

A Hybrid Access Model for Storage Area Networks

Aameek Singh^{*†} Kaladhar Voruganti[‡] Sandeep Gopisetty[‡] David Pease[‡] Ling Liu[†]

Abstract

We present HSAN - a hybrid storage area network, which uses both in-band (like NFS [9]) and out-of-band virtualization (like SAN FS [8]) access models. Using hybrid servers that can serve as both metadata and NAS servers, HSAN intelligently decides the access model per *each request*, based on the characteristics of requested data. This hybrid model is implemented using low overhead cache-admission and cache-replacement schemes and aims to improve overall response times for a *wide variety* of workloads. Preliminary analysis of the hybrid model indicates performance improvements over both models.

1 Introduction

Currently, there are two prevalent access models for Storage Area Networks (SANs). In an out-of-band virtualization model (henceforth called *direct* access model), clients access file *metadata* from dedicated metadata servers (MDS) and access data, directly from the storage controllers. This is in contrast to an in-band NAS access model, in which clients access data through an intermediate NAS server. Both access models have their advantages and disadvantages. We compare them on a few characteristics below:

1. Scalability: Direct access model is more scalable than the NAS model. Even with centralized MDS, the direct access model can serve a greater number of clients, since the hosts access data directly, without an intermediate server in the data path. The metadata transactions are much shorter and metadata caching at the client further improves scalability.

2. Caching: The NAS model of access provides a great opportunity to exploit access locality across multiple clients. The NAS servers typically maintain caches of objects being actively accessed (hot objects). This additional caching layer between the storage controller cache and client cache can significantly reduce response times. In a direct access model, only caches available are client cache and the storage controller cache. While it can be argued that storage controller cache can suffice for hot objects, for workloads where a single controller is being hit for many hot objects, the controller cache might end up swapping out the desired objects. In addition, it causes additional load on the controllers. The client cache fails to exploit similar accesses by different clients.

3. Workload Specific: It can be easily seen that both

of these models are better suited for certain kinds of workloads. For example, for accessing files with heavy read sharing across multiple clients, NAS model will perform better. Also, for small sized files the costs of establishing two connections (to metadata server and storage) makes the direct access model relatively expensive. On the other hand, for large files with little sharing across multiple clients, the direct access model is faster because of zero hops during data access.

4. Infrastructure: Currently, most SANs are based on Fibre Channel infrastructure, which is much more expensive than IP. However, with the advent of iSCSI, this cost difference is no longer an issue.

As can be seen, both of these models have their strengths and weaknesses. In this paper, we provide a hybrid framework called HSAN, that can provide the benefits of both approaches and thus, offer a single better solution for a variety of workloads. The aim of HSAN is to reduce client response times by selecting an access model most appropriate for the desired objects. The unique characteristic of the design is to allow the granularity of a single data request, or in other words the decision to choose a particular mode of access is made for each request. We implement this intelligence through cache-admission and cache-replacement policies at the hybrid servers. These policies are very low-overhead and utilize information already existing at the MDS or NAS servers. In this paper, we will discuss centralized direct access models (with dedicated MDS), though it will be easy to extend the proposed solutions to decentralized approaches [1].

It is also important to note that our solution requires hybrid servers that can act as both metadata or NAS servers. This will either require MDS to understand the NAS protocols (initiatives like Parallel NFS [4]) or NAS servers to be able to serve only metadata as well. We believe this to be a small change for the prospective benefits. The rest of the paper is organized as follows. We describe our proposed design in Section-2 and provide a brief preliminary analysis in Section-3. In Section-4, we discuss the related work, including caching work in other related areas like databases. We finally conclude in Section-5 with a note on future work.

2 Design

While it is possible to approach the design from both NAS as well as direct access models, we choose to describe it through additions to the direct access design. This choice gives us underlying infrastructure support of host connectivity to storage. To put it differently, we describe HSAN by adding caching and *data* access at MDS of the direct access

^{*}Work done while visiting IBM Almaden

[†]Georgia Tech, {aameek, lingliu}@cc.gatech.edu

[‡]IBM Almaden Research Center, kaladhar@us.ibm.com, {sandeep, pease}@almaden.ibm.com

design*

The primary motivation of HSAN is to reduce the response times seen by the clients. Using our design, the MDS will also maintain a data cache and in case of a cache hit, the MDS immediately sends the data in response to the metadata request, thus, saving the costs and delays of (a) parsing the metadata at the client, (b) connecting to appropriate storage controllers and (c) retrieving the data. This modified MDS is called the *hybrid server (HS)*. In case of a cache miss, the only penalty is to check for the existence of the data object in the cache, which is an $O(1)$ operation under our proposed scheme. In addition, the maintenance of statistics required for making a decision on the mode of access is cheap and does not add considerable overheads. Note that once the client has acquired metadata for a desired object, all subsequent accesses (for example, using cached metadata) are made directly to the storage controller and thus, the direct access model is used. As we discuss later, this condition can potentially be relaxed to save I/O operations.

As mentioned earlier, we implement our hybrid scheme to decide the model of access through cache-admission and cache-replacement policies. *Cache admission* policy determines the conditions necessary for an object to fulfill, before it can be admitted into a cache. *Cache replacement* policies determine the selection of the object (typically called the *victim* object) which is replaced when more space is required for an incoming object. Our scheme works as follows. We modify the metadata information for each object to also contain a pointer to the data object in the cache. In case the data is not in the cache, the pointer is set to null. When a client requests for metadata of a certain object, the HS first checks if the requested data object is in the cache. In case of the cache hit, the data is directly served to the client. In case of a miss, the HS checks if the requested object can pass the cache-admission test and can replace an existing data object in the cache (or there already exists enough space in the cache). If yes, it is accessed via the NAS model - HS making the access to storage and forwarding data to the client. If it fails on the cache admission test or cannot replace any existing cached object based on the cache replacement policy, it is accessed using the direct access model, with the HS only providing the metadata to the client and the client making an access to the storage. Thus using such a scheme, the problem of dynamically deciding the access model reduces to the design of cache admission and cache replacement policies. The workflow is presented in Figure-1.

2.1 Decision Factors

The main challenge in the design of such policies is identifying the factors that should influence the decision of whether an object should be cached or not. In addition, the evaluation of these factors should not be expensive, to prevent the caching overheads from becoming prohibitive. For our storage scenario, we believe the following factors to be critical in such decision-making:

- **Rate of Access (λ):** If an object is accessed more frequently, there is greater incentive of keeping the object

*For the other direction, we require host to storage connectivity and NAS servers to be able to serve only metadata as well

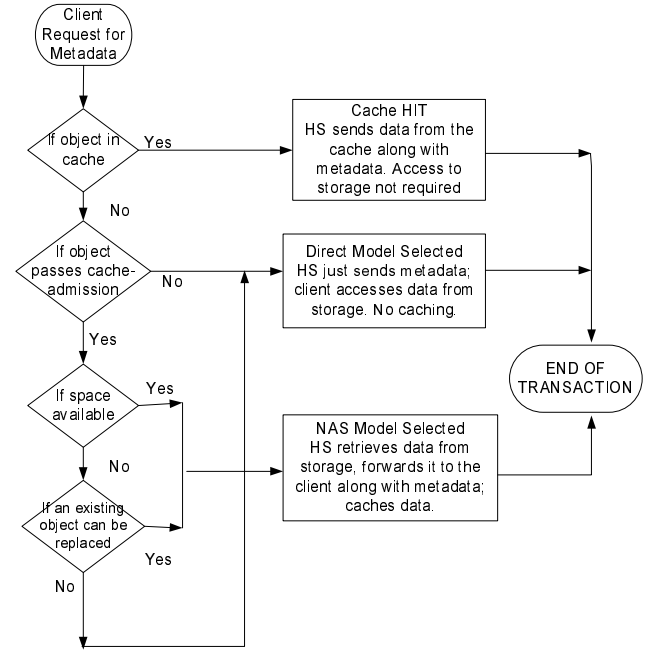


Figure 1: Workflow

in the cache. Frequent accesses will result in more cache hits and improved response times. However, as we describe later, this metric needs to be measured in conjunction with the data sharing and locking mechanisms, in order to obtain a good estimate for our caching policy.

- **Cost of Obtaining Object (c):** An object that is expensive to obtain from storage (for example, because of being on a slower or a heavily loaded storage controller) has a greater incentive to be cached. In case of cache hits for such objects, the response time improvement will be significant.

- **Size of the Object (s):** The size of objects also plays a critical component. An object that is smaller in size is more valuable to be cached, since it takes lesser cache space and also, in case of a cache hit, provides maximum response time improvement ratios compared to direct access models.

- **Load on HS:** Since a HS is also used for metadata transactions, it is important to prevent queuing delays at the HS due to various I/O operations. Thus, a heavily loaded HS should perform less I/O operations (promote direct access).

The first three parameters are used to define a *value* for each data object. A greater *value* indicates a greater incentive to cache the object (which in turn means, to access it via the NAS model).

$$Value(O_i) = \lambda_i * c_i / s_i^\alpha, \text{ where } \alpha \geq 1$$

The fourth parameter (load on HS) can be used to set up the admission threshold described later.

The *Value* metric will favor objects that are (a) accessed more frequently, (b) expensive to obtain in case of cache-miss and (c) smaller in size. The parameter α can be set depending upon the amount of favor desired for smaller objects (especially in order to favor metadata at the HS). We discuss this issue of interaction between data and metadata objects later. It is important to note that the rate of access is the rate of requests for the metadata at the HS and not the rate of access at the storage. This is because objects, which are being

accessed directly from storage (using cached metadata), have lesser incentives to be cached at the HS, as they are already being accessed using the fast direct access model.

2.2 Parameter Evaluation

As mentioned earlier, the evaluation of these parameters needs to be a low overhead operation. To achieve this goal, we use the following way of computing each:

– s_i : Size of the object is already available at the MDS and thus, is no added overhead.

– λ_i : We evaluate the rate of access as a moving average of the last K inter-arrival times of request to O_i , where K is a parameter determining the amount of history to be considered, (typically set to 10). Precisely, λ_i is defined as $\lambda_i = \frac{K}{t-t_K}$, where t is the current time and t_K is the time of the last K^{th} reference to O_i . Since all metadata requests come at the HS, this parameter can also be efficiently computed.

– c_i : We measure the cost of obtaining the data from the storage controller by means of average access times. Since in the direct access model, the clients directly access storage, this parameter has to be obtained from the clients. We achieve this in the following manner. For every object accessed from the storage controller, the client maintains its average access time and whenever it requests a lock or metadata for that object from the HS, it shares this statistic with the MDS. The MDS averages this access time across all clients[†].

2.2.1 Data Sharing

The calculation scheme described till now does not take into account the kind of locks held on the data objects. The locking mechanisms can have potential impact on object values. For example, consider a case, when an object is accessed very frequently, but always in an exclusive mode. All the attempts to access the object while it is being held in an exclusive lock will not be satisfied. Caching such an object has no incentive due to the inability to serve other clients with that object. Therefore, we modify the λ evaluation in the following manner. All access attempts that *would not* have been satisfied even if the object was in the cache (e.g. one client holds an exclusive lock) are not counted towards its rate of access. This scheme automatically prefers object which are more read-shared and thus, provide the maximum benefits of caching.

2.3 Cache Admission-Control

As mentioned before, an object is considered for caching (and thus, accessing via NAS model) only if it passes the cache admission test (CAT). The motivation for having such a test is to ensure that the object is *valued* enough to dedicate HS resources for I/O operations. One simple policy is to admit an object whose value is greater than the minimum value of the cached objects.

$$\text{CAT: } Value(O_{incoming}) > \min(Value(O_i))$$

However, this policy is insufficient. For example, consider the workload scenario in which only a few objects in the cache are being accessed frequently and the rest of the

[†]It is possible to use client-specific parameters instead of averaging across all clients. For example, the value of O_i can be defined as: $\sum_j \lambda_{ij} * c_{ij} / s_i^\alpha$ where λ_{ij} and c_{ij} are parameters for Client-j. In this case, λ can also be computed by the clients

objects are rarely being accessed, though not being replaced because of low or no contention of cache space. Thus, the minimum value of the objects in the cache will decrease with time and can potentially be a very small number. In such a scenario, we prefer to avoid bringing in new objects with low values (less workload for HS). We modify the policy as follows:

$$\text{CAT: } Value(O_{incoming}) > \max(\pi, \min(Value(O_i)))$$

where π is a threshold parameter, which dynamically adjusts based the workload seen so far. This is achieved by setting: $\pi = avg(N) \{ \min(Value(O_i)) \}$ i.e. π is computed periodically as the average minimum value of cached objects over the last N intervals. N determines the amount of history to be taken into account and can be statically set. The computation period can be set in terms of number of transactions at the HS. For example, it is computed after every 1000 metadata transactions.

In addition, π can be extended to incorporate load on HS. For example, for a heavily loaded HS, a factor β can be added to the threshold value, which raises the bar for accessing data using NAS model, reducing further load and queuing delays at HS.

2.4 Cache Replacement Policy

Once an incoming object, $O_{incoming}$, passes the cache-admission test, we try to evaluate if there is enough space in the cache to accommodate the object. In case, there is not enough space, we try to evict existing objects through cache replacement policy. We use the following algorithm:

1. Arrange all cache objects in a list in the increasing value order. Let the sorted list be $\{O_1, O_2, \dots, O_n\}$.
2. Let m be the minimal prefix, such that $size(O_1) + size(O_2) + \dots + size(O_m) \geq size(O_{incoming})$
3. If $Value(O_m) \leq Value(O_{incoming})$, then evict O_1, O_2, \dots, O_m else No replacement done and $O_{incoming}$ is not cached.

Step-3 ensures that we do not replace any higher value object at the expense of a lower value object. The *else* clause would in turn mean that the object is accessed via the direct access model. The cache-admission and cache replacement policies can be efficiently implemented by a low overhead priority queue (heap) [3, 5]. This issue will be further discussed in the detailed version of the paper.

2.5 Cache Consistency

For any caching solution, it is important to have efficient mechanisms of maintaining cache consistency. We use the following mechanism for achieving *strong consistency*. Whenever an object is required to be accessed for writes (exclusive mode access), the HS serves the initial metadata request, and then invalidates the object in the cache. This is an efficient mechanism, since an object being held in an exclusive manner cannot be shared at the cache anyway. For later accesses, the object is treated similar to any new object being accessed. This design avoids explicit cache consistency mechanisms, thus, preventing high caching overheads. We are also investigating mechanisms for maintaining weaker forms of consistency, with lazy writes by the HS.

2.6 Memory Model

It has been argued that the MDS are meant to provide only metadata information and are fine tuned for such workload characteristics (large number of small requests). Data caching at MDS competes for main memory with the metadata objects, thus, influencing the core task of the MDS. We propose following three approaches that can counter this and any one of them can be used depending upon the workload characteristics:

1. **Partitioned Memory Model:** In this model, there are statically assigned distinct spaces for metadata caching and data caching with no overlap between the two. This ensures that the data-caching component does not effect the regular metadata caching. This is dependent on the availability of enough memory at the HS. Such a model is best for workloads in which the size of the cached metadata does not fluctuate much.

2. **Shared Memory Model with strict priority to Metadata:** This model uses a shared space between metadata and data. However, metadata objects are given strict priority over data objects. Thus, a data object can never replace a metadata object (similar to setting metadata object values to ∞) and a metadata object always replaces the least valued data objects. Amongst metadata objects, the regular cache replacement policies can be used. This model is best used for workloads in which metadata cache size can vary significantly and metadata performance is critical to the application.

3. **Shared Memory Model with appropriate α :** There can be scenarios, when it is reasonable to swap out metadata objects, which are rarely accessed, in order to cache *valued data objects*. In such a scenario, we can use the shared memory model and appropriately set a value of α in the value function depending upon the priority given to smaller objects. Notice that α adds value to all small objects and not necessarily the metadata objects. If it is not desired, we can modify the value parameter for metadata objects to $(\pi + \lambda_i c_i / s_i^\alpha)$, thus, giving a head start of π , where π is the admission threshold parameter discussed earlier.

3 Analysis

In this section, we present a preliminary analysis of the hybrid model as compared to the NAS and direct access models. As part of our initial analysis, we have used simple models to describe certain empirical behavior. For example, we use a linear model to determine queue delays at servers based on the number of clients they are serving; though we use different parameters for metadata and I/O operations. Thus, it is important to emphasize that the model is not designed to predict accurately the response times, rather to **comparatively** analyze the three models. In the detailed version of the paper, we will include more experimental analysis.

Let γ be the time for sending a data/metadata request. Since the characteristics of metadata response (from MDS/HS) are similar (short messages), we assume the response to the request to be γ as well.

Let τ be the time to send a request to storage and retrieve the data for a particular file. Thus, this will be the time for a host to get data from storage for a direct access model and also for

a NAS (and HS) server to access data from storage in NAS (and hybrid) model.

Let τ' be the time to send a request and obtain the data from server's memory, e.g. from NAS/HS server cache.

Let ψ be the delay at the NAS/MDS/HS server due to contention with other simultaneous accesses. We set ψ to be a linear function of number of connections at the server, though the slope of the linear function is much lower for metadata connections as compared to data connections.

Below we give a short example of how the analysis would work for accessing a single file with space available in the caches. Assume the file is accessed by C clients and each client accesses it N times and only reads are issued. We do not consider client-data-caching since it is the same for all models. However, metadata can be cached in the direct and hybrid model.

For *NAS model*, only the first of the $N * C$ accesses results in I/O with the storage. All subsequent accesses are served from the cache of the NAS server. Therefore, total response time for NAS model is:

$$(\gamma + \tau + \psi_{NAS} + \tau') + (N * C - 1)(\gamma + \psi_{NAS} + \tau')$$

The first term includes - data request + fetching data from storage + delay at NAS server + forwarding content to client (once fetched, it is served from the memory) - for the first request to that data object. The second term includes for the subsequent $N * C - 1$ requests - data request + delay + serving from cache. So total response time is given by:

$$RT_{NAS} = \tau + NC(\gamma + \tau' + \psi_{NAS})$$

For *direct access* model, the first access of each client results in metadata request (γ), metadata response (γ) and I/O fetch (τ). Every subsequent access from each client only requires the I/O fetch (τ) because metadata is cached - again considering no data caching at client or controllers (which is the same scenario for NAS model). Thus, the total response time is given by:

$$C * (\gamma + \gamma + \psi_{Dir} + \tau) + (N - 1)C(\tau)$$

$$RT_{Dir} = C(2\gamma + \psi_{Dir} + N\tau)$$

The hybrid model would determine the value of the object, and if it considered valued enough, it will be cached at the HS. The HS would also return the metadata to the client which is cached at the clients and used for subsequent accesses. Also ψ_{Hyb} is the sum of delays due to concurrent metadata and data transactions. The response time would be given by[‡]:

$$(\gamma + \tau + \psi_{Hyb} + \tau') + (C - 1)(\gamma + \psi_{Hyb} + \tau') + (N - 1)C(\tau)$$

$$RT_{Hyb} = C(\gamma + \psi_{Hyb} + \tau') + \tau(1 + (N - 1)C)$$

Even though at first glance the analysis looks in favor of NAS model, it is important to note that ψ_{NAS} is much higher due to I/O costs in that model. In addition the above workload is an all-read workload. However, it also indicates that for a hybrid model, it might not be *always* a good strategy to obtain data from the storage when it is present in the HS cache. We plan to explore this extension in future. We also will consider delays due to implementation of caching policies for all the models in the detailed version of the paper.

3.1 Experimental Results

Using the above model, we evaluated the performance of the NAS, direct access and hybrid models, for a number of work-

[‡]assuming negligible additional cost for including metadata in response

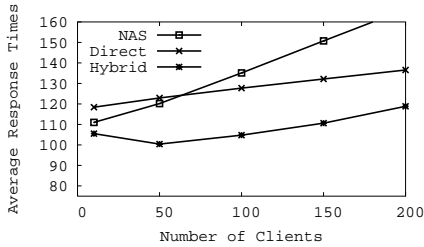


Figure 2: Varying Number of Clients

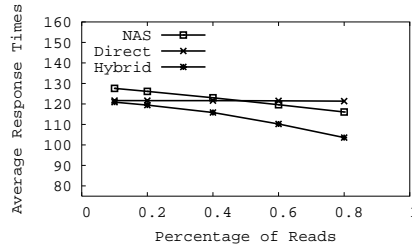


Figure 3: Varying Read-Write Ratio

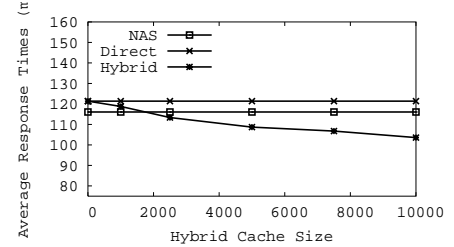


Figure 4: Varying Hybrid Cache Size

load and client load scenarios. We used a 1000 file data with file sizes distributed by a Poisson distribution with mean of 100 KB. Accesses to the files were based on a Zipf’s law with its $\alpha = 0.5$. The τ values were also assumed to be Poisson distributed with mean of 100ms. The NAS cache was implemented as an LRU cache. Also the HS cache was implemented using a partitioned memory model.

Figure-2 plots the average response times for the three models with varying number of clients (R-W ratio of 0.8 and NAS cache size = Hybrid data cache size = 10MB). As can be seen with increasing number of clients, the queuing delays at NAS servers increase significantly and thus, the overall response times increase. For direct access model also, the queuing delays increase though at a much smaller rate (metadata transactions). The hybrid model presents an interesting analysis. For the first jump in the number of clients, the response time decreases slightly. This is due to an initial increase in data locality at hybrid cache across multiple clients. Since there are lesser I/O data transactions at the hybrid server the queuing delays with increasing number of clients are not able to offset this. However, with later transactions, the delays become dominant. Figure-3 plots the models with increasing read percentage in the workload (1000 clients). As expected, both NAS and hybrid models perform better due to the caching effects. Figure-4 plots the models with varying cache size at the hybrid cache with the NAS cache fixed at 10MB. As expected, it performs better with bigger caches.

It is encouraging to note that the hybrid model is able to outperform both the other models due to its ability to make intelligent choices *per-request*. We continue to evaluate the models for other criteria and with real benchmarks.

4 Related Work

Even though there has been work on data management for file systems in storage area networks [2], to the best of our knowledge, there is no prior work describing a SAN with both co-existing in-band and out-of-band access models and ability to switch models per each request. In general, caching has been an important performance enhancing mechanism with application to a wide range of problems. While a number of caching algorithms like LRU, [6, 7] are targeted towards simplicity of implementation, more complex algorithms have also been extensively used in a variety of scenarios - like databases [10], web caching [3, 5, 11] etc. All of these approaches follow a similar principle of cost-aware caching. Our caching policy is similar to [10] algorithm, though we differ in cache admission and the parameter evaluation. Especially, our policy is closely aligned to the data sharing mechanisms, whereas other works do not focus on such environments.

5 Conclusions and Future Work

In this paper, we have presented a hybrid access model for Storage Area Networks. To the best of our knowledge, this is a first attempt at designing SANs that can exploit the strengths of both in-band NAS and out-of-band direct access models through a unified solution. An important characteristic of our design is to intelligently choose between the access models at a per-request granularity using low-overhead cache admission and cache replacement policies. Our initial analysis indicates that the hybrid model outperforms both NAS and direct access models for a variety of workloads. In future, we would like to evaluate the hybrid model on real benchmarks and also design extensions to the model.

References

- [1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. of SOSR*, pages 109–126, 1995.
- [2] R. Burns. Data management in distributed file system for storage area networks. *PhD Thesis, UCSC*, 2000.
- [3] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. USITS*, 1997.
- [4] G. Gibson and P. Corbett. pNFS Problem Statement. *IETF Internet Draft*, 2004.
- [5] S. Jin, A. Bestavros, and A. Iyengar. Accelerating internet streaming media delivery using network-aware partial caching. In *Proc. ICDCS*, page 153, 2002.
- [6] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proc. VLDB*, pages 439–450, 1995.
- [7] N. Megiddo and D. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. FAST*, 2003.
- [8] J. Menon, D. Pease, R. Rees, L. Duvanovich, and B. Hillsberg. IBM Storage Tank - A Heterogeneous Scalable SAN file system. *IBM Systems Journal*, 42(2).
- [9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proc. USENIX*, 1985.
- [10] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In *VLDB Journal*, pages 51–62, 1996.
- [11] A. Singh, A. Trivedi, K. Ramamritham, and P. Shenoy. PTC: Proxies that Transcode and Cache in Heterogeneous Web Client Environments. *World Wide Web Journal*, 7(1), 2004.