USENIX 1998 Annual Technical Conference,
June 15–19, 1998 in New Orleans, Lousiana

High-Performance Caching With The Lava Hit-Server

Jochen Liedtke Vsevolod Panteleenko Trent Jaeger Nayeem Islam

Thomas J. Watson Research Center
IBM
Hawthorne, NY 10532
{jochen,vvp,jaegert,nayeem}@us.ibm.com

Abstract

With the development of new client-server computing models, such as thin clients and network computers, the performance of servers becomes a bottleneck. In these models, servers support a large number of clients. They download significant amounts of data to their clients in the form of graphics, executables (e.g., applets), and video. We present an architecture for building high-performance server systems that can efficiently serve large local clusters of NCs or other clients. The key component in our architecture is a generic cache module that is designed to fully utilize available bus bandwidth. Our experiments show that such a server system can achieve throughput rates of up to 36,000 transactions per second. We detail the design and implementation of the generic cache component, describe its use in the implementation of a sample server system, and show how the architecture can be scaled.

1 Rationale

In the future, we envision local networks serving thousands up to hundreds-of-thousands of resource-poor clients, e.g., NCs. These networks might be intra-building, intra-organization or even intra-city. Customizable servers will be required that nevertheless offer extremely high performance.

The low cost and variety of future clients (e.g., PDAs, laptops, pagers, printers, and specialized appliances) will result in a larger number of client devices per user. Each office employee could have tens of client devices. Thin clients will have fast processors, but little or no disk storage so that they will download most of their data and executables. Some typical applications for these clients will also download very large objects, such as graphics and video.

The existence of cheap client hardware with high-resolution graphics and high-quality audio together with ubiquitous high-bandwidth networks will probably lead to applications with increasing demands on network and server performance. For a scenario with 10,000 users and 100,000 thin clients, we think that requirements to the server like “handle 20,000 requests in a second with a data bandwidth of 1 GByte/s” will be realistic. Perhaps even higher bandwidth will be requested. In the near future, we envision clusters of 1,000 up to 2,000 NCs.

The postulated server performance is about two orders of magnitude higher than current servers achieve [10, 15]. We are convinced that improving current systems by evolution is not sufficient: we need a new server architecture to achieve the mentioned goals. The basic requirements to this architecture are *customizability, performance, scalability* and *security support*.

As Kaashoek *et al.* have noted recently [10], traditional servers are designed either to run a variety of applications, but with abstractions that lead to poor performance, or run specialized applications efficiently, but without the flexibility to run other applications. They define a *server operating system* environment that is designed to provide abstractions for building specialized, high-performance servers. While their server operating system enables improvements in server performance and flexibility, we claim that further improvement in performance is necessary and possible without reducing the variety of server systems that can be developed.

We believe that in our envisioned scenarios, significant performance improvements are possible by providing clients with access to local servers that optimize cache response. For example, an organization could use a central server (or cluster of servers) and 1000 NCs, all connected by a local area network. The NCs boot from the central server, use it as a file system, as a Web proxy, as a server for organization-internal HTML documents, and perhaps also for video clips. Some objects the server deals with will come from the Web; however most objects will be local to the organization (software, forms, brochures, diagrams, custom data, etc.) so we expect a large but bounded working set.

In this scenario, the important problems are actually server latency and throughput, rather than network latency (as addressed by Web caching [3]). Network bandwidth for the central server to communicate with the clients is easily obtained (e.g., using multiple 100 Mbps Ethernets), so the problem in this scenario is to improve server performance such that it can utilize this bandwidth effectively.

We are aware of two principal ways for increasing a system's performance substantially beyond the bare performance growth of hardware: replication and caching. Massive replication of servers (e.g., IBM's Olympic server) is probably too expensive, makes write accesses complicated and slow, and needs sophisticated load balancing.

Therefore, we focus on the development of a high-performance, cache-based architecture that is general enough to support most type of server applications. Such an architecture should enable the server to achieve very close to the maximum performance that the architecture can achieve in principle for the fast path (i.e., hits on NC client requests in the local server's cache). In addition, customizability and handling heterogeneous objects are also relevant to the architecture because it must be general enough to support a wide variety of applications.

Our key decision for constructing high-performance customizable servers is to separate *generic cache modules* and *customizable miss-handling modules* and to map them to dedicated machines. Generic cache modules are responsible for high performance while customizable miss handlers enable flexibility. Single or multiple generic and customizable modules together build a general or specialized server (or server cluster). In the prototype, we use an off-the-shelf PC equipped with a 200 MHz Pentium Pro processor for a generic cache module.

In this paper, we focus on the generic cache module which is the centerpiece of the architecture. For it, we envision main-memory caches of 4 GB up to 64 GB. The challenge is to construct software that efficiently maintains such a cache, that does not restrict customizability, that supports scaling of multiple modules, and that is nevertheless highly specialized and optimized to achieve the demanded high throughput.

The proposed architecture heavily relies on the inherent "cache friendliness" of applications. Our hope is that due to the customizability of the architecture and due to the support of active objects in the generic cache module, most applications can benefit from the cache structure. However, we are still far from substantiating this hope. Currently, we have only implemented two small prototypes, an instructional video-clip server (see Section 2.1) and a web proxy. Using only a single custom module and a single generic cache module, the video-clip server is able to serve up to 402 clients with video clips simultaneously (MPEG I, 1.5 Mbps, full screen presentation). The performance is currently limited by the sub-optimal DMA of current PCI chipsets and memory buses.

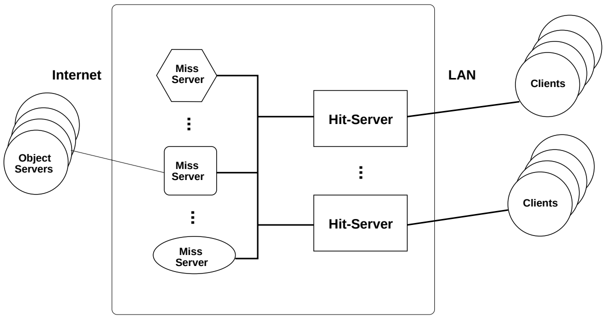

Figure 1: Server Architecture. A customized server has modules of two different types: hit-server(s) and miss-server(s). They are dedicated machines and are connected with a dedicated intra-server network. Client libraries reside on each client and are responsible for communicating with the hit-server via a LAN.

In Section 2, we describe the design of the server architecture, focused on the generic cache module. We detail the implementation of the hit-server in Section 3 and present performance results in Section 4. In Section 5, we discuss the scalability of the presented architecture. We review related work in Section 6 and conclude in Section 7.

2 The Server Architecture

The server architecture consists of two types of modules that cooperate to manage a large RAM cache of objects (shown in Figure 1). A *hit-server* is a highly-optimized generic module (a dedicated machine) that handles client requests that hit in the object cache. *Miss-servers* handle client requests that miss in the object cache. Miss-servers also implement application-specific policies for managing the hit-server cache and for distributing objects to clients. Multiple, different miss-servers can be combined with a single hit-server or with multiple hit-servers (see Section 5).

Clients read and write objects by performing *get* and *put* operations on server objects. For example, HTTP's *get* and *put* are mapped to Lava's *get* and *put* by client libraries. The Lava operations also work on partial objects. This feature permits clients to download or modify arbitrary, selected parts of the object (i.e., as opposed to reading the entire object). As well, client libraries can implement file-system like access. Furthermore, objects can have object-specific *get* and *put* operations supplied by the object creator. For example, a custom *get* can present an object in different formats based on the requesting client. Another example is HTTP *post*: A combined *putget* operation sends the *post* data to the active object which then calculates the response and sends it back to the client as *get* data.

In fact, *putget* is the only existing operation from the hit-server's point of view. The *put* data is sent to the object which then delivers the *get* data. Pure *get* and *put* operations are implemented by *putget* using empty put or get parameters respectively. For better intuitive understanding, we will, however, always refer to pure *get* and *put* operations in the following sections.

In the remainder of this section, we develop the design of the server architecture. First, we describe an example server which demonstrates the interplay between the architecture components. Next, we analyze the application scenario to determine the requirements of the architecture. The last two subsections develop the design.

2.1 An Example Server

We implemented a video-clip server that delivers video clips interactively to clients. Examples of such a system include museum kiosks, retail services (e.g., mall, information resources), educational services (e.g., library and encyclopedia), and entertainment services. For instance, upon entering a museum, information about exhibits, resources, and staff can be retrieved from computers located at kiosks throughout the museum.

A prototype version of the video-clip server system is built using Windows 95 PC clients running the ActiveMovie application to view videos and a miss-server that runs on Linux 2.0 to retrieve the videos. Our instructional server provides small video clips (varying from about 20 seconds and 4 MB to about 180 seconds and 50 MB each) to its clients.

The Windows 95 clients use Lava's reliable object protocol to communicate with the hit-server. The protocol is implemented as a Windows kernel extension (a so-called Vxd element) that communicates directly with the network driver using Window's NDIS network-driver interface. A special ActiveMovie source filter was built that transfers ActiveMovie requests into Lava's *get* transactions. The ActiveMovie source filter requests a series of video blocks that correspond to a consecutive intervals of the video. The size of each block is 32 KBytes.

The according miss-server executes as a user process on Linux. Lava's reliable object protocol (see Section 2.4) is incorporated into the Linux kernel to enable hit-server/miss-server communication. Application-specific policy is added to: (1) download video objects into the miss-server from the Web using (unmodified) HTTP and (2) implement a custom cache-replacement policy that controls the hit-server.

In conjunction with the mentioned video-clip miss-server, the hit-server can serve up to 402 clients simultaneously with different video clips (see Section 4).

2.2 Requirement Analysis

The hit-server has an object cache that it uses to process *get* and *put* operations from network clients. Furthermore, it is also linked to miss-servers (e.g., web proxy, file system servers, and databases) from which it can obtain other objects to fulfill its clients' requests. Each hit-server communicates with its clients via network controllers that transfer data between the network and the server's main memory over the PCI bus and the memory bus. On the memory bus, CPU-memory and PCI-memory traffic compete with each other.

Our original goal (which turned out to be not completely achievable, see Section 4) was to build server systems whose server-to-client throughput rates approach the current PCI bus bandwidth of 1 Gbps. Utilizing this rate even for moderately small objects of 1 K, would require 128,000 transactions per second. For comparison: commercial servers currently achieve rates up to 2100 [15] transactions per second, research servers [11] up to 7000 transactions per second.

Since the memory bus is the bottleneck of a hit-server, the server architecture must maximize the availability of the memory bus to the network controllers.

In order to achieve high throughput and low latency, the server must also make efficient use of its object cache, basically a problem of deriving cache replacement and prefetching protocols that keep the "right" objects in the cache. We refer to applications in which such protocols can be defined as *cache-friendly*. In this paper, we make no claims about the design of such protocols (as others do, e.g., Cao et al. [2]). However, for applications where such protocols exist, the server architecture must permit their implementation. Also, the effect of processing cache misses on server throughput must be minimized.

Communication over untrusted links must be authenticated to prevent attacks. An authenticated communication is one whose source, integrity, and freshness have been verified. We do not believe that privacy is required for our server, at present. Certainly communication with object servers over the Internet needs to be authenticated. In addition, given the number of insider attacks reported and the value of corporate data, client communication over the LAN may also need to be authenticated. Therefore, the server architecture must enable the ability to authenticate communication along any link. However, we must minimize the effect that message authentication has on the hit-server's throughput rate.

The server systems must also be scalable through the addition of new server modules. Scalability is limited primarily by interactions caused by objects being written. When an object write occurs, consistency requirements of that object must be enforced. Many applications enforce a strict consistency in which a reader sees the latest writes. However, less restrictive policies, such as the various types of release consistency, are also used by distributed applications, so the server must support application-specific consistency policies.

In summary, the major requirements relate to four dimensions: *Performance, flexibility, security,* and *scalability.*

2.3 Hit-Server Architecture

The hit-server provides efficient mechanisms for processing client requests that hit in the object cache. Upon a client request, the hit-server locates the requested object and either downloads it to the client (on a *get*), creates a new version (on a *put*), or forwards the request to the miss-server (on a miss).

The hit-server is free of policy. Its general mechanisms are intended to support any policy that the miss-servers can implement, so developers can create application-specific server systems. For example, when cache replacement is signaled by the hit-server, the miss-server is free to select the objects to be replaced. A miss-server library provides general miss-server primitives and a set of functions that use these primitives to implement predefined policies. However, the developer can choose to build a miss-server from any combination of predefined and custom policies or even build a new miss server from scratch.

The hit-server processes client *get/put* requests to access a large RAM cache of objects and processes miss-server *add/remove* requests to modify the cache. Requests that result in cache misses are forwarded to the miss-server.

The default *get* operation first checks whether the object is available. Then, the client is authorized against the object's ACL. Next, the object's status data and consistency matrix may indicate that the object's miss-server be signaled, so it can implement the object's consistency policy (see Section 5 for details). Finally, after verifying that a download is necessary by checking version numbers, the hit-server sends the requested object data to the client.

The *put* operation is similar except that each *put* generates a new version of the object: First, the entire object is copied (lazily); then, those object parts are modified that are addressed by the *put* data. The mentioned copy operation is based on copy-on-write techniques; basically, the newly received packets are simply linked into the new version's data descriptor. (The per-object data descriptor is similar to a multi-level page table but implements a granularity of 1 K.) This technique makes it easy to update an object while *gets* are concurrently active. After all *gets* on the old version are finished, the old version can be garbage collected.

Similar to the DynamicWeb cache [8], the hit-server interface permits to cache dynamic Web pages or other dynamic objects. The application running on the miss server constructs the dynamic object on demand and invalidates or updates it in the hit-server whenever necessary. Note that this requires only default *putget* operations. If the method is too expensive, e.g. for a dynamic object that delivers random numbers, an object-specifc *putget* can be used that executes directly on the hit-server.

The architecture enables flexible handling of objects through object-specific *putget* operations. An object-specific operation supersedes the default. Object-specific *putget* methods are run on the hit-server to enable efficient implementation of custom operations. An object-specific *put* operation is invoked after the new data is received, but prior to updating the object cache. E.g., it can be used to implement object-specific consistency protocols that are executed when an update is made. An object-specific *get* operation is invoked prior to delivery to the client. An object-specific *get* can be used to deliver modified object data to a client (e.g., for displaying the object effectively on the client).

In order to prevent corruption of the hit-server and denial-of-service to clients, the hit-server must control these object-specific *putget* operations. We assign each object-specific *putget* operation to its own address space to protect the hit-server and other object-specific operations from modification. Resource requests, such as access to a client descriptor, are intercepted and authorized against the object-specific operations access rights [9, 13]. For example, object-specific operations are permitted to read the requesting client's description and the requested object descriptor. If multiple objects share the same *putget* operation, they are all mapped into the same address space. Address spaces are a relatively lightweight resource, see Section 3.2. Custom *putget* operations can be multithreaded to execute multiple requests concurrently.

The hit-server also processes miss-server operations, *add* and *remove*, by which miss-servers can add or remove objects (specified by name and version) from the object cache, respectively. An *add* enables the miss-server to set the initial values for the object's attributes.

2.4 Reliable Network Communication

Although we envision the use of switched networks, packet losses are possible. They can occur in switches or even within the hit-server's Ethernet chips, even though the hit-server always has enough receive buffers available. The point of congestion is not main memory itself but the memory/PCI bus. As described in more detail in Section 3.1, the maximum memory-bus bandwidth for DMA is approximately 600 Mbps. As soon as the sum of all cards' incoming and outgoing Ethernet traffic exceeds[1] this value, the receiver FIFOs (4K each) in the Ethernet chips can run over. (The problem is real: the current hardware uses 7 full-duplex Ethernet cards enabling peaks of 1400 Mbps.)

The first obvious choice for a reliable transport protocol is TCP. Although some of its features are not necessary for our application (e.g., checksums, handling duplicates and out-of-order packets), adaptive flow control and retransmission of lost packets are required. It is well known that TCP is often costly in terms of processor cycles [18] and that a VMTP-like [4] protocol is better suited for transactions.

An even more important problem of TCP is that its congestion-avoidance policies (which are primarily based on end-to-end flow control) are tailored to current WANs and are not effective for our envisioned scenarios: On a highly loaded LAN, we experience dramatically changing loss rates, e.g. 0% loss for 5 ms, then 40% for 2 ms, etc. TCP would very quickly reduce its window size to a single packet. This would result in poor bandwidth utilization *and not avoid packet losses* since, in peak situations, the loss rate depends more on the synchronization between the clients than on the sender's transmission rate. Given that under peak load the round-trip time exceeds 1 ms (the client's and server's hardware FIFOs are even good for a 0.9 ms delay), it is very difficult to devise a flow-control protocol that can handle the described agility efficiently.

Instead, we use a late-retransmission protocol. Basically, any sender transmits the whole object in a burst, as fast as the network hardware permits. Afterwards, the receiver tells the sender which parts of the object it has received; finally, the sender retransmits the missing, i.e., lost, parts (if any). This procedure is repeated until all data is transferred.

The mentioned protocol does not work for any topology. However, it behaves nicely on a star topology as in our scenario where nearly all communication either goes to or comes from the hit-server. With a switched network, the protocol ensures that the hit-server receives data at its maximum rate and all clients get close to optimal bandwidth. At a first glance, this might be counterintuitive since the protocol seems to invite congestion rather than to avoid it. However, assume that two clients simultaneously send a 1 MByte object each to the same hit-server card. Due to congestion, always 1/2 of the data sent will be lost: On the first round, each client sends the entire object; on the second round, each client sends the lost 1/2 M, on the third the 1/4 M lost in the second round, etc. In total, each client sends 2 MByte with full speed to effectively transfer 1 MByte, i.e., gets 50% of the available bandwidth. In the same time, the hit-server receives 2×1 MByte at the highest possible rate. The point is that only such packets are lost that could not have been transmitted under ideal flow control, and that the "unnecessary" transmissions consume only resources that otherwise would be unused.[2]

Since this paper concentrates on the hit-server design, we will neither go into details of the protocol nor proof its properties nor discuss further "good" topologies here. For the context of this paper, it is relevant that the protocol is reliable, performs well under peak load, is cheap for low loss rates, is robust against random losses and highly fluctuating loss rates, and can be "asymmetrically" implemented such that it requires more processor cycles on the client side and less on the hit-server side.

3 Hit-Server Implementation

In this section, we describe the techniques used for implementing the generic hit-server. Miss-servers enable customizability and extensibility; the hit-server is responsible for performance. Consequently, its design is basically driven by performance requirements. In a first step, achievable performance goals are derived from the characteristics of the available hardware. Then, in an ideal-case micro analysis, we try to determine the load an optimal implementation would impose on processor, cache, memory bus, PCI bus and Ethernets. This analysis gives us a more realistic upper bound of the achievable throughput, and it reveals the bottlenecks of the system. Finally, guided by these results, we describe the actual construction of the hit-server core software.

[1] In reality, the situation is complicated by DMA bursts, bus arbitration policies and the existence of multi-level PCI buses. However, all this is hardware and most of its parameters and algorithms cannot be influenced by software. A detailed description is beyond the scope of this paper.

[2] There are some pathological situations. If, e.g., 100 clients simultaneously start sending an object of 100 packets, each round effectively transfers only 1 packet per client. Then we needed 1 ack per transferred packet, similar to 1-packet windows in TCP. (Nicely, we needed only 0.1 acks per transferred packet, if 1000-packet objects were sent.) Therefore, as soon as a client notices that the effective ratio of acks to effectively transferred packets becomes too high, it takes random rests while transmitting the packets. Since all active clients act in a similar way, the congestion and the loss rate decreases so that the ack ratios become better. For short objects, additional transmission rounds are needed: the packets are retransmitted a second or third time without waiting for an ack.

3.1 Analysis

Our current hit-server machine is an off-the-shelf PC, equipped with a 200-MHz Pentium Pro uniprocessor, an Intel 440 FX chipset, and 256-K of L2 cache memory. For our experiments, the hit-server was equipped with 256 M of main memory. External devices are connected to the processor and the memory by a 32-bit PCI bus with a cycle time of 30 ns (33 MHz). The PCI-bus specification [17] permits burst DMA transfers with a rate of 1 word per PCI-bus cycle, corresponding to 132 MByte/s or 1056 Mbps. However, the 440 FX chipset, at least in combination with the Ethernet controller chips we use, takes on average 1.5 cycles to transfer a word. So the maximum achievable transfer rate is 88 MByte/s or 704 Mbps.

The SMC EtherPower 10/100 PCI network cards we use support 100 Mbps Ethernets. They are based on the DEC 21140 AE ("Tulip") controller chip. Since the machine has only 4 PCI slots on its motherboard, we had to use an additional PCI bridge (DEC 21152) for connecting 7 Ethernet cards. In our experimental setup (Figure 2), 6 Ethernets are used as client networks, 4 of them are connected with the motherboard through the additional bridge. The seventh Ethernet connects the hit-server with the miss-servers.

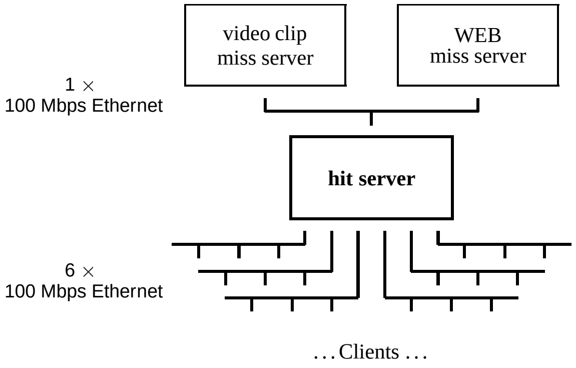

Figure 2: Single-Hit-Server Architecture.

For increased numbers of miss-servers and hit-servers in a server cluster, the inter-server network hardware can be upgraded: multiple Ethernets for point-to-point connections, an ATM switch or a Myrinet. Since the inter-server network connects only 2 to perhaps 15 nodes, the related costs are economically feasible.

Pre-Implementation Performance Analysis

From the performance point of view, the most relevant operations are delivering objects to clients and receiving requests. We start with an idealistic and optimistic *pre-implementation analysis* of these both basic functions. The purpose of this analysis is twofold: (1) estimate an upper bound of the achievable performance; (2) identify the system's potential bottlenecks. Of course, the thus determined idealistic performance is in practice not completely achievable. Nevertheless, it gives us a reasonable order-of-magnitude goal and helps us to concentrate on the relevant optimizations in the design. Furthermore, this methodology helps us checking whether the theory, i.e., our understanding of the system, is in accordance with the reality of the system. If later performance experiments roughly corroborate with the idealistic predictions, we have a certain confidence about theory and implementation. If experiments largely diverge with our theory, we either have the wrong model or made mistakes in implementing it.

Even if an ideal implementation of the hit-server core would spend no time for bookkeeping and OS overhead, sending and receiving packets through the Ethernet controller are unavoidable. So we first analyze the optimal costs for sending a packet. Figures 3 and 4 illustrate the interaction between processor and Ethernet controller.

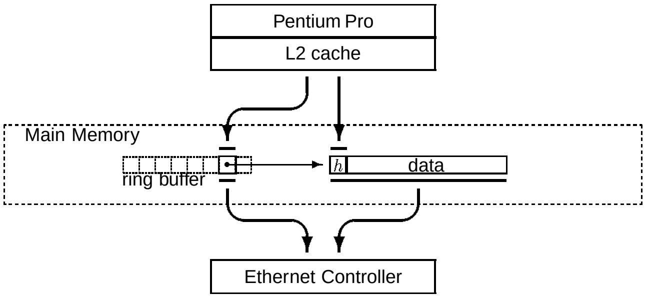

Figure 3: *Sending an Ethernet packet.* The *ring buffer* holds descriptors pointing (thin arrow) to the packets that the Ethernet controller should transmit. For each packet, the processor first writes the descriptor and the packet header; then the Ethernet controller reads the descriptor and the whole packet. Memory reads and writes are denoted by thick arrows.

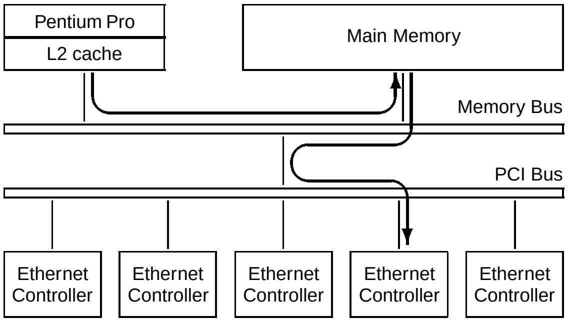

Figure 4: *Accessing Main Memory.* Processor read/writes use only the memory bus while transferring data to or from Ethernet controllers involves PCI bus and memory bus.

Both components communicate via the main memory:

the processor accesses the main memory through the memory bus and the Ethernet controller through the PCI bus *and* the memory bus.

For this analysis, we assume that the packets consist of 32 B header information and 1024 B object data. Transmitting a packet requires the following steps:

1. The system has to receive a device interrupt that indicates that the controller is ready to accept another packet. (Alternatively, the system has to poll the controllers, i.e., needs to read their status registers.) To generate the interrupt, the controller sends one word through the PCI bus. Furthermore, it writes its current status word, another PCI-bus write operation.
2. To ensure that no errors are pending, the system has to read the controller status (see step 1) from memory. Note that this is always an L2-cache miss, since the controller writes to memory and not into the L2-cache.
3. The system has to select a new packet. Under ideal assumptions, no L2-cache miss occurs for this. The main memory is not accessed.
4. The system has to prepare the new packet for transmission. This includes at least writing the client's Ethernet/IP address into the packet header: one cache line has to be written back.
5. The transmission has to be set up. For this purpose, the descriptor in the ring buffer has to be written. It needs at least the physical memory address of the new packet: one memory access. Furthermore, the Ethernet controller needs to be triggered for starting the transmission (one PCI word). Afterwards, the controller will read the according descriptor from the ring buffer, i.e. from memory: 4 words through the PCI bus.
6. Finally, the controller will transfer the packet from memory to its own bus: 256+8 words through the PCI bus.

In Table 1, the costs of these six steps are estimated and given for the critical components: processor, memory bus, PCI bus and Ethernet buses. Due to buffering and pipelining, these components can to a large degree work in parallel. However, main-memory reads through the PCI bus always require corresponding memory-bus activity.

Send Packet (1056 bytes)						
	processor [μs]	*lines by cpu*	**Memory** [μs]	*words by PCI*	**PCI** [μs]	$7\times$ **Ethernet** [μs]
1) device interrupt	1.90	–	0.30	2	0.48	–
2) inspect controller	0.40	1	0.29	–	–	–
3) select packet	0.10	–	–	–	–	–
4) prepare packet	0.57	1	0.29	–	–	–
5) setup transmission	0.72	1	0.57	5	0.36	–
6) transmit packet	–	–	12.09	264	12.09	84/7
total	3.69		**13.53**		12.45	12.00
utilization	27%		100%		92%	89%
max achievable rate: $\frac{1056 \times 8 \text{ bits}}{\mathbf{13.53\,\mu s}}$ = **624 Mbps**						

Table 1: *Pre-Implementation Micro Analysis for a Hit-Server.* Processor costs are derived from instruction estimates (disregarding memory costs) and from micro benchmarks of the underlying μ-kernel. Memory and PCI-bus costs are calculated from the derived number of transfers and the average throughput costs of these transfer measured by micro benchmarks. Ethernet costs are derived from the specified throughput of 100 Mbps.

3.2 Implementation Techniques

From the previous analysis, we know that every 14 μs a 1 K packet has to be sent to achieve maximum hardware utilization. However, we had to design the system for an even higher demand: with optimal PCI-bus DMA hardware (1 word per 30 ns cycle), a packet had to be transmitted every 8 μs.[3] Obviously, the hit-server software must be constructed carefully not to delay packet transmission. The following paragraphs discuss the methods we used to achieve these goals.

Early Evaluation

Early evaluation is a technique for reducing the latency and improving the throughput of operations. If an operation is requested several times on the same object, it needs to be executed only once. If an operation can be executed either at a place or at a time when free resources are available, its costs are hidden.

Object precompilation ensures that only negligible computations or data transformations are required by the hit-server for delivering a cached object to a client. When loading the object into the hit-server cache, the miss-server partitions it into network packets, generates the appropriate header information and computes a client-independent digest for each packet. On sending a packet, only the destination address, sequence number and sometimes a message-authentication code have to be generated.

To meet our security requirements, any delivered data has to be secured by a client-specific message-authentication code which is also unique in time to prevent replay attacks. By using a client-specific secret key, the authentication code is calculated from the precompiled client-independent digest and a nonce.

Many research experiments show that avoiding unnecessary data copies improves performance,(e.g., Fbufs [6], Unet [19]). Since we use precompiled packets in the object cache, we can always use unbuffered transmission

[3] From the above discussion, we know that transmitting a 1024 B packet and a 32 B header requires $(1 + 1)_{step1} + (1 + 4)_{step5} + (256 + 8)_{step6} = 271$ word transfers. So in the ideal situation, 1 word per 30 ns through the PCI bus and no delay by the memory bus, 217×30ns = 8.13μs.

for delivery. With *put* operations, the arriving 1 K packets are linked, not copied, into the new version of the object (see also Section 2.3).

Per-Object Address Spaces

The default *putget* operation is implemented by the hit-server in a single address space. Since we would like to enforce security requirements on active objects, they execute their specialized operations in per-object address spaces.

Remember that the object granularity for *put* updates is 1 K while hardware pages are 4 K. Therefore, once a 4 K region of an object with a non-default *putget* operation is no longer physically contiguous, the corresponding page must be removed from the object's address space. Upon the page fault on this page, the corresponding 1 K chunks are physically copied into a fresh 4 K page frame which is then mapped into the object's address space. Object-specific *putgets* often do not read the entire object data by itself but simply specify to the hit-server core what parts should be transmitted. In these cases, the above mentioned lazy-copying technique avoids copying of received *put* data packets even for non-default objects.

The μ-kernel can be configured to support up to 65,000 address spaces. The space costs per address space are low for small objects, about 22 bytes for an object of 16 K. Nevertheless, the maximum number of address spaces is only 6.5% of the maximum number of objects. Currently, we do not yet know whether this is in practice sufficient to preallocate and preconstruct an address space for any object that uses non-default *get* and *put* operations. Otherwise, the hit-server would have to multiplex address spaces for active objects.

Minimizing Memory Conflicts

In the hit-server case, copying data in main memory is not only "in principle avoidable" but belongs to the class of the most expensive operations. Section 3.1 illustrated that the memory bus is the time-critical bottleneck. *Therefore, processor accesses to main memory have to be minimized.* Since L1 and L2 caches use a write-back strategy, the processor's reads and writes are uncritical as long as they hit in the hardware cache and do not touch the memory bus.

Fortunately, early evaluation techniques, in particular object precompilation, prevents unnecessary memory-to-cache copies. Since we use a precalculated digest, there is no need for the default *get* operation to read the object data.

The code segments of the μ-kernel and the hit-server core are small enough so that their frequently used parts fit completely even into the L1 cache. The hit-server's memory bus is not burdened with handling instruction misses. For data, the situation is different:

1. *Client descriptors*, basically the client's secret key and some status information, are not expected to cause frequent L2-cache misses, since they need only 2 cache lines per client. 400 simultaneously active clients need 10% of the L2 cache.

2. *Hash and name table* serve to identify a requested object. Since they are large, most accesses will cause L2-cache misses. Finding a 100-byte name then requires to read 1 line of the hash table and 4 lines of the name table, provided there is no name-hash conflict.

3. *Object descriptors* will frequently miss the L2 cache. Due to the large number of objects, we generally assume that the object-descriptor data is never found in the L2-cache upon a *get* request. Applications with many hot-spot objects will perform slightly better. Memory accesses are minimized by keeping object descriptors small:

   (a) The *object root* holds pointers to the object-specific operation, the object-page descriptor list and the object's ACL. 2 cache lines are accessed per request, one from the ACL and one to find the packet descriptors.

   (b) Any *object-page descriptor* contains the pointer to 4 packets forming the page and the corresponding 4 precalculated digest values. Together with links and a length field (objects can be smaller than a page) this fits into one cache line.

4. *Object data* is never read by default *get* operations. So no L2-cache misses occur in this case. Writing an object using a *put* operation requires message-authentication codes of the received data to be verified. This costs $1024/32 = 32$ cache misses per 1 K packet. (Checking the authentication code is omitted if the packet comes from a miss-server, since miss-servers are trusted and the inter-server interconnection is a closed network.)

5. *Packet headers* have to be constructed per packet transmission. For our hardware, packet header and the buffer descriptor required for the Ethernet controller together fit into one cache line. Since the Ethernet controller always transfers data from/to main memory, one L2-cache-line write and one read is required per transmission.

6. *Request data* is placed in main memory by the receiving Ethernet controller. Obviously, a request packet has to be read by the hit-server. For requesting an object with a 100-byte name, 4 cache lines have to be transferred.

Therefore in total, a default *get* on an n K object requires at least $11 + \lceil 9n/4 \rceil$ cache-line transfers between L2 cache and main memory.

4 Performance

In two throughput experiments, *get* requests for objects of 10 K and 1 K size are generated at the highest possible rate. A single physical client sends requests for randomly chosen virtual clients. (The Ethernet driver and the hit-server software are constructed such that, provided the utilized bandwidth does not exceed that of one card, there is no measurable difference between the packets coming through a single or through multiple cards.) To ensure an infinite burst, the request generator does not wait for a request to be completed. Avoiding the request-generation problems mentioned in [1], this method is good for generating up to 161,000 requests per second. To be sure to saturate the hit-server in a realistic way, we send requests with random gaps such that the hit-server gets slightly more requests on average than it completes. For the experiments, all objects were resident in the hit-server so that no miss-server communication was required.

For 10 K objects, we achieve a throughput of 594 Mbps with the current hardware, 7,000 transactions per second. For 1 K objects, the bandwidth is 304 Mbps, approximately 36,000 transactions per second. Note that all delivered data is authenticated. Corroborating these results, our video-clip server can serve up to 402 clients with different MPEG I video clips (1.5 Mbps) simultaneously.

The crucial question is of course whether our approach performs significantly better than conventional server architectures. Therefore, we compare Lava with some other research systems and commercial Web servers *in our envisioned LAN scenario* (many NCs in a local cluster). The reader should note that the results cannot simply be extrapolated to other application fields, e.g., wide-area networks. In particular, the systems are not functionally equivalent: e.g., the commercial Web servers support TCP clients but cannot be customized like the Lava architecture.

Figure 5 illustrates that for our LAN-based application Lava's hit-server offers an order-of-magnitude increase over conventional systems. We report net data throughput, i.e., do not count headers etc.

The numbers for Harvest [3] and Cheetah are taken from [11] and converted to Mbps. Both systems run on a 200 MHz Pentium Pro machine like the Lava hit-server. Harvest runs on top of the BSD operating system, Cheetah on top of MIT's Xok system. IO-Lite [16] runs on a 233 MHz Alpha station with two 100 Mbps network adapters. For comparison with industry-standard servers, we include numbers for Microsoft's Internet Information Server 4.0 running on a 166 MHz Pentium with Windows NT 4.0. Afpa [15] is an NT-based server running on a Pentium 166 processor. Both the MIIS and Afpa peformance have been measured in our lab by eliminating all client and network bottlenecks and increasing load until server was saturated. Similar to our experiment, the measurements for Cheetah, Harvest, MIIS, and Afpa are based on a LAN with no competing traffic; *get* requests are always served from the systems' respective main-memory caches. All systems implement the HTTP functionality; for Lava however, the client library translates HTTP requests to object-protocol requests on the client side.

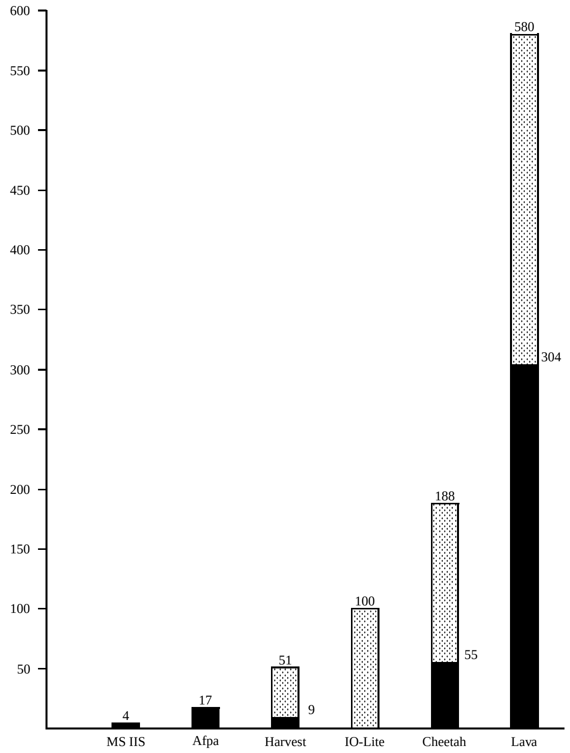

Figure 5: *Server Throughput.* Net data throughput in Mbps for get operations. (Headers, checksums, etc. are not considered to be net data.) Black bars denote the throughput for 1 K objects, shaded bars for 10 K objects. All systems are measured on a local area network and deliver data from their respective main-memory object cache.

Based on the get-throughput experiment, we simulated a system where a hit-server is used as a boot server for 1000 NCs. For booting, each NC had to download an individual set of objects, together 10 Mbyte per NC. We assumed that all 1000 NCs are turned on within the same 5-minute interval, equally distributed over time. We further assumed that each NC, once it is booted, starts working and then every second *get*s a 20 K object. When a user turns on her/his NC, how long would (s)he have to wait until the 10 M of boot data are downloaded? We found an average boot latency of 1.7 s with a standard deviation of 0.9 s (see Figure 6).

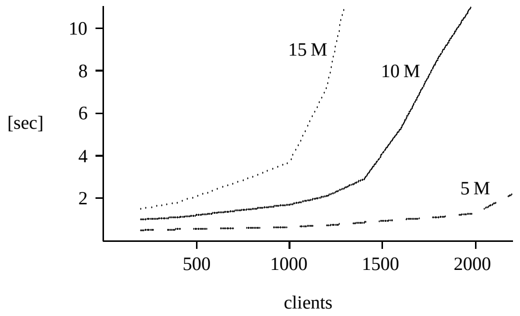

Figure 6: *Average Boot Latency.* All clients boot within the same 5-minute interval. Boot data is client-specific but of equal size for all clients (5 M, 10 M, or 15 M).

Miss handling does not substantially degrade the hit-server's throughput, since most of the work (loading the object) has to be done by the miss-server. Only during transferring an object from the miss-server into the hit-server, the hit-server's delivery rate decreases by 18%, basically because the miss-server communication consumes 100 Mbps from the total transfer bandwidth. (Transferring a 1 M object takes approximately 0.1 s.) Table 2 shows miss costs relative to hits for a single-disk Linux file system used as miss-server.

	1 K	10 K	100 K	1 M
miss latency	20 ms	21 ms	35 ms	162 ms
hit latency	0.1 ms	0.9 ms	9 ms	86 ms
hit : miss latency	1 : 200	1 : 23	1 : 3.9	1 : 1.9
hit : miss bandwidth	1200 : 1	140 : 1	22 : 1	10 : 1

Table 2: *Hit/miss costs.* The miss server file system runs on a 166 MHz Pentium with a Caviar 33100 disk. All data reflect ideal situations in which requests are not delayed by competing requests. Congestion at the miss-server or at a hit-server's Ethernet card would increase the latency. For the bandwidth ratios, we assume that the hit-server concurrently delivers objects of the same size on all cards.

For local networks, the throughput experiment gives some evidence that the Lava architecture enables an order-of-magnitude larger server/NC configurations than conventional server architectures. Whether the architecture can be modified to work efficiently in a wide area network is an open research problem.

5 Scalability

For many applications, a single hit-server might support up to 1000 clients. Future 66 MHz-PCI devices and 100 MHz memory buses might perhaps enable twice as many clients. However, for achieving our original goal of more than 10,000 clients, we must scale hit-servers. Miss-servers and hit-servers may also be scaled to reduce the miss latency or increase the effective cache size.

5.1 Adding Server Modules

There are three methods for adding server modules: (a) add miss-servers either to decrease miss latency or to handle heterogeneous objects, (b) add hit-servers to increase the cache size and hit rate, and (c) add hit-servers to improve the total bandwidth and handle more clients.

(a) Scalability is aided by an explicit separation of miss-server and hit-server hardware: miss-server CPU and IO consumption does not degrade hit-server throughput. Miss handling only influences the throughput of the hit-server when the miss-server stores an object into the hit-server cache.

(b) For certain cases, the bandwidth of a single hit-server might be sufficient but its main-memory cache might be too small for the application's working set. In particular, this can happen if the hit-server's motherboard supports less memory than the processor can address. Then, multiple hit-servers can be used to increase the object cache. Each hit-server holds the entire cache directory but the cached objects are partioned among all hit-servers. Client requests are multicasted to all hit-servers. If the request hits, the according hit-server executes it; the other ones classify the request as a hit but ignore it since they do not have the object. If a request misses, all hit-servers detect a miss and a dedicated hit-server signals it to the miss-server. This one then selects a hit-server for loading the new object. Fortunately, no complicated consistency protocol is required for this type of scaling. Network load, miss-server load and hit-server bandwidth are identical to the single hit-server case; only the resulting cache size is increased.

(c) When the number of clients becomes too large, hit-servers must be scaled to increase the total bandwidth of the system. This is simple as long as all objects are read-only. As soon as objects are write-shared between multiple hit-servers, we need consistency protocols.

5.2 Consistency

For sake of customizability and extensibility, the hit-server provides a consistency *mechanism* from which *per-object* consistency protocols can be implemented by miss-servers. So policies can be fully customized.

Unlike a hardware bus, a LAN does not enable snooping-based solutions for coherency. Instead, we enable the use of miss-servers as arbiters that can coordinate conflicting accesses to objects that are shared by multiple hit-servers.

The hit-server provides a single basic consistency mechanism: per-object *consistency-action matrices.* Two status bits are managed per object: *accessed* is set for any operation, *dirty* is set when a *put* operation occurs. The hit-server never resets these bits. The miss-server, however, can arbitrarily change them. Combination of the four states with the two possible operations *(get/put)* leads to a 2×4 consistency-action matrix. Miss-servers specify a consistency action for each of the 8 fields for each object (in an object descriptor's consistency-matrix attribute). Four different consistency actions are available:

Ignore. The *get/put* operation takes place without involving the miss-server.

Notify. The object's miss-server is notified about the *get/put* operation. This notification is non-blocking. The miss-server will be informed concurrently to serving the client's request.

Call. The object's miss-server is called before the *get/put* request is served. The request blocks until the miss-server grants or denies it. In its reply, the miss-server can define new settings for the object's *accessed/dirty* bits and the consistency-action matrix. Before replying to a call, the miss-server can itself read the object back or update its value. Call-associated actions are completely controlled by the corresponding miss-server, and have a higher latency than ignore-associated and notify-associated actions.

Propagate. Any *put* operation is directly propagated to the corresponding miss-server (*put-through*). The data received from the client is sent to the miss-server and concurrently used for updating the object in the hit-server. Receive and propagate activities are pipelined; however, the client is not acknowledged until the miss-server commits or aborts the operation. Prior to handling a client's *get* request, the hit-server itself "*gets*" the newest version of the object from the corresponding miss-server (*get-through*). (*Get* requests include no data transfer if the requestor already holds the current object version.) To minimize the latency, receiving the new object data from the miss-server and propagating it to the client overlap in time.

The consistency-action matrix is a generic mechanism that can be used to implement a variety of different cache-consistency protocols and also cache-replacement protocols.

For a simple *write-through* policy, *(i,i,i,i / p,p,p,p)*[4] could be used: get operations on this object do not involve the miss-server whereas any put operation is directly propagated. For implementing *write-back* together with LRU replacement, the miss-server can use *(n,i,n, i / n,n,n,i)* . Then the miss-server is notified (a) when the object is accessed the first time (read or write) and (b) when the object becomes dirty (first write). Subsequent accesses do not involve miss-server notification. For LRU bookkeeping, the miss-server will periodically reset all objects to *unaccessed* that have been accessed in the meantime. New accesses are then signaled (once per period) to the miss server. Inactive objects do not incur bookeeping overhead since there is no need for the miss-server to scan them periodically.

For consistency in a multi-hit-server system with write-shared objects, a MESI-like policy can be used:

(i ,i,i,i / i,i,i, i)	*(i ,i,i,i / c,c,c, c)*	*(c ,c,c,c / c,c,c, c)*
for **M** or **E** objets	for **S** objects	for **I** objects

Modified (M) and exclusive (E) objects reside only in one hit-server. Clean objects are called E, dirty ones are called M. Accessing an M or E object does not involve miss-server activity. Shared (S) objects can reside in multiple hit-servers: *get*s happen without involving the miss-server whereas *put*s invoke a *call* consistency action. As a result of the *call*, the miss-server can invalidate the replicas of the according object in all other hit servers by changing their respective consistency-action matrices or even by removing them completely. For classical MESI, the miss-server will afterwards change the original object's consistency-action matrix to the M state and permit the *put* operation. Invalid (I) objects *call* the miss-server for any access so that this one could update the object replica and permit the access or simply delay it by delaying its reply.

6 Related Work

Internet caching has attracted a substantial amount of work. Web proxy caches, e.g. the original CERN web cache [14], are client-oriented, while hierarchical internet caches like Harvest/Squid [3] aim at reducing both backbone traffic and end-to-end network latency. As Kroeger *et al.* [12] and Duska *et al.* [7] report, Web hit-rates on a wide-area network are only about 40% to 50%; latency can be reduced by 30% to 60%.

We use caching for a completely different purpose. Instead of reducing wide-area traffic and network latency, we aim at improving server latency and server throughput. In a way, that is similar to Iyengar and Challenger [8] who concentrate on how to use caches on a server to improve the generation of dynamic Web pages.

Server operating systems have been discussed recently by Kaashoek et al. [10]. Cheetah's design exploits the underlying characteristics of the Exokernel [11] to construct servers that can access the hardware at low overheads. Cheetah uses kernel extensions to achieve high performance. Our hit-server runs entirely at user level. In some sense, our active objects are similar to ASHes [20] but are geared to multimedia documents, provide an object-oriented structuring technique, run entirely at user level and use supervised IPC.

The facilities we used to securely execute the custom methods of active objects could be used securely support ActiveIP [21]. In this sense, our techniques could be used to build an extremely fast router. One key difference between activeIP and our techniques is that since we use

[4] We denote a consistency-action matrix by **(***get-clean-unacc, get-clean-acc, get-dirty-unacc, get-dirty-acc / put-clean-unacc, put-clean-acc, put-dirty-unacc, put-dirty-acc***)**, where the first four entries specify the actions for get operations, the second row for put operations. Consistency actions are qualified by their first letter, *i, n, c,* and *p*.

hardware based protection and fast authorized IPC, our code does not need to be interpreted to be supervised but regular binaries can be used instead.

ADC [5] and Fbufs[6] are techniques used to improve the performance of network protocols and drivers for high speed networks and focus on reducing the number of data copies. We use similar techniques but go beyond them as we concentrate more on server throughput and network scheduling, rather than point-to-point network protocol throughput.

7 Conclusions

We have described the centerpiece of a server architecture for designing high-performance LAN servers. The server is separated into generic cache modules and custom modules. The generic module is policy-free and implements general and optimized mechanisms to handle cache requests. The custom modules can enforce arbitrary policies on the management and use of the cache (including authentication and object consistency). For cache-friendly applications, the resulting servers can perform an order-of-magnitude faster than existing systems. Since the custom modules can implement application-specific cache management policies, the likelihood that an application is cache-friendly can be increased.

We have learned that reducing main memory conflicts between the CPU and the network conrollers is the key to achieve high network throughput. Therefore, we provide a detailed examination into how the cache module must be designed to make efficient use of all hardware components.

There remain many open questions about how to use this server architecture. It is difficult to determine the cache-friendliness of applications designed to support thousands of clients in a laboratory setting. In the future, we plan to investigate the breadth of applicability of this architecture to current applications and investigate new classes of applications that may be enabled by our architecture.

Acknowledgements

We thank Rich Neves for the Afpa and MIIS measurement data and Yoonho Park for multiple discussions. We furthermore appreciate the comments of the anonymous reviewers and Mike Schwartz, our shepherd.

References

[1] G. Banga and P. Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, pages 61–71, Monterey, CA, December 1997.

[2] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-contolled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[3] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrel. A hierarchical internet object cache. In *1996 USENIX Technical Conference*, pages 153–163, January 1996.

[4] D. Cheriton. VMTP versatile message transaction protocol. RFC 1045, NRL, February 1988.

[5] P. Druschel, L. Peterson, and B. Davie. Experiments with a high-speed network adaptor: A software perspective. In *SIGCOMM '94 Conference*, 1994.

[6] Peter Druschel and Larry L. Petersen. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, Asheville NC (USA), December 1993. ACM.

[7] B. M. Duska, D. Marwood, and M. J. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *USENIX Symposium on Internet Technologies and Systems*, pages 23–35, Monterey, CA, December 1997.

[8] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems*, pages 49–60, Monterey, CA, December 1997.

[9] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[10] F. Kaashoek, D. Engler, G. Ganger, and D. Wallach. Server operating systems. In *1996 SIGOPS European Workshop*, September 1996.

[11] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 52–65, St. Malo, October 1997.

[12] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USENIX Symposium on Internet Technologies and Systems*, pages 13–22, Monterey, CA, December 1997.

[13] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[14] A. Luotonen, H. Frystyk, and T. Berners-Lee. CERN httpd. http://www.w3.org/Daemon/Status.html.

[15] R. Neves. Personal communication, March 1997.

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. Technical Report CS TR97-294, Rice University, Houston, TX, 1997.

[17] PCI SIG, Hillsboro, OR. *PCI Specification, Rev. 2.1S*, August 1996.

[18] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local area communication with fast sockets. In *1997 USENIX Technical Conference*, pages 257–274, Anaheim, CA, January 1997.

[19] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain (USA), December 1995. ACM.

[20] D. Wallach, D. Engler, and F. Kaashoek. Ashs: Application-specific handlers for high-performance messaging. In *SIGCOMM '96 Conference*, August 1996.

[21] David Wetherall and David Tennenhouse. The Active IP Option. In *Proceedings of the 1996 SIGOPS European Workshop*. ACM, 1996.