the load balancing at the system level allows the developers of search applications to focus on building effective domain-specific filters.

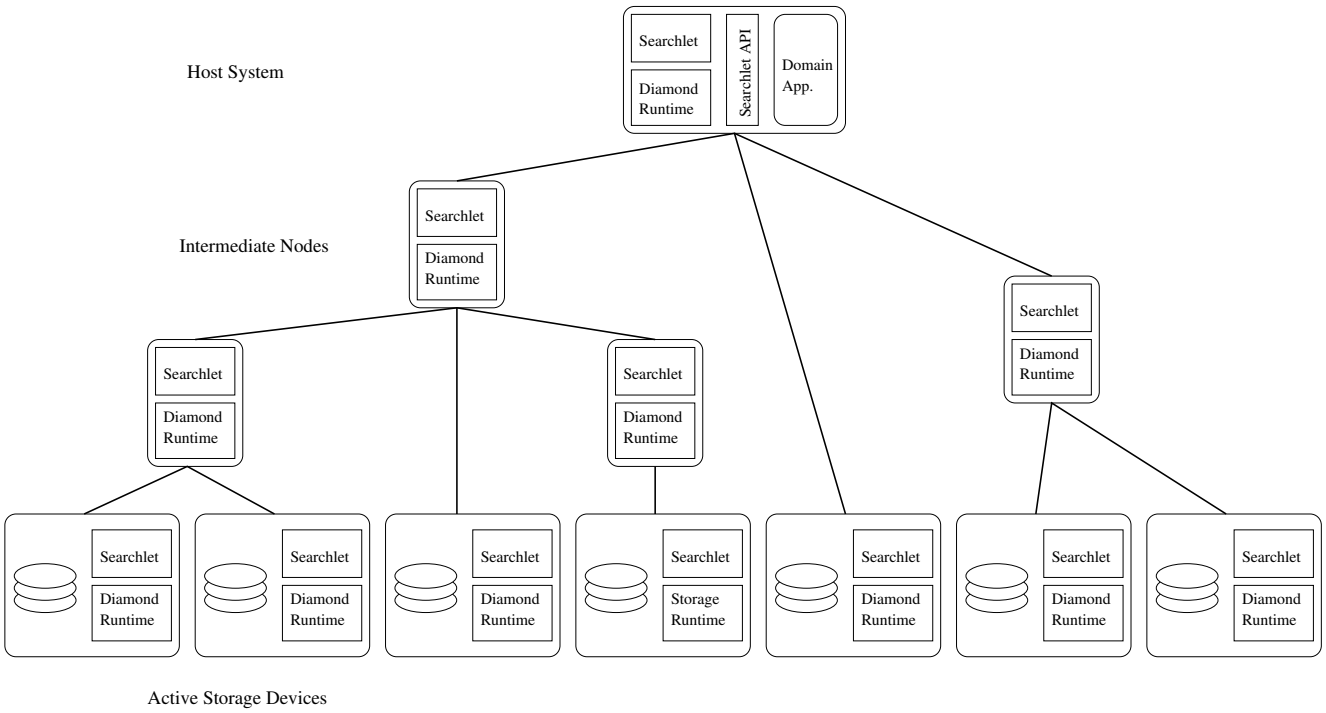

Figure 1: Distributed search

This paper proposes several techniques that exploit search characteristics to perform dynamic load balancing. These algorithms accommodate for variations in processing capacity, network bandwidth, storage capacity and heterogeneity of active storage components. Our experiments demonstrate that these techniques significantly improve search efficiency across a variety of system and query configurations.

This paper is organized as follows. Section 2 presents an overview of our system and discusses some related work. Section 3 describes our algorithms for dynamic load balancing. Section 4 presents experimental results. Section 5 concludes the paper.

2. Distributed Search

This section summarizes our system architecture for distributed search (see Figure 1). The user formulates a query using a domain-specific application on the host system. The application translates this query into a searchlet that the storage devices and compute nodes use to determine whether a particular data element matches the query. The searchlet is thus a proxy of the application that encapsulates the domain-specific knowledge necessary to perform the search task.

A searchlet consists of a set of machine-executable tasks (*filters*) and associated configuration state, such as filter parameters and dependencies between filters. For example, a searchlet to retrieve portraits of people in dark business suits might contain two filters: a color histogram filter that finds dark regions and a detector that locates human faces. Each filter's return value indicates whether the given object should be discarded, in which case the searchlet evaluation is terminated for the current object. Objects that pass through all of the filters are sent to the domain application for further processing.

A filter can pass state to another filter by adding attributes to a temporary copy of the data element being searched; these attributes can be read by any subsequent filter. For example, in a content-based image retrieval application, an earlier filter could preprocess the image to generate intermediate representations, such as color histograms, for use by later filters. These attributes can be very large, often larger than the original image and can impact the cost of sending the partially-processed object over the network. The Diamond runtime optimizes the order of filter execution based on measured rejection rates and execution times while ensuring that any partial ordering constraints imposed by the application are satisfied [9].

Diamond exploits several characteristics of the search task to minimize the problems associated with distributed computation. First, it assumes that data can be indepen-

dently processed in smaller units, termed objects (*e.g.*, individual images in a image retrieval system or segments of a movie). Second, search tasks typically permit stored objects to be examined in any order. This order-independence offers several benefits: easy parallelization within and across storage devices, significant flexibility in scheduling data reads, and simplified migration of computation. Third, most search tasks do not require maintaining state between objects. Finally, search tasks only require read access to data, allowing the system to avoid locking complexities and to simplify security issues.

Databases address a similar issue with query planning. Before a query is performed, the query planner maps a query to the available resources to get the best performance. This approach has several limitations for our problem, primarily in that it assumes perfect knowledge of resource availability at the planning stage. Our queries may run for long periods and as a result, the system must handle the case where available resources (*e.g.*, network bandwidth) may change during the execution of the search. Another limitation of the planning approach, is that the planner needs good *a priori* estimates for the cost of evaluating each filter as well as its selectivity. Unlike a typical database, our queries are difficult to predict in advance because the system can run arbitrary application code and the selectivity of a predicate can vary widely based on the parameter settings as well as the data being searched.

Several systems explore the problem of dynamically adapting the distribution of computation over a set of processors. Coign [7] profiles running programs and allocates components to machines so as to minimize communications cost. River [3] handles adaptive dataflow control generically in the presence of failures and heterogeneous hardware resources. Eddies [4] adaptively reshapes dataflow graphs to maximize performance by monitoring the rates at which data is produced and consumed at nodes. River and Eddies differ from our approach in that they work with small data elements and are primarily concerned with efficient usage of computational resources not with minimizing the cost of moving data across machine boundaries. Abacus [2] automatically moves computation between hosts or storage devices in a cluster based on performance and system load. Abacus tries to tackle a more general distributed computing problem while we focus specifically on search and exploit the characteristics of the search domain.

3. Dynamic Load Balancing

As described above, the search task (encapsulated by a searchlet) consists of a series of filters that is evaluated on each stored object. To achieve the best performance we want to efficiently distribute the processing of the filters across the set of available processors. One can view the path from a data source (storage node) to the data sink (the host) as a pipeline of processing stages separated by queues. Our goal is to dynamically adjust the processing performed at each stage so as to maximize system utilization and data throughput. In this section we introduce our dynamic load balancing algorithm in three steps. We first focus on a single processor and show how it can minimize the amount of data it forwards. Next we present the load balancing algorithm for a complete storage-to-host processor pipeline, and finally we present algorithms to handle heterogeneous configurations.

3.1. Mapping Search Filters to Processors

We first consider the problem of mapping filters to available compute nodes. In our model, an object passes through a chain of processors on its way to the host. Each processor tests the object against one or more filters, discarding it on failure. An important goal is to find an assignment of filters to processors that minimizes inter-processor network traffic. This is worthwhile for two reasons: (1) in many WAN configurations, the network may be the bottleneck; (2) even for searches that are CPU-bound, reducing network traffic can significantly reduce CPU load [5, 6, 12].

3.1.1 Bypass-based evaluation

We abstract the problem as follows. At each node, we define β to be the fraction of the arriving work that should be performed locally. This work consists of a set of filters $\{F_i\}$, that still need to be applied to a given object. We assume the execution order of the filters is fixed and given by the sequence $F_0, F_1, \ldots, F_{n-1}$. Clearly, there are multiple ways to achieve a desired β. Two simple examples are dividing the data into disjoint subsets, each of which is handled by particular compute nodes (data partitioning), or assigning specific filters to nodes (filter partitioning).

To more precisely specify the filters that a node will execute, we define the *bypass fraction* b_i of filter F_i as the average fraction of objects that are locally evaluated by F_i, as shown in Figure 2. Using this terminology, data partitioning is expressed via bypass assignments of the form $b_0 = \beta,\ b_1 = \cdots = b_{n-1} = 1$, while filter partitioning will correspond to $b_0 = 1,\ \cdots, b_j = 1, b_{j+1} = 0, \cdots, b_{n-1} = 0$. The run-time simply interprets b_i as a probability — when any object reaches filter F_i, it has a random chance, $1 - b_i$, of being immediately sent to the next compute node in the chain without further local evaluation.

We now present our partitioning algorithm for selecting the appropriate filters to run locally, while minimizing the data transferred. Details are in our technical report [11].

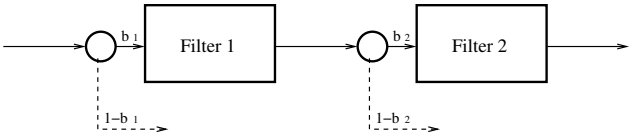

Figure 2: Bypass-based evaluation

3.1.2 Assigning bypass fractions

Let c_i be the average running time of filter F_i, and let the *conditional pass rate* p_i represent the fraction of objects evaluated by F_i that pass filter F_i. The average CPU time needed to evaluate all filters on each object is given by

$$C = \sum_{i=0}^{n-1} p_0 \cdots p_{i-1} c_i.$$

The CPU time spent locally on the average object under the bypass assignment $b = (b_0, \ldots, b_{n-1})$ is given by

$$C(b) = \sum_{i=0}^{n-1} p_0 \cdots p_{i-1} b_0 \cdots b_i c_i.$$

Thus a bypass assignment satisfying a computation partitioning of β is one such that $C(b) = \beta C$.

We define the *efficiency* of a filter as the decrease in the average bytes transmitted per unit computation; efficiency is affected by the filter's selectivity as well as the average amount of meta-data added by the filter. Efficiency can be used to compute bypass assignments. Intuitively, one can see that for a desired β, we want to set the bypass assignments to include the filters with the highest efficiency.

Let us first focus on a simple case where filters can be executed in any order. If the filters are ordered in decreasing efficiency, the most effective bypass assignment, for a given β, will execute as many of the early filters as possible [11]. To achieve this assignment, we define a partitioning scheme (termed Aggressive) that for each object always executes the first k filters locally, the next filter (F_{k+1}) locally a fraction of the time, and never executes the remaining filters locally. This corresponds to a bypass assignment of $b_i = 1$ for $i \leq k$, $b_i = 0$ for $i > k+1$, and b_{k+1} such that $C(b) = \beta C$. Figure 3 shows the relationship between β and the outgoing network traffic. The slopes of the line segments correspond to filter efficiencies. Note that increasing β reduces the average number of bytes per object sent downstream.

3.1.3 Filter clustering

In practice it is not possible to execute the filters in any order, since some filters may depend on other filters. Moreover, some filters can add meta-data to the object possibly

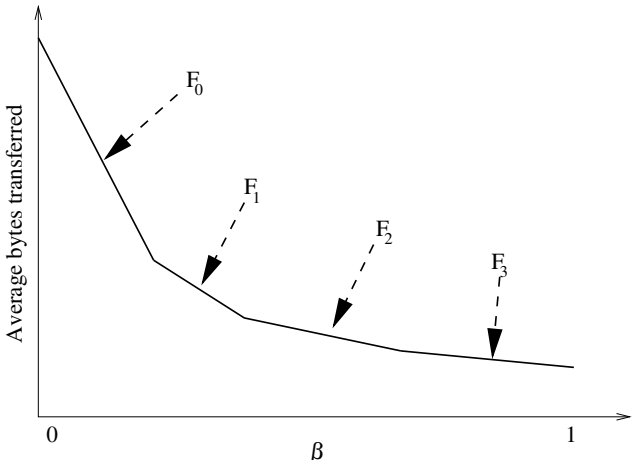

Figure 3: β vs. average bytes transferred for the ideal Aggressive scheme

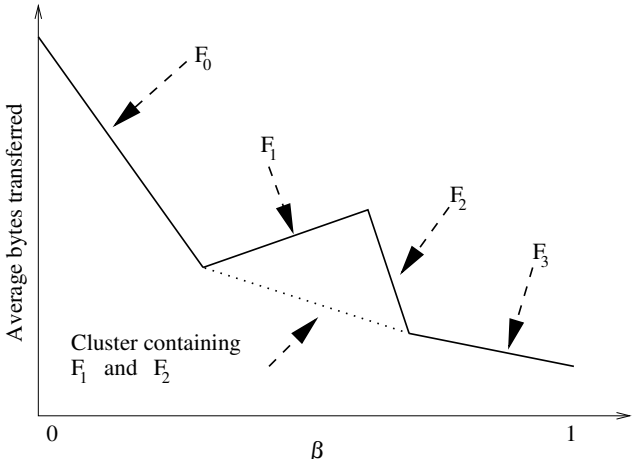

Figure 4: β vs. average bytes transferred for a typical Aggressive scheme

resulting in a negative efficiency. As a result we cannot always execute the most efficient filters first. The solid line in Figure 4 shows an example of what the Aggressive algorithm can achieve in this case; filter dependencies prevent us from moving F_2 before F_1. Note that in this case, increasing β does not necessarily decrease the network traffic.

Even if we cannot arbitrarily order filters, we can still apply the Aggressive scheme to minimize the outgoing network data by grouping adjacent filters to create a cluster. The Aggressive algorithm treats clusters as atomic units (if the first filter in the cluster is locally evaluated, the remaining filters in that cluster must also be executed locally). This clustering employs a simple agglomerative scheme where any adjacent filters that are not in order of decreasing efficiency are combined. The effect of combining two such filters is illustrated by the dashed line in Figure 4.

We implement and compare the following partitioning schemes:

- **Simple:** the data partitioning scheme introduced in Section 3.1.1. This scheme is equivalent to collapsing all of the filters into a single atomic unit.
- **Greedy:** an application of Aggressive without clustering.
- **Hybrid:** an application of Aggressive with clustering as described above.

3.2. Load balancing across a compute pipeline

To perform load balancing, each of the compute nodes should determine the fraction of computation that should be performed locally. For CPU-bound scenarios, an effective partitioning should keep all of the compute nodes continuously busy for the duration of the search, such that the nodes finish their tasks simultaneously. Given complete and accurate knowledge of processing rates at all nodes, objects stored on each active storage device, and the network bandwidths, one could determine the allocation of processing that minimizes search time. Unfortunately, such an analytical approach is likely to fail in practice due to variability in system behavior and the unpredictable impact of concurrent searches. Therefore, we advocate schemes that adapt each compute node's behavior. The challenge is for the compute nodes to find an effective partitioning based solely upon local observations.

We describe two methods for load balancing. The first, termed "queue adaptation", makes simple per-object decisions on when to queue an object. The idea is that each compute node should check its output queue after evaluating a filter (or cluster of filters). If the number of items in the queue drops below a specified threshold, the current (partially processed) object is enqueued. Thus, without explicitly calculating a β value, the given compute node automatically matches its processing to the observed drain rate on the output queue by delegating work to downstream nodes as necessary. A potential drawback is that the network traffic generated by this method may be higher than that using an explicit bypass assignment. Nevertheless, as shown in Section 4.2, this simple adaptive scheme is quite competitive for real-world tasks.

The beta estimation method monitors the enqueue rate on the input queue (e_1) and the drain rate of the output queue (d_2). Here we focus on the case where there is a single input and output queue; the more general case is discussed in the next section. Based on the observed rates (see Figure 5) this method adjusts β to match local processing to the drain rate of the output queue. However, when the input queue is the bottleneck (*e.g.*, a disk-bound case), we conserve network bandwidth by increasing local computation.

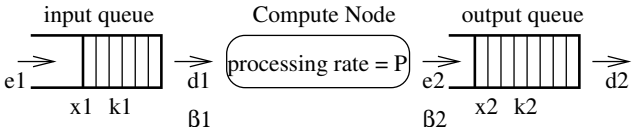

Figure 5: Beta Estimation

To achieve this goal, we independently compute an input and an output value for β based on the state of the upstream and downstream queues. The maximum β is used. As a secondary objective, we try to avoid overfull and underfull queues by driving the number of items in the queue, x_1 and x_2, toward specified target values, k_1 and k_2. The algorithm is summarized by:

$$\begin{aligned} d_1 &= e_1 + \frac{x_1 - k_1}{\Delta t} \\ e_2 &= d_2 - \frac{x_2 - k_2}{\Delta t} \\ \beta_1 &= \frac{P}{d_1} \\ \beta_2 &= \frac{1}{\frac{e_2}{P} + 1} \\ \beta &= \max(\beta_1, \beta_2). \end{aligned}$$

3.3. Heterogeneous Devices

As stated earlier, our goal is to finish processing all of the objects at the same time. To achieve this goal, downstream compute nodes should provide a disproportionate share of their compute resource to assist slower upstream compute nodes. Although the above discussion assumes a single aggregate input queue, each node actually maintains a separate queue for each upstream path and computes β values using the aggregate statistics.

To determine how we service the multiple input queues, we employ a credit-based mechanism. Each input queue, i, corresponds to a different upstream branch, and is alloted a certain number of credits, d_i, representing its share of processing. Objects are dequeued from the input stream with the highest credit count, which is decremented proportional to the processing time consumed by the object. The system is work conserving: if the particular queue selected is empty, the system simply continues with the next best one. Each credit count is replenished by the corresponding d_i value when no non-empty queues have a positive credit balance.

To aid in assigning credits, each upstream node periodically provides statistics about the number of remaining objects that it must process and its current processing rate. The given node uses this information to estimate each upstream

node's time-to-completion and allocates credits according to one of the two schemes described below.

The first scheme (termed "proportional allocation") finds the earliest completion time among all upstream nodes. Each upstream node is assigned credits proportional to the difference between its expected completion time and this minimum. The second scheme (termed "greedy allocation") finds the latest completion time among all upstream nodes and assigns it C credits. All of the other upstream nodes are assigned a single credit.

4. Experimental Evaluation

Diamond is implemented on Linux as user-level code with multiple threads. The host runtime is implemented as a library that links against the domain application. The storage runtime is implemented as a multi-threaded daemon running on the storage devices and the intermediate nodes. Background threads are used to read data objects to reduce I/O stalls. Network communication is implemented using sockets over TCP.

The storage devices and intermediate nodes are implemented using rack-mounted computers (1.2 GHz Intel® Pentium® III processors, 512 MB RAM and 73 GB SCSI disks), connected via a 1 Gbps Ethernet switch. The host system contains a 3.06 GHz Intel® Pentium® Xeon™processor, 2 GB RAM, and a 120 GB IDE disk. The host is connected via Ethernet to the storage platforms. We vary the link speed between 1 Gbps and 10 Mbps depending on the experiment. Some experiments require us to emulate slower active storage devices; this is done by running a real-time task that consumes a fixed percentage of the CPU.

Our search task was content-based image retrieval on a large, non-indexed collection of digital photos. Table 1 summarizes two queries generated using using a content-based image retrieval application (SnapFind [8]) that were used to evaluate our algorithms. Each of the storage devices was allocated 5,000 images (1.6 GB). As the number of storage devices increased, so did the total number of images involved in a search. For each experiment we performed 3 runs of each test, and reported the mean value. We chose the size of our data set to be large enough to avoid startup transients but manageable enough to enable the variety of experiments described below.

4.1. Impact of Partitioning Schemes

The first experiments examine how the three algorithms described in Section 3.1 affect the amount of data transmitted on the network. This experiment employs a single storage device directly connected to the host. We run several experiments with different fixed values for β and measure the network usage. Figures 6 and 7 show the results for the "Flower Pot" and the "Lawn" queries.

Query	Description
Flower Pot	Looks for images that contain flower pots by searching for multiple color distributions; terra-cotta for the pots, and red and green patches for the plants.
Lawn	Find images of lawns through color and texture filters.

Table 1: Test queries

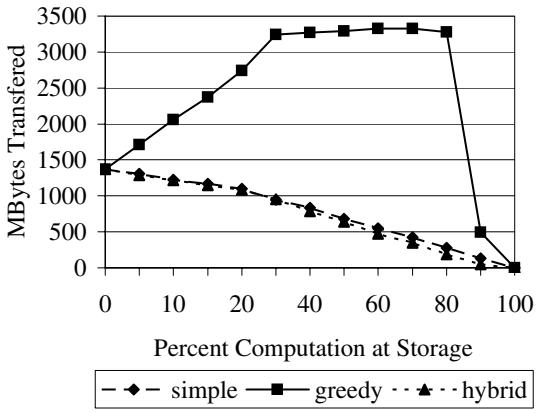

Figure 6: Bytes transferred for flower query.

The results show that the Hybrid algorithm minimizes the bytes transferred for both queries. In the "Flower Pot" query, the Hybrid and Simple algorithms are equivalent and minimize the bytes transferred for all settings of β. In the "Lawn" query, Hybrid is better than the other algorithms over large ranges of β. This validates our belief that the Hybrid scheme minimizes network usage for a target β; thus, we employ Hybrid for all of the remaining experiments.

4.2. Impact of CPU Load Balancing

The next set of experiments evaluate the adaptive CPU load balancing algorithms on system configurations with varying numbers of storage devices, network bandwidth and processor speeds, as shown in Table 2. For each configuration, we first generate a baseline measurement (for each query) by exhaustively searching for a fixed β that minimizes search time. This baseline is the best CPU load balancing using a static β for the given query and system configuration.

Figures 8 and 9 show the performance of the queue-based and the β-estimation load balancing algorithms relative to their respective baselines. As a comparison we also give two additional results: (1) where all of the computation

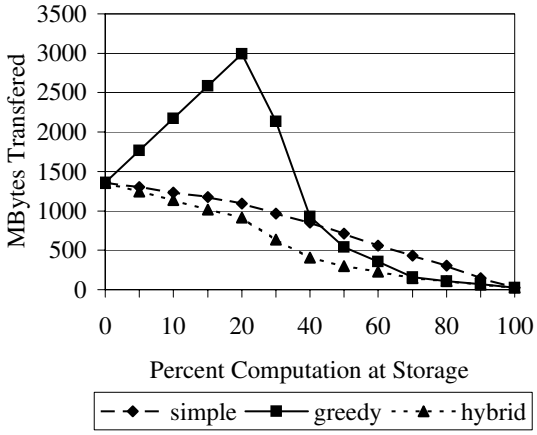

Figure 7: Bytes transferred for lawn query.

Name	Network Speed	Number of Devices	Processor Speed
4-slow	10 Mbps	4	25%
4-fast	1 Gbps	4	100%
8-slow	10 Mbps	8	25%
8-fast	1 Gbps	8	25%

Table 2: System configurations

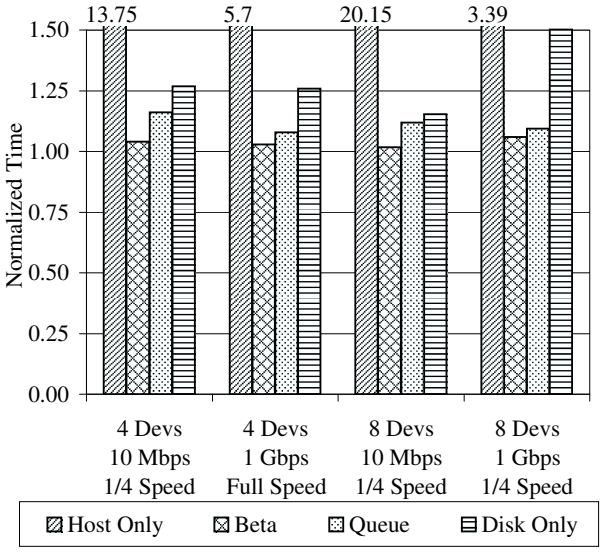

Figure 8: Dynamic CPU Partitioning for lawn query.

occurs at the host; (2) where all of the computation occurs at the active storage device.

These results show that both of the adaptive techniques perform well, particularly since they are fully automated and require no *a priori* knowledge. The β estimation is within 6% of the baseline while the queue-based approach has slightly longer runtimes (primarily due to transferring more data on the network). Our experiments confirm that adaptive approaches work well without making assumptions about the hardware configuration or tuning for specific queries.

4.3. Heterogeneous Configurations

This experiment evaluates the two algorithms for coping with heterogeneity in compute node processing power (described in Section 3.3). We evaluate the algorithms using two different configurations. In both cases we use four storage devices connected to the host computer via 1 Gbps Ethernet. The first configuration uses the"Lawn" query and the four storage devices are configured with the following relative speeds: CPU1 runs at 75%; CPU2 and CPU3 run at 50%; and CPU4 runs at 25%. The second configuration uses the"Flower Pot" query and the four storage devices are configured with the following relative speeds: CPU1 runs at 75%; CPU2 runs at 50%; and CPU3 and CPU4 run at 25%.

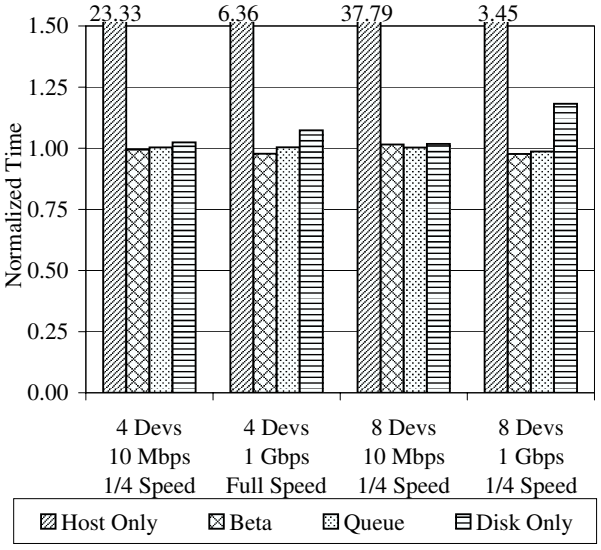

Figure 9: Dynamic CPU Partitioning for flower query.

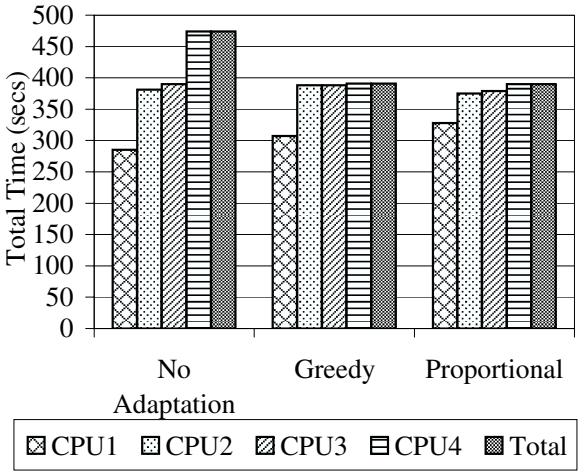

Figure 10: Heterogeneous adaptation for configuration 1

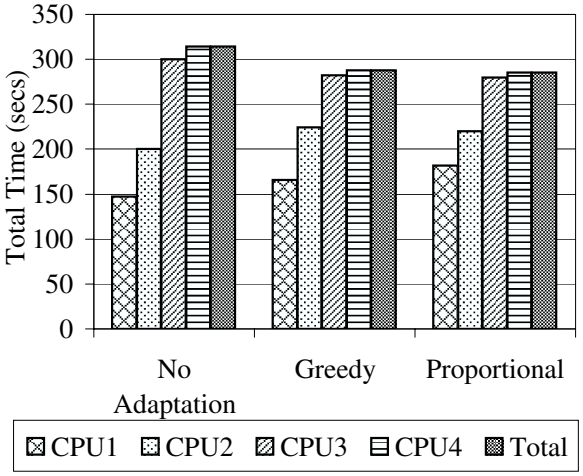

Figure 11: Heterogeneous adaptation for configuration 2

We evaluate the following three cases: (1) uniform, non-adaptive, credit allocation among the different devices; (2) greedy credit algorithm; (3) proportional credit allocation. Figure 10 and Figure 11 show the time for each node to finish processing its data, as well as the total search time (determined by the progress of the slowest node).

For the uniform credit allocation, we observe that there is a large difference between the completion times of the fastest and slowest nodes, leading to a longer total search time. Using either of the adaptive credit allocation schemes reduces the total search time by disproportionately allocating the host's compute resources to aid the slowest nodes. We observe no significant difference between Greedy and Proportional on these experiments. For the second configuration, the decrease in total run time is not as significant as the first configuration because the single host can not offer enough additional compute resources to offset the impact of the two slow disks. We expect to see larger improvements as the performance disparity between the host and the storage devices is increased.

4.4. Multi-Level Hierarchies

These experiments explore the performance of the adaptive schemes as intermediate compute nodes are added to the system. Our initial setup consists of 8 storage nodes, each running at 25% speed, connected to the host via a 10Mbps network, executing the "Flower Pot" query. This corresponds to running queries from a remote host over a WAN. We then examine the improvement achieved by adding two 100% speed intermediate compute nodes, each connected via 1Gbps network to four storage devices and 10Mbps to the host. These correspond to additional local processing available at the data center.

Configuration	Time (s)
No Intermediate	1067
Intermediate with Dynamic Adaptation	691
Intermediate with Static Partitioning	768

Table 3: Multi-level configurations

We conduct two experiments: (1) dynamic adaptation using β estimation load balancing for the intermediate nodes; (2) applying a static β calculated from relative processor speeds. Table 3 shows that, while both schemes reduce the total search time, the dynamic adaptation using β estimation outperforms the static scheme. We hypothesize that the static scheme performs less well because relative processor speeds do not account for the CPU overhead involved in networking and disk I/O.

4.5. Adapting to Dynamic Conditions

This experiment examines how well the Diamond system reacts to dynamic configurations. First, we execute the "Flower Pot" query on four storage nodes (CPU1–CPU4), without intermediate nodes. After 30 seconds (while the first query is still running), we initiate a concurrent search using the "Lawn" query from a different host computer on a subset of the storage nodes (CPU1, CPU2). In this experiment, Diamond uses β estimation for load balancing and the proportional credit allocation to handle heterogeneous environments.

On CPU1 and CPU2 we observe that β decreases for the first search and increases for the second. Both of these val-

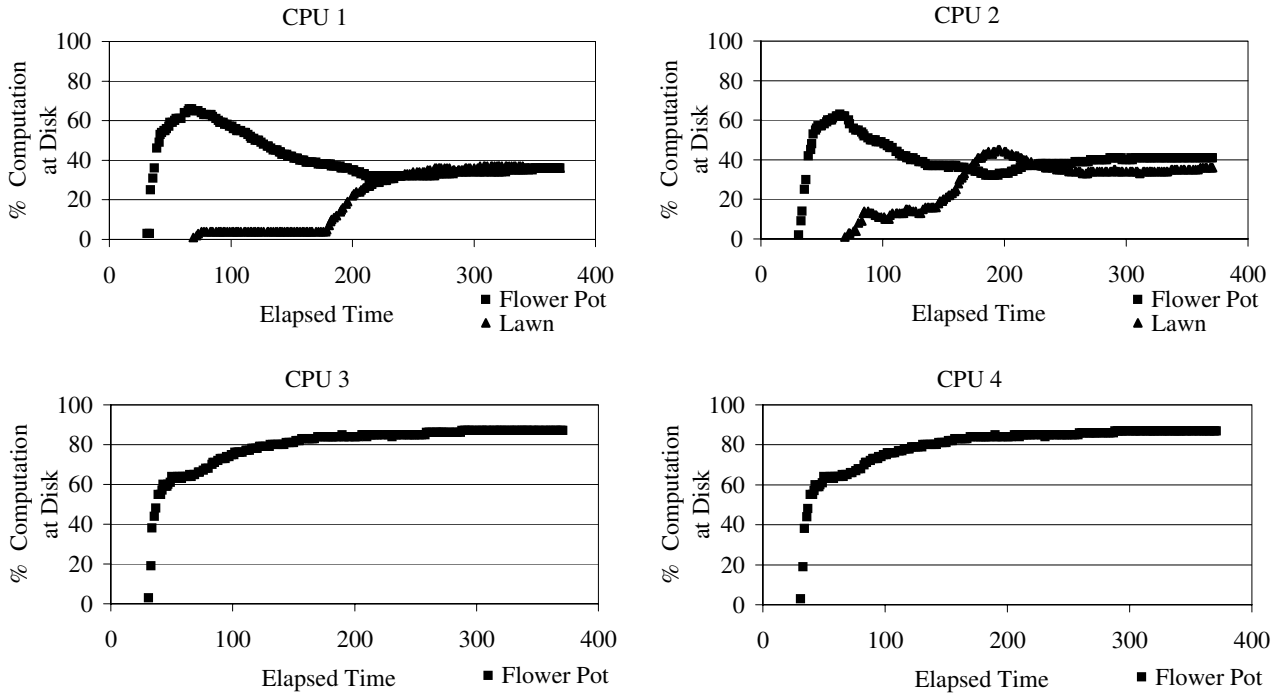

Figure 12: Dynamic adaptation

ues reach a stable point around 37%. This indicates that a significant fraction of the processing moves downstream to the host processor in response to the increased load. We see that the β values for CPU3 and CPU4 increase when the second search starts, in response to the greater allocation of host resources to CPU1 and CPU2 (which have been slowed by concurrent searches). This is the desired behavior as the host provides greater share of resources to the loaded nodes that have now become the bottleneck for the first query.

These results demonstrate that our adaptive algorithms for load balancing are effective at handling run-time changes in resource availability.

The above evaluation results show that, on average, the best performance results are obtained using the Hybrid clustering algorithm for selecting the filter to execute on each node, combined with the β estimation method for load balancing among processing nodes, and the proportional algorithm for distributing processing over multiple input queues.

5. Conclusion

This paper motivates the need for dynamic partitioning of computation among the components of a distributed search system. We present algorithms that efficiently distribute computation by considering the relative capabilities and number of compute nodes, the available interconnect bandwidth, the size and placement of the data on the storage nodes, and the characteristics of the current query. These techniques have been implemented for a content-based image retrieval application in Diamond, a distributed architecture that supports large-scale interactive brute-force search of complex data. Our experiments demonstrate that dynamic adaptation significantly improves efficiency by ensuring that compute nodes within the distributed system are maximally utilized. We believe that the techniques for load balancing described in this paper are broadly applicable to other forms of distributed computation (*e.g.*, parallel visualization). Finally, within our present application domain, we would like to investigate how our adaptation techniques interact with other performance optimizations such as computation caching.

Acknowledgments

The authors would like to thank M. Satyanarayanan for valuable discussions, and R. Wickremesinghe and D. Hoiem for their work on the SnapFind image retrieval application.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of ASPLOS*, 1998.

[2] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of USENIX*, 2000.

[3] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of Input/Output for Parallel and Distributed Systems*, 1999.

[4] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of SIGMOD*, 2000.

[5] J. Chase, A. Gallatin, and K. Yocum. End-System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[6] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6), June 1989.

[7] G. Hunt and M. Scott. The Coign automatic distributed partitioning system. In *Proceedings of OSDI*, 1999.

[8] L. Huston, R. Sukthankar, D. Hoiem, and J. Zhang. SnapFind: brute force interactive image retrieval. In *Proceedings of International Conference on Image Processing and Graphics*, 2004.

[9] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proc. USENIX Conference on File and Storage Technologies*, 2004.

[10] K. Keeton, D. Patterson, and J. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, 27(3), 1998.

[11] A. Nizhner, L. Huston, P. Steenkiste, and R. Sukthankar. Network-aware partitioning of computation in Diamond. Technical Report CMU-CS-04-148, School of Computer Science, Carnegie Mellon University, June 2004.

[12] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, and A. Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. *IEEE Micro*, 24(1):24–31, 2004.

[13] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of VLDB*, August 1998.