Locality-Aware Request Distribution in Cluster-based Network Servers

Vivek S. Pai[‡], Mohit Aron[†], Gaurav Banga[†],
Michael Svendsen[†], Peter Druschel[†], Willy Zwaenepoel[†], Erich Nahum[¶]

[‡] Department of Electrical and Computer Engineering, Rice University
[†] Department of Computer Science, Rice University
[¶] IBM T.J. Watson Research Center

Abstract

We consider cluster-based network servers in which a front-end directs incoming requests to one of a number of back-ends. Specifically, we consider *content-based request distribution*: the front-end uses the content requested, in addition to information about the load on the back-end nodes, to choose which back-end will handle this request. Content-based request distribution can improve locality in the back-ends' main memory caches, increase secondary storage scalability by partitioning the server's database, and provide the ability to employ back-end nodes that are specialized for certain types of requests.

As a specific policy for content-based request distribution, we introduce a simple, practical strategy for *locality-aware* request distribution (LARD). With LARD, the front-end distributes incoming requests in a manner that achieves high locality in the back-ends' main memory caches as well as load balancing. Locality is increased by dynamically subdividing the server's working set over the back-ends. Trace-based simulation results and measurements on a prototype implementation demonstrate substantial performance improvements over state-of-the-art approaches that use only load information to distribute requests. On workloads with working sets that do not fit in a single server node's main memory cache, the achieved throughput exceeds that of the state-of-the-art approach by a factor of two to four.

With content-based distribution, incoming requests must be handed off to a back-end in a manner transparent to the client, *after* the front-end has inspected the content of the request. To this end, we introduce an efficient *TCP handoff protocol* that can hand off an established TCP connection in a client-transparent manner.

Appears in the Proceedings of the ACM Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose, CA, Oct 1998. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, MIP-9521386, by Texas TATP Grant 003604, and by an IBM Partnership Award.

1 Introduction

Network servers based on clusters of commodity workstations or PCs connected by high-speed LANs combine cutting-edge performance and low cost. A cluster-based network server consists of a front-end, responsible for request distribution, and a number of back-end nodes, responsible for request processing. The use of a front-end makes the distributed nature of the server transparent to the clients. In most current cluster servers the front-end distributes requests to back-end nodes without regard to the type of service or the content requested. That is, all back-end nodes are considered equally capable of serving a given request and the only factor guiding the request distribution is the current load of the back-end nodes.

With *content-based request distribution*, the front-end takes into account both the service/content requested and the current load on the back-end nodes when deciding which back-end node should serve a given request. The potential advantages of content-based request distribution are: (1) increased performance due to improved hit rates in the back-end's main memory caches, (2) increased secondary storage scalability due to the ability to partition the server's database over the different back-end nodes, and (3) the ability to employ back-end nodes that are specialized for certain types of requests (e.g., audio and video).

The *locality-aware request distribution* (LARD) strategy presented in this paper is a form of content-based request distribution, focusing on obtaining the first of the advantages cited above, namely improved cache hit rates in the back-ends. Secondary storage scalability and special-purpose back-end nodes are not discussed any further in this paper.

Figure 1 illustrates the principle of LARD in a simple server with two back-ends and three targets[1] (A,B,C) in the incoming request stream. The front-end directs all requests for A to back-end 1, and all requests for B and C to back-end 2. By doing so, there is an increased likelihood that the request finds the requested target in the cache at the back-end. In contrast, with a round-robin distribution of incoming requests, requests of all three

[1]In the following discussion, the term *target* is being used to refer to a specific object requested from a server. For an HTTP server, for instance, a target is specified by a URL and any applicable arguments to the HTTP `GET` command.

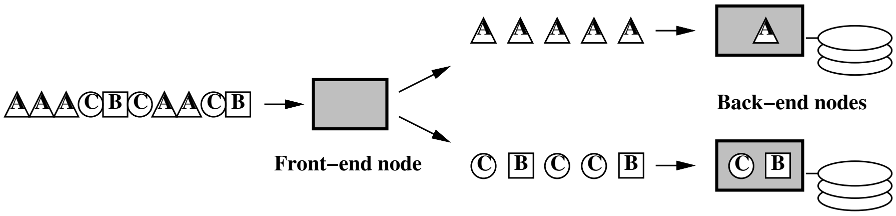

Figure 1: Locality-Aware Request Distribution

targets will arrive at both back-ends. This increases the likelihood of a cache miss, if the sum of the sizes of the three targets, or, more generally, if the size of the working set exceeds the size of the main memory cache at an individual back-end node.

Of course, by naively distributing incoming requests in a content-based manner as suggested in Figure 1, the load between different back-ends might become unbalanced, resulting in worse performance. The first major challenge in building a LARD cluster is therefore to design a practical and efficient strategy that *simultaneously* achieves load balancing and high cache hit rates on the back-ends. The second challenge stems from the need for a protocol that allows the front-end to hand off an established client connection to a back-end node, in a manner that is transparent to clients and is efficient enough not to render the front-end a bottleneck. This requirement results from the front-end's need to inspect the target content of a request *prior* to assigning the request to a back-end node. This paper demonstrates that these challenges can be met, and that LARD produces substantially higher throughput than the state-of-the-art approaches where request distribution is solely based on load balancing, for workloads whose working set exceeds the size of the individual node caches.

Increasing a server's cache effectiveness is an important step towards meeting the demands placed on current and future network servers. Being able to cache the working set is critical to achieving high throughput, as a state-of-the-art disk device can deliver no more than 120 block requests/sec, while high-end network servers will be expected to serve thousands of document requests per second. Moreover, typical working set sizes of web servers can be expected to grow over time, for two reasons. First, the amount of content made available by a single organization is typically growing over time. Second, there is a trend towards centralization of web servers within organizations. Issues such as cost and ease of administration, availability, security, and high-capacity backbone network access cause organizations to move towards large, centralized network servers that handle all of the organization's web presence. Such servers have to handle the combined working sets of all the servers they supersede.

With round-robin distribution, a cluster does not scale well to larger working sets, as *each* node's main memory cache has to fit the entire working set. With LARD, the effective cache size approaches the *sum* of the node cache sizes. Thus, adding nodes to a cluster can accommodate both increased traffic (due to additional CPU power) and larger working sets (due to the increased effective cache size).

This paper presents the following contributions:

1. a practical and efficient LARD strategy that achieves high cache hit rates and good load balancing,

2. a trace-driven simulation that demonstrates the performance potential of locality-aware request distribution,

3. an efficient *TCP handoff protocol*, that enables content-based request distribution by providing client-transparent connection handoff for TCP-based network services, and

4. a performance evaluation of a prototype LARD server cluster, incorporating the TCP handoff protocol and the LARD strategy.

The outline of the rest of this paper is as follows: In Section 2 we develop our strategy for locality-aware request distribution. In Section 3 we describe the model used to simulate the performance of LARD in comparison to other request distribution strategies. In Section 4 we present the results of the simulation. In Section 5 we move on to the practical implementation of LARD, particularly the TCP handoff protocol. We describe the experimental environment in which our LARD server is implemented and its measured performance in Section 6. We describe related work in Section 7 and we conclude in Section 8.

2 Strategies for Request Distribution

2.1 Assumptions

The following assumptions hold for all request distribution strategies considered in this paper:

- The front-end is responsible for handing off new connections and passing incoming data from the client to the back-end nodes. As a result, it must keep track of open and closed connections, and it can use this information in making load balancing decisions. The front-end is not involved in handling outgoing data, which is sent directly from the back-ends to the clients.
- The front-end limits the number of outstanding requests at the back-ends. This approach allows the front-end more flexibility in responding to changing load on the back-ends, since waiting requests can be directed to back-ends as capacity becomes available. In contrast, if we queued requests only on the back-end nodes, a slow node could cause many requests to be delayed even though other nodes might have free capacity.
- Any back-end node is capable of serving any target, although in certain request distribution strategies, the front-end may direct a request only to a subset of the back-ends.

2.2 Aiming for Balanced Load

In state-of-the-art cluster servers, the front-end uses *weighted round-robin* request distribution [7, 14]. The incoming requests are distributed in round-robin fashion, weighted by some measure of the load on the different back-ends. For instance, the CPU and disk utilization, or the number of open connections in each back-end may be used as an estimate of the load.

This strategy produces good load balancing among the back-ends. However, since it does not consider the type of service or requested document in choosing a back-end node, each back-end node is equally likely to receive a given type of request. Therefore, each back-end node receives an approximately identical working set of requests, and caches an approximately identical set of documents. If this working set exceeds the size of main memory available for caching documents, frequent cache misses will occur.

2.3 Aiming for Locality

In order to improve locality in the back-end's cache, a simple front-end strategy consists of partitioning the name space of the database in some way, and assigning request for all targets in a particular partition to a particular back-end. For instance, a hash function can be used to perform the partitioning. We will call this strategy *locality-based* [LB].

A good hashing function partitions both the name space and the working set more or less evenly among the back-ends. If this is the case, the cache in each back-end should achieve a much higher hit rate, since it is only trying to cache its subset of the working set, rather than the entire working set, as with load balancing based approaches. What is a good partitioning for locality may, however, easily prove a poor choice of partitioning for load balancing. For example, if a small set of targets in the working set account for a large fraction of the incoming requests, the back-ends serving those targets will be far more loaded than others.

2.4 Basic Locality-Aware Request Distribution

The goal of LARD is to combine good load balancing and high locality. We develop our strategy in two steps. The basic strategy, described in this subsection, always assigns a single back-end node to serve a given target, thus making the idealized assumption that a single target cannot by itself exceed the capacity of one node. This restriction is removed in the next subsection, where we present the complete strategy.

Figure 2 presents pseudo-code for the basic LARD. The front-end maintains a one-to-one mapping of targets to back-end nodes in the `server` array. When the first request arrives for a given target, it is assigned a back-end node by choosing a lightly loaded back-end node. Subsequent requests are directed to a target's assigned back-end node, unless that node is overloaded. In the latter case, the target is assigned a new back-end node from the current set of lightly loaded nodes.

A node's load is measured as the number of active connections, i.e., connections that have been handed off to the node, have not yet completed, and are showing request activity. Observe that an overloaded node will fall behind and the resulting queuing of requests will cause its number of active connections to increase, while the number of active connections at an under-loaded node will tend to zero. Monitoring the relative

```
while (true)
  fetch next request r;
  if server[r.target] = null then
    n, server[r.target] ← {least loaded node};
  else
    n ← server[r.target];
    if (n.load > T_high && ∃ node with load < T_low) ||
         n.load ≥ 2 * T_high then
       n, server[r.target] ← {least loaded node};
  send r to n;
```

Figure 2: The Basic LARD Strategy

number of active connections allows the front-end to estimate the amount of "outstanding work" and thus the relative load on a back-end without requiring explicit communication with the back-end node.

The intuition for the basic LARD strategy is as follows: The distribution of targets when they are first requested leads to a partitioning of the name space of the database, and indirectly to a partitioning of the working set, much in the same way as with the strategy purely aiming for locality. It also derives similar locality gains from doing so. Only when there is a significant load imbalance do we diverge from this strategy and re-assign targets. The definition of a "significant load imbalance" tries to reconcile two competing goals. On one hand, we do not want greatly diverging load values on different back-ends. On the other hand, given the cache misses and disk activity resulting from re-assignment, we do not want to re-assign targets to smooth out only minor or temporary load imbalances. It suffices to make sure that no node has idle resources while another back-end is dropping behind.

We define T_{low} as the load (in number of active connections) below which a back-end is likely to have idle resources. We define T_{high} as the load above which a node is likely to cause substantial delay in serving requests. If a situation is detected where a node has a load larger than T_{high} while another node has a load less than T_{low}, a target is moved from the high-load to the low-load back-end. In addition, to limit the delay variance among different nodes, once a node reaches a load of $2T_{high}$, a target is moved to a less loaded node, even if no node has a load of less than T_{low}.

If the front-end did not limit the total number of active connections admitted into the cluster, the load on all nodes could rise to $2T_{high}$, and LARD would then behave like WRR. To prevent this, the front-end limits the sum total of connections handed to all back-end nodes to the value $S = (n-1) * T_{high} + T_{low} - 1$, where n is the number of back-end nodes. Setting S to this value ensures that at most $n-2$ nodes can have a load $\geq T_{high}$ while no node has load $< T_{low}$. At the same time, enough connections are admitted to ensure all n nodes can have a load above T_{low} (i.e., be fully utilized) and still leave room for a limited amount of load imbalance between the nodes (to prevent unnecessary target reassignments in the interest of locality).

The two conditions for deciding when to move a target attempt to ensure that the cost of moving is incurred only when the load difference is substantial enough to warrant doing so. Whenever a target gets reassigned, our two tests combined with the definition of S ensure that the load difference between the old and new targets is at least $T_{high} - T_{low}$. To see this, note that the definition of S implies that there must always exist a node with a load $< T_{high}$. The maximal load imbalance that can arise is $2T_{high} - T_{low}$.

The appropriate setting for T_{low} depends on the speed of the back-end nodes. In practice, T_{low} should be chosen high enough to avoid idle resources on back-end nodes, which could cause throughput loss. Given T_{low}, choosing T_{high} involves a tradeoff. $T_{high} - T_{low}$ should be low enough to limit the delay variance among the back-ends to acceptable levels, but high enough to tolerate limited load imbalance and short-term load fluctuations without destroying locality.

Simulations to test the sensitivity of our strategy to these parameter settings show that the maximal delay difference increases approximately linearly with $T_{high} - T_{low}$. The throughput increases mildly and eventually flattens as $T_{high} - T_{low}$ increases. Therefore, T_{high} should be set to the largest possible value that still satisfies the desired bound on the delay difference between back-end nodes. Given a desired maximal delay difference of D secs and an average request service time of R secs, T_{high} should be set to $(T_{low} + D/R)/2$, subject to the obvious constraint that $T_{high} > T_{low}$. The setting of T_{low} can be conservatively high with no adverse impact on throughput and only a mild increase in the average delay. Furthermore, if desired, the setting of T_{low} can be easily automated by requesting explicit load information from the back-end nodes during a "training phase". In our simulations and in the prototype, we have found settings of $T_{low} = 25$ and $T_{high} = 65$ active connections to give good performance across all workloads we tested.

2.5 LARD with Replication

A potential problem with the basic LARD strategy is that a given target is served by only a single node at any given time. However, if a single target causes a back-end to go into an overload situation, the desirable action is to assign several back-end nodes to serve that document, and to distribute requests for that target among the serving nodes. This leads us to the second version of our strategy, which allows replication.

Pseudo-code for this strategy is shown in Figure 3. It differs from the original one as follows: The front-end maintains a mapping from targets to a *set* of nodes that serve the target. Requests for a target are assigned to the least loaded node in the target's server set. If a load imbalance occurs, the front-end checks if the requested document's server set has changed recently (within K seconds). If so, it picks a lightly loaded node and adds that node to the server set for the target. On the other hand, if a request target has multiple servers and has not moved or had a server node added for some time (K seconds), the front-end removes one node from the target's server set. This ensures that the degree of replication for a target does not remain unnecessarily high once it is requested less often. In our experiments, we used values of $K = 20$ secs.

2.6 Discussion

As will be seen in Sections 4 and 6, the LARD strategies result in a good combination of load balancing and locality. In addition, the strategies outlined above have several desirable features. First, they do not require any extra communication between the front-end and the back-ends. Second, the front-end need not keep track of any frequency of access information or try to model the contents of the caches of the back-ends. In particular, the strategy is independent of the local replacement policy used by the back-ends. Third, the absence of elaborate state in the front-end makes it rather straightforward to recover from a back-end node failure. The front-end simply re-assigns targets assigned to the failed back-end as if they had not been assigned before. For all these reasons, we argue that the proposed strategy can be implemented without undue complexity.

```
while (true)
  fetch next request r;
  if serverSet[r.target] = ∅ then
    n, serverSet[r.target] ← {least loaded node};
  else
    n ← {least loaded node in serverSet[r.target]};
    m ← {most loaded node in serverSet[r.target]};
    if (n.load > T_high && ∃ node with load < T_low) ||
        n.load ≥ 2T_high then
      p ← {least loaded node};
      add p to serverSet[r.target];
      n ← p;
    if |serverSet[r.target]| > 1 &&
        time() - serverSet[r.target].lastMod > K then
      remove m from serverSet[r.target];
  send r to n
  if serverSet[r.target] changed in this iteration then
    serverSet[r.target].lastMod ← time();
```

Figure 3: LARD with Replication

In a simple implementation of the two strategies, the size of the `server` or `serverSet` arrays, respectively, can grow to the number of targets in the server's database. Despite the low storage overhead per target, this can be of concern in servers with very large databases. In this case, the mappings can be maintained in an LRU cache, where assignments for targets that have not been accessed recently are discarded. Discarding mappings for such targets is of little consequence, as these targets have most likely been evicted from the back-end nodes' caches anyway.

3 Simulation

To study various request distribution policies for a range of cluster sizes under different assumptions for CPU speed, amount of memory, number of disks and other parameters, we developed a configurable web server cluster simulator. We also implemented a prototype of a LARD-based cluster, which is described in Section 6.

3.1 Simulation Model

The simulation model is depicted in Figure 4. Each back-end node consists of a CPU and locally-attached disk(s), with separate queues for each. In addition, each node maintains its own main memory cache of configurable size and replacement policy. For simplicity, caching is performed on a whole-file basis.

Processing a request requires the following steps:

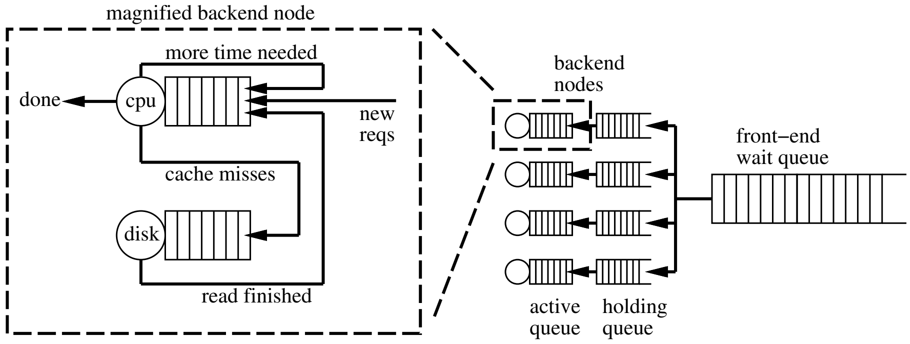

Figure 4: Cluster Simulation Model

connection establishment, disk reads (if needed), target data transmission, and connection teardown. The assumption is that front-end and networks are fast enough not to limit the cluster's performance, thus fully exposing the throughput limits of the back-ends. Therefore, the front-end is assumed to have no overhead and all networks have infinite capacity in the simulations.

The individual processing steps for a given request must be performed in sequence, but the CPU and disk times for differing requests can be overlapped. Also, large file reads are blocked, such that the data transmission immediately follows the disk read for each block. Multiple requests waiting on the same file from disk can be satisfied with only one disk read, since all the requests can access the data once it is cached in memory.

The costs for the basic request processing steps used in our simulations were derived by performing measurements on a 300 Mhz Pentium II machine running FreeBSD 2.2.5 and an aggressive experimental web server. Connection establishment and teardown costs are set at $145\mu s$ of CPU time each, while transmit processing incurs $40\mu s$ per 512 bytes. Using these numbers, an 8 KByte document can be served from the main memory cache at a rate of approximately 1075 requests/sec.

If disk access is required, reading a file from disk has a latency of 28 ms (2 seeks + rotational latency). The disk transfer time is $410\mu s$ per 4 KByte (resulting in approximately 10 MBytes/sec peak transfer rate). For files larger than 44 KBytes, an additional 14 ms (seek plus rotational latency) is charged for every 44 KBytes of file length in excess of 44 KBytes. 44 KBytes was measured as the average disk transfer size between seeks in our experimental server. Unless otherwise stated, each back-end node has one disk.

The cache replacement policy we chose for all simulations is Greedy-Dual-Size (GDS), as it appears to be the best known policy for Web workloads [5]. We have also performed simulations with LRU, where files with a size of more than 500KB are never cached. The relative performance of the various distribution strategies remained largely unaffected. However, the absolute throughput results were up to 30% lower with LRU than with GDS.

3.2 Simulation Inputs

The input to the simulator is a stream of tokenized target requests, where each token represents a unique target being served. Associated with each token is a target size in bytes. This tokenized stream can be synthetically created, or it can be generated by processing logs from existing web servers.

One of the traces we use was generated by combining logs from multiple departmental web servers at Rice University. This trace spans a two-month period. Another trace comes from IBM Corporation's main web server (www.ibm.com) and represents server logs for a period of 3.5 days starting at midnight, June 1, 1998.

Figures 5 and 6 show the cumulative distributions of request frequency and size for the Rice University trace and the IBM trace, respectively. Shown on the x-axis is the set of target files in the trace, sorted in decreasing order of request frequency. The y-axis shows the cumulative fraction of requests and target sizes, normalized to the total number of requests and total data set size, respectively. The data set for the Rice University trace consist of 37703 targets covering 1418 MB of space, whereas the IBM trace consists of 38527 targets and 1029 MB of space. While the data sets in both traces are of a comparable size, it is evident from the graphs that the Rice trace has much less locality than the IBM trace. In the Rice trace, 560/705/927 MB of memory is needed to cover 97/98/99% of all requests, respectively, while only 51/80/182 MB are needed to cover the same fractions of requests in the IBM trace.

This difference is likely to be caused in part by the different time spans that each trace covers. Also, the IBM trace is from a single high-traffic server, where the content designers have likely spent effort to minimize the sizes of high frequency documents in the interest of performance. The Rice trace, on the other hand, was merged from the logs of several departmental servers.

As with all caching studies, interesting effects can only be observed if the size of the working set exceeds that of the cache. Since even our larger trace has a relatively small data set (and thus a small working set), and also to anticipate future trends in working set sizes, we chose to set the default node cache size in our simulations to 32 MB. Since in reality, the cache has to share main memory with OS kernel and server applications, this typically requires at least 64 MB of memory in an actual server node.

3.3 Simulation Outputs

The simulator calculates overall throughput, hit rate, and underutilization time. Throughput is the number

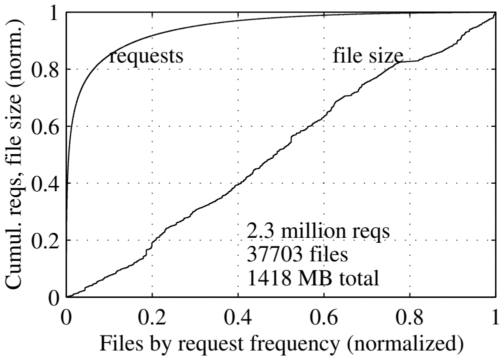

Figure 5: Rice University Trace

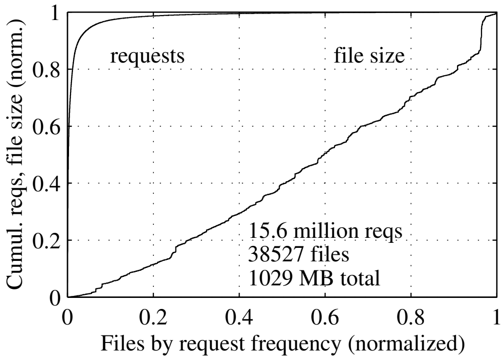

Figure 6: IBM Trace

of requests in the trace that were served per second by the entire cluster, calculated as the number of requests in the trace divided by the simulated time it took to finish serving all the requests in the trace. The request arrival rate was matched to the aggregate throughput of the server.

The cache hit ratio is the number of requests that hit in a back-end node's main memory cache divided by the number of requests in the trace. The idle time was measured as the fraction of simulated time during which a back-end node was underutilized, averaged over all back-end nodes.

Node underutilization is defined as the time that a node's load is less than 40% of T_{low}. This value was determined by inspection of the simulator's disk and CPU activity statistics as a point below which a node's disk and CPU both had some idle time in virtually all cases. The overall throughput is the best summary metric, since it is affected by all factors. The cache hit rate gives an indication of how well locality is being maintained, and the node underutilization times indicate how well load balancing is maintained.

4 Simulation Results

We simulate the four different request distribution strategies presented in Section 2.

1. weighted round-robin [WRR],
2. locality-based [LB],
3. basic LARD [LARD], and
4. LARD with replication [LARD/R].

In addition, observing the large amount of interest generated by global memory systems (GMS) and cooperative caching to improve hit rates in cluster main memory caches [8, 11, 17], we simulate a weighted round-robin strategy in the presence of a global memory system on the back-end nodes. We refer to this system as WRR/GMS. The GMS in WRR/GMS is loosely based on the GMS described in Feeley et al. [11].

We also simulate an idealized locality-based strategy, termed LB/GC, where the front-end keeps track of each back-end's cache state to achieve the effect of a global cache. On a cache hit, the front-end sends the requests to the back-end that caches the target. On a miss, the front-end sends the request to the back-end that caches the globally "oldest" target, thus causing eviction of that target.

4.1 Rice University Trace

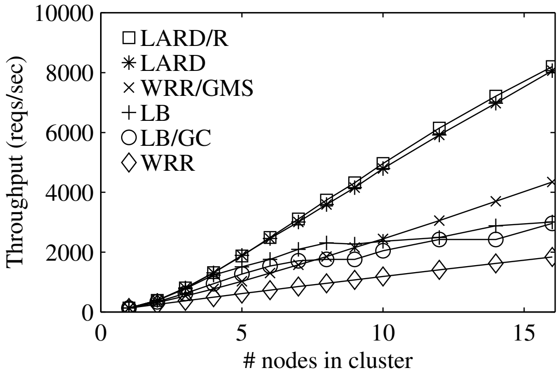

Figure 7: Throughput

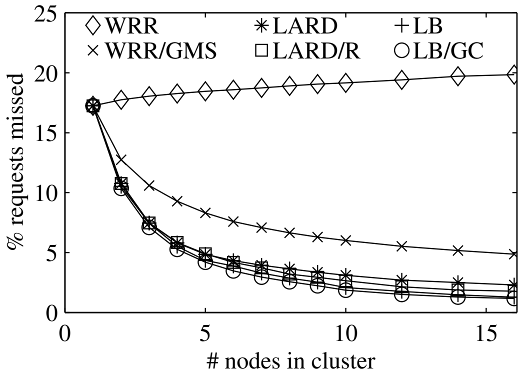

Figure 8: Cache Miss Ratio

Figures 7, 8, and 9 show the aggregate throughput, cache miss ratio, and idle time as a function of the number of back-end nodes for the combined Rice University trace. WRR achieves the lowest throughput, the highest cache miss ratio, but also the lowest idle time (i.e., the highest back-end node utilization) of all strategies. This confirms our reasoning that the weighted round-robin scheme achieves good load balancing (thus minimizing idle time). However, since it ignores locality, it suffers many cache misses. This latter effect dominates, and the net effect is that the server's throughput is limited by disk accesses. With WRR, the effective size of the server cache remains at the size of the individual node

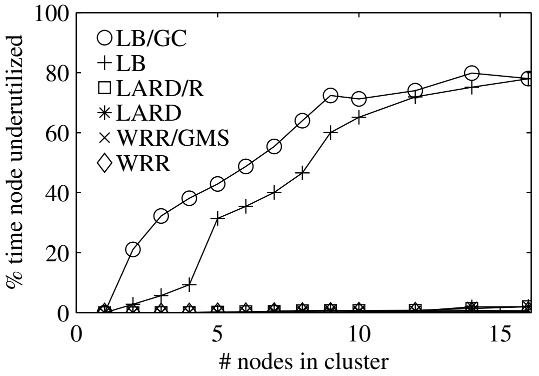

Figure 9: Idle Time

cache, independent of the number of nodes. This can be clearly seen in the flat cache miss ratio curve for WRR.

As expected, both LB schemes achieve a decrease in cache miss ratio as the number of nodes increases. This reflects the aggregation of effective cache size. However, this advantage is largely offset by a loss in load balancing (as evidenced by the increased idle time), resulting in only a modest throughput advantage over WRR.

An interesting result is that LB/GC, despite its greater complexity and sophistication, does not yield a significant advantage over the much simpler LB. This suggests that the hashing scheme used in LB achieves a fairly even partitioning of the server's working set, and that maintaining cache state in the front-end may not be necessary to achieve good cache hit ratios across the back-end nodes. This partly validates the approach we took in the design of LARD, which does not attempt to model the state of the back-end caches.

The throughput achieved with LARD/R exceeds that of the state-of-the-art WRR on this trace by a factor of 3.9 for a cluster size of eight nodes, and by about 4.5 for sixteen nodes. The Rice trace requires the combined cache size of eight to ten nodes to hold the working set. Since WRR cannot aggregate the cache size, the server remains disk bound for all cluster sizes. LARD and LARD/R, on the other hand, cause the system to become increasingly CPU bound for eight or more nodes, resulting in superlinear speedup in the 1-10 node region, with linear, but steeper speedup for more than ten nodes. Another way to read this result is that with WRR, it would take a ten times larger cache in each node to match the performance of LARD on this particular trace. We have verified this fact by simulating WRR with a tenfold node cache size.

The reason for the increased throughput and speedup can also be clearly seen in the graphs for idle time and cache miss ratio. LARD and LARD/R achieve average idle times around 1%, while achieving cache miss ratios that decrease with increasing cluster size and reach values below 4% for eight and more nodes in the case of LARD, going down to 2% at sixteen nodes in the case of LARD/R. Thus, LARD and LARD/R come close to WRR in terms of load balancing while simultaneously achieving cache miss ratios close to those obtained with LB/GC. Thus, LARD and LARD/R are able to translate most of the locality advantages of LB/GC into additional server throughput.

The throughput achieved with LARD/R exceeds that of LARD slightly for seven or more nodes, while achieving lower cache miss ratio and lower idle time. While WRR/GMS achieves a substantial performance advantage over WRR, its throughput remains below 50% of LARD and LARD/R's throughput for all cluster sizes.

4.2 Other Workloads

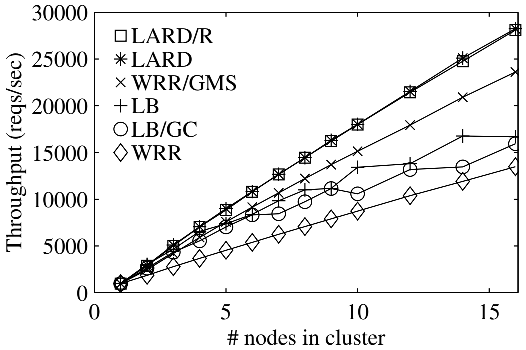

Figure 10: Throughput on IBM Trace

Figure 10 shows the throughput results obtained for the various strategies on the IBM trace (www.ibm.com). In this trace, the average file size is smaller than in the Rice trace, resulting in much larger throughput numbers for all strategies. The higher locality of the IBM trace demands a smaller effective cache size to cache the working set. Thus, LARD and LARD/R achieve superlinear speedup only up to 4 nodes in this trace, resulting in a throughput that is slighly more than twice that of WRR for 4 nodes and above.

WRR/GMS achieves much better relative performance on this trace than on the Rice trace and comes within 15% of LARD/R's throughput at 16 nodes. However, this result has to be seen in light of the very generous assumptions made in the simulations about the performance of the WRR/GMS system. It was assumed that maintaining the global cache directory and implementing global cache replacement has no cost.

The performance of LARD/R only slightly exceeds that of LARD on the Rice trace and matches that of LARD on the IBM trace. The reason is that neither trace contains high-frequency targets that can benefit from replication. The highest frequency files in the Rice and IBM traces account for only 2% and 5%, respectively, of all requests in the traces. However, it is clear that real workloads exist that contain targets with much higher request frequency (e.g. www.netscape.com). To evaluate LARD and LARD/R on such workloads, we modified the Rice trace to include a small number of artifical high-frequency targets and varied their request rate between 5 and 75% of the total number of requests in the trace. With this workload, the throughput achieved with LARD/R exceeds that of LARD by 0-15%. The most significant increase occurs when the size of the "hot" targets is larger than 20 KBytes and the combined access frequency of all hot targets accounts for 10-60% of the total number of requests.

We also ran simulations on a trace from the IBM web server hosting the Deep Blue/Kasparov Chess match in

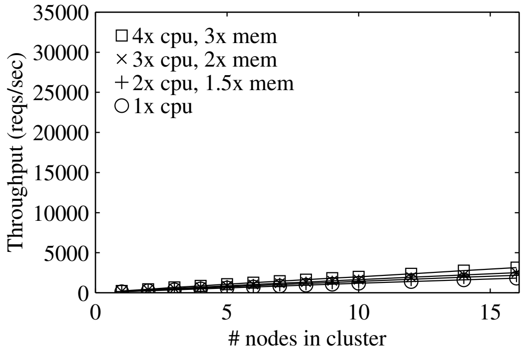

Figure 11: WRR vs CPU

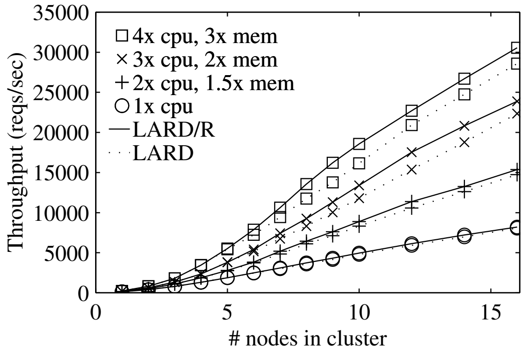

Figure 12: LARD vs CPU

May 1997. This trace is characterized by large numbers of requests to a small set of targets. The working set of this trace is very small and achieves a low miss ratio with a main memory cache of a single node (32 MB). This trace presents a best-case scenario for WRR and a worst-case scenario for LARD, as there is nothing to be gained from an aggregation of cache size, but there is the potential to lose performance due to imperfect load balancing. Our results show that both LARD and LARD/R closely match the performance of WRR on this trace. This is reassuring, as it demonstrates that our strategy can match the performance of WRR even under conditions that are favorable to WRR.

4.3 Sensitivity to CPU and Disk Speed

In our next set of simulations, we explore the impact of CPU speed on the relative performance of LARD versus the state-of-the-art WRR. We performed simulations on the Rice trace with the default CPU speed setting explained in Section 3, and with twice, three and four times the default speed setting. The [1x] speed setting represents a state-of-the-art inexpensive high-end PC (300 MHz Pentium II), and the higher speed settings project the speed of high-end PCs likely to be available in the the next few years. As the CPU speed increases while disk speed remains constant, higher cache hit rates are necessary to remain CPU bound at a given cluster size, requiring larger per-node caches. We made this adjustment by setting the node memory size to 1.5, 2, and 3 times the base amount (32 MB) for the [2x], [3x] and [4x] CPU speed settings, respectively.

As CPU speeds are expected to improve at a much faster rate than disk speeds, one would expect that the importance of caching and locality increases. Indeed, our simulations confirm this. Figures 11 and 12, respectively, show the throughput results for WRR and LARD/R on the combined Rice University trace with different CPU speed assumptions. It is clear that WRR cannot benefit from added CPU at all, since it is disk-bound on this trace. LARD and LARD/R, on the other hand, can capitalize on the added CPU power, because their cache aggregation makes the system increasingly CPU bound as nodes are added to the system. In addition, the results indicate the throughput advantage of LARD/R over LARD increases with CPU speed, even on a workload that presents little opportunity for replication.

In our final set of simulations, we explore the impact of using multiple disks in each back-end node on the relative performance of LARD/R versus WRR. Figures 13 and 14, respectively, show the throughput results for WRR and LARD/R on the combined Rice University trace with different numbers of disks per back-end node. With LARD/R, a second disk per node yields a mild throughput gain, but additional disks do not achieve any further benefit. This can be expected, as the increased cache effectiveness of LARD/R causes a reduced dependence on disk speed.

WRR, on the other hand, greatly benefits from multiple disks as its throughput is mainly bound by the performance of the disk subsystem. In fact, with four disks per node and 16 nodes, WRR comes within 15% of LARD/R's throughput. However, the are several things to note about this result. First, the assumptions made in the simulations about the performance of multiple disks are generous. It is assumed that both seek and disk transfer operations can be fully overlapped among all disks. In practice, this would require that each disk is attached through a separate SCSI bus/controller.

Second, it is assumed that the database is striped across the multiple disks in a manner that achieves good load balancing among the disks with respect to the workload (trace). In our simulations, the files were distributed across the disks in round-robin fashion based on decreasing order of request frequency in the trace[2].

Finally, WRR has the same scalability problems with respect to disks as it has with memory. To upgrade a cluster with WRR, it is not sufficient to add nodes as with LARD/R. Additional disks (and memory) have to be added to all nodes to achieve higher performance.

4.4 Delay

While most of our simulations focus on the server's throughput limits, we also monitored request delay in our simulations for both the Rice University trace as well as the IBM trace. On the Rice University trace, the average request delay for LARD/R is less than 25% that of WRR. With the IBM trace, LARD/R's average delay is one half that of WRR.

[2] Note that replicating the entire database on each disk as an approach to achieving disk load balancing would require special OS support to avoid double buffering and caching of replicated files and to assign requests to disks dynamically based on load.

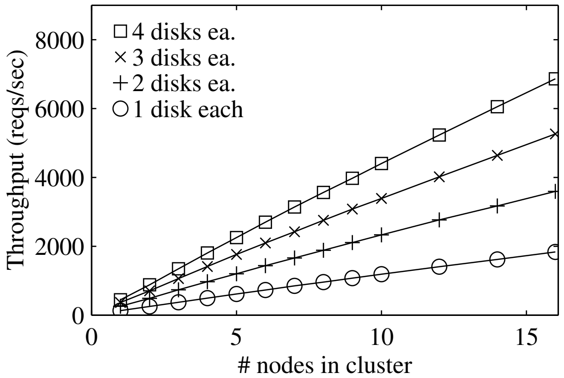

Figure 13: WRR vs disks

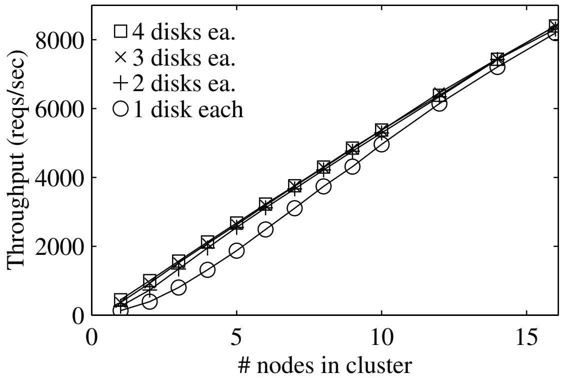

Figure 14: LARD/R vs disks

5 TCP Connection Handoff

In this section, we briefly discuss our TCP handoff protocol and present some performance results with a prototype implementation. A full description of the protocol is beyond the scope of this paper. The TCP handoff protocol is used to hand off established client TCP [23] connections between the front-end and the back-end of a cluster server that employs content-based request distribution.

A handoff protocol is necessary to enable content-based request distribution in a client-transparent manner. This is true for any service (like HTTP) that relies on a connection-oriented transport protocol like TCP. The front-end must establish a connection with the client to inspect the target content of a request *prior* to assigning the connection to a back-end node. The established connection must then be handed to the chosen back-end node. State-of-the-art commercial cluster front-ends (e.g., [7, 14]) assign requests without regard to the requested content and can therefore forward client requests to a back-end node prior to establishing a connection with the client.

Our handoff protocol is transparent to clients and also to the server applications running on the back-end nodes. That is, no changes are needed on the client side, and server applications can run unmodified on the back-end nodes. Figure 15 depicts the protocol stacks on the clients, front-end, and back-ends, respectively. The handoff protocol is layered on top of TCP and runs on the front-end and back-end nodes. Once a connection is handed off to a back-end node, incoming traffic on that connection (principally acknowledgment packets) is forwarded by an efficient forwarding module at the bottom of the front-end's protocol stack.

The TCP implementation running on the front-end and back-ends needs a small amount of additional support for handoff. In particular, the protocol module needs to support an operation that allows the TCP handoff protocol to create a TCP connection at the back-end without going through the TCP three-way handshake. Likewise, an operation is required that retrieves the state of an established connection and destroys the connection state without going through the normal message handshake required to close a TCP connection.

Figure 15 depicts a typical scenario: (1) a client connects to the front-end, (2) the dispatcher at the front-

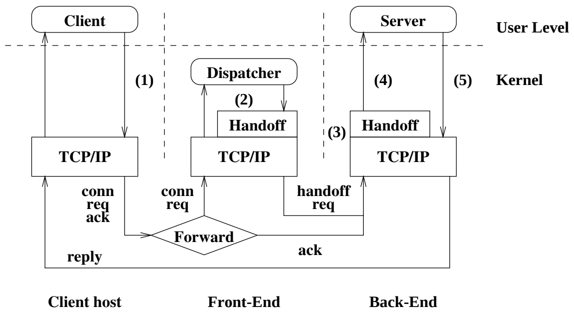

Figure 15: TCP connection handoff

end accepts the connection and hands it off to a back-end using the handoff protocol, (3) the back-end takes over the established connection received by the handoff protocols, (4) the server at the back-end accepts the created connection, and (5) the server at the back-end sends replies directly to the client. The dispatcher is a software module that implements the distribution policy, e.g. LARD.

Once a connection is handed off to a back-end node, the front-end must forward packets from the client to the appropriate back-end node. A single back-end node that fully utilizes a 100 Mb/s network sending data to clients will receive at least 4128 acknowledgments per second (assuming an IP packet size of 1500 and delayed TCP ACKs). Therefore, it is crucial that this packet forwarding is fast.

The forwarding module is designed to allow very fast forwarding of acknowledgment packets. The module operates directly above the network interface and executes in the context of the network interface interrupt handler. A simple hash table lookup is required to determine whether a packet should be forwarded. If so, the packet's header is updated and it is directly transmitted on the appropriate interface. Otherwise, the packet traverses the normal protocol stack.

Results of performance measurements with an implementation of the handoff protocol are presented in Section 6.2.

The design of our TCP handoff protocol includes provisions for HTTP 1.1 persistent connections, which allow a client to issue multiple requests. The protocol allows the front-end to either let one back-end serve all of the requests on a persistent connection, or to hand off a connection multiple times, so that different requests on the same connection can be served by different back-ends. However, further research is needed to determine the appropriate policy for handling persistent connections in a cluster with LARD. We have not yet experimented with HTTP 1.1 connections as part of this work.

6 Prototype Cluster Performance

In this section, we present performance results obtained with a prototype cluster that uses locality-aware request distribution. We describe the experimental setup used in the experiments, and then present the results.

6.1 Experimental Environment

Our testbed consists of 7 client machines connected to a cluster server. The configuration is shown in Figure 16. Traffic from the clients flows to the front-end (1) and is forwarded to the back-ends (2). Data packets transmitted from the back-ends to the clients bypass the front-end (3).

The front-end of the server cluster is a 300MHz Intel Pentium II based PC with 128MB of memory. The cluster back-end consists of six PCs of the same type and configuration as the front-end. All machines run FreeBSD 2.2.5. A loadable kernel module was added to the OS of the front-end and back-end nodes that implements the TCP handoff protocol, and, in the case of the front-end, the forwarding module. The clients are 166MHz Intel Pentium Pro PCs, each with 64MB of memory.

The clients and back-end nodes in the cluster are connected using switched Fast Ethernet (100Mbps). The front-end is equipped with two network interfaces, one for communication with the clients, one for communication with the back-ends. Clients, front-end, and back-end are connected through a single 24-port switch. All network interfaces are Intel EtherExpress Pro/100B running in full-duplex mode.

The Apache-1.2.4 [2] server was used on the back-end nodes. Our client software is an event-driven program that simulates multiple HTTP clients. Each simulated HTTP client makes HTTP requests as fast as the server cluster can handle them.

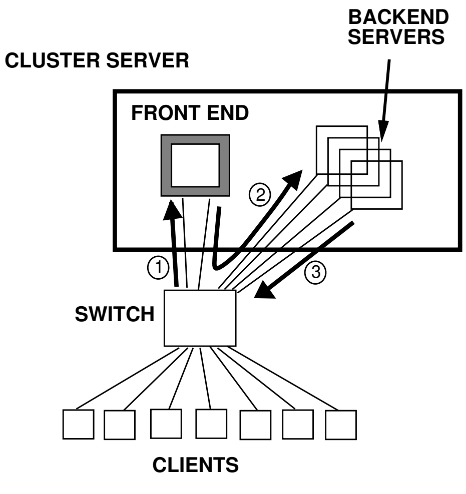

Figure 16: Experimental Testbed

6.2 Front-end Performance Results

Measurements were performed to evaluate the performance and overhead of the TCP handoff protocol and packet forwarding in the front-end. *Handoff latency* is the added latency a client experiences as a result of TCP handoff. *Handoff throughput* is the maximal rate at which the front-end can accept, handoff, and close connections. *Forwarding throughput* refers to the maximal aggregate rate of data transfers from all back-end nodes to clients. Since this data bypasses the front-end, this figure is limited only by the front-end's ability to forward acknowledgments from the clients to the back-ends.

The measured handoff latency is 194 μsecs and the maximal handoff throughput is approximately 5000 connections per second. Note that the added handoff latency is insignificant, given the connection establishment delay over a wide-area network. The measured ACK forwarding overhead is 9 μsecs, resulting in a theoretical maximal forwarding throughput of over 2.5 Gbits/s. We have not been able to measure such high throughput directly due to lack of network resources, but the measured remaining CPU idle time in the front-end at lower throughput is consistent with this figure. Further measurements indicate that with the Rice University trace as the workload, the handoff throughput and forwarding throughput are sufficient to support 10 back-end nodes of the same CPU speed as the front-end.

Moreover, the front-end can be relatively easily scaled to larger clusters either by upgrading to a faster CPU, or by employing an SMP machine. Connection establishment, handoff, and forwarding are independent for different connections, and can be easily parallelized [24]. The dispatcher, on the other hand, requires shared state and thus synchronization among the CPUs. However, with a simple policy such as LARD/R, the time spent in the dispatcher amounts to only a small fraction of the handoff overhead (10-20%). Therefore, we fully expect that the front-end performance can be scaled to larger clusters effectively using an inexpensive SMP platform equipped with multiple network interfaces.

6.3 Cluster Performance Results

A segment of the Rice University trace was used to drive the prototype cluster. A single back-end node running Apache can deliver about 167 req/sec on this trace. On cached, small files (less than 8 KB), an Apache back-end can complete about 800 req/sec.

The Apache Web server relies on the file caching services of the underlying operating system. FreeBSD uses a unified buffer cache, where cached files are competing with user processes for physical memory pages. All page replacement is controlled by FreeBSD's pageout daemon, which implements a variant of the clock algorithm [20]. The cache size is variable and depends on main memory pressure from user applications. In our 128 MB back-ends, memory demands from kernel and Apache server processes leave about 100 MB of free memory. In practice, we observed file cache sizes between 70 and 97 MB.

We measure the total HTTP throughput of the server cluster with increasing numbers of back-end nodes and with the front-end implementing either WRR

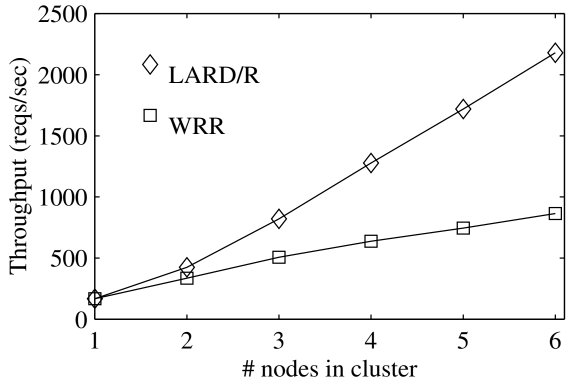

Figure 17: HTTP Throughput (Apache)

or LARD/R. The results are shown in Figure 17 and confirm the predictions of the simulator. The throughput achieved with LARD/R exceeds that of WRR by a factor of 2.5 for six nodes. Running LARD/R on a cluster with six nodes at maximal throughput and an aggregate server bandwidth of over 280 Mb/s, the front-end CPU was 60% utilized. This is consistent with our earlier projection that a single CPU front-end can support 10 back-ends of equal CPU speed.

7 Related Work

Much current research addresses the scalability problems posed by the Web. The work includes cooperative caching proxies inside the network, push-based document distribution, and other innovative techniques [3, 6, 10, 16, 19, 22]. Our proposal addresses the complementary issue of providing support for cost-effective, scalable network servers.

Network servers based on clusters of workstations are starting to be widely used [12]. Several products are available or have been announced for use as front-end nodes in such cluster servers [7, 14]. To the best of our knowledge, the request distribution strategies used in the cluster front-ends are all variations of weighted round-robin, and do not take into account a request's target content. An exception is the Dispatch product by Resonate, Inc., which supports content-based request distribution [21]. The product does not appear to use any dynamic distribution policies based on content and no attempt is made to achieve cache aggregation via content-based request distribution.

Hunt et al. proposed a TCP option designed to enable content-based load distribution in a cluster server [13]. The design has not been implemented and the performance potential of content-based distribution has not been evaluated as part of that work. Also, no policies for content-based load distribution were proposed. Our TCP handoff protocol design was informed by Hunt et al.'s design, but chooses the different approach of layering a separate handoff protocol on top of TCP.

Fox et al. [12] report on the cluster server technology used in the Inktomi search engine. The work focuses on the reliability and scalability aspects of the system and is complementary to our work. The request distribution policy used in their systems is based on weighted round-robin.

Loosely-coupled distributed servers are widely deployed on the Internet. Such servers use various techniques for load balancing including DNS round-robin [4], HTTP client re-direction [1], Smart clients [25], source-based forwarding [9] and hardware translation of network addresses [7]. Some of these schemes have problems related to the quality of the load balance achieved and the increased request latency. A detailed discussion of these issues can be found in Goldszmidt and Hunt [14] and Damani et al. [9]. None of these schemes support content-based request distribution.

IBM's Lava project [18] uses the concept of a "hit server". The hit server is a specially configured server node responsible for serving cached content. Its specialized OS and client-server protocols give it superior performance for handling HTTP requests of cached documents, but limits it to private Intranets. Requests for uncached documents and dynamic content are delegated to a separate, conventional HTTP server node. Our work shares some of the same goals, but maintains standard client-server protocols, maintains support for dynamic content generation, and focuses on cluster servers.

8 Conclusion

We present and evaluate a practical and efficient locality-aware request distribution (LARD) strategy that achieves high cache hit rates and good load balancing in a cluster server. Trace-driven simulations show that the performance of our strategy exceeds that of the state-of-the-art weighted round-robin (WRR) strategy substantially. On workloads with a working set that does not fit in a single server node's main memory cache, the achieved throughput exceeds that of WRR by a factor of two to four.

Additional simulations show that the performance advantages of LARD over WRR increase with the disparity between CPU and disk speeds. Also, our results indicate that the performance of a hypothetical cluster with WRR distribution and a global memory system (GMS) falls short of LARD under all workloads considered, despite generous assumptions about the performance of a GMS system.

We also propose and evaluate an efficient TCP handoff protocol that enables LARD and other content-based request distribution strategies by providing client-transparent connection handoff for TCP-based network services, like HTTP. Performance results indicate that in our prototype cluster environment and on our workloads, a single CPU front-end can support 10 back-end nodes with equal CPU speed as the front-end. Moreover, the design of the handoff protocols is expected to yield scalable performance on SMP-based front-ends, thus supporting larger clusters.

Finally, we present performance results from a prototype LARD server cluster that incorporates the TCP handoff protocol and the LARD strategy. The measured results confirm the simulation results with respect to the relative performance of LARD and WRR.

In this paper, we have focused on studying HTTP servers that serve static content. However, caching can also be effective for dynamically generated content [15].

Moreover, resources required for dynamic content generation like server processes, executables, and primary data files are also cacheable. While further research is required, we expect that increased locality can benefit dynamic content serving, and that therefore the advantages of LARD also apply to dynamic content.

9 Acknowledgments

Thanks to Ed Costello, Cameron Ferstat, Alister Lewis-Bowen and Chet Murthy, for their help in obtaining the IBM server logs.

References

[1] D. Andresen et al. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proccedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[2] Apache. http://www.apache.org/.

[3] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 Usenix Technical Conference*, Jan. 1997.

[4] T. Brisco. DNS Support for Load Balancing. RFC 1794, Apr. 1995.

[5] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[6] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[7] Cisco Systems Inc. LocalDirector. http://www.cisco.com.

[8] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[9] O. P. Damani, P.-Y. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29:1019–1027, 1997.

[10] P. Danzig, R. Hall, and M. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of the SIGCOMM '93 Conference*, Sept. 1993.

[11] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[13] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[14] IBM Corporation. IBM interactive network dispatcher. http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

[15] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[16] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[17] H. Levy, G. Voelker, A. Karlin, E. Anderson, and T. Kimbrel. Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System. In *Proceedings of the ACM SIGMETRICS '98 Conference*, Madison, WI, June 1998.

[18] J. Liedtke, V. Panteleenko, T. Jaeger, and N. Islam. High-performance caching with the Lava hit-server. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[19] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A push-based distribution substrate for Internet applications. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[20] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[21] Resonate Inc. Resonate dispatch. http://www.resonateinc.com.

[22] M. Seltzer and J. Gwertzman. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Topics in Operating Systems*, 1995.

[23] G. Wright and W. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.

[24] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. Towsley. Networking support for large scale multiprocessor servers. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Philadelphia, Pennsylvania, May 1996.

[25] B. Yoshikawa et al. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 Usenix Technical Conference*, Jan. 1997.