Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers *

Mohit Aron

Peter Druschel

Willy Zwaenepoel

Department of Computer Science
Rice University

{aron,druschel,willy}@cs.rice.edu

ABSTRACT

In network (e.g., Web) servers, it is often desirable to isolate the performance of different classes of requests from each other. That is, one seeks to achieve that a certain minimal proportion of server resources are available for a class of requests, independent of the load imposed by other requests. Recent work demonstrates how to achieve this performance isolation in servers consisting of a single, centralized node; however, achieving performance isolation in a distributed, cluster based server remains a problem.

This paper introduces a new abstraction, the cluster reserve, which represents a resource principal in a cluster based network server. We present a design and evaluate a prototype implementation that extends existing techniques for performance isolation on a single node server to cluster based servers.

In our design, the dynamic cluster-wide resource management problem is formulated as a constrained optimization problem, with the resource allocations on individual machines as independent variables, and the desired cluster-wide resource allocations as constraints. Periodically collected resource usages serve as further inputs to the problem.

Experimental results show that cluster reserves are effective in providing performance isolation in cluster based servers. We demonstrate that, in a number of different scenarios, cluster reserves are effective in ensuring performance isolation while enabling high utilization of the server resources.

1. INTRODUCTION

Web servers based on clusters of commodity PCs or workstations offer a cost-effective and scalable solution to the increasing performance demands placed on popular Web sites. As the volume, variety and sophistication of services offered on the World Wide Web (WWW) increases, such servers must host a rich set of services on a common hardware platform. Examples of such services are the retrieval of static and dynamic Web pages, online databases for information retrieval and electronic commerce, and search engines.

It is often desirable that individual services hosted by a Web site be *performance isolated* from each other. That is, a minimal proportion of server resources is reserved for a service, independent of the present demand for other services. Similarly, sites often wish to ensure that a minimal fraction of server resources be available to serve requests from a certain client community, independent of load generated by other clients. The need for performance isolation arises, for instance, when different services are being paid for by different content providers (e.g., Web hosting), when services have different priorities (e.g., purchasing versus browsing in an e-commerce site), or when an organization wishes to preferentially serve certain client communities (e.g., internal versus external clients).

In this paper, we use the term "service class" to refer to a set of requests for which the server wishes to reserve a certain minimal amount of resources. Each incoming request can be classified as belonging to exactly one service class. Requests can be classified based on the requested resource (i.e., the requested URI), the originator of the request (as indicated by the client network address or some stronger form of client authentication), or both.

State of the art Web sites typically achieve performance isolation by providing separate server nodes for each service class. For example, the various Yahoo! services (search, email, etc.) are hosted on dedicated and separate sets of server cluster nodes. Similarly, organizations often run separate server nodes for internal clients and for external clients. While this approach achieves performance isolation, it typically results in lower average utilization of cluster resources and higher average request latencies, because resources that are not currently utilized by one service class cannot be used by other service classes.

Recent advances in operating systems research [17, 10, 25, 7, 9] allow effective performance isolation in single node Web servers. In this paper, we address the problem of perfor-

*Appears in Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, June 2000.

mance isolation in cluster-based network servers. We propose a new cluster-wide abstraction called *cluster reserve* that is capable of achieving performance isolation between service classes that share the cluster nodes. Cluster reserves act as cluster-wide resource principals by extending the resource management facilities in individual nodes to the cluster. A cluster resource manager is responsible for mapping the resources assigned to a cluster reserve to individual nodes in the cluster. The mapping is dynamically adjusted based on prevailing load conditions and is independent of the request distribution strategy employed in the cluster.

We use a set of benchmarking scenarios to evaluate performance isolation in cluster-based Web servers and present performance results based on a prototype implementation of cluster reserves under synthetic and trace-based workloads. The results show that (1) hosting multiple service classes on a joint set of server nodes affords higher average resource utilization and higher average throughput when compared to the state-of-the-art approach of dedicating cluster nodes to service classes; and, (2) cluster reserves are capable of achieving effective performance isolation when different service classes share a common set of cluster nodes.

The rest of the paper is organized as follows. Section 2 presents some background information used in the rest of the paper. In Section 3 we propose the cluster reserves abstraction. Section 4 discusses our prototype implementation and Section 5 presents performance results obtained with the prototype in a number of benchmark scenarios. Related work is covered in Section 6 and Section 7 presents our conclusions.

2. BACKGROUND

This section presents some technical background on cluster-based servers and resource management in individual server nodes.

2.1 Cluster-based Servers

A cluster-based server consists of a number of commodity workstations or PCs connected by a network. Typically, the server nodes are connected by a high-speed LAN [14], but sub-clusters might be geographically distributed and communicate over the Internet. A geographically distributed cluster aims to place sub-clusters in closer proximity to the clients so as to reduce the client perceived latencies and minimize network load.

One or more nodes in the cluster act as front-ends; these nodes act as point(s) of contact for the clients. Client machines send requests to one of the front-end nodes across the Internet. The choice of a front-end node to contact is made either via DNS round-robin[1], or by asking the user which "mirror" server to contact.

Upon receiving a request, a front-end node decides which server node should serve the request by contacting the *dispatcher*. In making its decision, the dispatcher takes into account the current load and capabilities of the various server nodes, and/or locality of reference considerations [20, 4]. The chosen server node generates the content.

[1]With DNS round-robin, the server host-name is dynamically mapped into one of the IP addresses of the front-end nodes.

2.2 Resource Containers

It is well-known that the resource management facilities found in general-purpose operating systems are not effective in ensuring performance isolation for server applications [13, 7]. Recent research has addressed this problem [17, 10, 25, 7, 9]. Resource containers [7], for instance, can be used to ensure effective performance isolation in single-node Web servers. The cluster reserves proposed in the next section build upon this prior work and achieve cluster-wide performance isolation between service classes by using resource containers on individual nodes.

Resource containers are a first-class operating system abstraction for resource principals, which is independent of processes and threads. Using resource containers, a Web server can associate, for instance, a client network connection, a server thread, and a CGI process with a single resource principal that represents the client request being served. This principal competes with other principals representing other client requests for server resources. Resource containers allow accurate accounting and scheduling of resources consumed on behalf of a single client request or a class of client requests, and enable performance isolation and differentiated quality of service when combined with an appropriate resource scheduler.

3. CLUSTER RESERVES

This section presents cluster reserves, a facility for achieving performance isolation between service classes hosted on a cluster. Cluster reserves are cluster-wide resource principals obtained by logically combining the resource principals on individual cluster nodes. Resources can then be allocated to this cluster-wide resource principal and get translated into allocations for the resource principals (e.g., resource containers) on the individual cluster nodes. Performance isolation on individual cluster nodes, thus, is extended into performance isolation for the cluster.

We begin by defining some terminology:

- *Service class:* A service class defines a set of requests, such that the resources used in serving the requests are accounted for and scheduled separately from resources used to serve requests in different service classes. The notion of a service class, thus, coincides with that of a resource principal in the cluster. Service classes can be defined in terms of sets of requested content (i.e., URIs), in term of the client issuing the request, or a combination thereof.

- *Resource:* A resource is any shared system entity needed for execution by a service and multiplexed by the system between the various services it hosts. CPU time, memory, disk and network bandwidth are all examples of resources. This paper is primarily concerned with the CPU time resource. However, the proposed techniques can be applied to manage other resources as well.

- *Performance isolation:* A system is said to be capable of affording performance isolation between service classes hosted on it if (1) the system permits its resources to be proportioned between the service classes hosted on it, and (2) given sufficient request load, a service class receives at least as much resources as were assigned to it irrespective of the load on other service classes. A desirable third property is that resources not used by some service class may be distributed amongst other services classes.

As we will show, cluster reserves afford effective performance isolation for multiple service classes hosted on a joint set of cluster server nodes. Furthermore, this approach allows unused resources from one service class to be utilized by other service classes. This sharing yields increased utilization and increased throughput when compared to the current approach of hosting different service classes on separate server nodes.

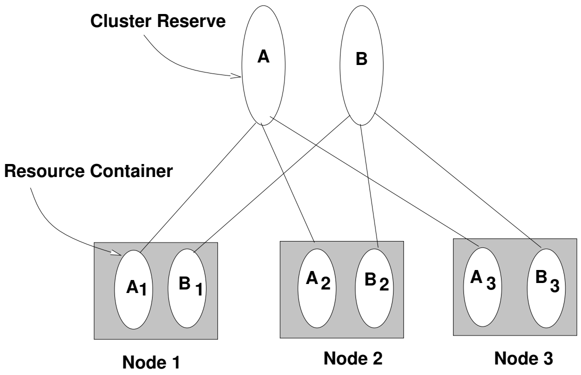

Figure 1: Cluster Reserves

Figure 1 depicts the hierarchical relationship between cluster reserves (cluster resource principals) and resource containers (node resource principals). The figure shows a cluster with three nodes and two cluster reserves A and B. Resources assigned to A and B are dynamically split into resource assignments for the corresponding resource containers on each node. Resource allocations to corresponding resource containers on each node add up to the desired allocation for the associated cluster reserve. For example, resources assigned to A_1, A_2 and A_3 add up to the desired allocation for cluster reserve A.

The partitioning of the resources allocated to a cluster reserve amongst the corresponding resource containers is determined by a cluster resource manager. In general, such a partitioning can be made in an infinite number of ways. For example, an allocation of 50% to cluster reserves A and B in Figure 1 can be achieved with the partitioning $(A_1 = B_1 = A_2 = B_2 = A_3 = B_3 = 50\%)$ or with $(A_1 = B_2 = 100\%, B_1 = A_2 = 0\%, A_3 = B_3 = 50\%)$. A good partitioning depends upon the resource usage on the individual nodes. For example, the first partitioning is suitable in situations when all nodes are equally loaded with requests from service classes associated with reserves A and B. However, the second partitioning is more suitable when *Node 1* and *Node 2* only get requests associated with service classes A and B, respectively.

To compute the partitioning, the cluster resource manager collects resource usage statistics from the cluster nodes and maps the allocation problem to an equivalent constrained optimization problem. The resource usage statistics and the target cluster allocations form the constraints for the problem and the solution yields the individual per-node resource allocations. This method is independent of the request distribution strategy deployed in the cluster.

We now formally show how to map the cluster resource management problem to a constrained optimization problem. First, we describe the inputs and outputs of the problem and state the goal. Then we outline the steps involved in formulating and solving the problem.

We first define the notion of a *resource sink*. A node is considered a resource sink with respect to a particular service class if all resources allocated to the corresponding resource container at that node are being fully utilized by the service class. Intuitively, a resource sink for a particular service class can potentially make use of an increase in resource allocation to the corresponding container.

Goal: Let the cluster consist of N nodes and S service classes. Each service class is associated with a distinct cluster reserve and a distinct resource container at each cluster node. Let r and u be NxS matrices such that r_{ij} and u_{ij} denote the percentage resource allocation and resource usage respectively at node i for service class j. Let D be a vector composed of S elements such that D_j gives the desired percentage resource allocation for the cluster reserve corresponding to service class j. Given input matrices r and u and the vector D, the resource manager computes a NxS matrix R such that R_{ij} gives the new percentage resource allocation for service class j on node i.

The constraints for the problem are formulated using two distinct steps, each of which solves a constrained optimization problem. The first step computes the least feasible deviation between the desired and actual allocations. The second step computes the new resource allocations such that (1) the deviation computed in the first step is achieved, and (2) the computed resource allocations are close to the service class usage on each node. Finally, a third step is used to distribute unassigned cluster resources to idle service classes whose allocations fall below their desired cluster-wide allocation.

Step 1: The objective in this step is to choose the matrix R such that the cluster-wide allocation deviates the least from its desired allocation. This can be stated formally as:

$$Minimize \sum_{j=1}^{S} | \sum_{i=1}^{N} R_{ij} - N * D_j | \qquad (1)$$

Additionally, the problem is constrained as follows:

- The resource allocations on any cluster-node should sum to no more than 100. That is,

$$\forall_{i=1}^{N} \sum_{j=1}^{S} R_{ij} \leq 100$$

- On any node, the new allocation should be no more than the usage if the node is not a resource sink i.e. if the previous allocation exceeds the usage.

$$\forall_{i,j}\ R_{ij} \le u_{ij}\ if\ r_{ij} > u_{ij}$$

- A minimum allocation of 1% is imposed so as to allow some progress to requests whose service classes happen to have the minimal allocation.

$$\forall_{i,j}\ R_{ij} \ge 1$$

The above problem can have infinitely many solutions (i.e. many different matrices R) each of which, however, yields the same value V for equation 1. In this step, we are primarily interested in the value V. Intuitively, V reflects the minimum feasible deviation of the new allocations from the desired ones. The purpose of Step 2 is to choose a solution R, that is the most desirable while still yielding the value V for equation 1.

The value of V computed from this step shall be zero if the resources can be assigned such that the desired cluster-wide allocation is met for all service classes. V can be greater than zero due to the presence of nodes that are not resource sinks with respect to some service classes. A simple example where V will be greater than zero is when no node is a resource sink with respect to a particular service class.

Step 2: A desirable solution matrix R is such that it deviates minimally from the reported resource usage values, while still yielding the minimal feasible value V for equation 1 in Step 1. For this reason, we formulate another constrained optimization problem that adds the following constraint to the constraints for the problem in Step 1:

$$\sum_{j=1}^{S} \left| \sum_{i=1}^{N} R_{ij} - N * D_j \right| = V$$

The objective can be stated as:

$$Minimize \sum_{i=1}^{N} \sum_{j=1}^{S} (R_{ij} - (u_{ij} + k_{ij}))^2 \qquad (2)$$

The term k_{ij} is a small offset (not more than 5 in absolute value) that is intended to bias the solution towards equal allocation to any service class across all the nodes. The small offset, if positive, serves to probe whether a particular service can make use of more resources on a node that is a resource sink with respect to that service[2].

$$\begin{array}{ll} k_{ij} \equiv min(5, 500 * (D_j - u_{ij})/D_j) & if\ u_{ij} < D_j \\ k_{ij} \equiv max(-5, 500 * (D_j - u_{ij})/D_j) & otherwise \end{array}$$

The solution to this problem yields the matrix R whose elements R_{ij} are further processed in Step 3.

Step 3: The solution from Step 2 might still result in some service classes whose allocations do not add up to their desired cluster-wide allocation (the value V from Step 1 shall be non-negative). This is possible only if some of the nodes are not resource sinks with respect to such service classes and hence would have unallocated resources in the solution matrix R. These unassigned resources are allocated to such service classes so as to bring up their aggregate cluster-wide allocation to the desired allocation level. This makes these resources immediately available to these service classes if needed. Any unused resources not used are proportioned amongst other service classes dynamically by the resource container mechanism. The resulting matrix R, therefore, is such that all cluster resources are fully assigned (all allocations on any node sum to 100). The values of R_{ij} are then used to set the allocation of service class j on node i.

[2] A resource sink can potentially make use of more resources; however, the absolute resource demand is not known a priori.

Constrained optimization problems are well understood and commercial tools are available for solving problems with thousands of variables and constraints. However, due to the relatively small problem size that needs to be solved by the resource manager, we used a freely available software tool called **LOQO**[24]. It is also possible to write a hand-tuned solver for our specific problem, but owing to the satisfactory speed afforded by LOQO, we did not adopt this approach.

In the following subsection we show a simple example that demonstrates the resource allocations computed by the resource manager after collecting usage statistics for the services from the various cluster nodes.

3.1 Dynamics of the resource manager

Our example consists of a cluster with two nodes that each host two service classes (Svc 1 and Svc 2). Each service class reserves 50% of the cluster capacity.

Call #		% usage (sink)		% allocation	
		Svc 1	Svc 2	Svc 1	Svc 2
1	Node 1	50 (1)	50 (1)	50	50
	Node 2	50 (1)	50 (1)	50	50
2	Node 1	40 (0)	60 (1)	40	60
	Node 2	50 (1)	50 (1)	60	40
3	Node 1	40 (1)	60 (1)	41	59
	Node 2	60 (1)	40 (1)	59	41
4	Node 1	41 (1)	59 (1)	42	58
	Node 2	59 (1)	41 (1)	58	42
5	Node 1	40 (0)	60 (1)	40	60
	Node 2	58 (1)	42 (1)	60	40
6	Node 1	40 (1)	60 (1)	41	59
	Node 2	60 (1)	40 (1)	59	41
27	Node 1	48.9 (1)	51.1 (1)	49	51
	Node 2	51.1 (1)	48.9 (1)	51	49

Table 1: Dynamics of the Resource Manager

Table 1 shows a series of successive calls invoking the optimizer. Every invocation uses the current resource usages for the two services on each node as input. Also used as input is the information whether a node is a sink with respect to a service (this is given in parenthesis along with the usage). Each call computes the corresponding allocations for each service on each node.

The resource manager is assumed to start at a time when the allocations for each service on all the nodes are 50% each. The reported usage in Call 1 is consistent with the previous allocations and the new allocations are computed to have the same values.

Call 2 reports that node 1 is no longer a resource sink with respect to service 1 and that the usage for service 1 has fallen to 40% on node 1 while it has increased to 60% for service 2. To equalize cluster-wide usage for the two services, the resource manager computes allocations such that the share for service 1 is raised to 60% on Node 2 while the share for service 2 is lowered to 40%.

Calls 3 and 4 report usages that are consistent with the previous allocations. Therefore, the resource manager computes the new allocations by offsetting the usages by a small amount in hope of ultimately equalizing the allocations for each service across all nodes.

Call 5, however, is similar to call 2 and reports node 1 to be no longer a resource sink with respect to service 1. The new allocations are similar to those for call 2. Call 6 again is similar to call 3.

From call 6 onwards, all nodes are reported to be resource sinks with respect to each service. The results are shown for call 27 – the allocations for each service have nearly equalized across all the nodes.

4. PROTOTYPE IMPLEMENTATION

In this section, we briefly describe our prototype implementation of cluster reserves. Experimental results obtained with this prototype are presented in Section 5.

The prototype cluster nodes consist of 300MHz Pentium II machines configured with 128 MB of RAM and run the FreeBSD-2.2.6 operating system. For achieving performance isolation on individual cluster nodes, we implemented resource containers [7] on this platform. The lottery scheduling [26] policy was employed for implementing proportional share CPU scheduling amongst resource containers. Lottery scheduling manages resources using tickets and currencies where each principal gets resources proportional to the number of tickets it possesses. Thus, to allocate a fixed percentage of a machine's CPU to a resource container, a proportional number of tickets are associated with it.

Cluster reserves are implemented by a resource manager that runs as a user process on a separate cluster node. For the purpose of communicating with the resource manager, each node in the cluster runs a user process called *tracker* that is capable of (1) collecting usage statistics from the kernel and sending them to the resource manager, (2) setting resource allocations once they are computed by the resource manager, and (3) sending requests to the resource manager for recomputing resource allocations. The resource manager communicates with the tracker on each back-end node through a persistent TCP connection.

In the event of a failure of the cluster resource manager, the cluster continues to serve requests, albeit with potentially suboptimal resource allocations (graceful degradation). As soon as a new resource manager is started, the resource allocations will be adjusted.

For our experiments, two different cluster configurations were employed. Figure 2 shows the first configuration where the cluster consists of a front-end machine that receives client

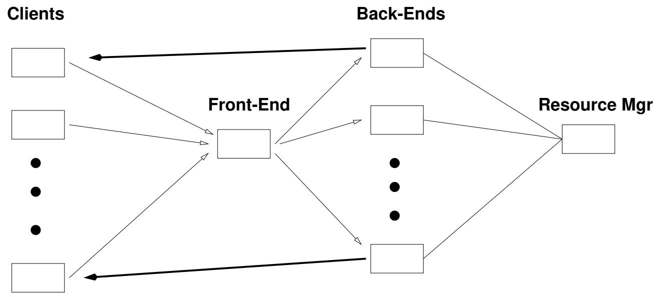

Figure 2: LAN configuration

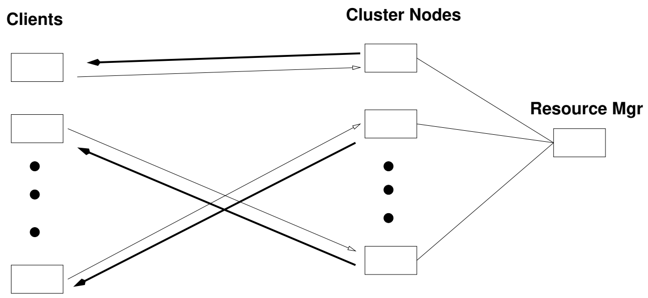

Figure 3: Geographically Distributed Nodes

requests and distributes them to the back-end nodes in the cluster. The second configuration shown in Figure 3 emulates a geographically distributed cluster where clients directly send requests to specific nodes.

The requests were generated by a client program based on the S-client architecture [6]. The program generates HTTP requests as fast as the Web server can handle them. Seven 166 MHz Pentium Pro machines were used as client machines. The client machines and all cluster nodes are connected via switched 100Mbps Ethernet. The Apache-1.3.9 [2] Web server was used at the server nodes.

As mentioned in Section 3, we used the freely available LOQO tool for solving the constrained optimization problem in the resource manager.

	Time (ms)	
	4 nodes 5 services	32 nodes 8 services
300 MHz PII	160	510
500 MHz PIII	85	290

Table 2: Computation time

Table 2 shows the computation time on some platforms for typical problems. For all the experimental results reported in Section 5, a 300 MHz Pentium II machine was used to run the resource manager.

5. RESOURCE MANAGEMENT IN CLUSTERS

In this section, we use a set of benchmarking scenarios to (1) demonstrate the need for performance isolation in Web servers, (2) demonstrate cases where cluster-wide resource management (as opposed to per-node resource management) is needed to achieve effective performance isolation, and (3) show that cluster reserves are an effective solution for providing performance isolation in cluster-based Web servers that run multiple service classes on a common set of cluster nodes.

5.1 Performance isolation via node separation

As mentioned earlier, the inadequate resource management facilities available in general-purpose operating systems render performance isolation between different service classes ineffective, even in single node Web servers. For this reason, state-of-the-art servers that host multiple service classes on individual nodes are not capable of affording performance isolation between the service classes.

For example, Internet Service Providers (ISPs) tend to host small Webs from different organizations on a common hardware platform. In the absence of performance isolation between service classes (i.e., requests for content from different organizations in this example) higher request loads for any one organization's web pages can unfairly steal server resources paid for by other organizations. That is, high load for one organization's Web can cause high latency for other customer's Webs.

Many state-of-the-art cluster-based servers achieve performance isolation by reserving a disjoint subset of the cluster nodes for each service class. While this approach is capable of achieving performance isolation, it can result in poor resource utilization and lower performance. First, every service requires a set of distinct cluster nodes specifically reserved for the service and enough hardware must be provided to cover the peak load expected for the service class. This can increase the capital and maintenance costs of the cluster. Second, this approach does not permit resource sharing—requests for an overloaded service cannot utilize idle cluster resources reserved for other service classes. For example, the nodes dedicated for running a database server might be idle even though the nodes serving static content might be fully saturated at a given time.

Our first experiment evaluates the performance advantages of using a shared cluster as opposed to using disjoint cluster nodes for different service classes. The setup consists of a front-end node that distributes incoming requests to four back-end nodes. Three front-end request distribution strategies are considered:

1. The front-end assigns requests for any resource class to a specific back-end node that is reserved for serving requests only for this class.
2. The WRR strategy (see Appendix A.1) is used for request distribution and requests for a service can be given to any cluster node. The strategy is unaware of service classes and resource management.
3. The LARD [20, 4] strategy (see Appendix A.2) is used for request distribution and requests for a service can be given to any cluster node. The strategy is unaware of service classes and resource management.

A trace obtained by merging logs of four different Rice University departmental Web servers was used to generate requests for the cluster. This trace spans a two-month period and its dataset consists of about 31000 different documents covering 1.015 GB of space. Our results show that this trace needs 526/619/ 745 MB of memory cache to cover 97/98/99% of all requests, respectively.

Four service classes were hosted on the cluster. Each service class provides resources for requests from one of the four original Web server logs. With shared use of the cluster (i.e., with WRR and LARD strategies), cluster reserves were employed and each service class was allotted a cluster-wide allocation of 25%.

	Disjoint	Shared	
		WRR	LARD
Xput (conn/s)	252 (1.0)	517 (2.0)	1214 (4.8)
CPU util. (%)	15	35	60

Table 3: Disjoint vs shared cluster use

Table 3 shows the results for the three different front-end request distribution strategies employed. The results show that disjoint use of the cluster nodes results in relatively low performance because of load imbalance when requests for some services exceed those of others. Substantially higher CPU utilization and throughput is achieved when the service classes are allowed to share the cluster nodes (WRR and LARD). The use of LARD yields an additional improvement in utilization and throughput, because it also aggregates the total cluster memory available for caching the documents [20, 4].

In summary, clusters that permit different service classes to share cluster nodes running general-purpose operating systems are incapable of achieving performance isolation. On the other hand, statically dedicating distinct sets of server nodes for different services can result in low utilization of server resources and lower throughput.

5.2 Performance isolation via per-node resource allocations

We next consider a scenario where static per-node resource assignments are used. This can be accomplished with an operating system mechanism for fine-grained resource management, like resource containers.

Resource containers afford performance isolation between service classes hosted on a single cluster node. We will show that statically assigning equal resources to each cluster node for every service class is sufficient for achieving performance isolation between services on a cluster-wide basis if, and only if, requests for all service classes are load balanced across the cluster nodes. More generally, static resource assignments to services on each node is sufficient for cluster-wide performance isolation if the load for each service class is always perfectly distributed in proportion to the resources assigned to it on each cluster node.

Our next experiment emulates conditions where the request load is balanced across all the cluster nodes for all service classes. For this purpose, we use a single front-end node in the cluster that employs the WRR request distribution strategy. The WRR strategy statistically balances the load for each service class on every cluster node. The cluster hosts five service classes configured on four back-end nodes. Each service is assigned a CPU allocation of 20% on each of the four back-end nodes. Seven client machines issue requests to the front-end node from a synthetic trace that contains a repeated set of five requests, one for each service class. The content size requested by each request is 6 KBytes. This value was chosen because the typical size of HTTP transfers has been observed to range between 5–13 KBytes [3, 18].

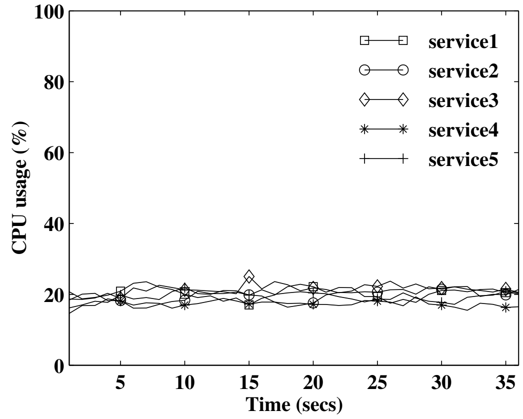

Figure 4: Typical node usage

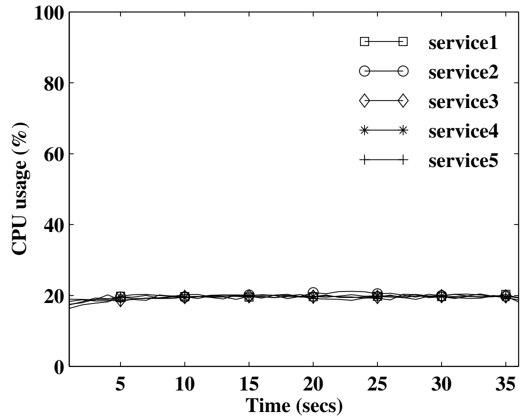

Figure 5: Cluster-wide usage

Figure 4 shows the resource usage for each of the five services on a typical back-end node. Figure 5 shows the cluster-wide resource usage for each service class. The results show that with a balanced request load, static allocation of per-node resources is sufficient to ensure cluster-wide performance isolation.

Unfortunately, a static assignment of per-node resources is not sufficient in situations where perfect load balance among each server node cannot be achieved for each service class. In the following subsections, we cover different scenarios where perfect load balance is not attainable and, therefore, cluster-wide resource management is needed.

5.3 Geographically distributed clusters

One case where it is not generally possible to balance request loads for each service class arises when the cluster nodes are geographically distributed. To emulate this scenario, we used a configuration consisting of four cluster nodes hosting five service classes. Only the first two nodes receive requests for the first service class, while all nodes receive requests for all other service classes. The CPU allocation for each service class at any cluster node is set to 20%.

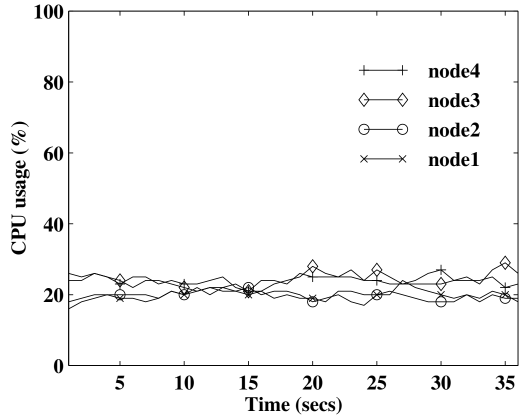

Figure 6: Typical service usage

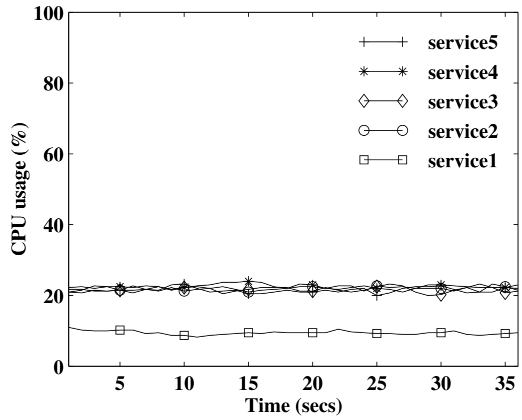

Figure 7: Cluster-wide usage

Figure 6 shows the resource usage of service class 2 on all the four nodes. The resource utilization on nodes 3 and 4 is more than that on nodes 1 and 2 because this service class is able to *steal* the un-utilized CPU allocated to service class 1. Figure 7 shows the cluster-wide resource usage for all the services classes. While service class 1 only gets 10% of the cluster CPU, all other service classes are able to get nearly 23% by utilizing unused cycles from service class 1 on nodes 3 and 4.

We next repeated the above experiment with cluster reserves for attaining performance isolation. To observe the effect the resource manager has on resource utilization, it was started after eight seconds from the beginning of the experiment. Further, the resource manager was made to recompute the resource allocations on the cluster nodes every five seconds. An alternative strategy of recomputing resource allocations upon demand is considered in experiments in the following subsections.

Figure 8 again shows the resource usage of service class 2. As can be observed, the resource manager decreases the CPU allocation on nodes 1 and 2 and increases it on nodes 3 and 4 so as to allow service class 1 to effectively utilize its cluster-wide resources. Figure 9 shows that after the resource manager is started, service class 1 is able to utilize its cluster-wide share of CPU allocation (20%).

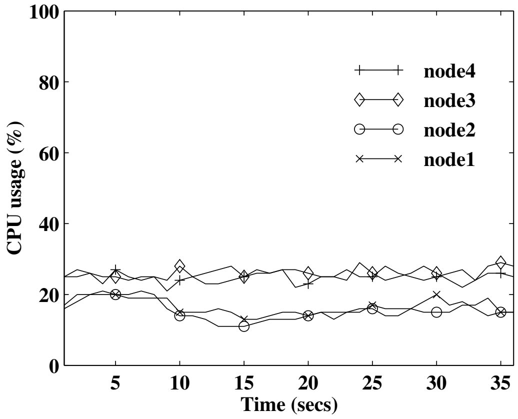

Figure 8: Typical service usage

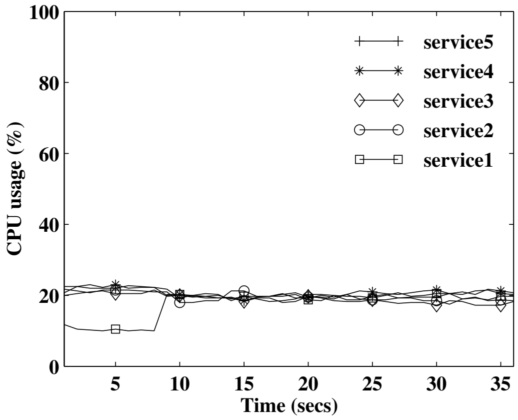

Figure 9: Cluster-wide usage

This experiment demonstrates that geographically distributed clusters require dynamic, global resource allocation for performance isolation and that static per-node allocation is insufficient in achieving effective performance isolation. The results also indicate that cluster reserves are effective in providing performance isolation and can redistribute resources to meet the desired allocations within 1 second.

5.4 Sparse, resource intensive requests

Our next experiment emulates a situation where a service class hosted on the cluster has a sparse incidence of compute intensive requests. Examples of services with this behavior include document translation services and rendering of maps in services that provide driving directions. These services are demanding of the resource manager, because the sparseness of requests prevents load balancing among nodes, and minimizing response time requires that reserved cluster resources are shifted quickly to a node that receives a request.

The experimental setup consists of five cluster nodes in a LAN environment. One of the nodes acts as a front-end, distributing requests to the four other back-end nodes. The request strategy employed at the front-end is WRR. Five services are hosted on the cluster and the desired resource allocation for each of them is 20%.

All requests for service class 1 access a CGI script that runs for 10 seconds before returning a 140 byte result to the client. Requests for all other service classes access a 6 KByte static document and are balanced across the cluster by the WRR policy employed at the front-end[3]. At any time, there is only one outstanding request for service class 1 and a new request is initiated as soon as the previous one finishes.

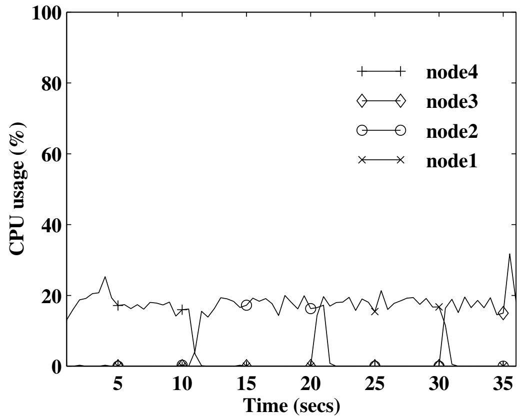

Figure 10: Usage for service 1

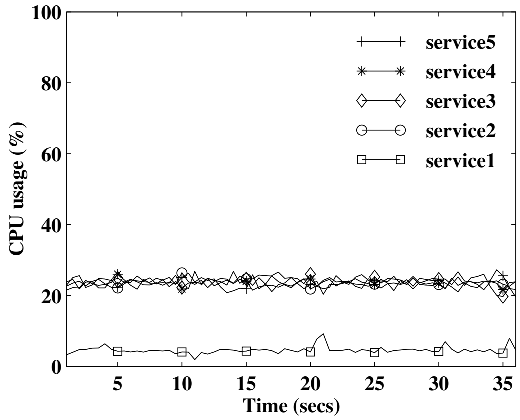

Figure 11: Cluster-wide usage

Figure 11 shows the cluster-wide resource usage of the five service classes with a static per-node resource allocation of 20% to each service class. Service class 1 gets only 5% of

[3] A small modification was made to the WRR policy that ensured that every request for service class 1 is assigned to a different node than the last one.

the cluster resources while all other service classes get nearly 24%. This is because the CGI script is only able to use the fractional capacity allocated to its service class on the node that serves the request; it cannot utilize the capacity allocated to its service class at other nodes, which is then used by other service classes. This is depicted in Figure 10.

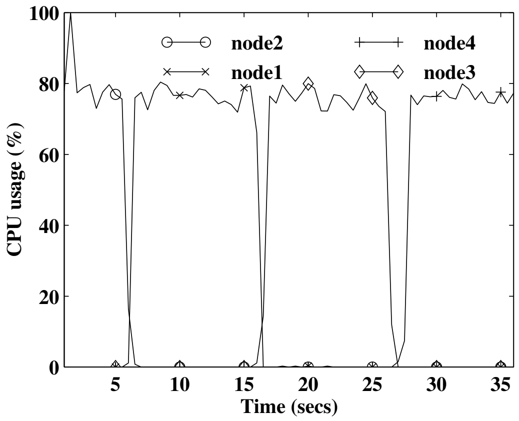

Figure 12: Usage for service 1

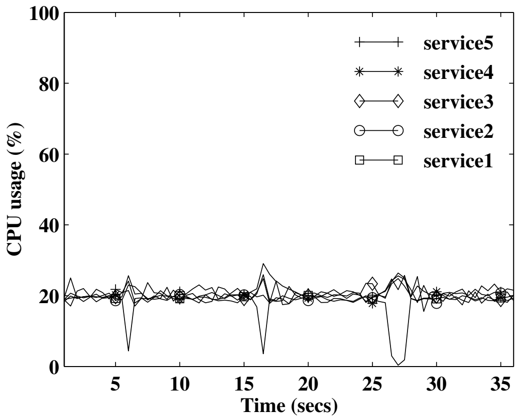

Figure 13: Cluster-wide usage

We next repeated the experiment with cluster reserves. Resource allocation decisions by the resource manager are made on an on-demand basis. A back-end requests a reallocation of cluster resources when (1) the node ceases to be a sink for any service, or conversely, (2) the CPU usage of a service falls significantly (more than 20% of its last measured usage) on that node.

Figure 12 shows that the resource manager is capable of dynamically assigning resources to service class 1 such that its cluster-wide usage meets its allocation. The dips in the graph correspond to the intervals between the instant when a CGI request finishes on one node, and the time when the resource manager reallocates resources to the node that just received the next CGI request.

Our results show that the time for re-allocation of resources ranges from 300ms to 1 second. This variation is the result of three factors: (1) the usage statistics at each node are computed every 250ms as a weighted mean of both the past and present usage; (2) the computation time taken by the resource manager ranges from 100–200ms; and, (3) new requests for resource allocation are sent to the resource manager only after the usage transients resulting from the last reallocation have stabilized (i.e., after 250ms).

Figure 13 shows the cluster-wide usage of all service classes with cluster reserves. The results show that resource usage corresponds to the allocation, except for the short intervals when a new request arrives at a different node and resources need to be reallocated.

5.5 Content-based request distribution

Our final experiment demonstrates cluster resource management with content-based request distribution schemes. The experimental setup consists of a front-end node distributing requests to four back-end nodes. The LARD [4, 20] request distribution strategy was employed at the front-end to distribute requests. The cluster hosts three service classes, each of which are allocated 33% of the cluster's CPU resources. In order to realistically reflect the resource usage across the cluster nodes with content-based request distribution, the requests for each service classes are played from actual web logs from three different departmental web servers. The logs corresponding to service classes 1, 2 and 3 consist of datasets of 358 MB, 24 MB and 193 MB respectively and need 248 MB, 16 MB and 67 MB respectively of memory cache to satisfy 98% of their requests from main memory.

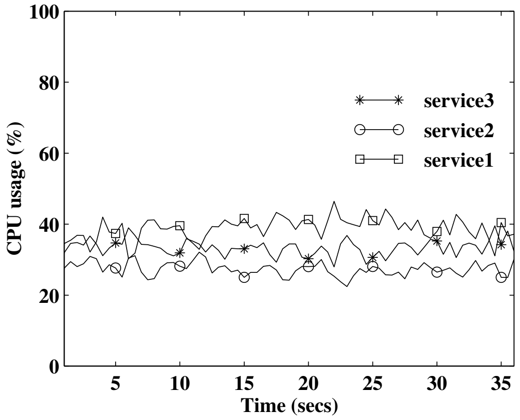

Figure 14: No Cluster Reserves

Figure 14 shows the cluster wide CPU usage of the three service classes when cluster reserves are not used. The results show that service class 1 is able to obtain more than its fair share (33%) of cluster CPU time while service class 2 gets less than its share. The high variation in the usage is mainly due to disk activity that is needed when requested documents are not found in the main memory cache. Figure 15 shows the results when cluster reserves are being used. With cluster reserves, all service classes achieve their allocated resource usage.

6. RELATED WORK

Network servers based on clusters of workstations are now widely used [14]. Several commercial products are available for use as front-end nodes in cluster servers [11, 16].

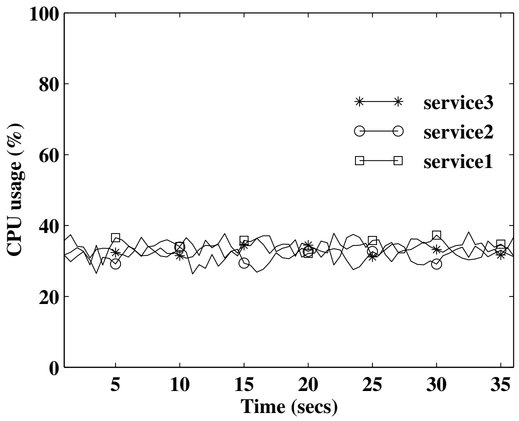

Figure 15: With Cluster Reserves

Most request distribution strategies employed are variations of WRR (described in Appendix A.1). The LARD [20, 4, 5] strategy is based on content-based request distribution and affords scalable performance by simultaneously achieving better cache locality and load balance. The Dispatch product by Resonate, Inc. supports a strategy that is also based on content-based request distribution [21]. Loosely-coupled distributed servers are also widely deployed in the Internet and use various techniques for load balancing including DNS round-robin [8], HTTP client re-direction [1], Smart clients [27], source-based forwarding [12] and hardware translation of network addresses [11]. Load balancing strategies that seek to satisfy response time based performance criteria are discussed in [22] and the citations therein. Our work focuses on resource management in cluster-based servers and our methodology can be applied independently of the request distribution scheme employed in the cluster.

Fox et al. [14] 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, which instead focuses on the resource management aspects.

Banga et al. [7] proposed the resource container abstraction that separates the notion of a resource principal from threads or processes and provides support for fine-grained resource management in the operating system. Coupled with LRP [13], resource containers are capable of affording performance isolation on a single node. Other related abstractions are Activities [17] in the Rialto real-time operating system, software performance units (SPU) [25] proposed in the context of shared-memory multiprocessors, reservation domains in the Eclipse operating system [10, 9] and paths in Scout [23].

Our proposed cluster reserves are cluster-wide resource principals that extend performance isolation on individual cluster nodes into performance isolation for the cluster. Our implementation uses resource containers to achieve performance isolation on individual cluster nodes. However, in principal any other abstraction capable of affording performance isolation on a single node can be used.

Various scheduling algorithms for achieving fixed share resource allocation have been discussed in past literature. Lottery scheduling [26] proposes to manage resources using tickets and currencies where each service class gets resources proportional to the number of tickets it possesses. Start time fair queuing [15] applies the weighted fair queuing algorithm developed for network switches to CPU scheduling for achieving fixed CPU allocations. The SMART [19] multimedia scheduler integrates priorities and weighted fair queuing to meet real-time constraints while simultaneously supporting non real-time applications. Our resource container implementation employs the Lottery scheduling algorithm to achieve proportional share CPU allocation.

7. CONCLUSIONS

This paper presents and evaluates *cluster reserves*, a resource management facility for cluster based Web servers that affords effective performance isolation in the presence of multiple Web services that share a server cluster. Cluster reserves extend existing mechanisms for performance isolation in single-node servers to a cluster environment.

Using a set of benchmark scenarios and both synthetic and trace based workloads, we evaluate a prototype implementation of cluster reserves. The results show that hosting multiple Web services on a joint set of cluster nodes can result in higher resource utilization and improved performance than the state-of-the-art approach of hosting different services on dedicated server nodes. The results also demonstrate that cluster-wide (as opposed to per-node) resource management is needed to achieve effective performance isolation among services that share a set of clusters.

Finally, our results show that cluster reserves are an effective solution for providing performance isolation in cluster-based Web servers. Cluster reserves afford performance isolation among multiple service classes, which can be defined based on the requested content, the client who issues the request, or both.

While we evaluate our implementation for the CPU time resource, cluster reserves can be extended to also provide performance isolation for other operating system resources like memory, disk and network bandwidth etc.

8. 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] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-based Web Servers. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[5] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[6] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 1999.

[7] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

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

[9] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[10] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proceedings of the USENIX 1998 Annual Technical Conference*, Berkeley, CA, June 1998.

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

[12] 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.

[13] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[14] 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.

[15] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating System. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

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

[17] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the Rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[18] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proceedings of the ACM SIGCOMM '95 Symposium*, 1995.

[19] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, New York, Oct. 1997.

[20] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

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

[22] J. Sethuraman and M. S. Squillante. Optimal stochastic scheduling in multiclass parallel queues. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Atlanta, GA, May 1999.

[23] O. Spatscheck and L. L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[24] R. Vanderbei. LOQO: An interior point code for quadratic programming. *Optimization Methods and Software*, 1999.

[25] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[26] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[27] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, Berkeley, CA, Jan. 1997.

APPENDIX

A. REQUEST DISTRIBUTION STRATEGIES

This section briefly describes the weighted round-robin (WRR) and the locality-aware request distribution (LARD) strategies employed by a front-end node to distribute requests to back-end nodes in a cluster.

$$cost\_balancing(target, server) = \begin{cases} 0 & Load(server) < L_{idle} \\ Infinity & Load(server) > L_{overload} \\ Load(server) - L_{idle} & otherwise \end{cases}$$

$$cost\_locality(target, server) = \begin{cases} 1 & target\ is\ mapped\ to\ server \\ Miss\ Cost & otherwise \end{cases}$$

$$cost\_replacement(target, server) = \begin{cases} 0 & Load(server) < L_{idle} \\ 0 & target\ is\ mapped\ to\ server \\ Miss\ Cost & otherwise \end{cases}$$

Figure 16: LARD Cost Metrics

A.1 The WRR strategy

The WRR request distribution strategy aims to efficiently utilize cluster resources by balancing the load across all the back-end cluster nodes. The pseudo-code for this strategy is given below:

```
while (true) {
  fetch next request r;
  chosen_backend ← (chosen_backend+1) mod num_backends;
  min_load ← Load(chosen_backend);
  cmp_backend ← (chosen_backend+1) mod num_backends;
  for (i←1; i < num_backends ;i++) {
    if (min_load > Load(cmp_backend)) {
      chosen_backend ← cmp_backend;
      min_load ← Load(chosen_backend);
    }
    cmp_backend ← (cmp_backend + 1) mod num_backends;
  }
  send r to chosen_backend;
}
```

A.2 The LARD strategy

The LARD [20, 4] strategy yields scalable performance by achieving both load balancing and cache locality at the back-end servers. For the purpose of achieving cache locality, LARD maintains mappings between targets and back-end nodes, such that a target is considered to be cached on its associated back-end nodes. To achieve a balance between load distribution and locality, LARD uses three cost metrics: *cost_balancing*, *cost_locality* and *cost_replacement*.

The unit of cost (and also of load) is defined to be the delay experienced by a request for a cached target at an otherwise unloaded server. The load point L_{idle} defines a value below which a back-end node is potentially underutilized. $L_{overload}$ is defined such that the difference in delay between a back-end node operating at or above this load, compared to a back-end node operating at the point L_{idle}, becomes unacceptable.

The aggregate cost for sending the request to a particular server is defined as the sum of the values returned by the above three cost metrics. When a request arrives at the front-end, the LARD policy assigns the request to the back-end node that yields the minimum aggregate cost among all nodes, and updates the mappings to reflect that the requested target will be cached at that back-end node.