# Adaptive Fast Path Architecture

by E. C. Hu
P. A. Joubert
R. B. King
J. D. LaVoie
J. M. Tracey

**Adaptive Fast Path Architecture (AFPA) is a software architecture that dramatically improves the efficiency, and therefore the capacity, of Web and other network servers. The architecture includes a RAM-based cache that serves static content and a reverse proxy that can distribute requests for dynamic content to multiple servers. These two mechanisms are combined using a flexible layer-7 (content-based) routing facility. The architecture defines interfaces that allow these generic mechanisms to be exploited to accelerate a variety of application protocols, including HTTP. Efficiency is derived from maximizing the number of requests that are handled entirely within the kernel, using a deferred-interrupt context instead of threads wherever possible. AFPA has been implemented on several server platforms including Microsoft Windows NT® and Windows® 2000, OS/390®, AIX®, and most recently Linux. By conservative estimates, AFPA more than doubles capacity for serving static content compared to conventional server architectures, and has allowed IBM to establish a leadership position in Web server performance. A prototype implementation of AFPA on Linux delivers more than 10000 SPECweb96 operations per second on a single processor.**

## 1. Introduction

The essential concept behind Adaptive Fast Path Architecture (AFPA) is to increase network server capacity using two techniques: caching static content in RAM, and serving that content as efficiently as possible from the kernel. AFPA takes each of these two established techniques to new heights. AFPA's central component, an in-kernel RAM-based cache, is intended to store not only the most frequently requested items, but the entire working set of static content. Serving content from RAM is not just a technique AFPA uses; it is what AFPA does. In-kernel implementation eliminates the overhead of switching from the kernel to a user-level context. AFPA further reduces overhead by performing processing in a deferred-interrupt context wherever possible, thus eliminating even the cost of switching to a thread. Building on the idea of in-kernel processing, AFPA features particularly tight integration with kernel components such as the file system and TCP/IP stack.

AFPA includes two components in addition to the RAM-based cache. A split-connection reverse proxy processes requests for dynamic content that cannot be served from the cache. A layer-7 router determines which requests can be served from the cache and which must be handled by the proxy on the basis of a configuration regarding URL paths and content types. An AFPA-based Web accelerator implements a simple Web server in the kernel. This in-kernel Web server delivers static HTTP responses from the cache, thus providing a *fast path* that short-circuits normal request processing at the user level. In the uncommon case of a request that cannot be served by the simple in-kernel Web server, AFPA passes the request to a conventional Web server via the proxy.

This paper discusses AFPA primarily in the context of Web servers. The architecture is, in fact, general-purpose and applicable to essentially any network server that uses a request–response protocol. Other protocols amenable to acceleration by AFPA include FTP, NFS, and DNS. The



**0018-8646/01/$5.00 © 2001 IBM**

architecture includes a framework that provides generic caching, proxy, and application-layer routing mechanisms. Accelerators for specific types of network servers are realized by augmenting the generic framework with protocol-specific support.

AFPA is a *reverse-proxy cache*. An AFPA instance is, therefore, associated with one or more Web *servers*. This is in contrast to a *forward-proxy cache*, which is associated with a set of Web *clients*. A reverse-proxy cache serves the relatively small working set of content associated with its designated servers. A forward-proxy cache, on the other hand, serves a much larger set of content consisting of all items requested by its clients. Forward-proxy caches necessarily store their large working sets on one or more disks. It is generally feasible for a reverse proxy to store its smaller working set in RAM, even though not all do. The effectiveness of AFPA relies on its ability to cache the entire working set it serves in RAM. Unlike many reverse-proxy caches, current AFPA implementations do not cache HTTP responses received from conventional Web servers. Instead, AFPA caches content retrieved via a file-system interface. AFPA is, in fact, not "just" a Web cache, but rather a caching Web server with reverse-proxy and layer-7 routing capabilities.

The primary design goal for AFPA is performance. The architecture and implementation display the effects of an emphasis on efficiency that has been sustained through several cycles of design, development, and performance evaluation. The results of our emphasis on efficiency are summarized in the following six techniques presented in roughly decreasing order of importance: 1) Serving content from RAM eliminates disk latency and bandwidth constraints. 2) In-kernel implementation avoids transitions between kernel and user mode. 3) Tight integration with the TCP stack enables efficient event notification and data transfer. In particular, a zero-copy send interface reduces data transfer by allowing responses to be sent directly from the RAM-based cache. 4) Performing processing in a deferred-interrupt context removes overhead associated with threads. 5) Pre-allocation and recycling of data structures and other resources reduce the lengths of critical code paths. 6) Finally, exploitation of aggregate APIs that perform multiple operations, such as send and disconnect, reduces overhead associated with invoking operating system services.

A comprehensive description of AFPA performance would require evaluation of each of its three main components. Such a complete description is beyond the scope of this paper. The purpose of this paper is to define AFPA and describe its performance in its primary role, which is to serve static content from RAM. Our analysis includes a cursory evaluation of the capacity of AFPA as a proxy. All other aspects of AFPA performance, including loading of content from disk, are excluded from this review.

The initial AFPA prototype was developed by the IBM Research Division on Windows NT** 4.0. The latest version for Windows runs on both Windows NT 4.0 and Windows** 2000 and exploits features specific to Windows 2000 when available. In this document, we use the term "Windows" to refer to both Windows NT 4.0 and Windows 2000. Specific versions of Windows are indicated when relevant. The Windows version of the AFPA cache has been incorporated into two IBM products: IBM HTTP server (based on the Apache Web server [1]) and Netfinity* Web Server Accelerator (NWSA). NWSA 2.0, released in March of 2000, is the most recent product deployment of AFPA. It includes the proxy and layer-7 routing features in addition to the cache. A second AFPA prototype has been developed by our research team on Linux but has not yet been released. The AFPA cache has also been implemented on AIX* and OS/390* by the respective product divisions under the name Fast Response Cache Accelerator (FRCA). The close integration of AFPA with the host operating system has not prevented it from being successfully implemented on several platforms.

The remainder of this paper proceeds as follows. We describe each of the three main components of AFPA—cache, proxy, and layer-7 router—in greater detail. We then discuss the framework by which AFPA provides general-purpose mechanisms in which accelerators for any of a number of network servers can be realized. The following section highlights the flexibility of AFPA by presenting four common deployment scenarios. Next we compare and contrast AFPA with relevant related work. Finally, we present performance results and draw some conclusions.

## 2. In-kernel RAM-based cache

Two of the performance techniques mentioned earlier, integration with the TCP stack and use of deferred interrupts, merit further discussion in the context of serving cached content. AFPA essentially extends the TCP stack. Implementation details vary across operating systems, but the key architectural characteristic is that the TCP stack invokes AFPA directly using a callback interface. When a client request is received, for example, the TCP driver calls AFPA to parse the request. If the request is parsed successfully, AFPA attempts to locate the corresponding cache object. In the typical case for which a matching object is found, AFPA begins sending the response immediately.

Because the TCP stack invokes AFPA directly, all of the above steps—parsing the request, looking up the cache object, and sending the response—occur in the same deferred-interrupt context in which TCP input processing occurs. A deferred interrupt performs tasks deferred by hardware-interrupt routines for devices such as network adapters. Blocking is prohibited in a deferred-interrupt

context, which means that only virtual memory guaranteed not to page-fault can be accessed. Therefore, cache objects can be served from a deferred-interrupt context only if they are pinned in memory. Because the amount of storage that can be pinned at one time is limited, AFPA pins only the most frequently requested cache objects smaller than a threshold size. Less frequently requested and larger cache objects remain pageable and are served by kernel threads.

Typical forward- and reverse-proxy caches retrieve content to be cached from conventional Web servers using the standard HTTP protocol. The AFPA cache, in contrast, is populated with content retrieved via one or more file systems. We now explain why AFPA uses a file-system interface instead of HTTP and describe an important resulting benefit.

Because a forward-proxy cache is not associated with any particular set of servers, it *must* interact with servers using a standard protocol such as HTTP. Caching creates the need to maintain *coherency* between cached data and its source. The HTTP/1.1 protocol includes provisions to help maintain coherency. HTTP responses can include an "Expires:" header that indicates how long the response remains valid. Cached items may be reused until they expire, thus reducing the number of server requests. Caches must validate expired entries before serving them, using a set of conditional headers. These allow a cache to identify the version of an item either by date and time or via an entity tag (essentially a serial number). Inclusion of conditional headers in a GET request causes the server to return an entity body only if all conditions are met. This reduces bandwidth utilization by eliminating retransmission of items that are already cached. These provisions promote coherency, but do not guarantee strict coherency. No provision is made for a server to explicitly invalidate a cache entry if the underlying content is modified before the entry expires.

The association of a reverse-proxy cache with a set of servers allows for closer integration than with a forward proxy. Instead of inventing a new coherency protocol, AFPA leverages existing protocols by accessing data to be cached via a file system interface. This allows AFPA to achieve better coherency than that provided by HTTP. Strict coherency can be achieved on some platforms via tight integration with the file system. On Windows, for example, AFPA exploits a callback mechanism that allows it to invalidate a cache object immediately when the associated file changes. This allows content to be "pushed" to the AFPA cache simply by writing it to a local or remote file system.

### Cache objects

AFPA cache objects combine platform-neutral architectural elements with platform-specific implementation details. Certain aspects, such as the naming scheme, are entirely portable. Other aspects, such as the specific format in which data is stored, vary considerably across operating systems. Cache objects exist to facilitate network transmission of responses to client requests. Network stacks typically require that transmitted data be described using a specific structure such as mbufs for BSD UNIX**, MDLs for Windows, and skbufs for Linux. AFPA cache objects, therefore, store data in the format required by the network stack for the given platform.

Responses to requests for static content typically include potentially dynamic header information in addition to static data. Responses to HTTP/1.1 requests, for example, include a "Connection:" header that indicates whether the connection will be closed after the current response. AFPA allows header information to be dynamically generated and combined with static cache objects. This feature is not required in cases for which a complete response with all required header information can be generated prior to receiving a request.

Sending a stream of data on a TCP connection involves dividing the data into segments and appending TCP, IP, and layer-2 (e.g., Ethernet) headers to each one. Some operating systems allow physically discontiguous data to be "gathered" for transmission in a single network frame. Others require the contents of a network frame to be assembled in physically contiguous memory. On platforms that lack support for scatter/gather I/O, data associated with a cache object is stored as a series of network frames with space allocated in each for TCP, IP, and layer-2 headers. The first frame also includes space for HTTP response headers.

On platforms that support scatter/gather I/O, cached data is stored as a "unit," for example, in a single MDL or mbuf chain. Each such unit may comprise a set of discontiguous buffers, but is treated as a single entity by the network stack. Cache objects that exceed a threshold size are passed to the network stack in chunks, typically 64 KB in size. This eliminates the need for the entire cache object to reside in pinned memory. It can also prevent errors, such as time-outs, that may occur if too much data is passed to the TCP/IP driver at one time.

We now describe two cache-object implementations. File-system cache objects, as the name suggests, are closely integrated with the file system. Pinned-memory cache objects occupy explicitly allocated nonpageable RAM. Relevant aspects of each on Linux and Windows are presented.

#### File-system cache objects

One characteristic that distinguishes cache-object implementations is the level of file-system integration. At one end of the spectrum lies the file-system cache object

that features close file-system ties. AFPA shares responsibility for managing storage backing file-system cache objects with the native file-system cache manager. With some file-system implementations, real memory is allocated implicitly by the file system as file data is read. With memory-mapped files, the virtual memory manager allocates real memory in response to page faults as the virtual address range mapping the file is accessed. In either case, the file-system cache manager works with the virtual memory manager to maintain some portion of real storage as a cache of recently accessed file data.

File-system cache objects leverage management of real memory performed by the file-system cache manager. Such objects are backed directly by file-system buffers. AFPA simply pins cache objects or portions thereof as required to transmit them via the network. This can be accomplished either by explicitly pinning pages in the file-system cache, as with a memory-mapped file system, or by intentionally failing to release file-system buffers after non-memory-mapped files are read. Objects below a threshold size are left pinned for some time after they are requested in order to increase the chances of frequently requested objects remaining resident in physical memory.

By default, AFPA limits the amount of storage it pins to 25% of real memory to ensure that sufficient pageable storage remains available for normal system operation. This guarantees only that AFPA does not pin more than 25% of real storage. It does not prevent the file-system cache manager and virtual memory manager from using more or less than 25% of real storage for the file-system cache.

Leveraging real-memory management performed by the file-system cache manager allows AFPA to influence resource management without assuming complete responsibility for a task it is ill-equipped to undertake. This approach provides a good combination of flexibility and performance. File-system cache objects do, however, have their drawbacks. Reliance on the file-system cache entails accepting its predispositions regarding resource allocation. Operating systems may allocate less memory to the file-system cache than is appropriate for a system whose primary purpose is caching. Another issue, encountered on Windows, is that system page-table entries may be exhausted by mapping files into the kernel address space before available real memory is exhausted. Finally, file-system cache objects require the ability to send response data directly from the file-system cache. Support for scatter/gather I/O is typically needed to implement a zero-copy send interface. AFPA must provide this support on platforms that lack it, such as Linux. (Support for scatter/gather I/O was added to Linux as of kernel version 2.4.)

#### *Pinned-memory cache objects*

Pinned-memory cache objects allow greater control of resource allocation and do not require support for scatter/gather I/O. AFPA explicitly allocates nonpaged RAM for pinned-memory cache objects. The primary questions regarding the implementation of such objects, therefore, concern allocation and management of pinned storage. We describe pinned-memory objects first on Linux, then on Windows.

Linux AFPA allocates a number of 128KB blocks of pinned memory which are managed by AFPA's own memory allocator. The maximum number of blocks is configurable. Once a block is allocated, it is never subsequently freed, although portions of it may be freed and reallocated. When a cache object is created, the corresponding file is opened and read into Ethernet-frame-sized buffers. Space is allocated in each buffer for TCP, IP, and layer-2 (e.g., Ethernet) headers. Additional space is reserved in the first buffer for HTTP headers. When a request is received, the HTTP header is filled in, and each frame associated with the object is queued for transmission. For large responses, frames are queued in 64KB chunks. If an additional request is received for a cache object that is in the process of being sent, an additional copy of the object is made. Old cache objects are freed when memory is needed for new ones. A modified LRU algorithm is used, which prefers to eject objects no smaller than the object being allocated.

On Windows, allocation of pinned memory is handled by the operating system. When a pinned-memory cache object is created, sufficient memory is allocated for it. The corresponding file is then opened and read into the pinned memory. No explicit limit is imposed on the amount of pinned memory which AFPA requests. Pinned memory is simply allocated until further allocations are refused. The operating system fails calls for relatively large amounts of pinned memory before the system runs dangerously low. When a call to allocate pinned memory does fail, a file-system cache object is used instead. Currently, a pinned-memory cache object is freed only when the object is invalidated by changes in the underlying file.

## 3. Reverse split-connection proxy

Having described AFPA's primary component, the cache, we now turn our focus to the reverse proxy. The purpose of the proxy is to distribute requests that cannot be served by the cache to one or more back-end Web servers. This allows AFPA to harness multiple servers in order to generate dynamic content. It also enables many features to be excluded from the in-kernel Web server by allowing complicated requests to be handled elsewhere.

The first step performed by the proxy when handling a request is to select a server. This is a two-part process.

First, the layer-7 router (described below) identifies the appropriate server group, which is the set of servers configured to handle a particular set of requests. The proxy selects a specific server within the group using a weighted round-robin algorithm. Each server has a numerical weight that specifies the number of requests AFPA directs to the given server before sending one or more requests to the next server in the group. The weight is not necessarily the percentage of requests handled by the given server.

After a server is selected, a TCP connection to it must be obtained. The proxy may already have one or more connections established with the server. An established connection is used if one is available; otherwise, a new connection is established. Once a connection is obtained, the proxy sends the request received from the client to the server.

Once the proxy has sent the request, it must convey the response received from the server to the client. AFPA implements a split-connection proxy; i.e., it maintains separate TCP connections with the client and the server. The proxy receives response data on the server connection and immediately queues it for transmission on the client connection. The proxy effectively splices the inbound side of the connection to the server with the outbound side of the connection to the client.

Efficiency is a key consideration when splicing connections. Colocating the proxy in the kernel with the TCP stack allows for efficient event notification and data transfer between the two. Typically, the proxy can be implemented to receive notification of network events, such as arrival of data or a connection request, via a function call. This causes the proxy to operate as an extension of the TCP stack. Notification by function call can be effected by modifying the stack, commandeering function pointers, or registering callback routines like those supported by the Windows Transport Driver Interface. Ideally, the proxy can send data received from the server to the client without having to copy it. Some environments, such as Windows NT 4.0, require a copy. Other platforms, such as Windows 2000, do not.

The final task to be performed by the proxy is identifying the end of each response. The keep-alive feature of HTTP/1.1 allows a client to send multiple requests and receive the corresponding responses on a single connection. Each request can be directed, by the layer-7 router (described below), to a different "destination" via either the proxy or the cache. In order to ensure that each request processed by the proxy is met with a complete response, the proxy must parse responses to determine where they end. Once the end of a proxied response is detected, AFPA marks the client connection as ready to receive a subsequent request. The HTTP/1.1 protocol includes five distinct methods by which a server can terminate a response. The proxy implements a state machine that allows it to identify the end of a response for each method.

Once the end of a response is identified, the proxy disassociates the client and server connections. If the response received from the server includes a "Connection: close" header, or is for a non-keep-alive HTTP/1.0 request, the proxy closes the *client* connection. The proxy places the *server* connection on one of two lists, depending on whether the server indicates its intention to close the connection. The proxy places all connections it expects to be closed by a server on a single "to be closed" list. Connections are removed from this list either when they are closed by the server or when they time out, at which point they are closed by the proxy. If the server indicates its willingness to keep the connection open, the proxy places the connection on a list of available connections established with the given server.

If the proxy is unable to connect to a selected server, it marks the server as "unavailable" to prevent further requests from being routed to it. This is done by setting the server's weight to zero. A connection attempt may fail with either of two errors: connection refused, or time-out. The former indicates that the server hardware is up and reachable but the HTTP server software is not. A connection-refused error causes the proxy to mark the server down immediately. A time-out error provides an ambiguous indication that the server is unreachable, overloaded, or not operational. Marking an overloaded server as down increases the load on the remaining servers, which may ultimately bring them down as well. The proxy marks a server down only if three successive connection attempts time-out. Every ten seconds the proxy attempts to connect to each unavailable server. The server's weight is restored to its original value if the connection succeeds.

The proxy allows AFPA to offload static content from a group of standard Web servers. Interaction between the proxy and the servers occurs via ordinary HTTP requests; therefore, no modification to the software on the back-end servers is required. The proxy can also be configured to route dynamic requests to a single conventional Web server colocated on the same machine. This scenario raises the possibility of closer integration of the AFPA cache with a conventional server.

An alternative to the proxy for the colocated scenario is for the cache to interpose itself into the stream of requests flowing to the user-level Web server. Static requests are harvested from the stream in the kernel and served from the cache. Dynamic requests proceed to the user-level server. This structure can be implemented in several ways. One option is to relink the user-level server with a special socket library. This is viable only when the server can be relinked. Modifications to a server that requires relinking may be motivated by the desire to have

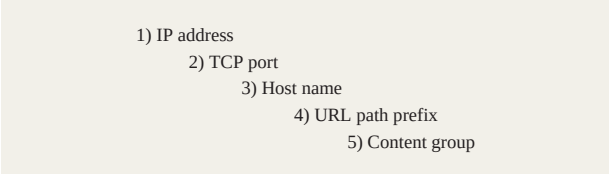


**Figure 1**

Layer-7 routing hierarchy.

the server control the kernel cache. This approach is used by the IBM HTTP server on Windows. Approaches that do not require any modification to the user-level server are also possible. If the socket library is dynamically linked, it can simply be replaced. Some environments, such as Linux, allow functionality to be added to the socket layer by commandeering pointers in the socket structure.

To date, the alternate mechanisms for interacting with a colocated Web server have been used only in conjunction with a relatively simple configuration method and not with the layer-7 router. With the simple method, requests that match both a list of directories and a list of extensions are served from the cache, and all other requests are deferred to the colocated Web server. The alternate mechanisms have been implemented such that once a request is deferred, all subsequent requests received on the same client connection are deferred as well, even if they match both the directory and extension list. This can significantly limit the effectiveness of AFPA when it serves a mix of static and dynamic content. The limitation could be eliminated in a more sophisticated implementation.

## 4. Layer-7 router

AFPA serves static requests from the cache and routes requests for dynamic content to one or more back-end servers via the proxy. The question arises, "How does AFPA distinguish between static and dynamic requests?" The answer involves AFPA's third component, the layer-7 router. The term *layer-7* refers to the uppermost or application layer in the OSI network model. For a Web server, the application protocol is HTTP. The AFPA layer-7 router, therefore, routes requests by examining HTTP headers.

In order to route requests based on HTTP headers, the router must *terminate* the client connection. This is because a client typically does not send an HTTP request until the connection is established. Before the client connection can become established, the layer-7 router must respond to the connection request, which fixes the server endpoint on the router. Layer-7 routing, therefore, requires a split-connection proxy. Terminating each connection at the router and establishing a separate connection with the server entails significant processing. Unlike layer-3 and layer-4 routers, which do not terminate connections, a layer-7 router performs TCP input and output processing. Layer-7 routers can direct requests based on HTTP headers but perform more processing per packet than layer-3 or layer-4 routers, which direct packets on the basis of destination IP address or TCP connections, respectively. The tradeoff is clear: greater functionality or higher capacity. Because of the tradeoff, no single type of router is inherently better than another. Each serves its own purpose. The layer-7 routing capability included with AFPA complements the proxy and cache by allowing requests to be handled by the most appropriate server.

The layer-7 router is configured with the hierarchical set of routing information shown in **Figure 1**. The top three levels of the hierarchy are related to virtual hosting.

The first two levels specify an IP address and TCP port number on which AFPA provides service. These levels allow AFPA to support multiple independent Web sites on different IP address–port pairs. Strictly speaking, the first two levels comprise layer-3 and layer-4 information, respectively. They are included here for completeness. The third level allows AFPA to distinguish requests based on the HTTP "Host:" header, if any. This allows multiple virtual sites to be supported on the same IP address–port pair. Any number of host names can be specified for a given site. Typically, the site's address, specified in dotted decimal notation (e.g., "198.193.16.99"), is included in any name list. This accounts for URLs that specify an address rather than a name. Default values can be used for each of the top three levels. The default IP address is a wild card, causing AFPA to serve the site on all local IP addresses. The default port number is 80, the standard HTTP port. The host name list is also a wild card by default, which prevents the router from distinguishing requests by host name.

The bottom two levels in the hierarchy identify requests within the context of a virtual site. Level four distinguishes requests based on URL path. Requests in the "/cgi-bin/" directory, for example, can be handled differently from those in "/2000/WORLD/europe/." The configured URL path prefix must match the beginning of a requested URL exactly. This provides a reasonable compromise between flexibility and performance. By default, behavior configured for a URL path prefix applies recursively to all subdirectories, but recursion can be disabled. The bottom level differentiates according to the requested file's extension, which identifies its MIME type. This allows files for a given path prefix to be processed according to their type. HTML files, for example, are usually cached, whereas Active Server Pages (ASPs) must be executed on a conventional server.

The same configuration is typically applied to a list of file extensions. For example, a set of extensions is typically cached. To facilitate this, AFPA includes the notion of a *content group*, which is simply a list of extensions. Each content group can then be configured for one or more URL paths on one or more virtual sites. The configuration is checked to make certain no extension is included in multiple content groups that are configured for the same URL path. This ensures that only a single behavior is specified for a given extension within the context of a given URL path. In addition to ordinary extensions such as ".gif," ".jpg," and ".html," two special-case extensions can be included in a content group. The "none" extension specifies requests with no extension, and the "all others" extension denotes all extensions not explicitly included in a configurable extension list.

The parameters for all five levels in the configuration hierarchy define a *request set*. This identifies requests with any of several extensions under a given URL path prefix for a given virtual site. Once a request set is defined, its behavior can be specified. A request set can be either cached or distributed. Cached request sets have two parameters. The first specifies the name of a default file to be served in response to a request that ends in a slash (/). AFPA appends the specified file name to the requested URL. The second specifies the file-system directory associated with the URL path. If, for example, the directory "E:\websites\acme\dogalog\" is associated with the URL path "/catalog/" for some content group that includes the extension ".gif," a request for the URL "http://www.acme.com/catalog/anvils/jumbo.gif" would be served from the file "E:\websites\acme\dogalog\anvils\jumbo.gif." The underlined portion of the URL, which matches the configured URL path prefix, is replaced with the specified file system directory (also underlined). The remainder of the requested URL (after the matching prefix) is then appended.

Distributed request sets have a different set of two parameters. The first specifies a *server group* to which requests in the set are routed. This is a list of servers identified by IP address and port number. Each server has a numerical weight that determines what portion of all requests is routed to that particular server. The second parameter indicates whether or not *affinity* is enabled. Enabling affinity for a distributed request set causes all requests received from the same client IP address to be routed to the same server. This is needed when state information is created on a Web server as a result of processing a request and the state is needed to correctly process subsequent requests. Affinity ensures that each request is routed to the server that contains the appropriate state information.

The hierarchical configuration structure is very flexible. Equally important, judicious use of default values allows simple configurations to be specified easily. A common simple configuration consists of a single virtual site with all default values. Two content groups are configured under the single root URL path "/." One content group contains all extensions to be cached. The other contains all other extensions and is distributed to a single server group. The server group consists of a single conventional Web server colocated on the local machine. This basic configuration can be expanded with additional virtual sites, content groups, server groups, and URL paths as needed.

## 5. AFPA framework

Although our discussion thus far has focused almost exclusively on Web servers, AFPA is, in fact, a general-purpose architecture that can be applied to essentially any network server. The architecture provides a generic framework that promotes efficient processing of network requests. A complete server is realized by adding to the framework support for one or more specific application-layer protocols, such as HTTP. We refer to the component that encapsulates application-specific functionality as an AFPA *module*. On Windows, the AFPA framework and modules are packaged as separate device drivers. On Linux both are implemented in a single kernel module.

AFPA generic functionality includes the in-kernel RAM-based cache and reverse split-connection proxy. Other generic services include resource allocation and deallocation and packet reception and transmission. This generic functionality is provided to modules through the use of exported functions. Most AFPA generic data structures can be extended to include module-specific data. For example, the HTTP module extends the cache-object data structure to allow HTTP header information to be associated with each cache object.

Each AFPA module is required to support certain interfaces and behaviors. For example, an initialization interface is invoked when a module is loaded. This interface must register a set of module entry points that are subsequently invoked by the framework to perform application-specific request processing. One such entry point is responsible for parsing a request; another invokes a framework function to look up a requested item in the cache. While the notion of a layer-7 router is generic, its implementation is heavily dependent on the specific application. Most layer-7 routing functionality, therefore, resides in a module and not in the framework.

AFPA supports an extensible API by which user-level code can interact with the framework and modules residing in the kernel. Modules can extend the API to provide application-specific control operations. The HTTP module supports an operation to pass down a list of extension/MIME-type pairs which is used when HTTP responses are generated. The API is typically implemented using "I/O control" operations supported by most

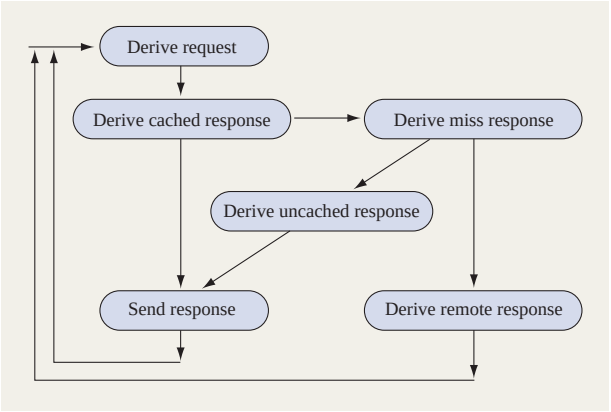


Figure 2

Simplified HTTP state machine.

operating systems. All I/O control operations are handled initially by the AFPA framework. Operations not recognized by the framework are passed to the appropriate module.

Some of the techniques AFPA uses to achieve efficiency require close integration with the host operating system. The implementation of the AFPA framework is, therefore, highly operating-system-dependent. As with any framework, a long-term goal in developing AFPA is to minimize the amount of operating system code needed to implement an AFPA module. Significant progress has been made to date. Ultimately, modules should be largely portable across operating systems.

### State machine

One significant characteristic of the framework is the use of state machines to control request processing. A distinct state machine is typically associated with each type of connection. The HTTP module includes two state machines: one for client connections and another for proxy connections. The state machine approach allows multiple network packets to be processed concurrently on the same connection. Modules are allowed to defer processing of packets or requests to limit concurrency or impose order. The HTTP protocol, for example, requires that multiple requests on the same connection are processed in the order they are received.

We distinguish among three types of state data: global data, which is independent of each connection/request, connection data, which changes on a per-connection basis, and request data, which changes per request within the same connection. For the HTTP module, global data includes server groups and content groups, connection data includes number of processed requests, and request data includes the HTTP version.

Each state in a state machine is represented by a function. These functions execute at one of two processor priority levels: a more restrictive priority level, which prevents process or thread scheduling (i.e., in a deferred-interrupt context), or a less restrictive level, which prevents nothing (i.e., in a kernel thread). On Windows, this two-level model maps to DISPATCH versus PASSIVE. On Linux, it maps to bottom half versus kernel thread. Functions that run at the more restrictive level must avoid page faults, because blocking is prohibited in a deferred-interrupt context.

After a response has been generated, there are two options available to send the response. The first possibility is to send the response directly from the deferred interrupt, the same interrupt as that on which TCP input processing occurs. The second possibility is to produce a work item to a FIFO queue, where it is later consumed by a worker kernel thread. The former approach is more efficient, but can be used only if the response resides in pinned memory. AFPA automatically adapts to serve responses using the most efficient context possible: deferred-interrupt context for pinned memory or thread context for pageable memory.

A simplified version of the state diagram used by the AFPA HTTP module is shown in **Figure 2**. This state diagram is used for client connections. Each state is described as follows:

- *Derive request* The function representing this initial state is invoked each time data is received on the client connection. When enough data has arrived to complete a valid request, the function finishes parsing the request and generates a *request object*. This function runs in the context of the deferred interrupt in which TCP input processing is performed.
- *Derive cached response* In this state, a key from the request object is used to search for the corresponding response in the AFPA cache. This function also runs in the deferred-interrupt context.
- *Send response* In this state, a cached response is sent to the client. The function representing this state runs either in a deferred-interrupt context, when sending a pinned-memory cache object, or in a thread context, when sending an unpinned cache object. This function is provided by the AFPA generic protocol subsystem and exported to the module for its use.
- *Derive miss response* This state is entered if a requested item is not found in the cache. This state performs the layer-7 routing function to determine whether the request should be served by loading an item into the cache or by passing it to a back-end server via the proxy.
- *Derive uncached response* This state is entered if the layer-7 router determines that the request should be served by loading an item into the cache. In this

state, the system creates a cache object, opens the corresponding file, and computes meta-data regarding the object, such as its MIME type. Depending on the implementation, the contents of the file may actually be read at this stage, or the file may simply be memory-mapped.

- *Derive remote response* This state is entered if the layer-7 router determines that the request should be served by a back-end server. In this state, the system establishes a connection with a server (or reuses a previously established connection) and sends the request received from the client to the server. Response data is subsequently sent to the client as it is received from the server. This state is executed at the deferred-interrupt priority level.

We examine two common scenarios for processing requests: one for a cached response and another for a remote response. All client connections begin in the derive request state. Once enough data arrives to form a complete request, the derive cached response state is entered. In this state, an attempt is made to find the response in the AFPA cache. Processing to this point occurs in a deferred-interrupt context.

If the appropriate response is found in the cache, it is immediately sent to the requesting client by entering the send response state. If the corresponding cache object is pinned in memory, the response is sent directly from the deferred-interrupt context. If the cache object is in pageable memory, a work item is created to send the response from a worker thread.

If the appropriate response is not found in the cache, but the data should be loaded into the cache, the derive uncached response state is entered to load the file into the cache. This processing must be deferred to a thread because it cannot be performed in a deferred-interrupt context. Once a cache object has been created, processing proceeds in the send response state as described above.

If the layer-7-based routing information indicates that the requested object should be handled by the proxy, the derive remote response state is entered. Its underlying function routes the request to an appropriate back-end server. Subsequent forwarding of the response is driven by the state machine for the proxy connection.

## 6. Deployment scenarios

Combining the cache with the reverse proxy and layer-7 router provides a great deal of flexibility, which allows AFPA to be deployed in a number of different scenarios. We briefly describe four scenarios to highlight the most typical deployments.

The simplest scenario features AFPA and a conventional user-level Web-server process colocated on a single machine. In this configuration, AFPA increases server capacity by increasing the efficiency with which static requests are processed. AFPA provides a fast path in the kernel that short-circuits normal processing of static requests at the user level. Implementing the fast path entails having AFPA parse each request to determine whether it can be served from the kernel. Requests that cannot be served from the kernel proceed to the user-level server process via the proxy. Adding the fast path in the kernel, therefore, introduces additional parsing and proxy processing to the path for requests served at the user level. This overhead is not significant and is more than compensated for by the increased efficiency for static requests.

The above scenario allows AFPA to be deployed as a single-box solution. High-traffic Web sites typically feature multiple servers that are in some cases specialized for particular purposes. A given set of servers, for example, may serve specific content such as images, advertisements, audio, or video. Dedicating servers to specific content types limits the total working set that must be delivered by any single server and allows the server's hardware configuration to be tailored to its content. One common approach to partitioning content, motivated by resource requirements, is to separate static and dynamic content. Serving static content typically requires more bandwidth and memory than dynamic content, which tends to require greater CPU capacity.

A second typical deployment scenario, usually associated with very-high-traffic Web sites, is for AFPA to be deployed only on servers dedicated to serving static content. This approach is typically viable only when the content of a site has already been manually partitioned among a set of specialized servers. Manually partitioning the content of a site requires not only distributing specific content types to the appropriate servers, but also modifying HTML pages that refer to the content to identify the server for each item delivered by a specialized server. Therefore, administrators may be reluctant, unwilling, or unable to manually partition content for a site after it is operational. Still, the second scenario is attractive for sites whose content is already partitioned.

AFPA's layer-7 routing and proxy features allow it to be deployed without requiring that content be partitioned. A third scenario, also associated with high-traffic Web sites, has AFPA installed on a single front-end server that offloads requests for static content from a set of back-end Web servers. This scenario, depicted in **Figure 3**, is similar to the single-box solution described above, except that the single user-level Web server process is replaced with one or more additional HTTP server boxes. This allows additional hardware resources to be devoted to the site as needed, which may be particularly useful for serving dynamic content. In this third scenario, AFPA's layer-7 router may be configured to direct requests for particular

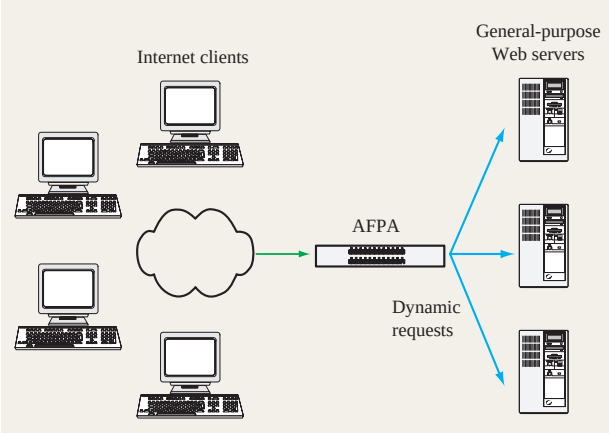


Figure 3

AFPA as a front-end server.

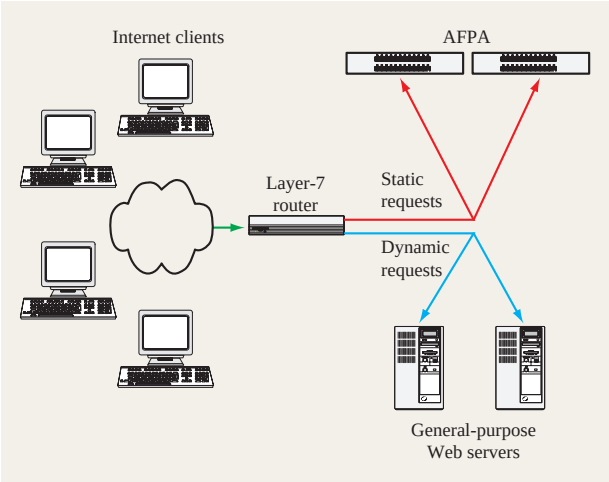


Figure 4

AFPA deployed behind a layer-7 router.

types of dynamic content to specific servers (i.e., content-based routing) or may simply be configured to distribute all requests for dynamic content across all back-end servers.

Although AFPA is capable of distributing requests for dynamic content to back-end servers, serving static content is what AFPA does best. Acting as a proxy for dynamic requests decreases AFPA's capacity to serve static content. This brings us to the fourth scenario, in which the AFPA cache is deployed in conjunction with a separate hardware layer-7 router, as seen in **Figure 4**. Using a separate router conveys the same benefit as using AFPA's layer-7 routing capability by allowing static requests to be served by the AFPA cache while dynamic requests are served by one or more general-purpose Web servers. Offloading the routing function to a separate router allows the entire AFPA server to be dedicated to serving static content.

The use of a separate router brings forth another important benefit. AFPA's capacity improvement depends on its ability to serve static content from RAM. There are limits as to the amount of content that can be served from the RAM of a single server. A separate router can partition a set of requests that exceed the RAM capacity of a single AFPA into multiple smaller request sets, each of which can be served from a single cache. Conceivably, the front-end layer-7 routing function could be performed by another AFPA server configured solely as a layer-7 router. In practice, hardware layer-7 routers provide more capacity than AFPA's software layer-7 routing implementation, and scenarios with such large working sets typically require very high routing capacity.

## 7. Related work

Much current research is focused on improving the performance of content retrieval via the Internet. Common goals are improving performance of origin Web servers, caching static content throughout the network, and directing requests to appropriate servers via content-based routing.

User-mode Web caches employ one of two architectures as defined by [2]: multiple-process/thread (MP) and single-process event-driven (SPED). In the MP model, a server creates a new process or thread (referred to here jointly as a "task") for each new request. Since creating a new task can be time-consuming, most MP caches reduce overhead by pre-allocating a pool of tasks. However, the scheduling overhead caused by large numbers of active tasks can also degrade performance. Apache [1] is the canonical example of MP architecture.

In the SPED model, a single process concurrently processes requests on behalf of multiple clients. SPED caches use asynchronous I/O systems calls, such as select( ) and I/O completion ports, to prevent blocking. Web servers such as Zeus [3], Microsoft Internet Information Services (IIS) [4], and Flash [2] use the single-process event-driven model.

Flash acts like a single-process event-driven architecture when request documents are cached, and acts like a multithread architecture when requests must be satisfied from disk. This is similar to AFPA, which uses a deferred-interrupt context when documents are cached and defers to worker threads when requests must be satisfied from disk or pageable memory. However, Flash is more full-featured than AFPA, since it is a user-mode Web server that can handle both static and dynamic content [2].

Squid is a high-performance proxy-caching server for Web clients, supporting FTP, gopher, and HTTP data objects. It handles all requests in a single, nonblocking,

I/O-driven process. Squid is designed to operate on any modern UNIX system and is an open-source software solution [5].

Web servers and caches are traditionally implemented as user-mode applications. However, in an effort to improve performance, several commercial Web caches have been implemented in kernel mode. The movement of services that are considered integral to a server's operation into kernel mode is not a new concept. Most commercial operating systems include kernel-mode file servers, for instance.

Static-content delivery is essentially a file-to-network copy operation and does not require extensive computation. An in-kernel Web cache can fetch response data from a file system or kernel-mode cache. If the kernel-mode accelerator determines that it cannot serve the request from its cache, it forwards the request to a full-featured user-mode Web cache or proxied server.

Kernel Web caches can be characterized according to the degree of their integration with the TCP/IP stack and their response-processing implementation. The Microsoft Scalable Web Cache (SWC) [6] is tightly integrated with the Windows 2000 TCP/IP stack. By contrast, the Linux kHTTPd [7] uses kernel-mode socket interfaces. Both SWC and kHTTPd handle response processing using kernel-mode threads. SWC supports only HTTP 1.0 and therefore lacks support for keep-alive connections.

TUX [8] is another in-kernel Web cache recently introduced by RedHat on Linux. Like kHTTPd, TUX uses a threaded model, but it offers more features and better performance. First, although TUX uses the file system to cache objects, it has its own cache-directory management, so that the URL-to-file-object resolution is not performed by the file system (which is very similar to the AFPA file-system object architecture on both Windows and Linux). Second, it implements zero-copy TCP send from the file-system memory along with a checksum cache for network adapters that do not support outbound hardware checksumming. Finally, TUX efficiently supports server-side includes for fast dynamic content generation.

The Cheetah HTTP server has been deployed in the Exokernel system [9]. It accelerates page transmission by transmitting file data directly from the file cache and uses precomputed file checksums which are stored with each file.

The locality-aware request distribution (LARD) strategy [10] is a content-based request distribution policy. It consists of a front-end router which uses the combination of the content requested and information about the load on back-end nodes to choose which back-end HTTP server will handle a request. AFPA uses a combination of the content requested and a static weighting on each back-end node to determine which server should process the request. LARD goes a step beyond achieving a balanced load by aiming for locality.

## 8. Performance

In this section, we present experimental results that indicate the performance of the Linux and Windows AFPA implementations along with several other Web cache architectures. We compare AFPA with widely used commercial user-mode HTTP servers: Apache 1.3.9 [1], Zeus 3.3.5 [3], and Internet Information Server 5.0 [4]. We also consider other kernel-mode Web caches: Linux kHTTPd [7] and Microsoft SWC 2.0 [6].

Event notification and data movement are the two areas that have the most significant effect on overall Web cache performance. Event notification uses operating-system-supplied mechanisms for notifying a Web cache of incoming network requests and for requesting responses. For example, select( ) is the event-notification mechanism commonly used on UNIX Web cache implementations, and I/O completion ports are commonly used by Windows 2000 Web caches. A poorly conceived event-notification mechanism causes a reschedule operation for every cache transaction. The ideal event-notification mechanism incurs no context switches or blocking.

Data movement addresses operating system support for the transfer of response data from a Web cache through the TCP/IP stack to the network hardware, as well as the transfer of request data in the opposite direction. Ideal data-movement support provides zero-copy interfaces for all data transfers. However, since the bulk of data movement in a Web caching application is from the cache to the network, the cache/network interface is most critical.

Web cache performance is also influenced by system call overhead, albeit to a lesser degree than event notification and data movement. Using more concise API calls reduces system-call overhead. Windows 2000 AcceptEx( ), for example, combines both the accept( ) of an incoming request and the read( ) of its first data packet into a single system call.

### *Workload*

We use two different synthetic workloads for our experiments: SPECweb96 [11] and WebStone 2.5 [12]. SPECweb96 was the first standard HTTP benchmark. The SPECweb96 working set comprises files that range in size from 100 bytes to 900 KB, where small files are referenced more often than large files (50% of the total number of requests reference files smaller than 10 KB). In addition, the SPECweb96 working set scales with the expected server throughput. In all of our experiments, the entire working set fit into the server's RAM, thus avoiding any performance distortion due to disk accesses.

SPECweb96 has been superseded by SPECweb99 as the industry-accepted Web-serving metric. SPECweb99 incorporates HTTP/1.1 features, such as persistent

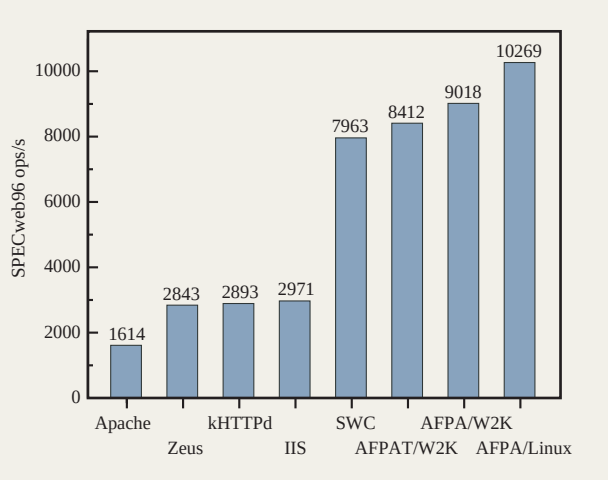


Figure 5

SPECweb96 workload.

connections, as well as requests for dynamically generated pages. Still, 70% of SPECweb99 requests are for static files.

Although SPECweb96 does not take into account some aspects of current HTTP workloads (e.g., no persistent connections, no dynamic content), it is well suited for measuring static-file-serving performance, which is the main purpose of our performance evaluation. Furthermore, large HTTP sites often use several servers that are partitioned into groups serving different types of content such as static files, user logins, and databases. The static-content servers are likely to experience workloads similar to the SPECweb96 workload. Finally, the SPECweb96 execution guidelines are strict enough to allow meaningful comparison of independently reported results. The results presented here do not meet SPECweb96 execution guidelines. We ran the benchmark for the highest load rather than running it for ten evenly spaced loads.

WebStone** [12] is another HTTP server benchmark. Unlike SPECweb96, it allows a user to change the workload characteristics, making it easier to identify performance bottlenecks for given file sizes. For WebStone, our workload consists of fixed-size files, ranging from 64 bytes to 1 MB. Several tests were run, each with a single file size.

### Test environment

The experiments were performed on two operating systems: Windows 2000 Advanced Server (build 2195) and RedHat Linux 6.1 with a Linux-2.3.51 kernel. AFPA for Windows 2000 (henceforth referred to as AFPA/W2K), IIS, and SWC were run on Windows 2000. AFPA/Linux, kHTTPd, Zeus, and Apache were run on Linux. To quantify the benefit of serving responses in a deferred-interrupt context, a version of AFPA (AFPAT/W2K) that does not include this optimization and instead serves all responses on threads was implemented.

All experiments used the same server hardware: an IBM Netfinity 7000 M10. The server was equipped with four 450-MHz Pentium** II Xeon** CPUs, 4 GB of RAM, two 33-MHz PCI buses (one 32-bit and one 64-bit), and four Alteon ACEnic Gigabit Ethernet adapters. For most experiments, only one of the server's four CPUs was used. Two Alteon ACEswitch 180 switches connected ten clients to the server. The clients were IBM Intellistation* Z-Pro (two 450-MHz Pentium II Xeon CPUs, 256Mb RAM) running RedHat Linux 6.1.

All experiments were performed with 9000-byte (jumbo) Ethernet frames. We chose jumbo Ethernet frames rather than standard 1500-byte Ethernet frames, since this allowed our SPECweb96 results to be compared with officially published results [11]. Limited experiments using standard Ethernet frames did not reveal any significant difference in the performance trends seen with 9000-byte frames.

We note the following limitations of our test methodology: All experiments were performed with the same limited number of client machines. Our results focus almost entirely on uniprocessor rather than multiprocessor servers. Experiments were performed solely with nonpersistent connections. Our analysis is constrained to static content only. Finally, results are reported only for the Linux and Windows 2000 operating systems, both running on Intel processors.

On the server side, Linux and Windows 2000, as well as each individual Web cache, were tuned in order to achieve maximum performance. To this end, we used some of the tuning parameters provided with submitted SPECweb96 results. For caches that support time-to-live values for cache objects, we tuned the cache time-out to prevent cache invalidations.

### Cache performance

The results for the SPECweb96 workload are presented in **Figure 5**. Results are presented for Apache, Zeus, kHTTPd, IIS, SWC, AFPAT/W2K (threaded implementation), AFPA/W2K (deferred-interrupt implementation) and AFPA/Linux (deferred-interrupt implementation).

The most obvious characteristic of the graph is that the results fall into two distinct ranges. There is a significant gap in performance, specifically a factor of 3, between kernel- and user-mode Web cache implementations. A second characteristic is that AFPA/Linux performance is close to the theoretical capacity of the hardware. The SPECweb result of 10269 represents more than 1.2Gb/s

server throughput. Although the primary intent was to analyze the uniprocessor performance, we did determine that on Windows 2000 AFPA is 50% faster on one CPU (9018 SPECweb operations per second) than IIS on four CPUs (6090 SPECweb operations per second).

We found kHTTPd to be relatively slow on the SPECweb96 workload as compared to other kernel Web caches. This is because 1) kHTTPd uses the file-system cache as its Web cache, and since the Linux file system does not provide a zero-copy interface, kHTTPd relies on a one-copy send; and 2) kHTTPd uses the socket interface instead of interfacing directly with the TCP/IP stack for networking. Therefore, kHTTPd's performance is not significantly different from that of Linux user-mode Web caches, which are also forced by Linux to use a one-copy send and a socket interface. The primary benefit of kHTTPd's design is that it avoids process scheduling overhead.

As mentioned before, both IIS and Zeus employ SPED architectures. Although Linux does not feature zero-copy send, Zeus was on par with IIS. This somewhat contradicts previous attempts at comparing user-mode Linux and Windows Web caches [13]. Without further investigation, we attribute this to optimizations integrated within the 2.3 Linux kernel [14].

The slowest in all experiments was Apache. This suggests that the SPED architecture used by Zeus and IIS outperforms the MP architecture used by Apache. This is consistent with other published results [2, 13]. Apache appears to be penalized by a significant process scheduling overhead. Note, however, that Apache 1.3.9 does not feature a memory-based static content cache; it uses the file-system cache. Among other optimizations, adding a memory-based cache to Apache reportedly increases its performance by 70% on Linux [15], which would bring Apache in line with IIS and Zeus.

We also note that AFPA on Linux outperforms AFPA on Windows 2000 by 14%. Since the AFPA implementation is the same on both operating systems, this suggests that the Linux TCP/IP stack implementation is faster than that of Windows 2000.

To determine the impact of file size on performance, we next compare the Web caches using a range of file sizes from 64 bytes to 1 MB. The connection rates and server bandwidths are reported in **Figure 6**. For small files, request latency was the dominant performance factor, which was, in turn, determined by the performance of the event-signaling mechanism.

For 64-byte files, AFPA on Linux was 21% faster (12522 requests per second) than AFPA on Windows 2000 (10321 requests per second). Using the Pentium performance counters, we measured the number of instructions executed in both cases. Using exactly the same source code within the deferred-interrupt handler, AFPA/Linux executed 19% fewer instructions than

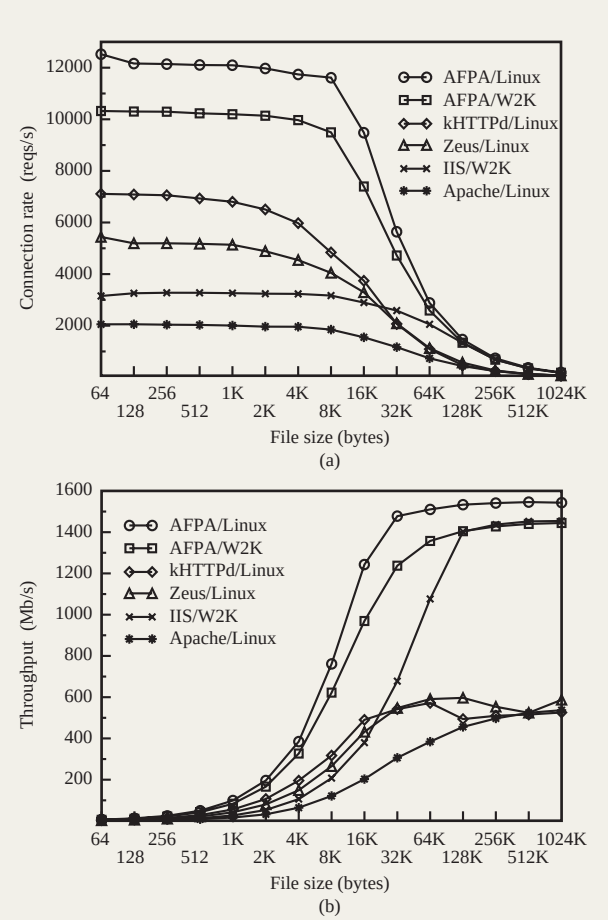


**Figure 6**

Fixed file-size workload: (a) Connection rates; (b) server bandwidths.

AFPA/W2K (26000 versus 31000 instructions per request). Two notable differences between the Linux and Windows implementations help account for this disparity. On Linux, the availability of operating system source code allows a kernel module to be optimally integrated, whereas on Windows 2000, a kernel module is constrained to use the TDI interface. We suspect that the additional 3% difference can be attributed to a larger number of instruction TLB misses (ten per request for Windows 2000 versus zero for Linux). The Linux kernel, TCP/IP stack, and kernel modules are stored entirely in nonpageable 4MB pages, so Linux does not experience any instruction TLB misses. Only the Windows 2000 kernel is mapped using 4MB pages; the TCP/IP stack is not.

Using the Pentium performance counters, we also compared the deferred-interrupt version of AFPA with the threaded version on Windows 2000. For 64-byte files, the deferred-interrupt version was 12% faster than

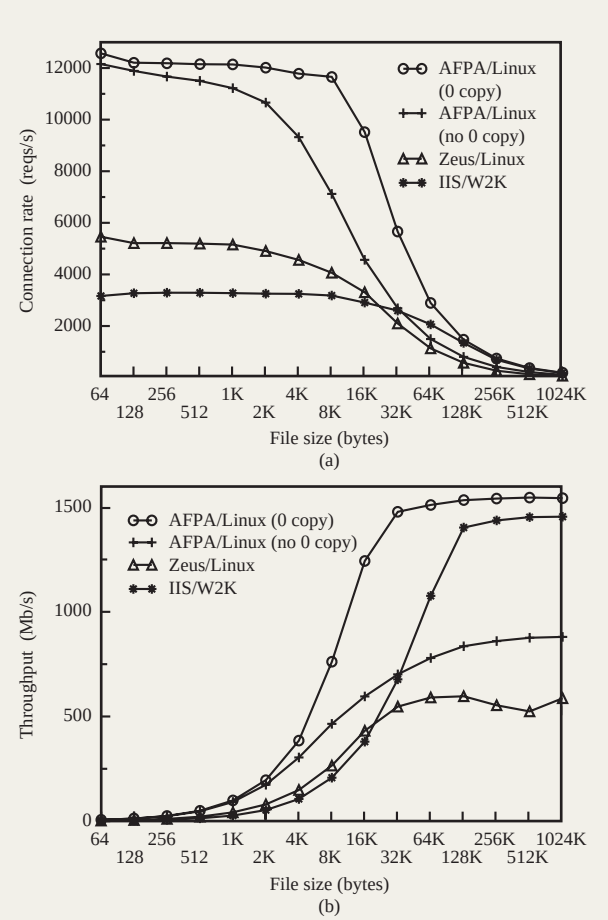


**Figure 7**

Zero-copy TCP performance: (a) Connection rates; (b) server bandwidths.

the threaded version (10321 versus 9209 requests per second). This closely matches the difference in the number of instructions executed. This difference corresponds to the overhead of queuing/dequeuing work items and scheduling the thread.

For small files, the performance of kHTTPd was closer to that of user-mode servers than other kernel accelerators, mirroring the SPECweb96 results. As mentioned earlier, this is because kHTTPd uses the same interfaces (e.g., sockets, file system) as user-mode servers.

For large files, performance is determined primarily by the speed at which the server can move data to the network. As file size increases, the operating system overhead for user-mode servers accounts for less and less in the overall cost of processing requests. This is because processing latency is completely amortized for large files. For example, Apache lags behind the other Web caches

**Table 1** Proxy performance.

| *No. of CPUs* | *Requests per second* | *CPU usage (%)* |
|---|---|---|
| 1 | 3128 | 100 |
| 2 | 3896 | 83 |
| 3 | 4361 | 70 |
| 4 | 4617 | 59 |

for performance on small files, but is just as good as other user-mode Linux servers for large files. On Windows 2000, IIS performs as well as AFPA for files of 128 KB and larger, while it is 3.27 times slower than AFPA for 64-byte files.

For user-mode Web servers, IIS was slower than Zeus for files smaller than 32 KB, but for larger files gained an advantage from having a zero-copy send interface. The importance of a zero-copy TCP send was further emphasized on the throughput graph. There was almost a threefold performance difference between Web servers using zero-copy send interfaces and those using one-copy send interfaces. We analyze this result further in the next section. We also investigated the upper bound on user-mode performance.

It is also interesting to compare the SPECweb96 results with the fixed-file-size results. The average size of a SPECweb96 request is 14.7 KB. On SPECweb96, AFPA/Linux was 3.6 times faster than Zeus, but we did not find such a high ratio on the fixed-file-size workload graph; the ratio varies from 2.30, for 64-byte files, to 2.95, for 512KB files. This seems to indicate that it is neither the connection setup cost nor the bandwidth that limits Zeus' performance. This result shows that SPECweb96 cannot be approximated by simply using fixed-file-size requests. Referencing different URLs has an impact on performance.

Another interesting result is the nearly flat connection rate for transfers of less than 8 KB. Even with jumbo frames enabled, one would expect a more significant decrease in the connection rate. It appears that the Alteon firmware is optimized for bulk data transfers rather than fast connection setup. We ran some tests using a single-client thread requesting 1KB transfers on Alteon, Intel gigabit, and Intel 100Mb adapters. This configuration measures connection latency. We found the Alteon adapter to be between two and three times slower than the Intel gigabit and 100Mb adapters, respectively. The high connection setup cost on the Alteon adapter probably accounts for the flat connection rate.

In order to evaluate the performance gain of a zero-copy send interface in the TCP/IP stack, we ran a modified version of AFPA/Linux that does not use the AFPA zero-copy cache architecture. In this version,

network buffers are allocated through the standard Linux `sock_wmalloc( )` primitive; file data is copied from the AFPA cache into network buffers and checksummed before being sent. **Figure 7** summarizes the performance of these two implementations plus Zeus on Linux (which does not use zero-copy sends) and IIS on Windows 2000 (which does use zero-copy sends through the `TransmitFile( )` API).

As expected, the performance advantage of a zero-copy send interface increased with the file size. It is important to note, however, that the benefits of a zero-copy interface can be seen for relatively small files. For 4KB files the performance difference is 25%; it grows to 111% for 32KB files.

We also ran an experiment on two CPUs with the Windows 2000 implementation. It shows a 30% performance improvement for SPECweb96 with two CPUs. In this situation, the performance is limited by the memory and I/O bandwidth rather than by the CPU. A more precise evaluation of scalability would require different hardware—either faster I/O or slower processors.

### *Proxy performance*

To evaluate the capacity of the proxy to establish and tear down connections, we used the test bed described above, but configured AFPA to distribute all requests to three back-end servers. The back-end servers were IBM Intellistation Z-Pros, with two 450-MHz Pentium II Xeon CPUs, and 256 MB RAM, running Windows NT 4.0 with service pack 6. For this experiment, the clients requested a small set of 102-byte files. Requests from six clients were uniformly distributed across the three back-end servers. The performance results, presented in **Table 1**, indicate a relatively high capacity for establishing and tearing down connections. Capacity is particularly high given that only a relatively small percentage of requests, typically 10–30%, would be handled by the proxy. The results provide little information about scalability because the experimental setup was insufficient to drive the multiple CPUs to 100% utilization.

## Conclusions

We have presented an overview of Adaptive Fast Path Architecture and demonstrated its ability to dramatically increase Web server capacity. AFPA's three mechanisms, the cache, proxy, and layer-7 router, provide a general-purpose platform for accelerating network servers. We continue to refine these mechanisms to further increase performance and to improve AFPA's applicability to other protocols. Our next step will be to deliver performance improvements for dynamic content similar to the gains we have achieved to date with static content.



## References

1. The Apache Group, Apache http Server Project, *http://www.apache.org*.
2. Vivek S. Pai, Peter Druschel, and Willy Zwaenpoel, "Flash: An Efficient and Portable Web Server," *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999, pp. 199–212.
3. Zeus Technology Ltd., Zeus Web Server, *http://www.zeus.com*.
4. Microsoft Corporation, Internet Information Services Features, *http://www.microsoft.com/windows2000/guide/server/features/web.asp*.
5. Squid Web Proxy Cache, *http://www.squid-cache.org*.
6. Microsoft Corporation, Installation and Performance Tuning of Microsoft Scalable Web Cache (swc 2.0), *http://www.microsoft.com/technet/iis/swc2.asp*.
7. Moshe Bar, "kHTTPd, a Kernel-Based Web Server," *Linux J.*, pp. 58–59 (August 2000).
8. "Answers from Planet TUX: Ingo Molnar Responds," *http://slashdot.org/articles/00/07/20/1440204.shtml*.
9. M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie, "Application Performance and Flexibility on Exokernel Systems," *Oper. Syst. Rev.* **31,** 52–65 (1997).
10. Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenpoel, and Erich Nahum, "Locality-Aware Request Distribution in Cluster-Based Network Servers," *Proceedings of the ACM Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Monterey, CA, October 1998, pp. 3–7.
11. The Standard Performance Evaluation Corporation, SPECweb96 Benchmark, *http://www.spec.org/osg/web96*.
12. Mindcraft, Inc., "WebStone—the Benchmark for Web Servers," *http://www.mindcraft.com/webstone*.
13. Mindcraft, Inc., Open Benchmark: Windows NT Server 4.0 and Linux, *http://www.mindcraft.com/whitepapers/openbench1.html*.
14. Joe Pranevich, "Wonderful World of Linux 2.4," *http://www.linuxtoday.com/stories/19955.html*.
15. SGI, Accelerating Apache, *http://www.oss.sgi.com/apache*.

*Received September 11, 2000; accepted for publication March 16, 2001*

**Elbert C. Hu** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (elbert@us.ibm.com).* Mr. Hu received his B.A. degree from Queens College, City University of New York, in 1979, and his M.S. degree from Polytechnic University, Brooklyn, in 1981, both in computer science. He joined the IBM Research Division in 1982 and has been involved in VM/Control Program, TCP/IP, network management, and distributed computing projects. Mr. Hu has received IBM Outstanding Technical Achievement and Outstanding Contribution Awards for his work on Netfinity and TCP/IP projects.

**Philippe A. Joubert** *ReefEdge, Inc., 2 Executive Drive, Suite 645, Fort Lee, New Jersey 07024 (philippe@reefedge.com).* Dr. Joubert received his Ph.D. degree in computer science from the University of Rennes, France, in 1993. From 1996 to 1999, he worked as a marketing product manager for Bull, both in France and in the United States. In 1997, he received the Bull Research and Development Award for work on software high-availability and load-sharing technology. In 1999 he joined the advanced OS Technology group at the IBM Thomas J. Watson Research Center, where he worked on high-performance in-kernel Web caches for the Linux operating system. In 2000 Dr. Joubert joined ReefEdge, Inc., where he is now working on core networking technology for in-building wireless infrastructures.

**Robert B. King** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (rbking2@us.ibm.com).* Dr. King is a Research Staff Member in the Internet Infrastructure and Computing Utilities Department at the IBM Thomas J. Watson Research Center. He received his B.A. degree in mathematics from Oberlin College in 1984 and his M.S.E. and Ph.D. degrees in computer and information science from the University of Pennsylvania in 1987 and 1991, respectively. In 1991 Dr. King joined IBM, working on system design and software implementation for gigabit and ATM network switches. In 1994 he received an IBM Research Division Award for his contributions to broadband network services. From 1996 to 1998, he was a member of the development team for the IBM AntiVirus family of products. Since 1998, he has been working on Adaptive Fast Path Architecture. His current research interests focus on operating systems and network servers.

**Jason D. LaVoie** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (lavoie@us.ibm.com).* Mr. LaVoie received his B.S. degree from Marist College, Poughkeepsie, New York, in 1996 and his M.S. degree from Rensselaer Polytechnic Institute in 2000, both in computer science. He is currently pursuing his Ph.D. in computer science at Polytechnic University. He joined IBM in 1996 and transferred into the Advanced OS Technology Department at the Watson Research Center in 2000. Mr. LaVoie's current research focuses on systems software for accelerating network servers.

**John M. Tracey** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (traceyj@us.ibm.com).* Dr. Tracey is a Senior Software Engineer and Manager, Advanced OS Technology, in the Infrastructure and Computing Utilities Department at the IBM Thomas J. Watson Research Center. He leads the Adaptive Fast Path Architecture project. Dr. Tracey received his B.S. and M.S. degrees, both in electrical engineering, from the University of Notre Dame in 1990 and 1992, respectively. In 1996, after receiving his Ph.D. in computer science, also from the University of Notre Dame, he joined the IBM Research Division as an Advisory Software Engineer. His primary professional interests are in the areas of operating systems and networking. He has contributed to several projects in these fields, including the Network Dispatcher and Netfinity Web Server Accelerator products.