# Wide-area Network Acceleration for the Developing World

Sunghwan Ihm[1], KyoungSoo Park[†2], and Vivek S. Pai[1]

[1]*Department of Computer Science, Princeton University*
[2]*Department of Electrical Engineering, KAIST*

## Abstract

Wide-area network (WAN) accelerators operate by compressing redundant network traffic from point-to-point communications, enabling higher effective bandwidth. Unfortunately, while network bandwidth is scarce and expensive in the developing world, current WAN accelerators are designed for enterprise use, and are a poor fit in these environments.

We present Wanax, a WAN accelerator designed for developing-world deployments. It uses a novel multi-resolution chunking (MRC) scheme that provides high compression rates and high disk performance for a variety of content, while using much less memory than existing approaches. Wanax exploits the design of MRC to perform intelligent load shedding to maximize throughput when running on resource-limited shared platforms. Finally, Wanax exploits the mesh network environments being deployed in the developing world, instead of just the star topologies common in enterprise branch offices.

## 1 Introduction

While low-cost laptops may soon improve computer access for the developing world, their widespread deployment will increase the demands on local networking infrastructure. Locally caching static Web content can alleviate some of this demand, but this approach has limits on its effectiveness, especially in smaller environments.

We propose to augment these caches with integrated wide area network (WAN) accelerators that have been specifically designed to operate in developing-world environments. WAN accelerators are deployed near edge routers, and work by compressing redundant traffic destined to locations with other WAN accelerators. To compress traffic, the accelerators break the data stream into smaller chunks, store these chunks at each accelerator, and then replace future instances of this data with reference to the cached chunks. By passing references to the chunks rather than the full data, the accelerator compresses the data stream.

Current WAN accelerators are not well-suited for the developing world. While they typically require server-class machines with a set of fast disks and a large pool of dedicated memory, the average school targeted by the One Laptop Per Child (OLPC) project will have 100 laptops in the price range of US $100-$200 each, for a total cost of $10K-$20K [22]. Requiring special server-class hardware for WAN acceleration alone could increase deployment cost. Other options would be to share the machine with other services (e.g, mail servers, Web servers, and proxies) or to use cheap, laptop-class hardware, both of which would reduce the RAM and disk available to the WAN accelerator. In addition, existing designs cannot exploit the mesh network environments being deployed in the developing world, limiting their potential utility.

We have developed a new WAN accelerator, Wanax, that is designed to meet these challenges in the developing world. Our technical contributions are the followings: (1) a novel multi-resolution chunking (MRC) technique, which provides high compression rates and high disk performance across workloads while having a small memory footprint; (2) an intelligent load shedding technique that exploits MRC to maximize effective bandwidth by adjusting disk and WAN usage as appropriate; and (3) a mesh peering protocol that exploits higher-speed local peers when possible, instead of fetching only over slow WAN links. The combination of these design techniques makes it possible to achieve high effective bandwidth even with resource-limited shared machines.

The rest of this paper is organized as follows: §2 provides background on WAN accelerators and new challenges in the developing world. §3 describes the design of Wanax, and we show the trace-based simulation analysis in §4. In § 5, we detail the prototype implementation, and §6 presents the experimental results. Finally, we discuss related work in §7, and conclude in §8.

## 2 Background and Motivation

Our goal is to improve Internet access in the developing world using WAN accelerators designed to use low-end hardware. We primarily focus on increasing the *effective bandwidth* (or throughput) of the expensive, low-bandwidth WAN link in the region. We first provide a brief introduction to WAN accelerators, and then discuss the specific problems.

### 2.1 WAN Accelerators

**Content Fingerprinting** Content fingerprinting (CF) forms the basis for WAN acceleration, since it provides a position-independent and history-independent technique for breaking a stream of data into smaller pieces, or chunks, based only on their content.

[†]Work partly done while at University of Pittsburgh.

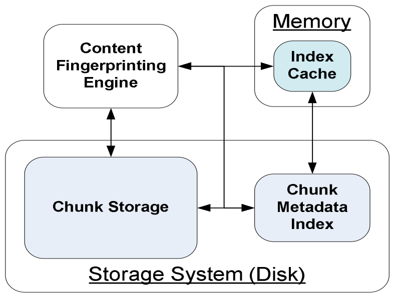


Figure 1: WAN Accelerator Architecture

While early systems used Manber's anchor technique to determine chunk boundaries [18], Rabin's fingerprinting technique is now widely used for its efficiency and flexibility [29]. It continuously generates integer values, or fingerprints, over a sliding window (e.g., 48 bytes) of a byte stream. When a fingerprint matches a specified global constant $K$, that region constitutes a chunk boundary. The average chunk size can be controlled with a parameter $n$, that defines how many low-order bits of $K$ are used to determine chunk boundaries. In the average case, the expected chunk size is $2^n$ bytes. To prevent chunks from being too large or too small, minimum and maximum chunk sizes can be specified as well. Since Rabin fingerprinting determines chunk boundaries by content, rather than offset, localized changes in the data stream only affect chunks that are near the changes.

Once a stream has been chunked, the WAN accelerator can cache the chunks and pass references to previously cached chunks, regardless of their origin. As a result, WAN accelerators can compress within a stream, across streams, and even across files and protocols.

**Performance Trade-offs** Figure 1 depicts the general architecture of modern WAN accelerators. Chunk data is stored on disk due to cost and capacity, but an index of chunk metadata is partially or completely kept in memory to avoid disk accesses. Memory also serves as a cache for chunk data, to reduce disk access for commonly-used content.

The performance of WAN accelerators is mainly determined by three factors - (1) *compression rate*, (2) *disk performance*, and (3) *memory pressure*. Compression rate refers to the fraction of the original data actually gets sent, and reflects network bandwidth savings by receiver-side caching. Disk performance determines the cached chunk access time (seek time) while memory pressure affects the efficiency of the chunk index and in-memory caching. These three factors affect the total latency, which is the time to reconstruct and deliver the original data. Delivering high effective bandwidth requires reducing the total latency – having *high compression*, *low disk seeks*, and *low memory pressure* simultaneously.

*Chunk size* directly impacts all three factors, and consequently the effective bandwidth as well. Small chunks can lead to better compression if changes are fine-grained, such as a word being changed in a paragraph. Only the chunk containing the word is modified, and the rest of the paragraph can be compressed. However, for the same storage size, smaller chunks create more total chunks, increasing the metadata index size, and increasing the memory pressure and disk seeks. Large chunks yield fewer chunks in total, reducing memory pressure from indexing and providing better disk usage since each read can provide more data. Large chunks, however, can miss fine-grained changes, leading to lower compression. No chunk size is standard in systems that use content fingerprinting – for example, VBWC [30] uses a 2KB chunk size, LBFS [21] uses 8KB, and Shark [5] uses 16KB.

## 2.2 Developing World Challenges

Our target environment, schools in the developing world, is very different from enterprise branch offices, the typical candidate for WAN accelerators.

**Limited RAM** Due to cost, schools want a *shared machine* or a cheap laptop with limited RAM running the WAN accelerator and other services, instead of using a dedicated server appliance. Also, school children may want to access any content on the Internet, rather than just a smaller set of work-related documents in the enterprise environment. This *larger working set* requires more disk storage, more chunks, and more metadata entries, increasing memory pressure.

**Poor Disk Performance** While disk capacity is cheap and large (1TB SATA per $100), disk seek performance is still limited and is often the bottleneck. Modern desktop drives typically perform roughly 100 seeks/second, but cheaper laptop/external drives we may expect in the developing world are even slower, and are much slower than the high-RPM SCSI disks commercial WAN accelerators use. Also, the larger working set and other services sharing the disks further increase the disk load.

**Low Compression Rate** To handle poor disk performance in the developing world, one choice is to use large chunks to reduce the number of disk accesses, but this reduces the compression rate, limiting bandwidth gains.

**Mesh Topology** Enterprise branch offices typically communicate with a central office in a star topology, whereas many schools in a local region may prefer to get content from each other over cheaper local links rather than over the WAN link. Current WAN accelerators are not designed to exploit this opportunity.

## 3 Wanax Design

Motivated by the challenges in the developing world, we design Wanax around four goals - (1) maximize compression, (2) minimize disk seeks, (3) minimize memory

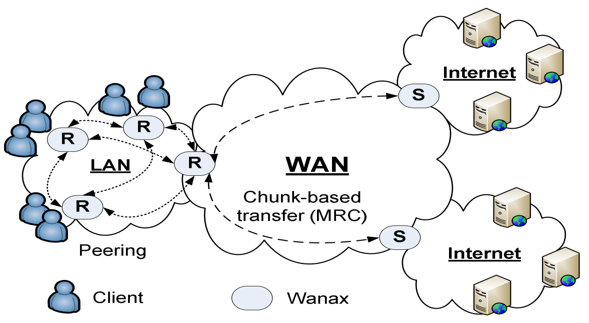


Figure 2: Wanax System Overview

pressure, and (4) exploit local resources.

Wanax works by compressing redundant traffic between a pair of servers – one near the clients, called a R-Wanax, and one closer to the content, called an S-Wanax. For developing regions, the S-Wanax is likely to be placed where bandwidth is cheaper. For example, in Africa, where Internet connectivity is often backhauled to Europe via slow and expensive satellite, the S-Wanax may reside in Europe.

Since we expect most Wanax usage will be Web-related, Wanax operates on TCP streams rather than IP packets since buffering TCP flows can yield larger regions for content fingerprinting. The remote Wanax divides the incoming TCP stream into chunks and sends chunk identifiers (such as SHA-1 hashes) to the local Wanax. If the local Wanax has the chunks cached, the data is reassembled and delivered to the client. Any chunks that are not cached can be fetched from the remote Wanax or other nearby peer. Figure 2 shows the overall system architecture. Each machine is capable of acting as both S-Wanax and R-Wanax, based on the direction of communication.

## 3.1 Basic Protocol

Wanax uses three kinds of communication channels between the accelerators – control, data, and monitoring channels. The control channel is used for connection management and chunk name exchange. The data channels are used to request and deliver uncached chunks, so it is stateless and implemented as a simple request-reply protocol. Finally, the monitoring channel is used for checking the liveness and load levels of the peers using a simple heartbeat protocol. Figure 3 shows typical data transfer between two Wanax gateways.

**Control Channel** When client A initiates a TCP connection to client B in the WAN, that connection is transparently intercepted by the Wanax gateway accelerator [1], R-Wanax. R-Wanax selects S-Wanax which is network topologically closer to B, and sends it an *open connection* message with the IP and the port number of B. S-Wanax then opens a TCP connection to B and a logical end-to-end user connection between A and B is established.

[1]Non-cacheable protocols(e.g., SSH, HTTPS) are bypassed.

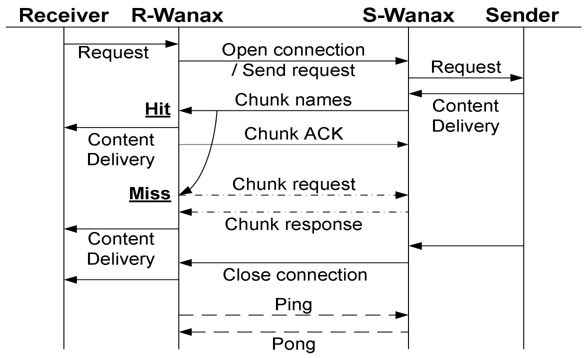


Figure 3: Basic Protocol

When the client B sends data back to S-Wanax, S-Wanax generates chunk names from the data and sends them to R-Wanax in a *chunk name* message. Each chunk name message contains a sequence number so that R-Wanax can reconstruct the original content in the right order. After R-Wanax reconstructs and delivers the chunk data to the original client, it sends a *chunk acknowledgment* (ACK) message to S-Wanax. S-Wanax can then safely discard the delivered chunks from its memory, and proceed with sending more chunk names.

When the sender or receiver closes the connection, the corresponding Wanax sends a *close connection* message to other gateway and the connections between the gateways and the clients are closed once all the data is delivered. The control channel, however, remains connected. All control messages carry flow identifiers, so one control channel can be multiplexed for many data flows. Control messages can be batched for efficiency.

**Data and Monitoring Channels** The data channel uses *chunk request* and *chunk response* messages to deliver the actual chunk content in case of a cache miss at R-Wanax. We also have the *chunk peek* message which is used to query if a given chunk is cached, which is used in our load shedding system.

Each Wanax accelerator monitors the status of its peers by exchanging heartbeats on the monitoring channel. The heartbeat response carries the load level of disk and network I/Os of the peer so that we can balance the request load among peers.

## 3.2 Multi-Resolution Chunking

MRC combines the advantages of both large and small chunks by allowing multiple chunk sizes to co-exist in the system. Wanax uses MRC to achieve (1) high compression rate, (2) low disk seeks, and (3) low memory pressure. When content overlap is high, Wanax can use larger chunks to reduce disk seeks and memory pressure. However, when larger chunks miss compression opportunities, Wanax uses smaller chunk sizes to achieve higher compression. In contrast, existing WAN accelerators typically use a fixed chunk size, which we term *single-resolution chunking*, or SRC.

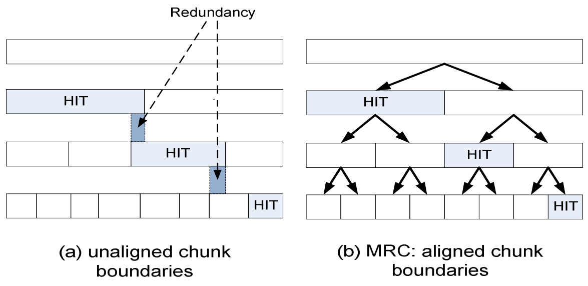

Redundancy
HIT
HIT
HIT
HIT
HIT
(a) unaligned chunk boundaries
(b) MRC: aligned chunk boundaries

Figure 4: Multi-Resolution Chunking

**Generating Chunks** Generating multiple chunk sizes requires careful processing, not only for efficiency, but also to ensure that chunk boundaries are aligned. A naive approach to generating chunks can yield unaligned chunk boundaries, as shown in Figure 4(a). Here, the fingerprinting algorithm was run multiple times with multiple sizes. However, due to different boundary-detection mechanisms, chunk size limits, or other issues, the boundaries for larger chunks are not aligned with those of smaller chunks. As a result, when fetching chunks to reconstruct data, some areas of chunks overlap, while some chunks only partly overlap, causing wasted bandwidth when a partially-hit chunk must be fetched to satisfy a smaller missing range.

Instead, we perform a single-pass fingerprinting step, in which all of the smallest boundaries are detected, and then larger chunks are generated by matching different numbers of bits of the same boundary detection constraint. This process produces the *MRC tree* shown in Figure 4(b), where the largest chunk is the root, and all smaller chunks share boundaries with some of their leaf chunks. Performing this process using one fingerprinting pass not only produces a cleaner chunk alignment, but also requires less CPU.

**Storing Chunks** All chunks generated by the MRC process are stored to disk, even though the smaller chunks contain the same data as their parent. The rationale behind this decision is based on the observation that disk space is cheap, and having all chunks be fully independent simplifies the metadata [2] indexing process, reducing memory pressure in the system, also minimizing disk seeks as well. For example, when reading a chunk content from disk, MRC requires only *one* index entry access, and only *one* disk seek.

Two other options would be to reconstruct large chunks from smaller chunks, which we call *MRC-Small*, and storing the smaller chunks as offsets into the root chunk, which we call *MRC-Large*.

While both MRC-Small and MRC-Large can reduce disk space consumption by saving only unique data, they suffer from more disk seeks and higher memory pressure.

[2]chunk name, disk location of chunk content, and chunk length at a minimum.

| Scheme | Compression Rate | Disk I/O | Memory Pressure | Index Update |
|---|---|---|---|---|
| SRC-Small | High | High | High | Simple |
| SRC-Large | Low | Low | Low | Simple |
| MRC-Small | High | High | High | Complex |
| MRC-Large | High | Low | High | Complex |
| MRC | High | Low | Low | Simple |

Table 1: Comparison of Chunking Schemes

To reconstruct a larger chunk, MRC-Small needs to fetch all the smaller chunks sharing the content, which can significantly increase disk access. The metadata for each small chunk is accessed in this process and loaded in memory, increasing memory pressure compared to standard MRC with only one chunk index entry. MRC-Large avoids multiple disk seeks but complicates chunk index management. When a chunk is evicted from disk or overwritten, all dependent chunks must also be invalidated. This requires either that each metadata entry grows to include all sub-chunk names, or that all sub-chunk metadata entries contain backpointers to their parents.

MRC avoids these problems by making all chunks independent of each other. This choice greatly simplifies the design at the cost of more disk space consumption. In practice, however, we can store more than one month's worth of chunk data on a single 1 TB disk assuming a 1 Mbps WAN connection. Table 1 summarizes the tradeoffs of different schemes.

**Content Reconstruction** When an R-Wanax receives an MRC tree (chunk names only) from an S-Wanax, it builds a *candidate list* to determine which chunks can be fetched locally, at peers, and from the S-Wanax. To get this information, it queries its local cache and peers for each chunk's status, starting from the root. Since Wanax uses the in-memory index to handle this query, it does not require extra disk access. If a chunk is a hit, R-Wanax stops querying for any children of the chunk. For misses, we find the root of the subtree containing only misses, and fetch that from S-Wanax. After reconstructing the content, Wanax stores each uncached chunk in the MRC to disk for future reference.

**Chunk Name Hints Optimization** Sending full MRC trees would waste bandwidth if there is a cache hit at a high level in the tree or when subtrees are all cache misses. Sending one level of the tree at a time avoids the wasted bandwidth, but increases the transmission latency with a large number of round trips. Instead, we have S-Wanax predict chunk hits or misses at R-Wanax and prune the MRC tree accordingly. We augment S-Wanax with a hint table that contains recently-seen chunk names along with timestamps. Before sending the MRC tree, S-Wanax checks all chunk names against the hint table. For any hit in the hint table, S-Wanax avoids sending the subtrees below the chunk. If it is a miss or the chunk name hint is stale, S-Wanax determines the largest subtree that is a miss and sends one chunk content for the entire subtree. This way, we eliminate any inefficiency exchanging MRC trees, further increasing effective compression rate.

Here, we assume the S-Wanax and the R-Wanax will be roughly synchronized over time – what an R-Wanax receives from an S-Wanax now is likely to be fetched from the same S-Wanax in the future. We use the timestamps to invalidate old hint entries, but even if prediction is wrong, it does not affect correctness.

## 3.3 Resource Sharing via Peering

Wanax incorporates a peering mechanism to share the resources such as disks, memory, and CPU with nearby peers using cheaper/faster local connectivity. It allows Wanax to distribute the chunk fetching load among the peers and utilize multiple chunk cache stores in parallel, improving performance. In comparison, existing WAN accelerators support only point-to-point communication.

To reduce scalability problems resulting from querying peers [45], Wanax uses a variant of consistent hashing called Highest Random Weight (HRW) [40]. Regardless of node churn, HRW deterministically chooses the responsible peer for a chunk. We considered other approaches like Summary cache [12], but HRW consumes small memory at the expense of more CPU cycles, and this trade-off fits well in the developing world scenario. In comparison, periodic rebuilds of a Bloom filter would require re-scanning all chunk metadata, causing significant memory pressure and possibly disk access.

Here is how it works. On receiving the *chunk name* message from S-Wanax, R-Wanax sends a *chunk request* message to its responsible peer Wanax. The message includes the missing chunk name and the address of S-Wanax from whom the name of the missing chunk originates. If the peer Wanax has the chunk, it sends the requested chunk content back to R-Wanax with a *chunk response* message. If not, the peer proxy can fetch the missing chunk from S-Wanax, deliver it to R-Wanax, and save the chunk locally for future requests. If peers are not in the same LAN and could incur separate bandwidth cost, fetching the missing chunk falls back to the R-Wanax instead of the peer. After finishing data reconstruction, R-Wanax also distributes any uncached chunk to its corresponding peers. We introduce a *chunk put* message in the data channel for this purpose.

## 3.4 Intelligent Load Shedding

While chunk cache hits are desirable in general since they reduce bandwidth consumption, too many disk accesses may degrade the effective bandwidth by increasing the overall latency. This problem becomes even worse in the developing world where the disk performance is poor. In such cases, we can opportunistically use network bandwidth instead of queueing more requests to the disk. By using the disk for larger chunks and fetching smaller chunks over the network, we can sustain high effective bandwidth without disk overload.

**Algorithm 1** Intelligent Load Shedding

```
Require: $C$: all the chunk names to be scheduled
    $BW$, $RTT$: link bandwidth and RTT
    $Q_i$: # of pending disk requests for peer $i$
    $B_i$: pending network bytes to receive for peer $i$
    $S$: per chunk disk latency
 1: partition $C$ with HRW
 2: resolve $C$ with *chunk peek* message in parallel
 3: generate the candidate list
    $D_i$: cache-hit chunks on peer $i$
    $N$: cache-miss chunks
 4: estimate each latency
    $T_{D_i} = (|D_i| + Q_i) \times S$
    $T_N = RTT + \{\sum_i B_i + \sum_{c \in N} length(c)\}/BW$
 5: while $max(T_{D_i}) > T_N$ do
 6:     pick the peer $k$ where $max(T_{D_i}) = T_{D_k}$
 7:     move the smallest chunk from $D_k$ to $N$
 8:     update $T_{D_k}$ and $T_N$
 9: end while
10: return $D_i$ and $N$
```

We introduce intelligent load shedding (ILS), which exploits the structure of the MRC tree and dynamically schedules chunk fetches to maximize the effective bandwidth given a resource budget. The ILS algorithm is presented in Algorithm 1, and takes the link bandwidth ($BW$) and round-trip latency ($RTT$) of the R-Wanax as input. Each peer Wanax also uses the monitoring channel to send heartbeats that contain its network and disk load status in the form of the number of pending disk requests ($Q_i$), and the pending bytes to receive from network ($B_i$). We assume per-chunk disk read latency ($S$), or seek time is uniform for all peers for simplicity.

The first step in the ILS process is generating the candidate list. On receiving the chunk names from S-Wanax, R-Wanax runs the HRW algorithm to partition the chunk names ($C$) into responsible peers. Some chunk names are assigned to R-Wanax itself. Then R-Wanax checks if the chunks are cache hits by sending the *chunk peek* messages to the corresponding peers in parallel. Based on the lookup results, R-Wanax generates the candidate list (§3.2). Note that this lookup and candidate list generation process (line 2 and 3 in Algorithm 1) can be saved by name hints from S-Wanax, which R-Wanax uses to determine the results without actual lookups.

The next step in the ILS process is estimating fetch latencies for the network and disk queues. From the candidate list, we know which chunks need to be fetched over network (*network queue*, $N$) and which chunks need to be fetched either from local disk or a peer (*disk queues*, $D_i$). Based on this information, we estimate the latency for each chunk source. For each disk queue, the estimated *disk* latency will be per-chunk disk latency ($S$) multiplied by the number of cache hits. For the net-

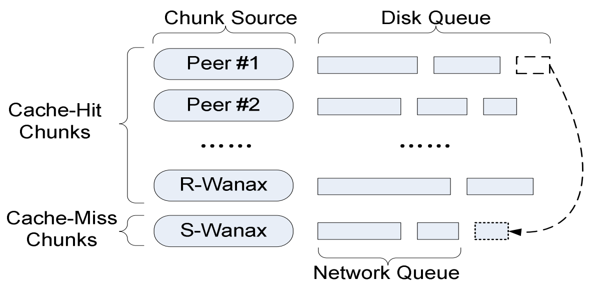


Figure 5: Intelligent Load Shedding: by moving smaller chunks from the disk queue to the network queue, the overall latency is further reduced.

work queue, the estimated *network* latency will be one $RTT$ plus the total size of cache-miss chunks divided by $BW$. If there were pending chunks in the network or disk queues, each latency is accordingly adjusted. We assume the latency between the R-Wanax and peers is small, and do not incorporate it in our model.

The final step in ILS is balancing the expected queue latencies, but doing so in a bandwidth-sensitive manner. We decide if we need to move some cache hit chunks from a disk queue to a network queue – since fetching chunks from each source can be done in parallel, the total latency will be the maximum latency among them. If the network is expected to cause the highest latency, we stop here because no further productive scheduling is possible. When disk latency dominates, we can reduce it by fetching some chunks from the network. We choose the *smallest* chunk because it reduces one disk seek latency while increasing the minimum network latency. We update the estimated latencies, and repeat this process until the latencies equalize, as shown in Figure 5. After finishing ILS, R-Wanax distributes *chunk request* messages to corresponding peers. We send the requests in the order they appear in the candidate list, in order to avoid possible head-of-line (HOL) blocking.

Note that ILS algorithm works with both MRC and SRC. However, by moving the smallest chunk from the disk queue to the network queue, MRC could further reduce the disk latency than SRC, which results in smaller overall latency. Combined with MRC's better overall disk performance and compression, it gives much higher effective bandwidth.

## 4 Simulation Analysis

To understand the trade-offs between MRC and other schemes, we simulate their behavior under a variety of workloads, comparing bandwidth savings, disk access overheads, memory pressure, and performance.

### 4.1 Simulator

We develop a simulator that reads the packet-level traces from tcpdump [38] and simulates various scenarios using SRC and MRC. The simulator uses libnids [16] for stream reconstruction, and consists of 7,000 lines of C code. The outputs are actual and ideal bandwidth savings with and without chunk indexing metadata overhead, disk access overhead for chunk content fetching, and total memory usage. We use 20-byte SHA-1 hashes for the chunk names, and model point-to-point deployments with one S-Wanax and one R-Wanax with no peers. The simulator implements all of the Wanax design mentioned earlier, including the chunk name hint optimization used for both SRC and MRC.

We vary the chunk size for both schemes, with SRC using chunks from 32 bytes to 64 KB, and MRC using three tree configurations, with a 64 KB root chunk with tree degrees 2, 4 and 8 each. The child chunk size is obtained by dividing the parent chunk size by the degree. For example, a degree-2 tree ($d = 2$) starts with a 64 KB root chunk and two 32 KB children chunks. Each child chunk recursively forms a subtree with the same degree until the chunk size reaches 32 bytes. A degree-4 tree has 64 bytes as leaf node size while a degree-8 tree has 128 bytes as the minimum size. If needed, we also change the height of the MRC tree of the same degree, by controlling the smallest chunk size, $m$.

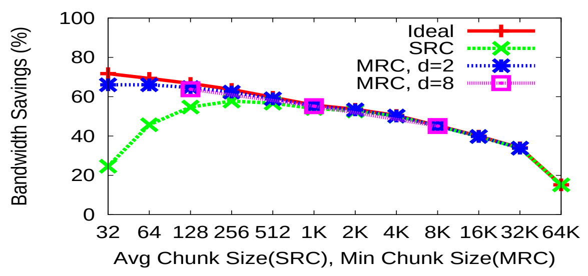
### 4.2 Workload

We choose two types of workloads – dynamically-generated Web content and redundant large files. We focus on dynamic content because the static content is likely to be handled by a standard Web proxy, and we can further reduce bandwidth consumption on uncacheable content with Wanax. We select a number of popular news sites, fetch the front pages every five minutes, and measure the redundancy between the fetches. [3] To generate traffic close to what actual users would produce, we use Firefox 3.0 [13] to fetch the content, and we enable the browser cache to avoid re-fetching cacheable content. We collect packet-level traces for three days, yielding a 1GB trace with 102K TCP sessions and a 72% redundancy. We refer to this workload as "news sites".

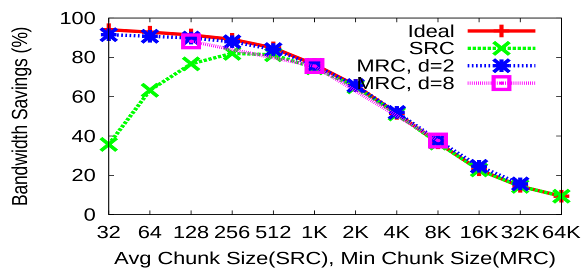
The large-file workload represents long-lived connections for videos or software packages. For this, we download two different versions of the Linux kernel source tar files, 2.6.26.4 and 2.6.26.5, one at a time and gather packet-level traces as well. The size of each tar file is about 276 MB, and the two files are 94% redundant. We refer to this workload as "Linux kernel".

**Cacheability Breakdown** Table 2 separates the potential bandwidth savings on the news sites by their HTTP cacheability, as determined by checking the cache control directives in the response headers. The top two numbers represent the portion of HTTP-uncacheable bytes (H-U), while the bottom two indicate HTTP-cacheable bytes (H-C). The middle two numbers show the portion

[3] CNN, Google News, NYTimes, Slashdot, Digg, Fark, Salon, Yahoo News, and Drudgereport.

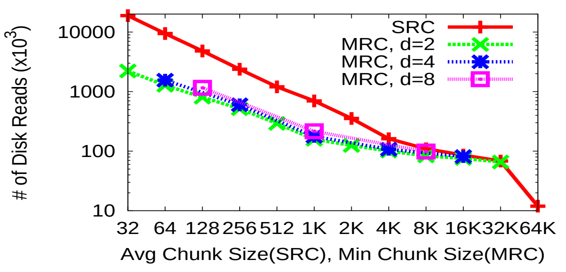


(a) News Sites

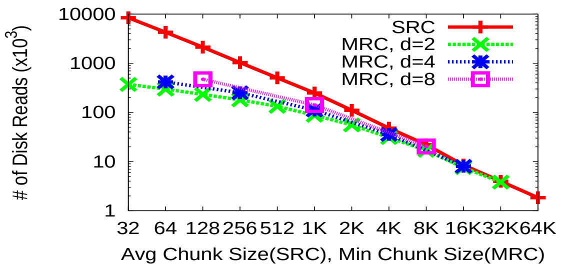


(b) Linux Kernel

Figure 6: Potential Bandwidth Savings (d:degree) – SRC overheads prevent it from reaching ideal savings for smaller chunk sizes. MRC savings are close to ideal across all chunk sizes.



(a) News Sites



(b) Linux Kernel

Figure 7: Disk Operation Cost (d:degree) – By using larger chunks when possible, MRC dramatically reduces the number of disk operations needed for a given workload. Note: Y axis is thousands of operations.

| | SRC | MRC-2 | MRC-4 | MRC-8 |
|---|---|---|---|---|
| H-U/W-U | 20 | 20 | 21 | 23 |
| H-U/W-C | 62 | 62 | 61 | 59 |
| H-C/W-C | 10 | 10 | 9 | 8 |
| H-C/W-U | 8 | 8 | 9 | 10 |

Table 2: News Sites Cacheability Breakdown (%) – as a result of browser caching, most traffic in this workload is HTTP-uncacheable (H-U). However, it still has much redundancy, making most bytes Wanax-cacheable (W-C).

of Wanax-cacheable bytes (W-C), while the outer two depict the Wanax-uncacheable portion (W-U).

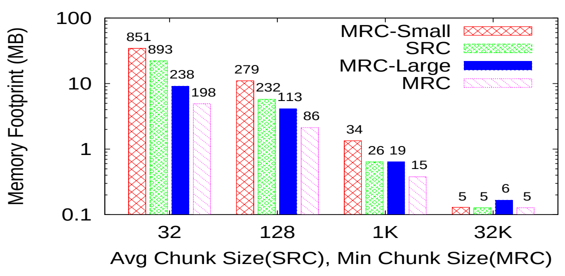
We see that most of the bytes are not cacheable by HTTP, but are cacheable by Wanax. Of the bytes that are not HTTP cacheable, about 75% are redundant and can benefit from Wanax. Of the HTTP-cacheable bytes, more than half are Wanax-cacheable as well. This result suggests that Wanax plus a browser cache can handle much of the traffic, but that Wanax with an HTTP proxy can provide even greater savings. Using an HTTP proxy with Wanax also allows HTTP-cacheable responses to be served directly from the proxy without re-contacting the content provider.

## 4.3 Results

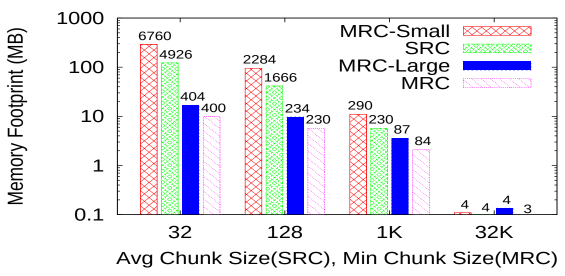
**Potential Bandwidth Savings** Figure 6 shows the ideal and actual bandwidth savings on both workloads for various chunk sizes. As expected, the ideal bandwidth savings increases as the chunk size decreases. However, due to the chunk indexing metadata transmission overhead, the actual savings with SRC peaks at a chunk size of 256 bytes with 58% bandwidth savings on the news sites, and 82% on the Linux kernel. The bandwidth savings drops as the chunk size further decreases, and when the chunk size is 32 bytes, the actual savings is only 25% on the news sites and 36% on the Linux kernel.

On the other hand, MRC approaches the ideal savings regardless of the minimum chunk size. With 32 byte minimum chunks, it achieves close to the maximum savings on both workloads – about 66% on the news sites and 92% on the Linux kernel. This is because MRC uses larger chunks whenever possible and the chunk name hint significantly reduces metadata transmission overheads. When comparing the best compression rates, MRC's effective bandwidth is 125% higher than SRC's on the Linux kernel while it shows 24% improvement on the news sites.

**Disk Operation Cost** MRC's reduced per-chunk indexing overhead becomes clearer if we look at the number of disk I/Os for each configuration, shown in Figure 7. SRC's disk fetch cost increases dramatically as the chunk size decreases, making the use of small chunks almost impossible with SRC. MRC requires far fewer disk operations even at small chunk sizes. When the leaf node chunk size is 32 bytes, SRC performs 8.5 times as many disk operations on the news sites, and 22.7 times more on the Linux kernel.

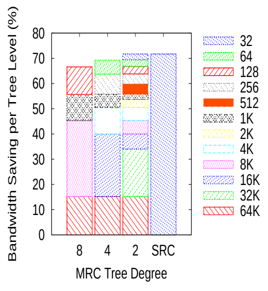


(a) News Sites

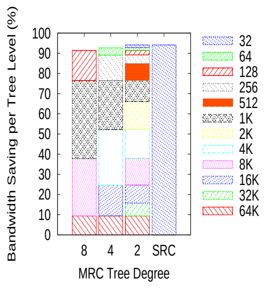


(b) Linux Kernel

Figure 8: Memory Footprint Comparison. Note log-scale Y axis. MRC's memory pressure is typically one-tenth that of SRC and MRC-Small. MRC-Large typically uses twice the memory due to backpointer overhead.



(a) News Site



(b) Linux Kernel

Figure 9: Per-level Bandwidth Savings in the MRC Tree – most MRC savings are from larger chunk sizes, reducing disk access and memory pressure.

**Memory Pressure** Memory pressure limits the amount of cache storage that a WAN accelerator can serve and the amount of RAM it requires for that storage. Figure 8 compares the memory footprint with different chunking approaches. We count the number of chunk index entries that are used during the simulation, and calculate the actual memory footprint. Each bar represents the memory footprint (MB), and the numbers on top of each bar show the number of used cache entries in thousands. Due to space constraints, we show only the MRC trees with the degree 2, but other results follow the same trend.

MRC incurs much less memory pressure than SRC does, since MRC requires one cache entry for any large chunk while SRC needs several cache entries for the same content. MRC-Small, however, requires even more cache entries than SRC does since reconstructing a larger chunk requires accessing all of its child entries. At a 32-byte chunk size, MRC-Small consumes almost 300 MB for the linux kernel while MRC requires only about 10 MB for the cache entries. MRC-Large shows a similar number of cache entries as MRC. However, the actual memory consumption of MRC-Large is much worse than MRC because every child chunk has a back pointer to its parent. MRC-Large consumes almost twice as much memory as MRC on the news workload.

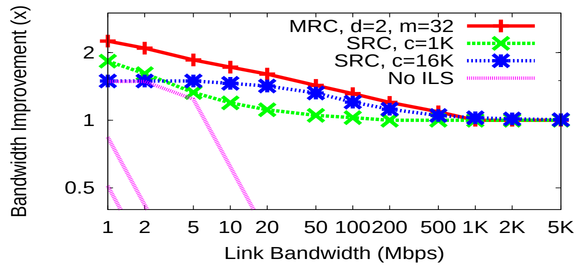
**MRC Chunk Size Breakdown** Figure 9 shows the breakdown of bandwidth savings by different chunk sizes. We present all three MRC configurations and SRC with a 32-byte minimum chunk. For MRC, chunk sizes are sorted from smallest at top to largest at bottom, and the bottom bar shows the root chunk size of 64KB.

The results explain MRC's low disk overhead and low memory pressure – only a small fraction of the total savings is handled by the smallest chunks with MRC, whereas all of the savings is handled by 32-byte chunks with SRC. Most of MRC's bandwidth reduction comes from larger chunks, which results in a much smaller number of disk I/Os and cache entries. We can see the similar trend across different MRC degrees. For example, the portion handled by a 4KB chunk size in MRC degree 4 is handled by 8KB chunk size as well in MRC degree 2. This means that some portion of 4KB chunks are merged into 8KB chunks in MRC degree 2. In all the MRC scenarios, chunks that are 4KB or larger provide 40-50% of the bandwidth savings, drastically reducing disk I/O.

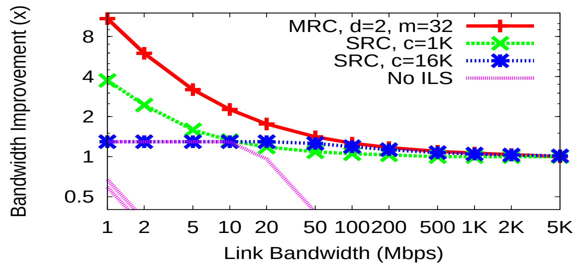
**Intelligent Load Shedding** Based on the previous results of bandwidth savings and disk performance, we simulate the effective bandwidth improvement (times) given a target link capacity using ILS in Figure 10. We vary the link capacity from 1Mbps to 5Gbps, and assume one 7200RPM SATA disk.

We see that the effective bandwidth improvement of both MRC and SRC approaches one as link capacity increases, but SRC drops much faster than MRC. With smaller chunk sizes, SRC shows a high effective bandwidth with slow links due to its high compression rate, but the effective bandwidth quickly degrades as the link capacity grows. This is because with small chunks, the disk soon becomes the bottleneck of the system. In the same context, SRC with larger chunk sizes performs better with fast links, but shows a worse bandwidth improvement for slow links due to its low compression rate.

MRC outperforms SRC regardless of link speed, and it sustains high effective bandwidth by leveraging multiple chunk sizes. If the link is slow, MRC fetches even the smallest chunks from disk, suppressing most redundancy. As the link capacity increases, MRC stops



(a) News Sites



(b) Linux Kernel

Figure 10: Effective Bandwidth Improvement over Link Capacity (c: avg chunk size, d: degree, m: min chunk size) – as link capacity increases and disk performance becomes a bottleneck, MRC sheds cache hits on smaller chunks first, leading to a graceful degradation in effective bandwidth. With ILS disabled, the bandwidth collapses to the bottleneck disk speed. Note log-scale Y-axis.

fetching the smaller chunks from disk, and focuses on the larger chunks rather than completely disabling compression, gracefully degrading the effective bandwidth. When ILS is disabled, the effective bandwidth of all three configurations collapses to the bottleneck disk speed.

## 5 Implementation

The Wanax prototype consists of about 18,000 lines of C code sharing the same MRC/SRC code base with the simulator in §4.

**PPTP/GRE Tunneling** To provide easy access to end users, Wanax is implemented as an Internet gateway with PPTP/GRE tunneling, with TUN/TAP [42] support planned for the near future. Currently, users need to specify the IP address of Wanax in their PPTP client on Linux (or to set up a VPN client on Microsoft Windows), after which all traffic from the user is forwarded to the Wanax system. Wanax performs content fingerprinting only on TCP streams, and bypasses all non-TCP packets.

**Reconstructing TCP Byte Streams** While a fully transparent solution could intercept all IP packets and reconstruct TCP streams, that creates unnecessary complexity between layer 3 and 4. Instead, we intercept each TCP connection from the client, and redirect it to Wanax. This greatly simplifies the buffering process since Wanax can use the regular socket interface to recover the original content. We implement this in the PPTP server [26] by modifying the destination address and port of the incoming packets from the client, to those of Wanax. Similar to network address translation (NAT), we store this mapping in the address translation table, and recover the original address and port for the outgoing packets from Wanax to the client. This requires about 500 lines of PPTP server code modification.

**Storage System** We use HashCache [6] not only as an HTTP proxy, but also as scalable storage for storing and retrieving the chunk content as well as the chunk name hint. With a highly memory-efficient indexing scheme, HashCache fully utilizes a Terabytes-sized disk with less than 256 MB of physical memory, which is an ideal storage system for developing regions. HashCache is designed to use at most one disk seek for reading a random chunk, and performs group writes of related chunks to minimize disk latency for future reading. Wanax uses two special HashCache APIs, `hc_peek()` and `hc_hint()`. `hc_peek()` tells the existence of a chunk without performing actual disk I/O, and we use it for ILS and chunk name hints. `hc_hint()` exports the queuing status of the disk I/Os and is used for ILS calculations.

**Optimizing Transport Protocol** Inter-Wanax communication uses a set of techniques to improve network performance over high-latency WANs. While implementing a fully-custom transport protocol might yield some additional benefit, we opt for simplicity and use TCP variants optimized for high-delay, low-bandwidth links [14, 15]. They modify the congestion avoidance algorithm so that they can quickly increase the congestion window even under high latency. In addition, Wanax multiplexes all communication over a set of long-lived TCP connections, avoiding an extra connection setup overhead of one RTT [23]. We also disable slow-start after idle time because we carefully control the number of connections per link. [4] These techniques are helpful especially for short-lived HTTP connections, which dominates traffic in the developing world [11]. In our tests, we find this combination yields close to the line speed even for many short connections.

**Minimizing MRC Computation Overhead** While MRC preserves high bandwidth savings without sacrificing disk performance, it consumes more CPU cycles in fingerprinting and hash calculation due to an increased number of chunks. Figure 11 shows average time for running Rabin's fingerprinting algorithm and SHA-1 on one chunk with an average size of 64 KB from a 10 MB file. Surprisingly, Rabin's fingerprinting, though it is known to be computationally efficient, turns out to be still quite expensive, taking three times more than SHA-1. How-

[4] sysctl '`tcp_slow_start_after_idle`' in Linux.

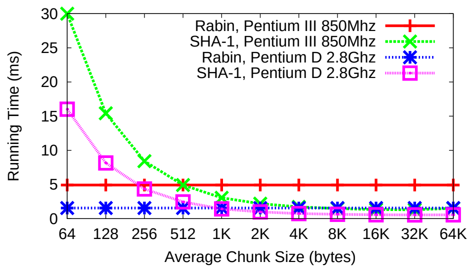
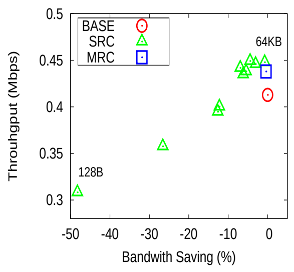
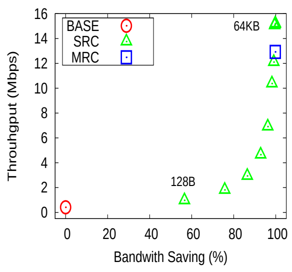


Figure 11: MRC Computation Overhead for 64KB Block



(a) 100% Cache Miss



(b) 100% Cache Hit

Figure 12: Cache Miss and Cache Hit Performance – even on all-hit or all-miss workloads, the extra overheads of MRC are small compared to SRC. The best SRC performers on this set use large chunk sizes, which would produce poor compression on realistic workloads.

ever, the aggregate SHA-1 cost increases as MRC's leaf chunk size decreases. If naively implemented, the total CPU cost of an MRC tree with a height $n$ would be $n \times$ Rabin's fingerprinting time + sum of SHA-1 calculation of each level.

We consider two general optimizations which can be applied to both S-Wanax and R-Wanax. First, we run Rabin's fingerprinting on content only once, detect the smallest chunk boundaries, and derive the larger chunk boundaries from them. Second, we compute SHA-1 hashes only when necessary using the chunk name hint. For example, if S-Wanax knows that this chunk has been sent to R-Wanax before, S-Wanax assumes all of its children are already in R-Wanax and sends only the name of the parent. Likewise, if R-Wanax knows that a chunk has been stored on disk before, it does not re-store its children.

In addition, we implement an R-Wanax specific optimization. When the top-level chunk is a miss with R-Wanax but there are some chunk hits in the lower levels in the MRC tree, we only need to run fingerprinting with the cache-missed candidate list chunks. In order to support this, we now store a Rabin's fingerprint value (8 bytes) along with each chunk name hint. If a chunk in the candidate list is a cache hit, we can retrieve the fingerprint value for the chunk. If a chunk is a cache miss, we run the fingerprinting function to find and store any smaller chunks. We now know Rabin's fingerprint values for all chunks in the candidate list, so we can also reconstruct any parents without running the fingerprinting on the cache-hit chunks.

These optimizations are mainly for the case of chunk cache hits, where more CPU cycles are needed to deliver the chunks to the client. In case of a chunk cache miss, the bottleneck will still be in the slow WAN link for the developing worlds and consuming extra CPU cycles will not affect the download throughput.

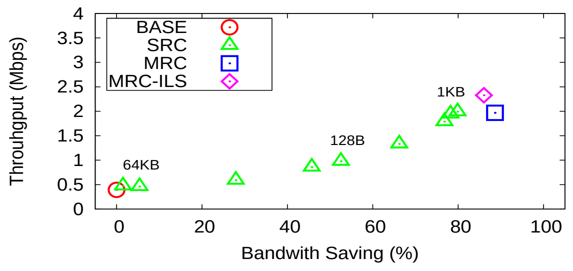
## 6 Evaluation

In this section, we evaluate our prototype implementation of Wanax. Except for the realistic traffic test in the middle of this section, our tests use 1GHz AMD Athlon 64 X2 CPU machines equipped with 1GB RAM and a SATA disk. We divide them into two regions to represent the content provider and the developing region, with intra-region bandwidths set to 100Mbps. We vary the bandwidth and latency of the bottleneck WAN link connecting the two regions, depending on the evaluation scenarios. We have an origin server and an S-Wanax in the content provider side, and a client and two R-Wanax nodes in the developing region. Both the SRC and MRC tests are conducted using the same Wanax servers with the same TCP optimizations. To emulate the effect of large working sets which do not fit in memory, we disable in-memory cache for serving chunk content.

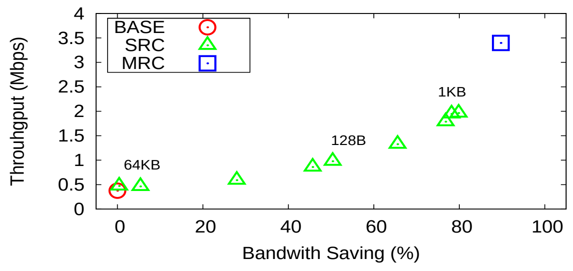
**Microbenchmark** For our microbenchmark, we use two 1 MB files that have 90% redundancy using a 64-byte chunk size. The bottleneck WAN link is set to 512Kbps with a 200ms RTT. We download the first file twice to generate a cold cache miss and a complete cache hit, and then download the second file to generate a partial cache hit. We repeat the experiment by increasing the number of peers, and performing ILS. The downloading throughput (effective bandwidth) without Wanax (BASE) is only 0.41 Mbps due to the high WAN latency. We test SRC with chunk sizes from 128 bytes to 64KB, and a degree-8 MRC using a 128-byte minimum and 64KB maximum chunks.

Figure 12 (a) shows the bandwidth savings and throughputs when downloading the first file. Since every chunk is a cache miss, S-Wanax sends the content as well as the chunk name. Due to the chunk name overhead, SRC consumes more bandwidth than BASE, with up to 48% overhead for 128-byte chunks. However, the throughput is higher than BASE, reaching 0.45Mbps for 64KB chunks, due to the optimized TCP between Wanax nodes. On the other hand, the overhead of MRC is negligible since it uses the largest chunk size of 64KB for most cache misses, yielding an overhead of 5.6% and a throughput of 0.43Mbps.

Figure 12 (b) compares MRC with SRC for a second download of the same file. As expected, SRC with the



(a) One node



(b) Two nodes

Figure 13: Performance with 90% redundancy and 512Kbps WAN link – MRC without ILS produces much better compression than any SRC configuration, and throughput is comparable to the best SRC. With ILS enabled, MRC produces better compression and throughput than any SRC configuration. When peering is used, disk is not a bottleneck, and enabling ILS has no effect.

large chunk sizes (16, 32, and 64KB) shows the best throughput of 15Mbps. [5] As the chunk size decreases, the throughput degrades, and the bandwidth savings is also reduced due to the per-chunk metadata overhead. However, MRC achieves both high throughput and bandwidth savings since they use the largest chunk size in this case. The slightly lower throughput of MRC versus SRC with large chunks is because MRC generates multiple chunk sizes for the first download, spreading the layout of the large chunks on disk, whereas the SRC download stores all of the chunks in sequence on disk.

Figure 13 (a) depicts the performance of downloading the second file after warming the cache with the first file (90% redundancy). In this particular workload, SRC with 1KB chunks is the best configuration achieving both the highest bandwidth savings (80%) and highest throughput (2Mbps). MRC, in comparison, provides a higher bandwidth savings (89%) than any SRC scheme, but without ILS, the disk becomes the bottleneck and the throughput is almost the same as the best SRC. Enabling ILS raises the MRC throughput to 2.4Mbps at the cost of bandwidth savings, but beats every SRC configuration on both bandwidth savings and throughput – ILS automatically finds the sweet spot regardless of the workload.

Figure 13 (b) presents the effect of peering. The experiment is the same as the previous test, but now includes another Wanax peer in the developing region. Since peering allows Wanax to access multiple disks in parallel, we can expect improved throughputs by mitigating the disk bottleneck. However, for SRC, the lower compression rate causes the WAN bandwidth to be the bottleneck, so peering does not help. In comparison, MRC benefits significantly from peering, achieving 3.4Mbps throughput. With disk no longer the bottleneck, ILS is not necessary, and enabling it does not shed any load.

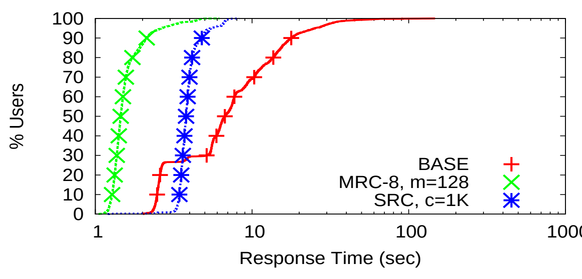
**Realistic Traffic** To test more general Web browsing in the developing regions, we use Alexa Top Sites [3] and YouTube [46] for testing using realistic traffic. We use the "pc850" nodes on Emulab [43], each equipped with an 850MHz Pentium III CPU and 512MB RAM. The bottleneck WAN link is set to 1Mbps with a 1000ms RTT, mimicking a satellite link commonly found in the developing world. First, we collect packet-level traces from Alexa's top 10 sites for Ghana and Nigeria, to reflect common Web browsing activity in these regions, including both cacheable and uncacheable objects. We replay 5,000 connections with 200 simultaneous clients on the traffic, and measure the response time. We also pick one of the most popular videos at the time of testing [6] from YouTube, and have 100 clients simultaneously download the whole 18 MB clip. YouTube's video content is not cacheable by standard Web proxies since its URL is in a customized format and changes for each download. This test is intended to reflect a classroom scenario where a number of students watch the same clip roughly at the same time. We introduce an 1 second interval between the client requests, and measure the throughput of each transfer. For these experiments, we use only one R-Wanax, configured with either a degree-8 MRC tree or a 1 KB SRC configuration, which has shown good performance and bandwidth savings.

Figure 14 (a) shows the response time CDFs for the Alexa workload. The average object size is 5,425 bytes and the median is 570 bytes. MRC outperforms both SRC and direct transfer (BASE), and shows the median response time of 1.5 seconds while BASE and SRC show 6.7 and 3.8 seconds each. MRC and SRC are generally faster than BASE because they fetch most objects from the local disk cache. However, on this workload, MRC typically uses one disk read per object while SRC frequently uses multiple disk I/Os per object. This behavior explains the performance difference between the two, and the disk latency sometimes makes SRC worse than BASE.

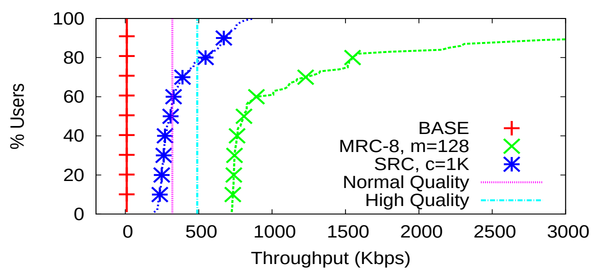
Figure 14 (b) shows the YouTube results. The bitrate of the video is 490 Kbps and the BASE curve shows

[5] The throughput is limited by the 200ms link latency since the total download time is 500-600ms. Downloading a larger file (10 MB) yields 44 Mbps throughput.

[6] The first weekly address by President Obama on 01/24/09



(a) Alexa



(b) YouTube

Figure 14: Realistic Traffic – both MRC and SRC provide compression on the Alexa workload, but MRC's median response time is 1.5 seconds, compared to 3.8 for SRC. For the YouTube test, all students would be able to view the video without interruption using MRC, while with SRC, it would be 20% for the high-quality version and 50% for the low-quality version.

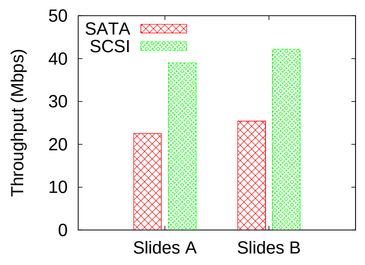


Figure 15: Enterprise Environment – with no link bottleneck, the underlying system performance can be measured. No standard test exists for these systems, but these figures are comparable to those published for commercial systems.

that nobody would be able to watch the clip reliably on a 1 Mbps link. SRC would satisfy only about 20% of the users while MRC would deliver the video to all 100 clients without interruption. The median throughputs are 809 Kbps and 309 Kbps for MRC and SRC each. We test the lower quality video (320 Kbps) of the same content, and find that SRC satisfies half of the users. Without a WAN accelerator, only two or three clients can watch the clip at any given time, which makes using classroom video problematic.

**Enterprise Environment** Finally, we evaluate Wanax in an enterprise-like environment to determine how well it performs compared to commercial WAN accelerators. Unfortunately, while vendors publish performance figures, none appear to publish the test scenarios they use. Testing in industry magazines uses LANs to remove network capacity as the bottleneck, which we also use in this test. That is, we focus on the impact of disk performance by separating the network delay from the overall throughput. This is because the disk performance is the bottleneck in higher link capacity enterprise environments. A high-end commercial product targeting large offices or data centers uses multiple small capacity SCSI disks, [7] rather than one large capacity disk [32].

We create three sets of PowerPoint slides – an original deck that is 11.9 MB, and two modified decks that add slides to this deck, yielding a 13.9 MB file (Slides A) and a 14.9 MB file (Slides B). Compared with the original deck, these have redundancies of 86% and 81%. This represents a scenario where multiple people in different offices are collaborating on a presentation. We first warm the Wanax cache with the original file, and measure the throughput of the two modified slide decks. The measured bandwidth savings correspond to the redundancy in the files. We use MRC degree 8 with the minimum chunk size of 128 bytes, and repeat the experiments with two different disks.

As shown in Figure 15, both file downloads achieve slightly more than 20 Mbps with a single 7200 RPM SATA disk at R-Wanax. The slightly larger redundancy of Slides A (86%) incurs more disk hits than Slides B (81%), and it is reflected in B's slightly larger throughput. With a faster 15K RPM SCSI disk, the throughput almost doubles in both cases. In examining the configurations of one of the leading WAN accelerator companies [32], we see that their per-disk performance ranges from 8 Mbps to 20 Mbps depending on the configuration. Since we have incomplete information about the testing scenario, we cannot draw any firm conclusions, but our range of 20-40 Mbps suggests that we have at least comparable performance to commercial solutions in these higher-end configurations, and our memory pressure analysis suggests that Wanax does so using a small fraction of the memory of these systems.

## 7 Related Work

Much work, both commercial and academic, has been done in the broad area of redundancy elimination for network traffic. Web caching has been an active field, with the first-generation caches [8, 17] storing unchanging objects in their entirety, often with protocol support. Later techniques included delta encoding [19] to reduce traffic for object updates, and duplicate detection to suppress downloading of aliased HTTP objects [20].

Spring and Wetherall [36] further extend the pre-

[7] 1U product supporting 45Mbps uses 4 disks, 3U product supporting 310Mbps uses 16 disks.

vious approaches to sub-packet granularity, and develop a protocol-independent content fingerprinting (CF) scheme that eliminates redundancy over a single link. Recently, Anand *et al.* [4] extend this idea on ISP routers, with an emphasis on redundancy-aware routing algorithms. RTS-id [2] also eliminates redundancy in the wireless environment by caching recently transferred packets through eavesdropping. However, they all work on a per-packet basis at the link layer, which limits the potential bandwidth saving to the packet size. Since Wanax operates on byte streams, it does not have such limits.

Content fingerprinting has been widely adapted in many applications, including network file systems [5, 21], Web proxies [7, 30], file transfer services [27, 28], and Web servers [24]. However, all of these systems are application-specific, and do not work across protocols. DOT [41] proposes a flexible architecture for generic data transfer, which is protocol independent, but not transparent, and requires application-level modification. Ditto [10] extends DOT, and targets wireless mesh network environments. It is complementary to Wanax since Wanax focuses on eliminating redundancy on the bottleneck WAN link.

There are a number of commercial WAN accelerators [9, 33, 35] as well. They operate below the application layer, so they are both transparent and protocol independent. However, they are designed to run on dedicated server-class appliances with fast disks and a large pool of memory. Also, their typical enterprise deployment scenario is a star topology where branch offices are speaking only to a central office. Running them on the resource-limited shared machines with mesh topology in the developing world would be problematic leading to poor performance if possible at all. Instead, Wanax is designed from the scratch to specifically address the developing world's needs, and we believe some of our techniques such as MRC and ILS can also be applied to the enterprise scenarios to reduce the deployment cost.

To the best of our knowledge, Wanax is the first system to simultaneously use multiple chunk sizes. Riverbed [33] uses a bottom-up segmentation scheme [1] that first uses 100 byte chunks, and then creates larger pseudo-chunks that contain the names of the smaller chunks [31], which is similar to MRC-Small. This approach provides some of the disk efficiency and bandwidth benefits of MRC, but still requires access to all of the metadata of the 100-byte chunks, thereby retaining the memory pressure of the smaller chunks. In the context of large file replication, Remote Differential Compression [39] uses a similar recursive segmentation scheme with a minimum chunk size of 1 KB, in order to reduce the size of chunk names sent over the network. Most recently, multi-resolution handprinting [37] proposes an efficient technique for choosing the best chunk sizes for the given similar files, by comparing handprints - a deterministic subset of chunk hashes with different chunk sizes. We share the same spirit of exploiting trade-offs of multiple chunk sizes. However, their method is based on static analysis on the files they already have. MRC is a dynamic counterpart, and is directly applicable for online processing.

Finally, there are a number of active research projects for the developing world. DitTorrent [34] shares the same idea of exploiting better regional connectivity as Wanax, but focuses on scheduling P2P dialup connections. As systems like rural WiFi [25] or WiMAX [44] extend the Internet to new regions, Wanax can help improve the effective bandwidth delivered.

## 8 Conclusion

We have presented the design and implementation of Wanax, a flexible and scalable WAN accelerator targeting developing regions. Using a novel chunking technique, MRC, Wanax provides high compression and high throughput, while maintaining a small memory footprint. This profile enables it to run on resource-limited shared hardware, an important requirement in developing-world deployments. By exploiting MRC to direct load shedding, Wanax is designed to maximize the effective bandwidth even when disk performance is poor due to overloading. The peering scheme used in Wanax allows multiple servers in a region to share their resources, and thereby exploit faster and cheaper local-area connectivity instead of always using the WAN. In summary, through a careful design addressing the developing world challenges, Wanax provides customized, cost-effective WAN acceleration to the region with commodity hardware. We have begun deploying Wanax at a few partner sites in Africa, and expect to have more results about real-world operation in the future.

## Acknowledgment

We would like to thank Anirudh Badam for providing HashCache, and Marc Fiuczynski for arranging and coordinating our deployments in Africa. We also thank our shepherd, Michael Isard as well as anonymous USENIX ATC reviewers. This research was partially supported by NSF awards CNS-0615237 and CNS-0916204, and KAIST awards G04100004.

## References

[1] US patent #7,116,249: Content-based segmentation scheme for data compression in storage and transmission including hierarchical segment representation, 2006.

[2] Afanasyev, M., Andersen, D. G., and Snoeren, A. C. Efficiency through eavesdropping: Link-layer packet caching. In *USENIX NSDI* (Apr. 2008).

[3] ALEXA THE WEB INFORMATION COMPANY. `http://www.alexa.com/`.

[4] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet caches on routers: The implications of universal redundant traffic elimination. In *SIGCOMM* (2008).

[5] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIERES, D. Shark: Scaling file servers via cooperative caching. In *USENIX NSDI* (2005).

[6] BADAM, A., PARK, K., PAI, V., AND PETERSON, L. Hashcache: Cache storage for the next billion. In *Proceedings of the 6th conference on Networked Systems Design and Implementation (NSDI'09)* (2009).

[7] CHAKRAVORTY, R., CLARK, A., AND PRATT, I. Optimizing web delivery over wireless links: Design, implementation and experiences. In *IEEE Journal of Selected Areas in Communications (JSAC)* (2003).

[8] CHANKHUNTHOD, A., DANZIG, P. B., NEERDAELS, C., SCHWARTZ, M. F., AND WORRELL, K. J. A hierarchical internet object cache. In *USENIX ATC* (1996), pp. 153–163.

[9] CITRIX SYSTEMS. `http://www.citrix.com/`.

[10] DOGAR, F., PHANISHAYEE, A., PUCHA, H., RUWASE, O., AND ANDERSEN, D. Ditto - A System for Opportunistic Caching in Multi-hop Wireless Mesh Networks. In *MobiCom* (2008).

[11] DU, B., DEMMER, M., AND BREWER, E. Analysis of WWW traffic in Cambodia and Ghana. In *WWW* (2006).

[12] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking 8*, 3 (2000), 281–293.

[13] FIREFOX WEB BROWSER. `http://www.mozilla.com/firefox/`.

[14] FLOYD, S. Highspeed tcp for large congestion windows. RFC 3229, 2003.

[15] HA, S., RHEE, I., AND XU, L. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Operating Systems Review. 42*, 5 (2008), 64–74.

[16] LIBNIDS. `http://libnids.sourceforge.net/`.

[17] MALTZAHN, C., RICHARDSON, K. J., AND GRUNWALD, D. Performance issues of enterprise level web proxies. In *In Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1997).

[18] MANBER, U. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference* (1994).

[19] MOGUL, J., KRISHNAMURTHY, B., DOUGLIS, F., FELDMANN, A., GOLAND, Y., VAN HOFF, A., AND HELLERSTEIN, D. Delta encoding in HTTP. RFC 3229, January 2002.

[20] MOGUL, J. C., CHAN, Y. M., AND KELLY, T. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *USENIX NSDI* (2004).

[21] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *SOSP* (2001).

[22] ONE LAPTOP PER CHILD. `http://www.laptop.org/`.

[23] PADMANABHAN, V. N., AND MOGUL, J. C. Improving http latency. *Computer Networks and ISDN Systems 28*, 1-2 (1995), 25–35.

[24] PARK, K., IHM, S., BOWMAN, M., AND PAI, V. S. Supporting practical content-addressable caching with czip compression. In *USENIX ATC* (2007).

[25] PATRA, R., NEDEVSCHI, S., SURANA, S., SHETH, A., SUBRAMANIAN, L., AND BREWER, E. Wildnet: Design and implementation of high performance wifi based long distance networks. In *USENIX NSDI* (2007).

[26] POPTOP - THE PPTP SERVER FOR LINUX. `http://www.poptop.org/`.

[27] PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Exploiting similarity for multi-source downloads using file handprints. In *USENIX NSDI* (Cambridge, MA, Apr. 2007).

[28] PUCHA, H., KAMINSKY, M., ANDERSEN, D. G., AND KOZUCH, M. A. Adaptive file transfers for diverse environments. In *USENIX ATC* (2008).

[29] RABIN, M. O. Fingerprinting by random polynomials. Tech. Rep. TR-15-81, Harvard University, 1981.

[30] RHEA, S., LIANG, K., AND BREWER, E. Value-based web caching. In *Proceedings of the Twelfth International World Wide Web Conference* (May 2003).

[31] RiOS 5.5 Technical Whitepaper. `http://www.riverbed.com/docs/TechOverview-Riverbed-RiOS\_5.5.pdf`.

[32] RIVERBED STEELHEAD PRODUCT FAMILY DATASHEET. `http://www.riverbed.com/docs/DataSheet-Riverbed-FamilyProduct.pdf`.

[33] RIVERBED TECHNOLOGY, INC. `http://www.riverbed.com/`.

[34] SAIF, U., CHUDHARY, A. L., BUTT, S., AND BUTT, N. F. Poor man's broadband: peer-to-peer dialup networking. *SIGCOMM Computer Communication Review. 37*, 5 (2007), 5–16.

[35] SILVER PEAK SYSTEMS, INC. `http://www.silver-peak.com/`.

[36] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *ACM SIGCOMM* (2000).

[37] TANGWONGSAN, K., PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Efficient similarity estimation for systems exploiting data redundancy. In *Proc. IEEE INFOCOM* (San Diego, CA, Mar. 2010).

[38] TCPDUMP. `http://www.tcpdump.org/`.

[39] TEODOSIU, D., BJRNER, N., GUREVICH, Y., MANASSE, M., AND PORKKA, J. Optimizing file replication over limited-bandwidth networks using remote differential compression. Tech. Rep. MSR-TR-2006-157, Microsoft Research, Nov. 2006.

[40] THALER, D. G., AND RAVISHANKAR, C. V. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking 6*, 1 (Feb. 1998), 1–14.

[41] TOLIA, N., KAMINSKY, M., ANDERSEN, D. G., AND PATIL, S. An architecture for internet data transfer. In *USENIX NSDI* (2006).

[42] UNIVERSAL TUN/TAP DRIVER. `http://vtun.sourceforge.net/tun/`.

[43] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI* (2002).

[44] WIMAX. `http://www.wimaxforum.org/home/`.

[45] WOLMAN, A., VOELKER, G. M., SHARMA, N., CARDWELL, N., KARLIN, A. R., AND LEVY, H. M. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles* (1999).

[46] YOUTUBE. `http://www.youtube.com/`.