RC23044 (W0312-120) December 23, 2003
Computer Science

IBM Research Report

Kernel-Level Caching of Dynamic HTTP Content from Many Sources

Jason LaVoie, John M. Tracey
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g, payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Kernel-level Caching of Dynamic HTTP Content from Many Sources

Jason LaVoie and John M. Tracey
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
{lavoie, traceyj}@us.ibm.com

Abstract

Network file serving applications often use caching to increase throughput and capacity. Furthermore, caching in the operating system's kernel multiplies these benefits. What these caches lack are the ability to receive or retrieve content from many disparate sources. Adaptive Fast Path Architecture (AFPA) [3], a network server accelerator implemented for HTTP, suffered from this same limitation. AFPA not only required the caching of static content from the file system but dynamic content generated from a myriad of sources.

This paper introduces an architecture designed specifically to address the many combinations of HTTP content sources and caching techniques required by AFPA. N-source In-kernel Cache (NICache) is an in-kernel cache capable of caching both application-specific and application-independent data and delivering it via any application protocol. Data may come from a variety of sources and may be stored in the kernel using numerous kernel specific mechanisms. NICache provides a framework for a generic multiple source, in-kernel cache that provides extensive flexibility without negatively affecting performance for static content and potentially significantly increasing the performance of serving dynamic content. NICache has been implemented on Linux and Windows 2000 for an HTTP reverse-proxy cache, AFPA.

1.0 Introduction

HTTP servers can be accelerated via in-kernel caches caching static files [1, 2, 10]. Adaptive Fast Path Architecture (AFPA) [3] demonstrated the performance gains of caching static HTML files in memory via the Windows file cache and pinned memory[1]. As the World Wide Web has moved towards dynamic content (estimated at 30 percent [4, 19]), the need for caching dynamically generated content has grown. Caching of dynamic content is especially important because it is compute intensive; a dynamic page may require several orders of magnitude more computation time than a static page of comparable size [5]. Dynamic content can be generated from various mechanisms (JSP, PHP, CGI, Apache modules, databases) either locally and remotely. This disparity places the requirement of caching content from many sources on the caches such as AFPA. This content may arrive in different formats and may need to be cached using different kernel mechanisms.

AFPA also includes a reverse split-connection, thereby, forwarding requests to other HTTP servers. Proxy caching has been established as an efficient means of offloading work from busy back-end servers [4], and AFPA, already having the caching capability, had the requirement to cache the eligible responses. The addition of caching dynamically generated content and responses from back-end servers led to convoluted algorithms and structures with type fields for both the source and the storage mechanism. Each cache object, that is, an application specific set of cached data, contained many different elements for each of the different sources of data resulting in wasted memory. A more architected solution was needed.

To solve this problem, AFPA has been enhanced with N-source In-kernel Cache (NICache) - an architecture for building in-kernel caching applications to handle the ever-increasing combinations of sources and storage mechanisms in a flexible and extensible fashion without sacrificing performance. N-source In-kernel Cache separates information about the source of the data from the caching mechanism itself, thus allowing for new data sources or caching techniques to be added with minimal

[1] Pinned Memory, also know as non-pageable memory or non-paged pool, is guaranteed to reside in physical memory thus can be accessed at all times and all IRQL levels. [23]

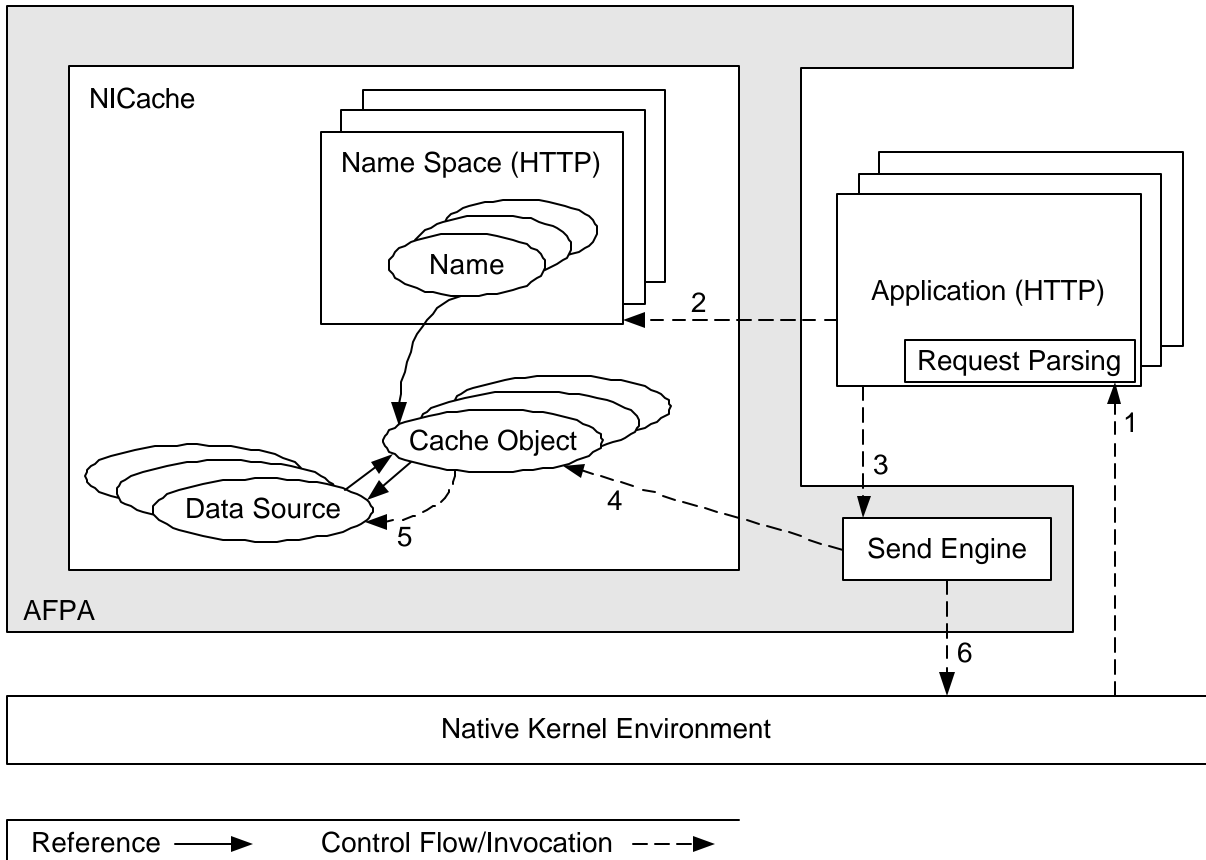


Figure 1: Overview of AFPA with NICache

impact on existing code. The architecture is application protocol and operating system independent and includes policies and APIs for managing the cache. Once data is cached, NICache manages the cached data efficiently and effectively in areas such as expiration and time to live for weak consistency [21] while managing the system resources it uses. Since NICache can choose between different kernel mechanisms for content storage, resource utilization for various resources can be closely monitored and controlled.

Furthermore, caching of entire files and application responses is no longer sufficient. Fragment caching, that is caching portions of HTTP responses, saves a significant amount of CPU time [5]. The NICache architecture allows for the composition of content - either from the cache or retrieved for assembly.

The primary design goal of NICache is achieving flexibility and extensibility without negatively impacting performance. Given NICache's integration with AFPA, performance is always a key concern. Strict resource management and full control of the cache from the application is also important. Meeting these design goals results in an architecture allowing customization of the cache while maintaining performance levels.

In this paper the overall design of AFPA with NICache is introduced, followed by a description of the NICache architecture. After the components are defined, the important relationships and interactions between the components are discussed. Performance data is then provided presenting the impact of NICache, followed by implementation limitations. Lastly, related work and a conclusion are presented.

2.0 Architecture

This section presents and overview of the entire architecture. Subsequently, the NICache architecture is explored in greater detail.

2.1 Overall Design

Figure 1 provides an overview of AFPA including

NICache. The architecture provides a general-purpose framework that is augmented with support for one or more applications such as HTTP or LDAP. We refer to the application support simply as “the application” or, in some cases, “the kernel application” to distinguish it from application functionality implemented at user-level. NICache allows caching of data from multiple sources by separating caching from retrieval. This separation is mirrored by encapsulation of functionality into Cache Objects and Data Sources. The division of responsibility between a Cache Object and Data Source can be highlighted by example. One Cache Object might leverage an in-memory file system cache for backing store, while another could explicitly allocate and manage its own pinned or pageable memory. Different Data Sources might retrieve data from a file system, an HTTP server, and a user-level application server respectively. Any Cache Object can be used with any Data Source.

AFPA provides a name facility that allows one or more application-specific names to be associated with a Cache Object. In the case of an HTTP application, a name might simply be a URL. A more sophisticated name implementation might also allow for constraints on various fields of an HTTP request such as cookie or accept headers. AFPA allows an application to instantiate one or more name spaces in which it can create, delete and look up names. Specific functionality such as comparison of names is provided by the application.

In Figure 1, the processing of a request for static content begins when AFPA receives the request and parses it to obtain the name (URL in this case) in its application-specific request parser (1). Next, the application invokes a routine in NICache to look up the name in the name space associated with the application (2). If the name is found, then a reference to its corresponding Cache Object is returned to the application. Now, the application calls the generic send engine to send the Cache Object (3). The send engine is provided by AFPA for the application to use when sending responses to clients. The send engine invokes a get data routine in the Cache Object to obtain the data for the response (4). If the Cache Object has valid and non-expired cached data, then it will immediately return the content. Otherwise, the Cache Object obtains the required response data by invoking a get data routine in the corresponding Data Source (5). The Data Source retrieves the data from the appropriate source, e.g., a file in the file system or another server via HTTP, and returns it to the Cache Object. The Cache Object subsequently returns the content to the application. Finally, the send engine generates the appropriate [20] HTTP response headers and sends the response (6). The interaction between AFPA, Cache Objects, and Data Sources will be covered in more detail in section 2.2.6.

2.2 NICache Architecture

The requirements for closely tying certain data structures with their behaviors and inheritance within the design led to an object-oriented design. Due to the many-to-many relationship between the sources of content and the storage of content, an object-oriented approach aides in solving many design problems. Figure 2 provides a UML-style [6] object diagram. This section contains a description of each of NICache’s objects followed by a discussing of their relationships. More detail on the implementation of specific derivations of NICache’s primary objects can be found in section 3.0.

Though NICache is discussed in the context of AFPA, the design presented herein can be applied to a variety of network file serving applications. To that end, it is built as a library to be used by in-kernel applications such as AFPA.

2.2.1 Cache Objects

The fundamental object in NICache is the Cache Object. The Cache Object represents the cached form of a single unit of data and supplies functions for efficient access to the content. NICache supports any number of varying types of Cache Objects as long as they inherit from the parent Cache Object. The abstract parent class contains members standard in all Cache Object types, such as reference count, a unique name, locks, access rate and expiration time along with other metadata. Likewise, the parent Cache Object specifies those functions, often referred to as virtual functions, that must be defined by the children Cache Objects. These functions, which will be discussed in detail later, include GetData, Suspend, Cleanup, and Create. Each instance of a Cache Object represents a single logical unit of cached data, such as a file or output from a program, identified by a unique name. A Cache Object’s name will be mentioned throughout this paper; however, its attributes and construction are application specific and outside the scope of this paper.

All Cache Objects have metadata as a member. Metadata has two components: one application specific, the other application independent. The application independent portion of the Cache Object’s metadata contains that information which is generic across all

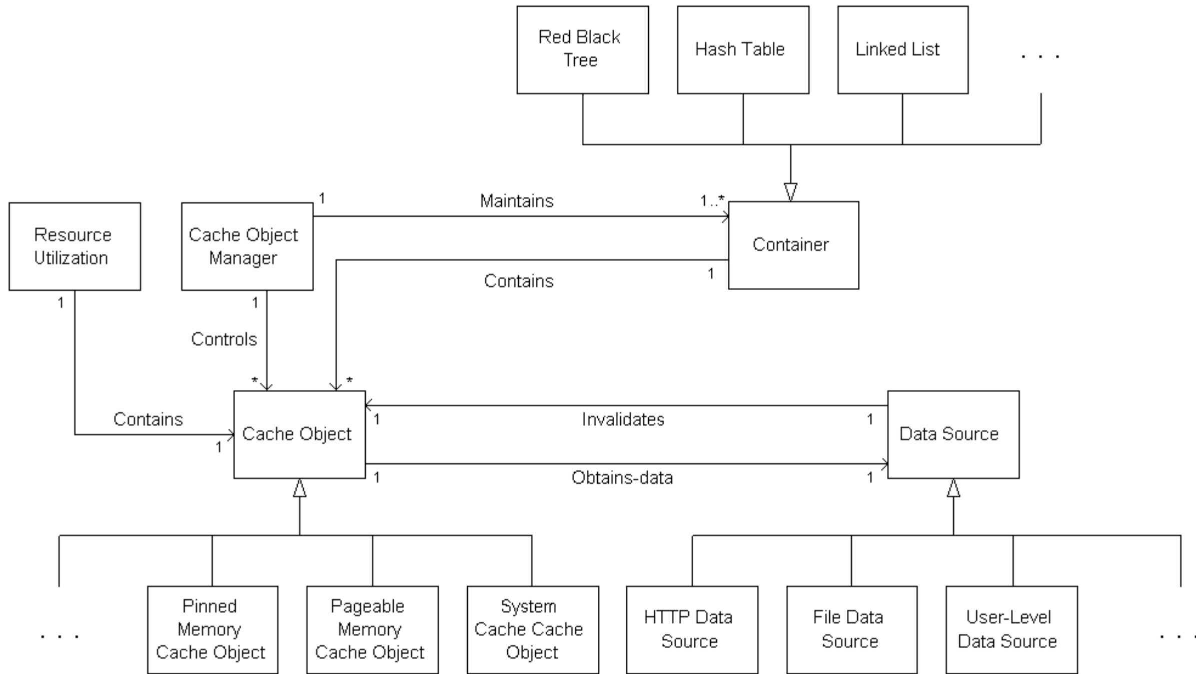


Figure 2: UML Object Diagram

applications. This includes expiration time, size, last modified time, content type, and content encoding. The application dependent portion of the metadata is a generic reference to a buffer that is set and filled out by the application and/or Data Source.

By design, Cache Objects are completely independent from the actual source of the data. Fundamentally, Cache Objects store data and return it upon request. Cache Objects can contain references to other Cache Objects of any type. This inclusion allows Cache Objects to be composed of other objects, thus removing duplicate cached data and allowing for object assembly. The collection of all instantiated Cache Objects is "the cache".

2.2.2 Data Sources

Data Sources identify the source of content and possess logic to obtain the latest version of the content. As with Cache Objects, Data Source types have an abstract parent class containing members and methods standard to all Data Sources. Members for Data Sources include a reference to actual data (if any) and an expiration time of that data (if any). A Data Source may temporarily have a reference (or copy) of the data while obtaining it from the actual source. Common methods in Data Sources are GetData, Validate, Create, and Cleanup. The GetData routine is called by the Cache Object to retrieve the latest version of the data. The Validate method is used by the Cache Object to determine whether the current version of the cached content is still valid. Testing for validity allows a Cache Object with expired content to verify it has up-to-date data without actually transferring data if it has not changed.

Each Data Source instance is responsible for retrieving content at the request of a Cache Object; therefore, each instance is associated with a single Cache Object instance. Data can arrive from a variety a sources: files, HTTP servers, FTP servers, user-level programs, etc.

Data Sources maintain coherency between the source of the data and the cached version in the Cache Object. If the content changes (file-change notification, database trigger [8, 9], HTTP push), then the Data Source must invalidate the Cache Object's version of the data. For example, File Data Sources have a FileChangeCallback function that is registered as a callback with file systems that support such a feature (e.g. NTFS). In the event the file changes, this callback function will invalidate the

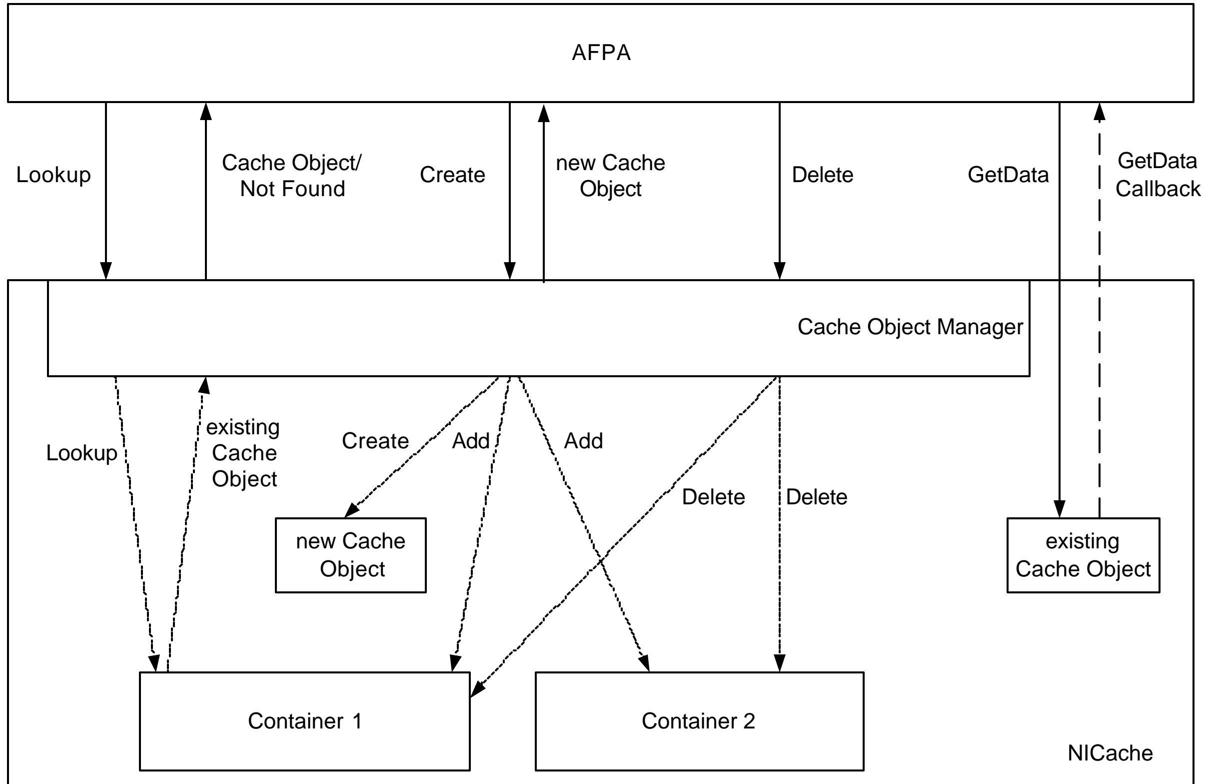


Figure 3: NICache - AFPA Interaction

Cache Object's data. Invalidation is encouraged whenever applicable because it maintains strong consistency (versus time to live), and it performs better than other methods used for strong consistency [21].

2.2.3 Resource Utilization Objects

Resource Utilization Objects are simply a list of system resources used by a particular Cache Object. System resources tracked by these Utilization Objects include the usage of pageable memory, non-pageable memory, and system page table entries. The latter is a limited resource on some operating systems (e.g. Windows) that is consumed when chunks of memory must be mapped into the kernel address space. The system page table, which is limited in size, maintains the kernel address to physical page mapping. Resource Utilization Objects are used to determine resource utilization by NICache and to determine on which Cache Objects to perform resource balancing operations. Each Cache Object has a single Resource Utilization describing the resources consumed by that Cache Object.

2.2.4 Cache Object Containers

Cache Object Containers (also referred to as just Containers) are standard searchable/sortable data structures conforming to an interface specified by the abstract parent Cache Object Container class. This standard interface, including methods for adding, removing, and searching, allows NICache to use many disparate Containers for different purposes with little regard to what the actual Container is. This allows the application utilizing NICache, AFPA, to employ Containers best suited to its task. A Hash-Table and Linked-List were two types of Containers used in the AFPA implementation. Containers contain references to Cache Objects.

2.2.5 Cache Object Manager

The Cache Object Manager manages all Cache Objects in NICache in regards to lookup, creation, and deletion requests from the kernel application. These interactions are shown in Figure 3. The lookup API is used by AFPA to retrieve cached content based upon the object's unique name. The Cache Object creation function in the Cache Object Manager takes as input a Data Source and returns a newly constructed Cache Object. When a

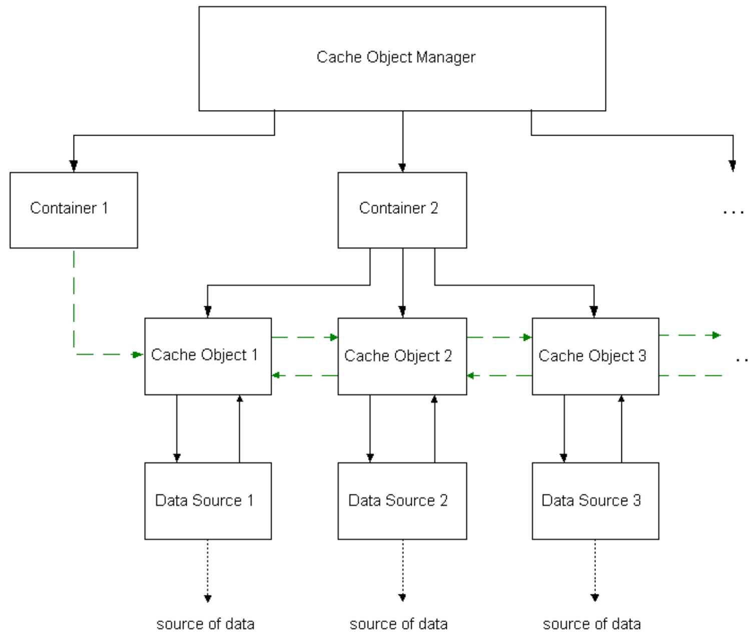


Figure 4: Object References

request to cache new content is made into the Cache Object Manager, it must choose the "best" mechanism (a.k.a. Cache Object type) to cache the data. This choice is made based upon the resources available on the system, the type of data in the Data Source, and those resources used by current instances of Cache Objects via the Resource Utilization objects. Cache Objects can be deleted when they expire or when they are determined to be expendable in a resource-constrained environment. The Cache Object Manager chooses which Cache Objects to remove based upon access rates, last accessed times, and the resources utilized by the Cache Object. For example, if pinned memory becomes constrained, the Cache Object Manager may decide to remove a number of Pinned Memory Cache Objects with low access rates that are utilizing large amounts of pinned memory. To aid in this process the Cache Object Manager could have a Priority Queue Container that stores only Pinned Memory Cache Objects sorted by pinned memory utilization or access rate.

Using the Cache Object Manager as the focus of control allows strict resource tracking to be done. At any time the Cache Object Manager can calculate NICache's approximate utilization of many system resources. Furthermore, upon finding out a high-water mark for a system resource has been crossed, the Cache Object Manager can make appropriate adjustments, such as deleting, suspending or morphing Cache Objects. When a Cache Object is suspended, its cached content is removed and the resources used to store it are freed; however, the Cache Object instance is allowed to persist. This is done to avoid the overhead of instantiating a new Cache Object if heap memory is not constrained. Morphing a Cache Object from one type to another type can be done if one resource used for caching is constrained, but another is not. Morphing essentially results in copying a Cache Object's metadata into a new object of a different type. The data is then converted from one caching mechanism to another. Morphing a Cache Object eliminates the overhead of deleting a popular Cache Object and replacing it right away.

The Cache Object Manager also handles expiration processing. Using timers, Cache Objects can be inspected at regular intervals to determine if the content has expired. A Cache Object with a low or zero access rate that has expired may be deleted, while a Cache Object with a higher access rate that has expired will be invalidated. By simply invalidating an expired Cache Object, during the next request for that object, NICache avoids the overhead of creating a new Cache Object and retrieving the content from the source. Furthermore, if the data has not changed, it can be validated thus avoiding the population of the Cache Object which could involved a costly transfer of data. The population of Cache Objects and the validation of expired content is discussed in more detail in the next section

2.2.6 Object Interaction

Figure 4 illustrates the reference relationship between instantiated objects. Each Cache Object instance in the diagram can be any Cache Object type, and each Data Source instance can be any Data Source type. Container 1 is some sort of a list structure, while Container 2 is more of an array structure. As seen in this figure, Cache Objects are associated with (maintain a reference to) a single Data Source and vice versa. This is because each piece of cached content needs to be retrieved from a distinct source. For example, if a file is being cached, the ability to locate and reload that particular file is needed. Although it is intuitive to believe System Cache Objects will always have a File Data Source, it is not a requirement for NICache.

Containers hold references to Cache Objects as seen in Figure 4, and a Cache Object must be present in at least one container at all times (so it can always be located). Typically, the Cache Object Manager will employ many different containers: one for fast searches (Hash Table), one for deletion processing (a weighted Linked List or a tree structure), and perhaps one for each type of Cache Object (Linked Lists) for resource recovery. Many Containers are needed because a Container that is efficient at lookups may not be best suited for sorting certain types of Cache Objects based on utilization or other criteria. All additions, deletions, and lookups done on Containers are done via the Cache Object Manager. Figure 4 shows two containers holding references to every instantiated Cache Object. The behavior of the cache can easily be altered by replacing or adding Containers. Figure 3 shows the Cache Object Manager's interaction with the Containers when responding to calls from AFPA.

After AFPA has obtained a needed Cache Object via the Cache Object Manager (i.e. Lookup in Figure 3), it must now request the content stored within the Cache Object. The GetData function shown in Figure 3 must be called, passing in such information as the length and offset of the needed data and a pointer to a callback function. NICache makes no assumption as to whether the caller of GetData can block or not. To that end, NICache uses an asynchronous interface (represented as dashed lines in Figure 3) for retrieving the data from Cache Objects. By extension, the data returned to AFPA is always pinned into memory (i.e. it is not pageable) so AFPA will not encounter a page fault when referencing the content. Pageable Memory Cache Objects need to temporarily pin data when it is passed to AFPA. Cache Objects do not have any data when created; therefore, upon the first request for data from AFPA, the Cache Object must request data from the Data Source.

Each Data Source must also support an asynchronous GetData call, which is made by a corresponding Cache Object. The interface between Cache Objects and Data Sources is also non-blocking because a Cache Object instance may have partial data that is ready to be returned to the application but is waiting for the next chunk of data from the source. There is no need to delay returning part of the data while waiting for the Data Source to respond. More importantly, there is no need to hold up a process for the unbounded amount of time it may take for a source to respond. After retrieving data from the source, the Cache Object's callback function is invoked, passing in a reference to the data or file handle and metadata. Without knowledge of the caching mechanism employed by the Cache Object, it is not enough for every Data Source to return references for buffer addresses. Typically, most Data Sources will return references; however, a File Data Source passing buffers to a System Cache Object results in a frivolous file read. In addition it prevents the GetData from behaving properly (because it needs to issue its own read with the file handle). In the special case of the pairing between a System Cache Object and a File Data Source, a file handle should be passed to the Cache Object. Since all sources should work with all Cache Objects, File Data Sources always return file handles and places the responsibility of reading the file onto the Cache Object. Each type of Cache Object should have the capability of receiving data in either format. What happens if the Cache Object Manager decides an HTTP response should be cached by a System Cache Object? The HTTP response does not have a file name or directory on the local machine, but to be resident in the system (page) cache a file must exist. It is up to the Cache Object to create a temporary file and write the HTTP response to the file. Removal of the file is the Cache Object's responsibility. If a new response for the same URL arrives from the source, then the file must be refreshed. Other Cache Object types may need to do something similar because not all data arrives in the format required by the storage mechanism.

Once the content received from the client is cached within the Cache Object, AFPA's callback function is called, passing in the content. The Cache Object's callback function and AFPA's callback function can be called many times for the same respective GetData request. This allows the data to trickle in from the source, but still be served to the application. Figure 5 demonstrates the double GetData asynchronous call pattern between AFPA, a Cache Object, and a Data

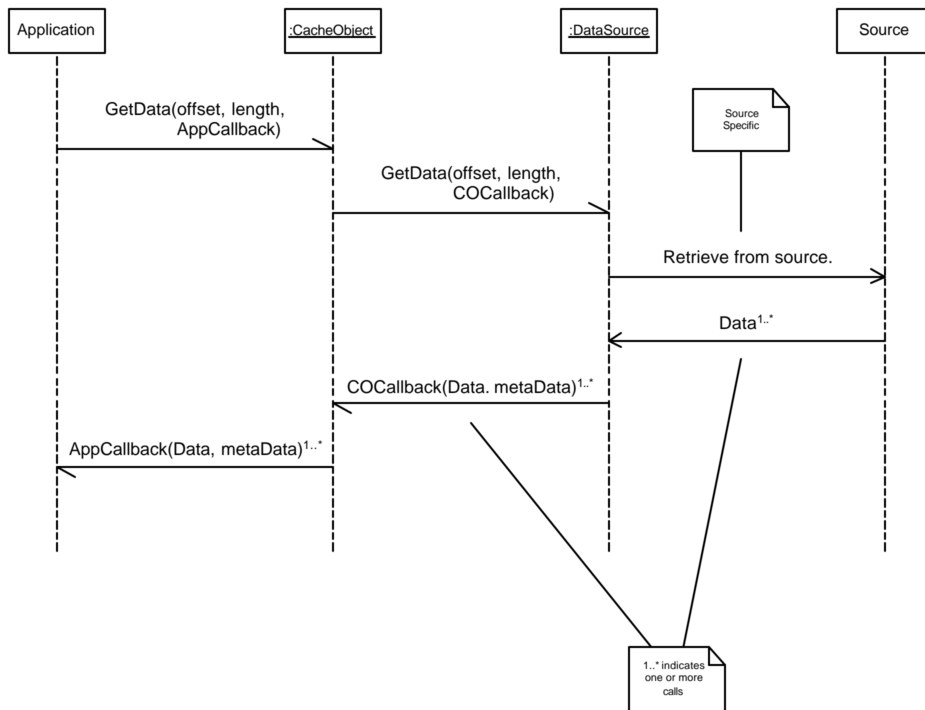


Figure 5: Sequence Diagram

Source. Subsequent requests into a Cache Object with valid data result in AFPA's callback being called without contacting the Data Source until the content becomes invalid.

A Cache Object's data may become invalid by the Data Source responding to an update from the source, the Cache Object Manager flagging the data as expired, or the Cache Object itself. Each request for data results in a self-check of the expiration time by the Cache Object. When the Cache Object receives a request for content that has become invalid, the Cache Object submits an asynchronous Validate call to its associated Data Source. Dykes et al [20] discuss the benefits of validating data instead of blindly retrieving it for large enough files. If a mechanism for validating data exists within the source (e.g. HTTP HEAD request or a file's last modified time), then the Data Source should check the validity of the data. If the data stored in the Cache Object is still valid, the Cache Object's validation callback is called with a new expiration time. If the data is no longer valid, the callback is returned with a status indicating as such, and the Cache Object instance must submit a GetData request to the Data Source.

Each GetData request into an instantiated Cache Object may get forwarded to subordinate Cache Objects if the data requested falls within that subordinate Cache Object. For example, if a Cache Object has ten bytes of data and includes another Cache Object containing 50 bytes, a request for the first 20 bytes will result in ten bytes served by the first Cache Object and the first ten bytes of the subordinate Cache Object. In the event many GetData requests are made into a Cache Object that does not have the needed content, then all requests are queued at the Cache Object. A single GetData call is made into the Data Source for the requested data. When the Cache Object's callback function is called, all pending requests for the needed data are completed. This queuing prevents many GetData calls into the Data Source, and thus many calls out to the source of the data. GetData calls into the Cache Object for different ranges of the data that are not present result in respective GetData calls to the Data Source.

3.0 Component Detail

3.1 Cache Objects

Cache Object types differ by the technique in which data is stored. System Cache Objects store data by pinning it in the kernel's system cache (page cache). On Windows, this is done by issuing a CcMdlRead system

call [22] passing in a file handle, offset, and length. The result of this call is a reference to a pinned Memory Descriptor List (MDL) that can be returned to AFPA in response to a GetData call into the Cache Object. On Linux the System Cache Object issues a do_generic_file_read[2] system call passing in a file handle and a custom actor function[3]. The actor function increments the page_count on those pages holding file data in the page cache. By incrementing the page_count, the pages will not be removed from the page cache because they will be considered 'in use'. The System Cache Object maintains the file handle as part of its member data.

Pinned Memory Cache Objects store the data in pinned memory, that is, non-paged pool on Windows and kernel memory on Linux. GetData calls are satisfied by returning the page or pages that contain the requested data either as MDLs on Windows or a list of pages on Linux.

Many other mechanism can be used to store content. If supported by the operating system, pageable memory can be used in a Pageable Memory Cache Object. Custom mechanisms, such as a HashTableOnDisk Cache Object could store large amounts of data using efficient persistent storage [7]. Virtually any kernel mechanism for storage can be used as a Cache Object as long as the implementation conforms to the attributes and methods parent Cache Object.

3.2 Data Sources

Data Source types differ by the source they represent. An HTTP Data Source, for example, maintains a URL and host name for the content stored in its associated Cache Object. Additionally it stores any pertinent headers (e.g. range) specified by AFPA. The HTTP Data Source's GetData function formulates an HTTP GET request and submits it to the TCP stack. When data begins streaming in from the HTTP server, a callback within the HTTP Data Source that was registered with TCP is called. This callback, in turn, parses the HTTP response headers and streams the data buffers along with metadata stripped from the response(s) to the Cache Object. The HTTP Data Source's Validate method works much like the GetData request. A call to the HTTP Data Source's Validate function results in the generation of an HTTP HEAD request. This request is submitted to the TCP stack (after registering the appropriate callback). When this callback is called with a response, it will examine the response code. If the response indicates the data is still valid, then new metadata (gathered from the response headers) including the expiration time is passed to the Cache Object. Otherwise, the Cache Object is notified that its content is invalid.

A File Data Source, on the other hand, maintains a process identifier and a file name. The process identifier is the thread id of the thread that opened the file. This is needed for closing the file in the appropriate context. A File Data Source's GetData function entails checking the interrupt request level, opening the file and collecting metadata (size, last modified time, file type) about the file. If GetData is called at elevated IRQ (i.e. at interrupt/dispatch level), then the rest of the processing is deferred to a thread. Upon opening the file and collecting metadata, the Cache Object is passed the file handle and metadata. A File Data Source's Validate method consists of checking to see if the last modified time is greater than that in the metadata currently stored in the Cache Object. The Cache Object is notified as to whether the current form of the data is current.

A Buffer Data Source is a simplistic Data Source describing a pageable buffer that is used to pass buffers of generated content from user level to kernel level. The GetData function for a Buffer Data Source entailed returning requested portions of the buffer. The Validate function for a Buffer Data Source always returns "Invalid" because there is no way to verify the contents with user level. A more robust user-level Data Source that eliminates copying user-level data into the kernel is planned. This more robust data source based on IO-Lite [28] will allow two way communication, therefore validation, between the kernel and user mode operating environments.

4.0 Performance Analysis

In this section we present an analysis of the performance of AFPA employing NICache. We will demonstrate the effect caching of dynamic content has on throughput based on the cacheability of the content. Furthermore, we will show the impact of NICache on AFPA when only static content is requested to show the effect of the overhead of NICache.

---

[2] Defined in the 2.4 and 2.5 Linux kernel source trees in /mm/filemap.c.

[3] In Linux, an actor function is a function passed into the generic file read function that performs application specific operations during the read (such as copy into a buffer).

4.1 Workload

We used a single synthetic workload in our experiments: SPECweb99 [11]. SPECweb99 is an industry accepted benchmark that requests files ranging in size from 100 bytes to 900 kB. This benchmark requests a mix of both static and dynamic content of various types. We vary the mixes for different tests to highlight different aspects of the software, and we will note these variations when discussing the tests. Depending on the test, we also restrict the number of directories for requested files.

Because of these modifications, the results presented here do not meet SPECweb99 reporting guidelines and are not certified SPECweb99 results. Reasons for these modifications follow. When evaluating AFPA against TUX [10], our intent was to evaluate the ability of AFPA and TUX to serve static content from RAM, not to generate dynamic content. When evaluating NICache's ability to manage and deliver dynamic content, we eliminate static content because it becomes noise in the results. SPECweb99 is designed to increase the number of directories requested by the clients as the load the server is subjected to (in terms of the number of simultaneous connections) increases. The number of directories is related to the number of simultaneous connections according to the following formula: $D = (3.27 * C) / 5 + 25$, where D is the number of directories and C is the number of simultaneous connections. Each directory includes 36 files totaling approximately 4.88 MB. For a C of 2000, for example, the clients would request 1336 directories (or approximately 6.3 GB). As the server we used was configured with "only" 2 GB of RAM, we limited the number of directories so we could evaluate each (software) server's performance serving content from RAM, which is their primary purpose. Allowing the clients to request the SPECweb99 mandated number of directories would have essentially amounted to evaluating the disk performance of the server, which was not our intent.

4.2 Test Environment

Our experiments were performed on an unmodified Red Hat Linux 2.4.18-5 kernel running on an IBM eServer xSeries 300 with dual 866 MHz Pentium III, 4 x 512 MB RAM (2 GB total), and two ACEnic gigabit Ethernet adapters. No attempt was made to tune the TCP stack (other than any settings AFPA or TUX may have modified themselves.) Although an SMP kernel was used, only a single processor was used for our experiments.

Our test environment included twelve SPECweb99 client machines: IBM Intellistation Z-pros, running Windows 2000 SP2. Each of these machines contained two 450 MHz Pentium II Xeon processors, 512 MB RAM, and a single ACEnic gigabit Ethernet adapter. An Extreme Networks Summit 7i gigabit Ethernet switch connected the twelve clients to the server. The network was configured so six clients communicated with a single adapter on the server.

4.3 Static Content

	Simultaneous Connections
AFPA with NICache	2,253
TUX	2,185

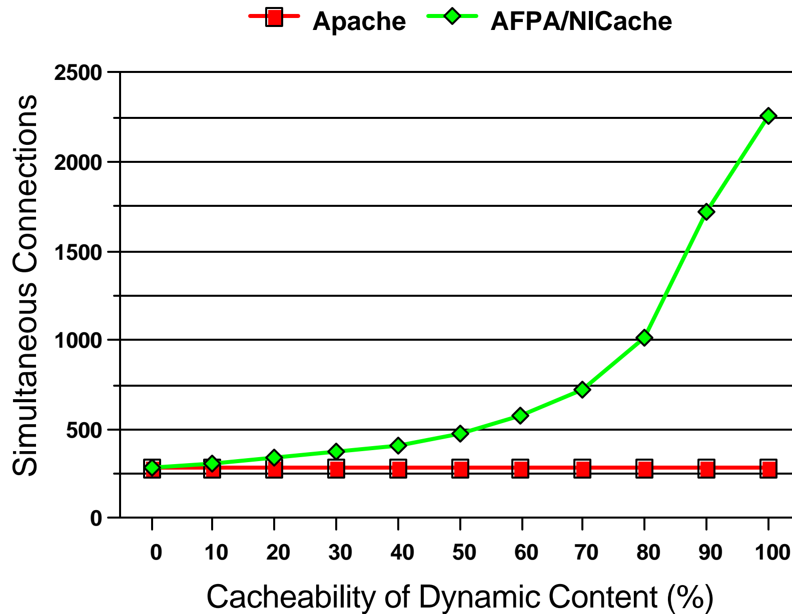
Table 1

Table 1 contains the results from our first experiment. Caching only static content which completely fits into RAM (25 SPECweb99 directories), AFPA achieved about 3% more simultaneous connections than TUX under identical conditions. AFPA with NICache used File Data Sources to describe the static files and used System Cache Objects to cache the content of those files.

The added flexibility and extensibility of NICache does not have a negative effect on the performance of AFPA. Due to major changes to the infrastructure of AFPA to accommodate NICache, a before and after comparison will not yield a valid comparison.

4.4 Dynamic Content

Since NICache provides the capability to cache dynamic content, this next experiment only uses SPECweb99 dynamic content generated by Apache 2.0.43 and a statically linked Apache module. Apache was configured to use all standard defaults. For the AFPA/NICache results, the dynamic content generation module was augmented with logic to populate the AFPA cache based on a percentage of cacheability. For a given request, AFPA checked to see if it can serve the desired content from the cache. If not, the request was "punted" to Apache. Apache then invoked the module to generate the content. If the content was deemed "cacheable", then the module would call an API to populate the AFPA cache. Apache would then respond



	0	10	20	30	40	50	60	70	80	90	100
Apache	288	288	288	288	288	288	288	288	288	288	288
AFPA/NICache	286	310	340	372	414	475	576	725	1,020	1,716	2,259

Figure 6: Dynamic Content Results

to the initial request. Subsequent requests for that content (i.e. URL) could then be served out of the cache. Cacheability is a percentage dictated by a parameter passed into the Apache module. Configured to use 20 SPECweb99 directories, 90% cacheability means the module demand-populated AFPA with directories 0 - 17. Fifty-percent cacheability means AFPA is populated with directories 0 - 9. When cacheability was 0%, all HTTP requests still pass through AFPA and are handed off to Apache. Cached content inside of NICache was given an infinite expiration time. Buffer Data Sources were used to pass dynamically generated content to the kernel. Pinned Memory Cache Objects were used for caching this content.

Figure 6 shows the results of varying cacheability. Of course, the Apache case was not using AFPA/NICache; therefore, the results are a straight line with a slope of zero. When cacheability is set to 0%, AFPA/NICache provide negligible overhead in terms of throughput. AFPA/NICache doubles the throughput of Apache 2.0.43 at 60% cacheability. At 100%, AFPA/NICache provide a benefit of a factor of about 6.8.

The NICache results were expected to be approximately linear; however, the curve is concave with a knee around 70%. We believe this is due to the connection hand-off from AFPA to the user-space server. This process is detailed in [1]. When keep-alive is used (and it was for these experiments), after a connection is handed to Apache, that connection remains in user-space until MaxKeepAliveRequests is reached or the maximum number of requests per connection in SPECweb99 is attained. For these tests, the Apache default of 100 was used for MaxKeepAliveRequests, and SPECweb99's default of ten was used for the average number of requests per keep-alive connection. Seventy-percent of the requests in SPECweb99 were keep-alive. Returning to Figure 6, as cacheability increases, the likelihood of passing a connection to the user-level server decreases, thus causing a sharp increase after 70%. We believe this makes a case for more cacheable dynamic content to increase benefit. We also assert this gives validation to the notion of adding a configurable layer-7 router [3] to AFPA to give locality to dynamic requests on back-end HTTP servers.

Attributes:
AccessRate
referenceCount
cacheObjectLock;
CacheObjectMutex
*pDataSource;
*pCacheObjectManager
*pSearchKey;
*pMetaData;

Methods:
GetData
Suspend
Cleanup
GetDataComplete
ClearData
SendComplete

Figure 7: Cache Object Specification

4.5 Conclusions

The amount of dynamic content on the Internet will most likely increase. As long as a percentage of that content is cacheable, even for a short duration, benefits can be derived from caching. Studies have shown dynamic content to be more cacheable than previously thought. [27] Although the results presented only contain two sources (file system for static content files and an Apache module for dynamic content), NICache provides a flexible and extensible framework for caching content from a variety sources. It does this while maintaining high performance and not impacting the miss case.

5.0 Implementation

AFPA with NICache has been implemented in C for both Linux and Windows. The Linux version contains the NICache framework, two Cache Object containers - a hash table and linked list, two Cache Object implementations - System Cache Object and Pinned Memory Cache Object, and two Data Sources - a File Data Source and a Buffer Data Source. NICache was implemented in about 4,800 non-blank lines of code, and the AFPA portion consists of about 14,000 non-blank lines of code. Although the aforementioned experiments were run with only one Cache Object and one Data Source implementation each, many Cache Object and

Attributes:
*pCacheObject;
union {
*pData[] (temp buffer)
fileDesc
}
dataSize;
expirationTime

Methods:
Cleanup
GetData
Validate

Figure 8: Data Source Specification

Data Source types can be used simultaneously.

The Windows version of NICache includes the framework, a single Cache Object Container - a hash table, two Cache Objects - System Cache Object and Pinned Memory Cache Object, as well as two Data Sources - File Data Source and HTTP Data Source. Of the 50,000 lines of code reported by the Windows' driver build utility [25], about 45,000 is the AFPA implementation[4] and 5,000 the NICache implementation.

As mentioned previously, the NICache architecture was constructed using an object oriented paradigm. Figures 7 and 8 show the main components of the parent Cache Object and Data Source 'classes' respectively. The attributes and methods are common across both platforms.

Performance numbers for fragment caching were not presented because that portion of the software is not yet implemented.

6.0 Related Work

A wide variety of research in the area of Web caching [12] has been done. We outline some of the most relevant work here. To our knowledge, no other studies on multiple source in-kernel caches have been performed.

Several Web caches have been implemented for the kernel and can be categorized by their integration with the TCP/IP stack. Microsoft's Scalable Web Cache

[4] The AFPA source code is much larger on Windows than Linux because the Windows code contains a layer-7 router and a proxy. Details can be found in [3].

(SWC) [14] is highly integrated with the Windows TCP/IP implementation. Linux's kHTTPd [24] employs kernel-mode socket interfaces. Both of these Web caches handle requests using kernel-mode threads. Red Hat Content Accelerator [10], otherwise known as TUX uses a threaded model but offers greater performance and features over kHTTPd on Linux. Although TUX uses the file system to cache content, it has its own cache directory management so that the URL to file object resolution is not performed by the file system (which is similar to the AFPA file system object architecture on both Windows and Linux.) Furthermore, 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. Lastly, TUX efficiently supports server side includes for fast dynamic content generation. All three of these solutions lack the capability to cache dynamic content in the kernel. Furthermore, SWC and TUX each use separate proprietary API for forwarding requests for dynamic content to a user-level server.

Several commercial products provide user-level caching functionality for both static and dynamic content, that is actual HTML output from Servlets/JSPs, including DynaCache [15], IBM WebSphere Application Server [16], Oracle9i AS Web Cache [17], and SpiderCache from Spider Software [18]. None of these products contain a kernel-based caching component.

Arun Iyengar et al [9] cache dynamically generated content by pushing evolving content out to the Web cache. Though the content they generate can be considered dynamic, it does not change for individual user request's. Therefore, this approach only works when the dynamic data is not dependent on a user's state (e.g. a cookie).

The General-Purpose Software Cache [24] GPS is a generic cache that can store contents using memory or disk. Instead of a concept of a Data Source, the GPS cache accepts data as a buffer through the API. When the data changes, it is up to the application using GPS to update the data not GPS. Finally, GPS only stores data using memory (provided by the application) or disk. NICache permits the use of many kernel-level storage techniques.

7.0 Acknowledgments

We would like to thank Raymond Jennings and Elbert Hu for assisting with the performance results and gathering certain background material. We would also like to thank Fred Douglis, Rob Saccone, and Catherine Zhang for providing insightful feedback.

8.0 References

[1] Phillipe Joubert, Robert King, Richard Neves, Mark Russinovich, and John. Tracey. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. *In Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[2] Ardan van de Ven. kHTTPd Linux HTTP accelerator. **http://www.fenrus.demon.nl**.

[3] Elbert Hu, Phillipe Joubert, Robert King, Jason LaVoie, and John Tracey. Adaptive Fast Path Architecture. *In IBM Journal of Research and Development*, April 2001.

[4] Cáceres Ramón, Fred Douglis, Anja Feldmann, Gideon Glass and Michael Rabinovich. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. In *Proceedings of IEEE Infocom 1999*, June 1999.

[5] Jim Challenger, Arun Iyengar, Karen Whitting, Cameron Ferstat, and Paul Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of IEEE Infocom 2000*, March 2000.

[6] Craig Larman. *Applying UML and Patterns: an introduction to object-oriented analysis and design.* 2nd ed. Prentice Hall, 2001.

[7] Arun Iyengar, Shudong Jin, and Jim Challenger. Techniques for Efficiently Allocating Persistent Storage. In *Proceedings of IEEE Symposium on Large Scale Storage In the Web*, April 2001.

[8] Jim Challenger, Paul Dantzig, and Arun Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE Infocom 1999*, March 1999.

[9] Arun Iyengar, Jim Challenger, Daniel Dias, and Paul Dantzig. Techniques for Designing High-Performance Web Sites. *IEEE Journal of Internet Computing*, March/April 2000.

[10] Red Hat Inc. Red Hat Content Accelerator. **Http://www.redhat.com/docs/manuals/tux/**.

[11] The Standard Performance Evaluation Corporation. SPECweb99 Benchmark. **http://www.spec.org/osg/web99**.

[12] Greg Barish and Katia Obraczka. World Wide Web Caching: Trends and Techniques. In *IEEE Communications Magazine, Internet Technology Series*, May 2000.

[13] IBM Corporation. Netfinity Solutions - NWSA v2.0 (Netfinity Web Server Accelerator). http**://www-1.ibm.com/support/docview.wss?uid=psg1MIGR-4HDSMK**.

[14] Microsoft Corporation. SWC 3.0: Microsoft Scalable Web Cache. Http://www.microsoft.com/downloads/release.asp?ReleaseID=29211.

[15] Abdelsalam Heddaya. Dynacache: Weaving caching into the internet. In *Proceedings of the Third International WWW Caching Workshop*, June 1998.

[16] IBM Corporation. IBM WebSphere Application Server. **http://www.ibm.com/software/websphere**.

[17] Oracle Corporation. Oracle 9iAS Web Cache. **http://otn.oracle.com/products/ias/pdf/9iaswebcache_v2_twp.pdf**.

[18] SpiderSoftware. SpiderCache. **http://www.spidersoftware.com**.

[19] Weisong Shi, Randy Wright, Eli Collins, and Vijay Karamcheti. Workload Characterization of a Personalized Web Site - And Its Implications for Dynamic Content Caching. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*, August 2002.

[20] Sangra G. Dykes, Kay A. Robbins, and Clinton L. Jeffery. ``Uncacheable documents and cold starts in Web proxy cache simulations: How two wrongs appear right". *Technical Report CS-2001-01*, University of Texas at San Antonio, Division of Computer Science, San Antonio, TX 78249-0664, Jan. 2001.

[21] Y. Cao and M.T. Özsu. "Evaluation of Strong Consistency Web Caching Techniques," *World Wide Web Journal*, 5(2): 95-123, 2002.

[22] Rajeev Nagar. *Windows NT File System Internals. A Developer's Guide*. O'Reilly, 1997.

[23] David Solomon and Mark Russinovich. *Inside Microsoft Windows 2000 Third Edition.* Microsoft Press, 2000.

[24] Arun Iyengar. Design and Performance of a General-Purpose Software Cache. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference*, February 1999.

[25] Microsoft Corporation. Microsoft Windows Driver Development Kits (DDK). **http://www.microsoft.com/ddk/**.

[26] The Apache Software Foundation. The Apache HTTP Server Project. **Http://httpd.apache.org/**.

[27] Craig Wills and Mikhail Mikhailov. Examining the Cacheability of User-Requested Web Resources. In *Proceedings of the 4th International Web Caching Workshop*, March/April 1999.

[28] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, Vol. 18, No. 1, pp.37-66, February 2000.