WebSphere Dynamic Cache: Improving J2EE application performance

by R. Bakalova
A. Chow
C. Fricano
P. Jain
N. Kodali
D. Poirier
S. Sankaran
D. Shupp

The Dynamic Cache is part of the IBM solution for improving performance of Java 2 Platform, Enterprise Edition (J2EE™) applications running within WebSphere Application Server. It supports caching of Java™ servlets, JavaServer Pages™ (JSP™), WebSphere® command objects, Web services objects, and Java objects. This paper describes the techniques used by the Dynamic Cache for caching these objects and demonstrates the performance improvement gained by applying these techniques to a typical enterprise Web application.

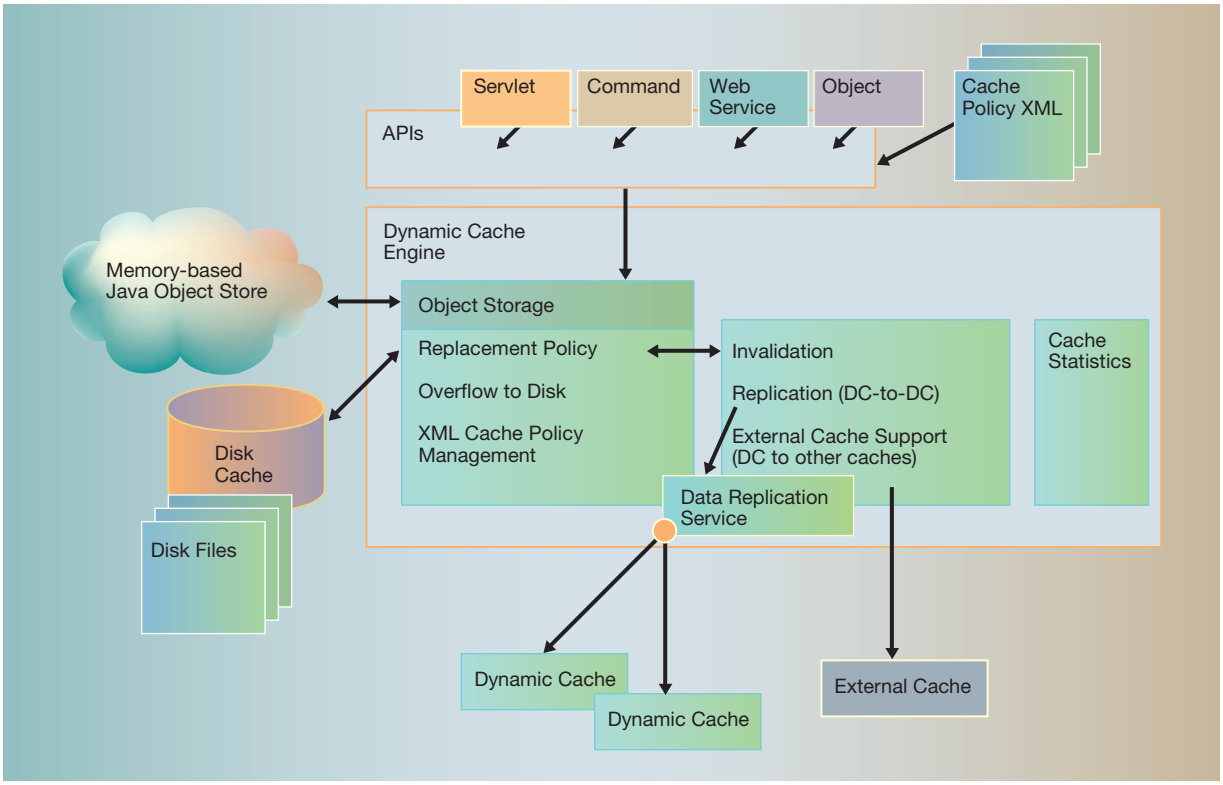
Modern Web applications carry a heavy burden: they are required to deliver high performance and be full featured. They must feature personalization for individual users, assemble information from disparate and volatile data sources, and process data intelligently depending on user requests. Also, the applications are fragmented into multiple operations executed for the same request, the results of which are assembled into a structured response to the user. For example, the request to view an account balance may require checking for a valid session, querying a backend database, and finally putting together an HTML (HyperText Markup Language) page using the results from the database query. As some of these operations are time-consuming, such dynamic applications have high inherent costs in performance and scalability.

Caching can significantly improve the response time for such applications by saving the computed results in a cache, associating the saved results with properties of the request, and obviating computations for subsequent similar requests by serving their responses from the cache. Existing caching mechanisms, however, fail to operate suitably for Java 2 Platform, Enterprise Edition (J2EE**) applications. They either do not operate at a fine enough granularity, cannot handle the complex associations between fragments that are used across multiple operations, or require one-of-a-kind solutions. When application developers build their own caches, they are often hard to re-use or modify, costly (in time) to develop, and still cannot function at every tier of a J2EE application.

The WebSphere Dynamic Cache solution responds to the unique demands of J2EE applications. It is integrated into the WebSphere Application Server[1] and utilizes the J2EE architecture to cache objects and results at every level. Web services output, presentation layer servlets, JavaServer Pages** (JSPs**), WebSphere* command classes, and Java** objects can all be cached by means of application programming interfaces (APIs) or by declaring caching policies and adding them to an application. Significant performance advantages, especially for high-volume Web applications, can be achieved by use of the Dynamic Cache.[2] In most cases, these policies can be written and enabled without modifying application code and can be easily updated whenever the underlying code changes. This method makes enabling dynamic caching for existing J2EE applications

©Copyright 2004 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

Figure 1 WebSphere Application Server Dynamic Cache overview



straightforward while keeping the application independent of a specific J2EE platform.

By using the dynamic caching service, application designers can define a range of conditions to govern the freshness of objects in the cache. Explicit timeouts can be set up to ensure that an object's data is current. Also, because data in a cached result may be modified by requests to other objects or may become invalid based on other conditions like the end of a session, a cache policy may declare dependencies on other objects, causing a stored result to be deleted from the cache when underlying data is modified or some other condition is met. Because multiple entries may be dependent on the same conditions, grouping different cache entries to be deleted together is also allowed through the policies.

Figure 1 presents an overview of the Dynamic Cache engine. The Dynamic Cache stores its content in a memory-based Java object store. These objects can be accessed and manipulated by APIs that are provided with the service. The WebSphere Application Server also offers a Web application called Cache Monitor, which plugs into the cache to provide a view of its contents and statistics. The Dynamic Cache uses a replacement policy like the least recently used (LRU) algorithm to create space for incoming entries when the assigned space is full by deleting contents based on the replacement policy. It can also be configured to push data to a disk cache from which it can reclaim the data if needed in the future. Removal of entries from a cache can also occur due to data invalidations, based on cache policies defined by the administrator.

The Dynamic Cache can take advantage of the data replication services in the application server to replicate cached data across servers in a cluster. Invalidations are transmitted across the cluster to keep data consistent and valid. The Dynamic Cache can also send cached data and invalidations to various external caches like the Akamai ESI (Edge Side Includes) server and the WebSphere Edge Server to

bring data closer to the end user and further enhance performance of J2EE applications.

This paper illustrates the caching techniques offered by the Dynamic Cache service, shows examples of their use, and measures their effects on performance using the Trade3 application.[3] Trade3 is IBM's end-to-end benchmark and performance sample application. For each of the key dynamic caching technologies, we provide a technical discussion of its unique requirements and the specific methods WebSphere uses to cache in that context. We also provide examples that apply those methods to relevant parts of the Trade3 application and measure the resulting changes in performance to understand the relative effect of caching within the scope of a whole application.

Common aspects of Web application caching. While J2EE application performance can be significantly improved by the use of caching techniques, the J2EE standard does not require implementations to perform any caching of dynamic results. There is currently an effort to define a standard programming interface for caching from Java programs (Java Specification Request [JSR] 107[4]), and IBM is participating in the process of defining this standard.

Dynamic caching requires cache policies to be configured for an application, or for the application to explicitly use the available caching APIs. Caching policies must be carefully chosen because incorrect cache policies may result in users viewing outdated data. The IBM WebSphere Application Server toolkit provides an Eclipse plug-in to generate a set of default policies for an application, along with a custom GUI (graphical user interface) editor and wizards to create and edit policies. No changes need to be made to existing J2EE applications to take advantage of the Dynamic Cache technology.

The Dynamic Cache (in the *cachespec.xml* file) stores a caching policy for each cacheable object. This policy defines a set of rules specifying when and how to cache an object (i.e., based on certain parameters and arguments), and how to set up dependency relationships for individual or group removal of entries in the cache.

Each data request (i.e., any invocation of a cacheable servlet, JSP, Web service, or other object) is associated with a set of input parameters that are combined to form a unique key, called a *cache identifier* or cache ID. A key policy defines which cache identifiers result in a cacheable response. If a subsequent request generates the same key, the response is served from the cache. If a unique key is not created through any of the rules, the response for the request is considered not cacheable.[5]

A second critical policy defines when to remove the cached results from the cache. Removing out-of-date cache entries is as important as populating them. Cache entries may expire after a given amount of time or be removed from the cache based on its dependencies or on the basis of an LRU algorithm when the cache needs space for new entries.

The method of removing entries from the cache is called cache invalidation. The easiest method for cache invalidation is to set a "time-to-live" (TTL) value on the cache entry. If an entry needs to be invalidated before its TTL has expired, an explicit invalidation policy is written for that cache entry in the *cachespec.xml* file. This policy defines an invalidation rule very similar to the cache ID rule, and an invalidation ID is generated based on that rule. When an invalidation rule is specified for an object, WebSphere generates an invalidation ID during the execution of that object and checks it against the cache IDs of all entries currently in the cache. If a cache ID matching the invalidation ID is found in the cache, the cache entry associated with this cache ID is removed from the cache (i.e., invalidated).

The Dynamic Cache also provides a group-based invalidation mechanism through *dependency ID*s. A dependency ID identifies a cache entry's dependency on certain factors, such that when those factors occur, they trigger an invalidation of all the cache entries that share this dependency. Different objects, based on their defined rules, may generate the same dependency ID. If an invalidation ID is generated for a request that matches a given dependency ID, all the cache entries associated with that dependency ID are removed from cache.

Competing technologies. Alternative approaches for caching the dynamic content of J2EE applications include Oracle Application Server Web Cache,[6] the JCache specification,[4] the Tangosol Coherence Cache,[7] and SpiritCache.[8]

Like the WebSphere Dynamic Cache, Oracle's caching technology includes fragment caching, ESI caching, and invalidations based on triggers and APIs. In an effort to standardize caching APIs, Oracle Corporation submitted JSR-107 (JCache–Java Tempo-

rary Caching API) with support from several other companies including IBM. JCache is an open-source caching-API specification for J2EE applications to programmatically cache data. It includes APIs to temporarily store Java objects in memory for reuse and invalidation APIs to maintain freshness and consistency. It also requires data replication across caches on servers in a site. There are several open-source implementations of JCache as well as several JCache-compliant products in the market.

The Tangosol Coherence Cache is a JCache-compliant technology that can be deployed on multiple application servers. In addition, it includes a distributed cache that can be used by applications to programmatically replicate cached data across servers in a cluster.

SpiritCache by SpiritSoft provides a multitiered distributed caching framework based on JCache standards. It offers API-based caching for storing Java objects and tag-based caching for HTML responses from JSPs.

Similarly to these approaches, WebSphere Dynamic Cache includes servlet caching for full pages as well as fragments, ESI caching, object caching, invalidations, and cache replication. Additionally, it features caching of WebSphere commands and Web services responses on the server.

The dynamic caching design specified by Viriri[9] attempts to split dynamic pages into relatively static and more dynamic parts by using custom tags. Its suggests the use of existing static cache technologies for storing and reusing the relatively static parts of the dynamic pages. This method introduces two problems. The first problem is that any existing J2EE applications would have to be redesigned to accommodate this method of caching. The second problem is that because the relatively static pages are cached using existing static cache technologies, application vendors would have to customize their existing static caches to recognize the new tags that separate the static and dynamic portions of a page and reconstruct the static portion when it does change.

Performance measurement. In the remainder of this paper, performance numbers for the Dynamic Cache for all caching types except Object Cache were obtained using the Trade3 application. (For Object Cache, a basic J2EE application was created to do simple cache "puts," "gets," and "invalidates" by using the DistributedMap APIs.)

Trade3 is an application developed and used by the WebSphere performance team for characterizing the performance of WebSphere Application Server Version 5.0. This application was created to emulate online brokerage services. It is a collection of Java servlets, JSPs, Enterprise JavaBeans** (EJBs**), commands, Web services, and message-driven beans that together form an application. It allows a user, typically using a browser, to perform the following actions: register to create a user profile, log in to validate a registered user, browse a list of stocks for ticker prices, sell and buy shares, view a portfolio, and log out. Figure 2 provides the architectural overview of Trade3.

Trade3 provides a trading scenario servlet, emulating a single trade, to simplify benchmarking. This servlet can be driven by multiple threads to model the activity of a population of users. The trading scenario servlet models a user performing any of the standard trading operations listed previously.

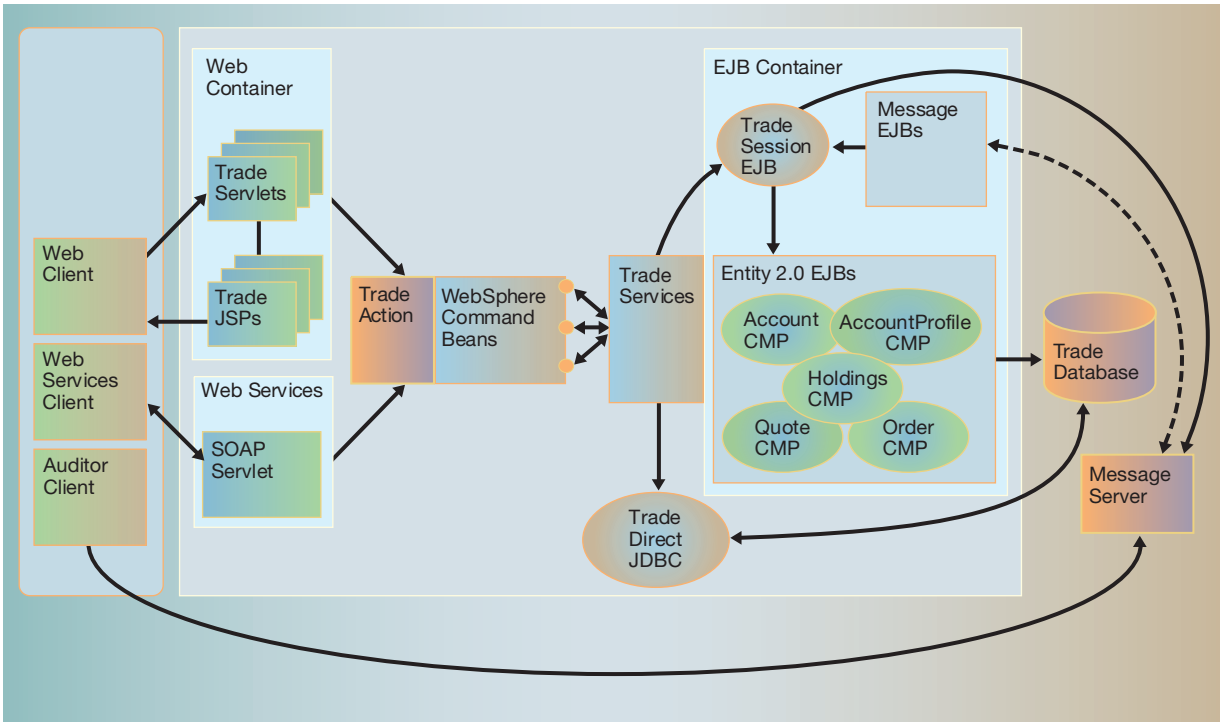
Trade3 implements new and significant features of the EJB 2.0 component specification. These include:

- Container-managed relationships—One-to-one, one-to-many and many-to-many relationships of objects to relational data which are managed by the EJB container and defined by an abstract persistence schema. This provides an extended, real world data model with foreign key relationships, cascaded updates and deletes, and so forth.
- EJB query language—Standardized, portable query language for EJB finder and select methods with container-managed persistence.
- Local or remote interfaces—Optimized local interfaces providing pass-by reference objects and reduced security overhead.

In our tests, the database is accessed using the EJB 2.0 technology. The operations provided by the Trade3 server model are defined in an interface called TradeServices. The TradeServices interface is implemented by an EJB implementation (Trade) using session, entity, and message EJBs. Access to the EJB model implementations is through the "TradeAction" Java bean, which acts as a facade to the Trade server model.

Trade3 also provides two access modes which determine the protocol used by the Trade Web application to access server-side services. The *standard* access mode uses the default Java RMI (remote method invocation) protocol and was used in the servlet and command caching tests. The *Web services*

Figure 2 Trade3 application architecture



access mode uses WebSphere's implementation of Web services including SOAP (Simple Object Access Protocol), WSDL (Web Service Definition Language), and UDDI (Universal Description, Discovery, and Integration). This mode was used in the Web services caching tests.

Figure 3 shows the four-tier test configuration used to gather performance data. The first tier consists of a single client machine which runs the AKStress tool. AKStress is an IBM internally developed tool to generate HTTP (HyperText Transfer Protocol) requests. The second tier consists of a single processor Linux** machine running IBM HTTP Server 2.0, which is an Apache-based Web server developed and supported by IBM. The third tier consists of a multiprocessor machine, which runs the WebSphere Application Server and the IBM DB2* runtime client. The fourth tier consists of a machine functioning as a DB2 server.

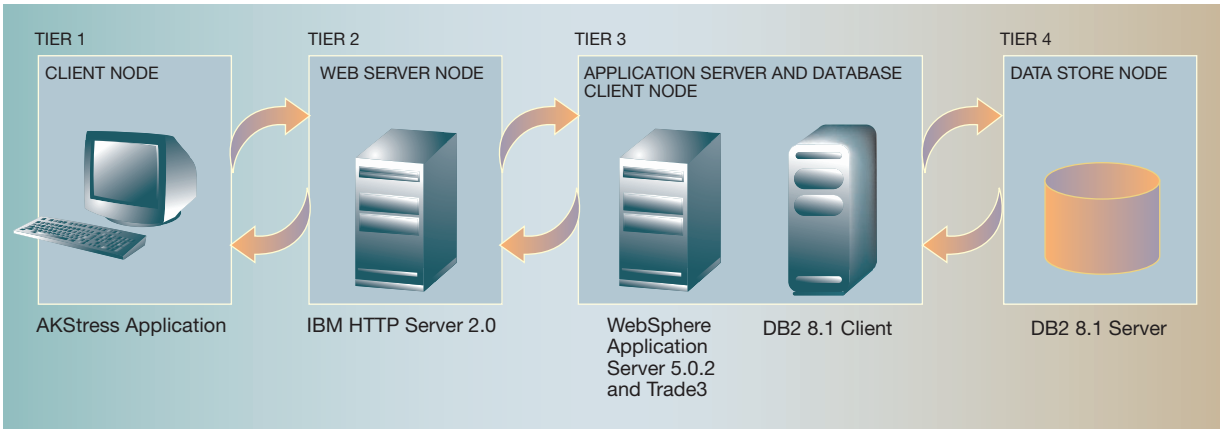
Performance is primarily measured by how many requests the server can handle per second (i.e., throughput) and how fast it responds to the client request (i.e., response time). This is accomplished by using AKStress to simulate multiple users making hundreds of requests to the server. AKStress was configured to run with 11 threads per run.

For connectivity, a gigabit network adapter was installed on the tier-3 machine for the DB2 client hosted on this machine to connect to the DB2 server machine. A 100-Mbps Ethernet card was installed on the same machine for the Web server to connect to the application server also hosted on this machine. 100-Mbps Ethernet cards were installed on each of the tier-1 and tier-4 machines, hosting the client and the DB2 server respectively. Two 100-Mbps Ethernet cards were installed on the tier-2 machine, one to connect to the application server and one for the client to connect to the Web server hosted on this machine.

Servlet and JSP caching

This section describes issues involved in caching Java-servlet or JSP responses in a Web application. It ex-

Figure 3 Dynamic Cache performance test setup using Trade3



plains what information is needed for caching and how a Web application developer might specify this data.

A servlet or JSP is a Java technology-based Web component, managed by a container that generates dynamic contents. This is a widely used technology in building Web applications, and the performance of a Web application can be improved by intelligently caching servlet and JSP responses. The golden rule of servlet caching is to isolate the servlets and JSPs that perform complex processing, and cache them and the pages they depend on. Although caching a simple presentation JSP file gives moderate performance gains, caching servlets that request information from EJBs or a database saves the WebSphere Application Server significant processing power and decreases load on the back end.

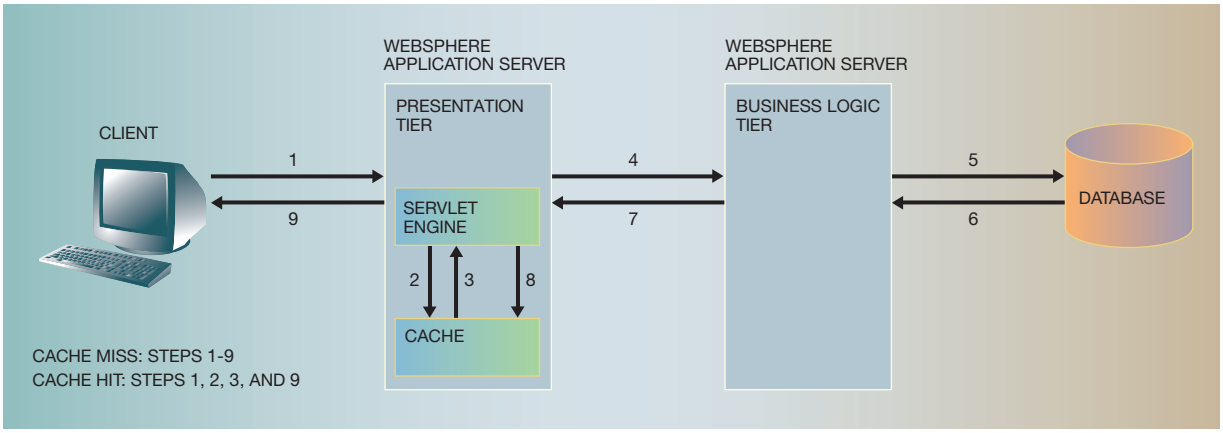
The Dynamic Cache service of the WebSphere Application Server is an in-memory cache capable of caching servlet and JSP responses as well as other dynamic content. The servlet and JSP cache can be enabled or disabled using the Web container settings of the WebSphere administrative console. Because all JSP files are compiled into servlets by the WebSphere Application Server, from the point of view of the Dynamic Cache, JSPs and servlets are identical.

Figure 4 shows the steps used in handling a servlet request. In general, serving a servlet request involves (step 1) loading the servlet class, creating an instance of the servlet class, initializing the servlet instance by calling its initialization method, and finally invoking the service method of the servlet class. When servlet caching is enabled, it uses hooks in WebSphere Application Server's servlet engine to (step 2) intercept calls to a servlet's service method and tries to create a key based on the policy defined for it. If a key is created, it looks for the response to that servlet request in the cache. If found, the response is served from the cache (steps 3 and 9); otherwise, the service method of that servlet is executed (steps 4-7) and the response is cached (step 8). If a policy is not defined for this servlet or a key is not generated based on the policy, the response is not cached.

Cache ID of a servlet or JSP. When a servlet request is received, the cache attempts to build a unique key for that invocation from the values of certain request variables. When the cache does not contain an entry associated with this key, the variable HttpServletResponse is stored in the cache. For the purpose of servlets and JSP caching, these cache IDs can be composed using the following:

- *parameter*—Named parameter from the servlet request object
- *attribute*—Named attribute from the servlet request object
- *servletpath*—Directory path of the servlet; that is, the path section that corresponds to the component alias that activated this request
- *pathinfo*—Part of the request path that is not part of the context path or the servlet path
- *session*—Named value from the HTTPSession bean
- *header*—Named header from the servlet request

Figure 4 Data flow for WebSphere Application Server servlet/JSP response cache



- *locale*—Request locale
- *cookie*—Value of named cookie

Whole page versus fragment caching. Most dynamic Web pages are constructed from smaller and simpler page fragments. Some fragments are static (such as headers and footers), and some are dynamic (such as fragments containing stock quotes). The goal of creating fragments or components is to maximize reuse and cache utilization. A fragment may or may not contain one or more fragments, and it may or may not be contained by one or more "parents." The Dynamic Cache can be configured to cache either the whole page or individual fragments.[10] For example, Figure 5 shows the home page of a specific user in the Trade3 application. This page, "tradeHome.jsp," includes another dynamic fragment called "marketSummary.jsp." The marketSummary.jsp fragment is not user-specific and hence is the same for all users. In such a case, caching marketSummary.jsp as a separate fragment and reusing its cached value in every page that includes it would result in higher gains than caching the tradeHome.jsp page as a whole.

ESI caching. ESI is an open standard[11] for composing Web pages from smaller fragments that reside at the edge of the network. A Web server or proxy that supports the ESI standard can take advantage of the caching of Web page fragments close to the end user.

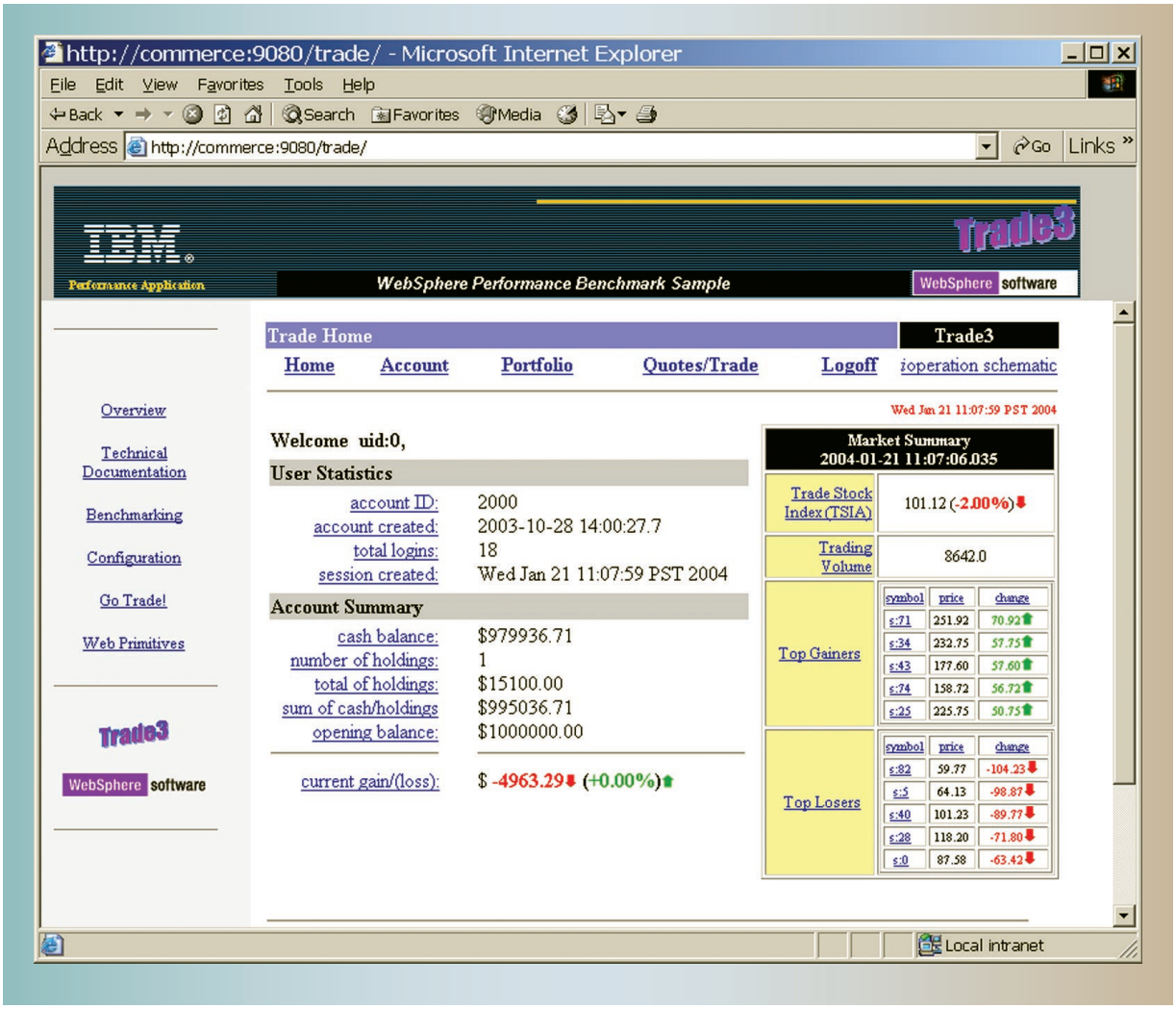
The WebSphere Dynamic Cache can recognize the presence of an edge server and automatically generate ESI tags and appropriate ESI cache policies for fragments to be cached at the edge, based on the cache policies defined in the *cachespec.xml* file. The purpose of this integration is to allow WebSphere dynamic content to be assembled and cached at edge nodes throughout the Internet, resulting in increased throughput and lower response times for WebSphere applications. It provides edge caching of applications without requiring any changes to the WebSphere/J2EE programming model.

Whole pages, as well as fragments, can be cached at the edge. However, for a fragment to be edge-cacheable, it has to be accessible via the HTTP protocol. In other words, if users enter the URL (Uniform Resource Locator) in their browser with the appropriate parameters and cookies for the fragment, the WebSphere Application Server returns the content for that fragment. In addition, cache IDs of edge-cacheable fragments must be formed using only data that is available at the edge, that is, query parameters, request headers, and cookies, but no session data. A fragment is configured as edge-cacheable by adding the following XML (eXtensible Markup Language) tag to its *cachespec.xml* file:

```
<property name="EdgeCacheable">true</property>.
```

The presence of an ESI processor in the network is indicated to the WebSphere Dynamic Cache by the ESI processor by adding a "surrogate capabilities" header to a request before it is forwarded to WebSphere Application Server. Upon receipt of this header, WebSphere Dynamic Cache will switch into the edge caching mode in which it first checks if the

Figure 5 A user's Trade3 home page

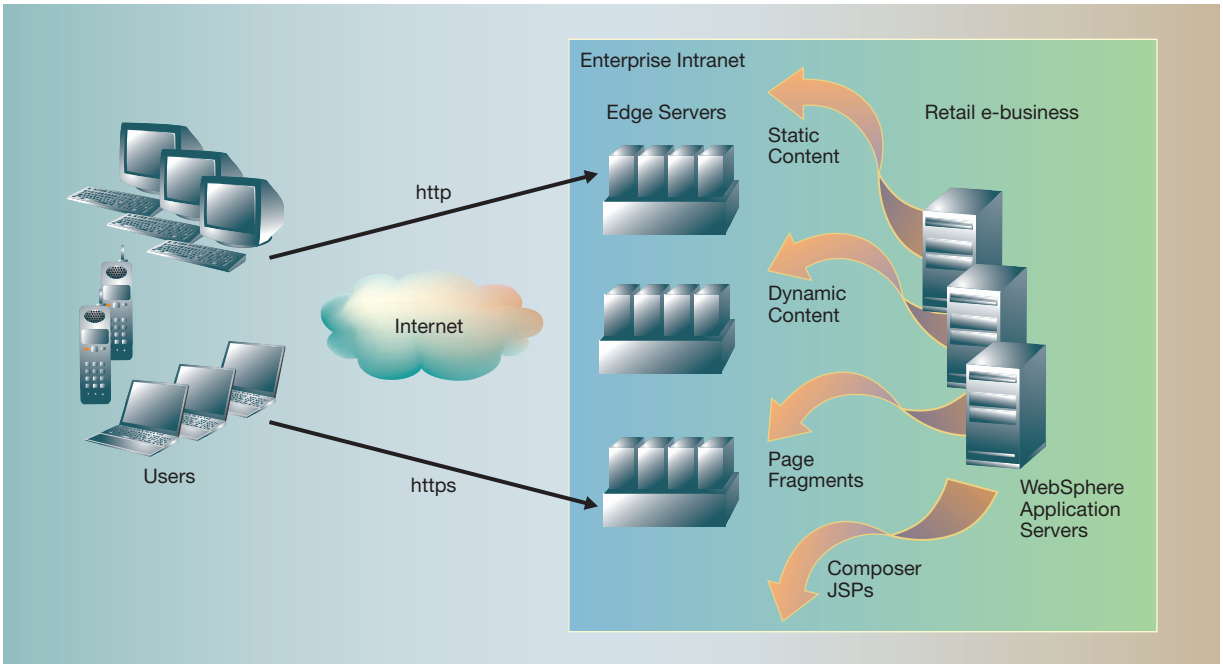


servlet is configured as edge-cacheable. If the fragment is not edge-cacheable, then a header indicating this is added to the response to indicate to the edge server that this response is not edge-cacheable. If the fragment is edge-cacheable, then a header indicating this is added to the response with the cache ID, dependency ID, and TTL value for the fragment. The ESI processor builds a cache and stores the response in the ESI cache.

The Dynamic Cache converts all servlet and JSP include tags to ESI include tags by using the ESI syntax. For each ESI include tag in the body of the response, the ESI processor processes a new request, such that each nested include tag results in either a cache hit or another request forwarded to the application server. When all nested include tags have been processed, the page is assembled and returned to the client.

Figure 6 shows content being pushed by the Dynamic Cache service that is running on the WebSphere Application Servers to the edge servers, thus bringing the content closer to the end user. Pushing content saves the steps of the edge server forwarding a request to the application server and the application

Figure 6 Pushing content to WebSphere edge servers



server sending the response back to the edge server, thus resulting in lower response time.

The cache entries on the edge server can be invalidated either through timeouts or by sending explicit invalidation messages to the edge server caches. In the case of the standard ESI Version 1.0 protocol, these cache invalidation messages include the full URLs of the fragments that need to be invalidated. The WebSphere Dynamic Cache will push all invalidations that occur out to the edge server through an ESI invalidation gateway.

Servlet and JSP caching in Trade3. Trade3 has been built based on the standard model-view-controller (MVC) architecture, where a call to a controller servlet might include one or more JSPs that are used to construct the view. In the Trade3 application, the controller servlet used is "TradeAppServlet," and the following JSPs are used for creating the view:

- Welcome.jsp—This is the login page.
- TradeHome.jsp—This page is displayed after a user logs into his account. It displays the user account information.
- Portfolio.jsp—This page displays the stock portfolio of a specific user.
- Account.jsp—This page displays information, such as the account balance of a specific user.
- MarketSummary.jsp—This page displays the stock market summary for that day.

Figure 7 shows the sample policies for generating the cache ID, dependency ID, and invalidation ID for the portfolio page in Trade3. After a user logs into his account, he can view his portfolio by accessing the controller servlet with the query parameter "action" set to "portfolio." When a user logs in, his user ID is stored as "uidBean" in the HTTPSession object. Because a portfolio page should be unique for a specific user, its cache ID should include the value of uidBean stored in the session. But as mentioned earlier, for a page to be edge cacheable, its cache ID should be composed of only query parameters, headers, and cookie values; therefore, in order to make the portfolio page edge-cacheable, its cache policy is based on the JSESSIONID cookie value (instead of the uidBean value). In Trade3, a new JSESSIONID cookie is set every time a user logs into his account; hence, it would be unique for a specific user.

Figure 7 Sample cache policies

```
<cache-entry>
        <class>servlet</class>
        <name>/app</name>
        <cache-id>
                <component id="action" type="parameter">
                        <value>portfolio</value>
                        <required>true</required>
                </component>
                <component id="JSESSIONID" type="cookie">
                        <required>true</required>
                </component>
                <property name="EdgeCacheable">true</property>
        </cache-id>
        <dependency-id>Holdings_UserID
                <component id="action" type="parameter" ignore-value="true">
                        <value>portfolio</value>
                        <required>true</required>
                </component>
                <component id="uidBean" type="session">
                        <required>true</required>
                </component>
        </dependency-id>
</cache-entry>
<cache-entry>
        <class>command</class>
        <sharing-policy>not-shared</sharing-policy>
        <name>com.ibm.websphere.samples.trade.command.SellCommand</name>
        <invalidation>Holdings_UserID
                <component id="getUserID" type="method">
                        <required>true</required>
                </component>
        </invalidation>
</cache-entry>
```

In Figure 7, the <cache-id> element defines a rule for caching the portfolio page and is composed of two <component> subelements, each generating a portion of the cache ID. As mentioned earlier, the controller servlet forwards the request to the portfolio page if its query parameter "action" is set to "portfolio." Hence, the first <component> subelement is of type "parameter" named "action" and with a value of "portfolio." Because the portfolio page should be unique to a specific user, a second <component> subelement of type "cookie" with ID "JSESSIONID" is defined. Similarly, a dependency ID named "Holdings_UserID" is associated with the cache ID of the portfolio page, and it is also composed of two <component> subelements. The dependency ID is based only on the user ID of that specific user.

Every time a user sells or buys a stock, his stock portfolio changes, which requires the portfolio page to be removed from cache. An invalidation ID named "Holdings_UserID" is defined for the sell command, and it is based on the user ID of a specific user. When a user "uid1" accesses his portfolio, a cache ID is generated, and a dependency ID "Holdings_UserID:uid1" is associated with that cache ID. When the same user tries to sell one of his holdings, an invalidation ID "Holdings_UserID:uid1" is generated, and because it matches the dependency ID of the portfolio page, the portfolio page is invalidated from cache. For more information on writing cache policies, see Reference 12.

The performance measurements shown in Table 1 and obtained with Trade3 demonstrate the advantage of servlet caching in a typical J2EE application. Note that on a completely loaded system, as is the case in our stress tests, the throughput multiplied by the response time is a constant. Table 1 shows the performance improvement achieved for the Trade3 application by enabling servlet caching. The response time of the application has decreased to 44 percent

of its original value, improving the throughput by a factor of 2.21. Enabling ESI caching further improves the response time to 35 percent of its original value and results in a throughput improvement of 2.8 times.

Note that in a typical live scenario the performance improvement gained by using ESI caching is in fact much more striking than seen in our results. This is because Trade3 is more dynamic than real-world applications in that it has about the same number of buy and sell requests as requests to view data. In a real-world usage environment, the number of data retrievals is much higher than the number of data modifications, resulting in a lower number of invalidations, higher number of cache hits, and hence more outstanding performance gains.

Command caching

In the WebSphere Application Server, commands written to the WebSphere Command Architecture[12] encapsulate business logic operations and provide a standard way to invoke business tasks. Commands may access databases, file systems, and connectors to perform business logic and often execute remotely on another server. Significant performance gains can be achieved by caching command results and avoiding their repeated execution.[13]

Commands are Java objects that follow a "set, execute, get" usage pattern. The set method is used to initialize the input properties of the command object. The execute method is used to perform the specific business logic for which the command was written. Finally, the get methods are used to read the output properties that are set by the execution. Each command can be in one of three states based on which methods have been executed: "new"—meaning the command has been created but the input properties have not been set; "initialized"—meaning the input properties have been set; and "executed"—meaning the execute method has been called and output properties have been set.

Executed command objects can be stored in the cache so that subsequent instances of that command object's execute method can be served by copying output properties of the cached command to the current command for retrieval by its get methods.

The Dynamic Cache supports caching of command objects for later reuse by servlets, JSPs, EJBs, or other business logic programming (see Figure 6). To identify these cached command objects, a unique cache ID is generated based on the fields and methods that represent or are used to retrieve input properties of the command. To cache a command in an application, a cache ID creation rule must be written in the cache policy file, and the command must be changed to extend the "CacheableCommandImpl" class instead of implementing the standard "TargetableCommand" interface.

Table 1 Performance measurement with dynamic caching disabled, servlet caching enabled, and ESI caching enabled

Caching	Throughput (pages/second)	Response Time (ms)
None	51.43	214.83
Servlet Caching	113.89	94.76
Servlet + ESI Caching	143.26	75.65

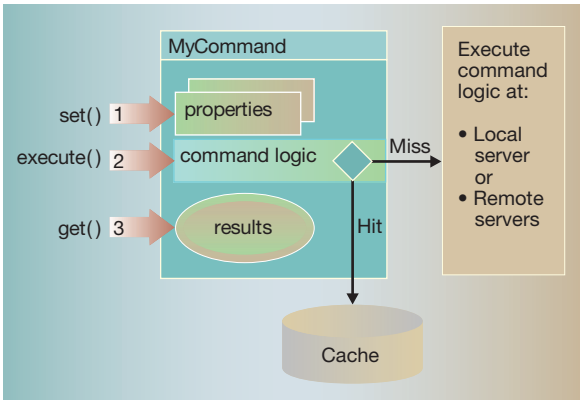
CacheableCommandImpl is an abstract class that implements the methods necessary for the command to interact with the caching framework. Because the CacheableCommand interface extends the TargetableCommand interface, the command in the application must continue to implement the methods needed for the TargetableCommand interface.

The standard TargetableCommand interface provides only the client-side interface for generic commands and declares three basic methods as described in Reference 11:

1. *IsReadyToCallExecute*. This method is called on the client side before the command passes to the server for execution.
2. *Execute*. This method passes the command to the target and returns any data.
3. *Reset.* This method resets any output properties to the values they had before the execute method was called so that the object can be reused.

When a command is called to execute, the request is intercepted by the cache, and a cache ID is generated based on the values of the input properties specified in its cache policy. If a cache entry exists for this cache ID, the output properties are copied from the cached object to this instance of the command, and the state of this instance is changed to "executed" without actually executing the business logic. If an entry with the generated ID is not found, the execute method is called, and the executed command object is stored in the cache. If an ID is not

Figure 8 WebSphere Application Server command cache



generated, the command is considered not cacheable. Figure 8 illustrates this process.

The cache policies for cacheable command objects are defined in the *cachespec.xml* file. The <cache-entry> elements identify the command objects. To cache an object, the server must know how to generate unique IDs for different invocations of that object. The <cache-id> element performs that task. Each cache entry has its own properties, including its sharing policy, priority, TTL, and single or multiple cache ID rules. Multiple cache ID rules execute in sequence until a rule either returns a non-empty cache ID or no more rules remain to execute. The ID can be generated by using the <component> subelement defined in the cache policy of a cache entry or by writing custom Java code to build the ID from input variables and system state.

Command caching and Trade3. In Trade3, cache policies are defined in the *cachespec.xml* file for a variety of brokerage application commands. The "QuoteCommand" class, used to get the stock price for a given stock symbol, is shown in Figure 9. Quote Commands are invalidated for each individual symbol when the symbol is updated with the command "UpdateQuotePriceCommand."

Because QuoteCommand is used to get the stock price of a given symbol, its cache ID is based on the getSymbol method. Similarly, the Quote_Symbol dependency ID is associated with this cache ID and is based on the getSymbol method as well. For example, when a user tries to look up the stock price of s:120, a cache ID of "QuoteCommand:getSymbol=s:120" is generated to cache the QuoteCommand and a dependency ID of "Quote_Symbol:s:120" is associated with that cache ID. The sample policies to generate the cache ID and dependency ID of QuoteCommand are shown in Figure 10.

Table 2 shows the performance improvement seen in the Trade3 application by enabling command caching. Our tests show that the response time of the application decreased to 27 percent of its original value, thus improving the throughput by a factor of 3.5.

Caching using Web services

Web services are self-contained, modular applications that are described, published, located, and invoked over a network. Key components of Web services are WSDL, UDDI, and SOAP. WSDL and UDDI are used for describing and locating an application that is provided as a Web service. SOAP provides the standard protocol for passing XML-encoded data between the servers and clients for Web services. The underlying technology for all these components is XML, which allows systems of different technologies to communicate with each other.

Web services provide effective cross-vendor and cross-organization interoperability by making use of Web protocols and defining a common framework for communication. The remote application components provided as Web services usually perform complex business tasks involving database queries and computations. To communicate with a Web service application, a client converts requests from its language to an XML format and sends the request to the server. The server parses these XML messages and converts them to the language of the server before the service is invoked. The response is again converted into an XML message format and sent back to the client. Due to this extensive amount of parsing and the complex business operations on the provider's end, the overall latency of a Web service could pose a concern for application performance. Caching Web service responses is an effective way of improving performance of a Web service application. The WebSphere Dynamic Cache supports caching of Web service responses on the server side and could improve performance by avoiding parsing and conversion of XML data during service invocation and response and/or avoiding the expense of executing the service.

Dynamic caching and Web services. A Web service message essentially consists of an HTTP header and

Figure 9 Example of class using command caching

```
public class QuoteCommand extends CacheableCommandImpl {

        public TradeServices trade = null;
        public QuoteDataBean quoteData = null;
        public String symbol = null;

        public QuoteCommand() {}

        public QuoteCommand(TradeServices trade, String symbol) {
          this.trade = trade;
          this.symbol = symbol;
        }

        public void setTradeServices(TradeServices trade) {
          this.trade = trade;
        }

        public void setSymbol(String symbol) {
          this.symbol = symbol;
        }

        public String getSymbol() {
          return this.symbol;
        }

        public QuoteDataBean getQuote() {
          return this.quoteData;
        }

        ////////////////////////// Cacheable command methods ////////////////////////////

        public boolean isReadyToCallExecute() {
                //We can only execute the command and populate this
                //bean if we have a TradeServices object on which to act
          return this.trade != null && symbol != null;
        }

        public void performExecute() throws Exception {
          this.quoteData = this.trade.getQuote(this.symbol);
        }

}
```

a SOAP envelope. The HTTP request header contains a required field called "SOAPAction" that indicates the intent of this request by using the URI (Universal Resource Identifier) of the requested service as the value of this field. The SOAPAction HTTP header is defined in the SOAP specification and is used by HTTP proxy servers to dispatch requests to particular HTTP servers. When no parameters exist for a Web service request, the cache policies can be written to build cache IDs based on this SOAPAction header and avoid parsing the SOAP envelope altogether. This method would save all parsing and execution on the server side, providing significant performance gains as well as reduction of load on the server. For example, if a Web service from a business provided its current stock quote, this service would not require parameters and could be cached based on the SOAPAction header.

The following is an example of a SOAP message with accompanying HTTP request headers. The message sample contains a SOAP message for a "getIBMQuote" operation, requesting a stock price. This is a read-only operation that gets its data from the back end with no parameters, and is cacheable. In this example, the SOAP message is cached, and a timeout

Figure 10 Sample command caching policies

```
<cache>
        <cache-entry>
                <class>command</class>
                <sharing-policy>not-shared</sharing-policy>
                <name>
                com.ibm.websphere.samples.trade.command.QuoteCommand
                </name>
                <cache-id>
                        <component type="method" id="getSymbol">
                                <required>true</required>
                        </component>
                </cache-id>

                <!-- This dependency ID is set up to identify stocks by their symbol -->
                <dependency-id>Quote_Symbol
                        <component id="getSymbol" type="method">
                                <required>true</required>
                        </component>
                </dependency-id>
        </cache-entry>
</cache>
```

Table 2 Performance measurement with all dynamic caching disabled and with command caching enabled

Caching	Throughput (pages/second)	Response Time (ms)
None	51.43	214.83
Command Caching	179.73	59.42

is placed on its entries to ensure that the quotes it returns are not out of date.

```
POST /soap/servlet/soaprouter
HTTP/1.1
Host: www.myhost.com
Content-Type: text/xml; charset="utf-8"
SOAPAction: urn:ibmstockquote-lookup
<SOAP-ENV:Envelope  xmlns:
SOAP-ENV="http://schemas.xmlsoap.
                        org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.
                xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:getIBMQuotexmlns:m=
                        "urn:ibmstockquote:>
</m:getIBMQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The cache policy for the above service could be defined as follows:

```
<cache-entry>
  <class>webservice</class>
  <name>/soap/servlet/soaprouter</name>
  <cache-id>
    <component id="" type=SOAPAction>
      <value>urn:ibmstockquote-lookup
      </value>
    </component>
  </cache-id>
</cache-entry>
```

Most Web services are more complex and require further details for a request, such as the service name, method name, and parameter values for the method. These details are provided in the SOAP envelope as part of the SOAP header and the SOAP body. The Dynamic Cache also provides an alternative method to generate cache IDs, based on the content of the SOAP envelope. In this method, the entire SOAP envelope is hashed, and the result is used as the cache ID to save the Web service response. This method, like using the SOAPAction header, saves all the parsing and repeated execution of the Web service on the server side. A cache policy using the hash of the SOAP envelope could be defined as follows:

```
<cache-entry>
<class>webservice</class>
<name>/soap/servlet/soaprouter</name>
<cache-id>
  <component id="Hash"
                    type="SOAPEnvelope"/>
  <timeout>3600</timeout>
</cache-id>
</cache-entry>
```

Various SOAP engines on different Web service clients could generate different formats of SOAP envelopes for exactly the same request. Therefore, this method could potentially result in the caching of the same response multiple times with different cache IDs. To overcome this problem, the WebSphere Dynamic Cache also provides an option to generate the cache ID based on the specific details in the SOAP envelope. This method requires parsing of the XML in the consumer's request, but prevents the recurring execution of the Web service as well as the conversion of the response to XML. This method of cache ID generation for Web services has the potential to provide the maximum number of cache hits. Selecting the right method for cache ID generation based on the type of Web service can provide high gains in performance and scalability.

In the following example, the service operation name is used to build the cache key. This name is always found in the SOAP message body and also sometimes exists in the SOAPAction HTTP header. Because reading a header is fast and simple compared to extracting the operation from the message body (which requires parsing XML and searching it), the Dynamic Cache provides different component types for the two places where the service operation name might be found.

```
<cache>
<cache-entry>
<class>webservice<class>
<name>/soap/servlet/soaprouter</name>
 <cache-id>
 <component id="" type="serviceOperation">
  <value>urn:stockquote:getQuote</value>
 </component>
 <timeout>3600</timeout>
</cache-id>
</cache-entry>
</cache>
```

Web services in Trade3. All server-side trading services are provided as Web services in Trade3. Trade3 provides a SOAP-EJB runtime mode, in which the Trade3 Web application accesses these server-side services via the SOAP protocol using the WebSphere UDDI registry and Trade3 WSDL. This mode was used to run the performance tests for Web services caching.

Table 3 Performance measurements with Trade3 in Web services access mode and Web services dynamic caching disabled and enabled

Caching	Throughput (pages/second)	Response Time (ms)
None	37.39	292.90
Web services caching	61.91	176.38

The results of the stress tests with Web services caching turned off and on are shown in Table 3. The setup used is the same as that in the previous tests except that the trade services are now being accessed using the Web services access mode. Table 3 shows the performance improvement seen in the Trade3 application by enabling Web services caching. The response time of the application decreased to 60 percent of its original value, thus improving the throughput by a factor of 1.6.

Object caching

Distributed Java object cache. Another powerful feature of the Dynamic Cache is its ability to store and share Java objects. Using the Enterprise Edition of WebSphere Application Server, a J2EE application can leverage this functionality by defining unique instances of the Dynamic Cache. Public APIs in the DistributedMap interface will then give the application direct access to that instance, which seamlessly distributes its objects across servers. An application can store and retrieve any Java object using an extended java.util.Map interface, and if the stored object implements the java.io.Serializable or java.io.Externalizable interface, WebSphere can automatically share these objects with other servers in a clustered environment. This section describes the use of the Dynamic Cache in a J2EE application to improve performance and share common information between servers in a clustered environment.

Cache instances. Cache instances are used to cache often used Java objects and to share these objects with other servers within a cluster. They are also used to logically group related object types. Objects stored in a particular cache instance are not affected by

other cache instances. This means that if an object with a given name and value is stored in cache instance *x*, an object with the same name, and possibly a different value, can also be stored in cache instance *y*. For example, if an object named ADDRESS is stored in cache instance SHOE_STORE, another object with the same name can be stored in cache instance DRESS_STORE. Each cache instance will operate independently on its own ADDRESS.

In the WebSphere network deployment environment, objects stored in a particular cache instance are available to applications on other servers by accessing a cache instance with the same JNDI (Java Naming and Directory Interface) name. The servers must be within the same replication domain to share data. Cache instances are considered resources similar in concept to a JDBC** (Java Database Connectivity) resource. The WebSphere administrative console is used to configure a cache instance.

For the system described here, two cache instances were defined with JNDI names "services/cache/instance_one" and "services/cache/instance_two." Each cache instance can be configured independently for the parameters JNDI name, cache size, disk offload enabled/disabled (which determines cache behavior if a new entry is created while the cache is full), and other parameters.

DistributedMap interface. The DistributedMap interface extends java.util.Map to access a cache instance. Using the DistributedMap interface, J2EE applications and system components can cache and share Java objects by storing a reference to the object in the cache. To store and retrieve objects in a cache instance, a DistributedMap reference for the named cache instance is needed. The following code added to the application will return a DistributedMap reference for two cache instances:

```
import javax.naming.InitialContext;
import com.ibm.websphere.cache.DistributedMap;
. . .
InitialContext ic = new InitialContext( );
DistributedMap dMap1 =
  (DistributedMap)ic.lookup("services/cache/
                                    instance_one");
DistributedMap dMap2 =
  (DistributedMap)ic.lookup("services/cache/
                                    instance_two");
. . .
```

Placing an entry in cache is performed simply by using the put and get methods. For example,

```
dMap1.put("someId",someJavaObject);
sjo = (SomeJavaObject)dMap1.get("someId");
```

In a clustered environment, an application can put an entry on one server and get it from another server. Objects can be removed from the cache by allowing the object's TTL (which is set during a put operation) to expire. For example, to place an object in cache for 30 seconds:

```
int TTL = 30; // Allow this entry to live for 30 seconds
int priority = 1;
int shareType = EntryInfo.SHARED_PUSH;
dmap1.put( "someId", someJavaObject, priority, TTL,
                                        shareType, null );
```

In the preceding example, "someJavaObject" will be placed in cache and shared (via the PUSH method) with other application servers in the cluster. After 30 seconds, the object will expire and thus will be removed from all the caches in the cluster. Alternatively, cache removal can be performed by using the explicit "invalidate" method (e.g., "dmap1.invalidate("someId")").

Invalidation events. Invalidation events allow an application to be notified when a cache entry is removed from cache. The application can then repopulate the cache with current information. This helps keep the cache up to date with current entries and prevents delays (caused by cache misses) upon user requests for the information. To use invalidation events, an event handler that implements the InvalidationListener interface must be created. The following code repopulates the cache when an entry is invalidated due to a timeout.

```
public class InvalidationEventHandler implements
                                    InvalidationListener {
  public void fireEvent(final InvalidationEvent e) {
    String name = (String)e.getId( );
    if ( e.getCauseOfInvalidation( ) ==
                        InvalidationEvent.TIMEOUT ) {
      StockQuote stockQuote =
        BackendSimulator.getStockQuote(name);
        BackendSimulator.setStockQuote(stockQuote);
    }
  }
}
```

The following code activates "invalidate" events for the above handler:

```
dMap1.enableListener(true);
InvalidationEventHandler handler = new
                        InvalidationEventHandler( );
dMap1.addInvalidationListener(handler);
```

Sharing information in a cluster. One can control how data is shared between servers in a clustered environment by setting the sharing policy at the time an object is put into the cache. There are three sharing types available for use with the DistributedMap interface that control the amount and type of information (including the object name, its value, and invalidation messages) that flows between servers: NOT_SHARED, SHARED_PUSH, and SHARED_PUSH_PULL. The sharing types are defined in the EntryInfo class. With all share types, object invalidation messages are always sent to other servers to ensure that outdated information is never served to a user.

In the case of SHARED_PUSH, the cached object and its name are sent to all servers in the replication domain at the time the object is placed in cache. This makes the object immediately available to the applications on other servers and speeds up application server performance at the expense of greater network traffic. In the case of SHARED_PUSH_PULL, the cached object is kept local to the server that created it, but the object's name is shared with other servers. If a remote server needs the object, it will request the object by name from the creating server. This performance compromise is useful when network bandwidth comes at a premium.

The NOT_SHARED option is useful when there is an affinity between a client and a Web application server and the data is used only by that client (e.g., a shopping cart). The SHARED_PUSH option is useful when the entry is very heavily used by all clients (e.g., a popular product display). The SHARED_PUSH_PULL option is useful when the entry is shared by clients but is not accessed often enough to warrant pushing the entire cached object to all other servers (e.g., a product display that is not popular).

Performance improvement for object caching. Because the Trade3 application does not make use of the DistributedMap APIs for object caching, a simple J2EE application was created to do cache puts, gets, and invalidates using the DistributedMap APIs. This application was stress tested to obtain the results that are presented here. The method shown in Figure 11 was used to gather the hit/miss counters while running the test. Each pass through this method constituted one request for a stock quote. If a request was the first quote request for a particular stock symbol, the quote was not found in the cache, and a request was made to the back-end stock quote system. Once the back-end stock quote system returned the stock quote, the quote was cached. Subsequent requests for the same stock symbol were retrieved from the cache. One thousand unique stock symbols were used. Stock quotes were randomly invalidated at the rate of 500/sec, and the back-end database delay was set to 10 ms.

The performance improvement experienced depends on many factors. Some of these factors include the request rate, back-end database response time, invalidation rate, and cache size. The hit/miss ratios shown in Tables 4, 5, and 6 represent the number of cache hits for every cache miss for the simple J2EE application. These numbers primarily confirm that the use of cache instances and the DistributedMap APIs can greatly improve throughput, and the use of InvalidationEvents can improve the hit/miss ratio. As with all dynamic caching, performance gains are tightly tied to the usage and design patterns of the application.

Performance results summary

Tables 7, 8, and 9 summarize the performance improvement results measured for dynamic caching with the Trade3 application with the setup described earlier in this paper. The performance improvements were measured in terms of gain in throughput and decrease in response time. The throughput and response time are inversely proportional to each other.

Using servlet caching, the response time of the application decreased to 44 percent of its original value, improving the throughput by a factor of 2.21. Enabling ESI caching further improved the response time to 35 percent of its original value and resulted in a throughput 2.8 times higher. Using command caching, the response time of the application decreased to 27 percent of its original value, improving the throughput by a factor of 3.5. Using servlet caching in conjunction with command caching resulted in a 75 percent decrease in the response time, lowering it to less than 25 percent of its original value. The throughput in this case improved by a factor of almost 4.

Using Web services caching in the Web services access mode, the response time of the application de-

Figure 11 Method used for generating cache hit ratios

```
public StockQuote getStockQuote(String stockSymbol, boolean useCache){

        StockQuote stockQuote = null;

        //-------------------------------------------------------------------
        // If using cache, check for stock quote in stockQuoteCache
        //-------------------------------------------------------------------
        if (useCache){
                stockQuote = (StockQuote)stockQuoteCache.get(stockSymbol);
                if (stockQuote != null){                    // Was stockQuote found?
                  statusBean.incrementCacheHit();           // If so,  log a cache hit
                  return stockQuote;                        // Exit the method with the stockQuote value
                }
        }

        //-------------------------------------------------------------------
        // Go to the back end for stock quote if
        //          1) not using cache, or
        //          2) using cache and get() results in cache miss
        //-------------------------------------------------------------------
        stockQuote = BackendStockQuoteSystem.getStockQuote(stockSymbol); // get a stockQuote

        //-------------------------------------------------------------------------------------
        // If using cache, log a cache miss and save stock quote in stockQuoteCache
        //-------------------------------------------------------------------------------------
        if (useCache){
              statusBean.setCacheMiss();
              stockQuoteCache.put(stockSymbol, stockQuote);
        }
        return stockQuote;
```

Table 4 Performance measurements with no object caching

Threads	Throughput (requests/second)
1	100
2	200
10	985
20	1950

Table 5 Performance measurements using object cache

Threads	Throughput (requests/second)	Hit/Miss Ratio
1	136	0.40
2	381	0.93
10	48000	132
20	59000	149

creased to 60 percent of its original value, thus improving the throughput by a factor of 1.6. Using object caching resulted in a drastic increase in the number of requests completed per second, and using invalidation listeners in conjunction with this resulted in an 8 times higher hit/miss ratio.

Table 6 Performance measurements using object cache and invalidation listeners

Threads	Throughput (requests/second)	Hit/Miss Ratio
1	26000	539
2	42000	838
10	61000	1171
20	64000	1169

Concluding remarks

As the industry trend moves towards conducting more business over the Web, competitive pressures drive e-business sites to deliver more personalized and hence, more dynamically generated content. Due to variations in workload, providers of dynamic Web sites and e-commerce services need to find a cost-effective solution to keep applications scalable and

Table 7 Performance measurements with Trade3 in standard access mode and Web services dynamic caching disabled and enabled

Caching	Throughput (pages/second)	Response Time (ms)
None	51.43	214.83
Servlet caching	113.89	94.76
Servlet and ESI caching	143.26	75.65
Command caching	179.73	59.42
Servlet and command caching	200.19	53.62

Table 8 Performance measurements with Trade3 in Web services access mode and Web services dynamic caching disabled and enabled

Caching	Throughput (pages/second)	Response Time (ms)
None	37.39	292.90
Web services caching	61.9	176.38

Table 9 Performance measurements with ten threads and object caching and invalidation events disabled and enabled

Caching	Throughput (requests/second)	Hit/Miss Ratio
None	985	N/A
Object caching	48000	132/1
Object caching and invalidation events	61000	1171/1

to avoid server bottlenecks while still providing viable response times. Dynamic caching at various tiers of a J2EE application provides such a solution by avoiding CPU-intensive or back-end-intensive computations on the servers and reusing response data when possible. In most cases no change is required to the application in order to use the Dynamic Cache service. Also, the service is conveniently offered as a built-in solution with WebSphere Application Server and can be easily enabled through the administrative console.

Using Trade3, the IBM J2EE benchmark application, our performance tests showed an improvement of 2.2 times in throughput using Dynamic Cache servlet caching without ESI caching and an improvement of 2.8 times using servlet caching with ESI caching. Command caching and Web services caching resulted in improvements of 3.5 and 1.6 times respectively and, using a basic sample application, object caching showed sweeping gains in performance. Improvements in performance vary based on how dynamic the data is, the intensity of saved computations, and most important, the cache policy configuration. Because of this, the WebSphere Application Server also provides tools to configure, tune, and monitor the cache in order to optimize the benefits from the Dynamic Cache.

The Dynamic Cache is also capable of leveraging the services of distributed notification and replication services to efficiently spread the benefit of saved computation across processes and further improve scalability. Moreover, the cache is designed to work together with external caches that exist outside the J2EE application space such as WebSphere Edge Server, WebSphere HTTP Plug-in, or the Akamai ESI Server. The Dynamic Cache thus provides an ideal solution for high volume Web sites serving dynamic content, enabling them to achieve high levels of scalability and performance.

Acknowledgments

We would like to thank Kenichiroh Ueno, Harley Stenzel, and Greg Ames for help with performance measurements setup, and Stan Cox for help with Trade3 information.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc. or Linus Torvalds.

Cited references and notes

1. *WebSphere Application Server Library*, IBM Software, http://www.ibm.com/software/webservers/appserv/was/library/.
2. *High-volume Web Sites, Library of Best Practices*, IBM Corp., http://www7b.software.ibm.com/wsdd/zones/hvws/library.html.
3. Trade3 is available for download at http://www.ibm.com/software/webservers/appserv/benchmark3.html.
4. *JCACHE - Java Temporary Caching API*, Java Community Process (2001), http://www.jcp.org/en/jsr/detail?id=107.
5. The policy for an object may contain multiple rules to generate the unique key (cache ID), based on different sets of input parameters. These rules are executed in order until one of the rules returns a non-empty cache ID (based on the current input parameters) or until no more rules remain to be executed.
6. *Caching In on the Enterprise Grid: Turbo-Charge Your Applications with OracleAS Web Cache*, Oracle Technology Network, Oracle Corporation (September 2003), http://otn.

oracle.com/products/ias/web_cache/pdf/OracleAS-Web-Cache-10g-904-twp.pdf.
7. *Tangosol Coherence User Guide*, Tangosol, Inc. (February 2004), http://www.tangosol.com/userguide.jsp.
8. *Welcome to SpiritCache 3.0*, Spritsoft, Ltd., (2004), http://www.beyondjms.com/documentation/cache/3.0/index.html.
9. S. Viriri, "Dynamic Caching Design Proto-Pattern for J2EE Web Component Development," *Journal of Object Technology* **2,** No. 4, 113–117 (July 2003).
10. G. Copeland and M. McClain, "Web Caching With Dynamic Content," http://research.microsoft.com/~gray/HPTS99/papers/Copeland_McClain.pdf..
11. *ESI Language Specification 1.0*, Edge Side Includes (2001), http://www.esi.org/language_spec_1-0.html.
12. *WebSphere Application Server Network Deployment - Command Class*, IBM Corp., (October 2003), http://publib.boulder.ibm.com/infocenter/wsphelp/topic/com.ibm.websphere.nd.doc/info/ae/ae/rprf_commandclass.html.
13. G. Cuomo and C. Diep, "The Dynamic Caching Services: Eliminate Bottlenecks and Improve Response Time," *WebSphere Developer's Journal* (February 2003), http://www.findarticles.com/m0MLX/2_2/103192558/pl/article.jhtml?cf=dls..

General references

Java 2 Platform, Enterprise Edition (J2EE), Sun Microsystems, http://java.sun.com/j2ee.

WebSphere Software Platform, IBM Corp., http://www.ibm.com/websphere.

Accepted for publication December 8, 2003.

Radoslava Bakalova *IBM Software Group, 4205 S. Miami Blvd., Durham, NC 27703 (rgbakalo@us.ibm.com)* Ms. Bakalova is a software developer in the IBM WebSphere Application Server cache development group in Research Triangle Park, NC. She received her Bachelor's degree in business economics in July 1996 and her M.B.A. degree in November 1997 from the University of National and World Economy in Sofia, Bulgaria. Ms. Bakalova is currently finishing her M.S. degree in information technology at the Rochester Institute of Technology in Rochester, New York. Her concentrations are object-oriented programming and database administration. She joined IBM as a supplemental employee in December 2002.

Andy Chow *IBM Software Group, 4205 S. Miami Blvd., Durham, NC 27703 (cachow@us.ibm.com).* Mr. Chow is a staff software engineer in IBM's WebSphere Application Server development group in Research Triangle Park, NC. He received a B.S. degree in electrical engineering from the University of Michigan at Ann Arbor and an M.S. degree in electrical engineering from the Georgia Institute of Technology. He joined IBM in 1983 and has worked on developing diagnostics and configuration software for the network adapters used in the IBM 3174 and 3745 communications controllers and personal computers. He has also worked on developing software for host integration products and edge services components. His current assignment is in WebSphere Dynamic Cache development.

Chuck Fricano *IBM Software Group, 4205 S. Miami Blvd., Durham, NC 27703 (fricano@us.ibm.com).* Mr. Fricano is an advisory software engineer with IBM's WebSphere Application Server development group in Research Triangle Park, NC. He received a degree in electronics from the Penn Technical Institute in Pittsburgh, Pennsylvania in 1976. He joined IBM in 1979 and worked in the Pittsburgh Field Engineering division for 11 years. In 1990 he was transferred to the Networking Support Center in Research Triangle Park where he provided remote technical support for the IBM 3745 communications controller and related software products, including VTAM (Virtual Telecommunications Access Method) and NCP (Network Control Program). In 1995, he transferred to the programming lab in Research Triangle Park where he has been developing Internet-related products. His current assignment is in WebSphere Dynamic Cache development.

Priyanka Jain *IBM Software Group, 4205 S. Miami Blvd., Durham, NC 27703 (pjain@us.ibm.com).* Ms. Jain is a software engineer in the IBM WebSphere Application Server cache development group in Research Triangle Park, NC. She received a Bachelor of Engineering degree in computer science and engineering from Bangalore University in India in 1998 and an M.S. degree in computer science from Louisiana State University in Baton Rouge, LA in 2001. She joined IBM the same year, and has been working with the dynamic caching group since May 2002.

Nirmala Kodali *IBM Software Group, 4205 S. Miami Blvd., Durham, NC 27703 (kodalin@us.ibm.com).* Ms. Kodali is a staff software engineer in IBM's WebSphere Application Server development group in Research Triangle Park, NC. She received an M.S. degree in computer science from the University of Tennessee at Knoxville in 1998 and joined IBM in the same year. Since joining IBM, she worked on the IBM WebSphere HostPublisher product, then joined WebSphere Application Server development in August 2002.

Dan Poirier *IBM Software Group, 4205 S. Miami Blvd., Durham, NC 27703 (poirier@us.ibm.com).* Mr. Poirier is an advisory software engineer in IBM's WebSphere Application Server development group in Research Triangle Park, NC. He received an M.S. degree in computer science from the University of North Carolina at Chapel Hill in 1991. He joined IBM in 1992, and has worked on numerous networking products ranging from an SNA implementation for PC-DOS to his current work on the WebSphere Application Server.

Sajan Sankaran *IBM Software Group, 4205 S. Miami Blvd., Durham, NC 27703 (sajan@us.ibm.com).* Mr. Sankaran is an advisory software engineer working on dynamic caching in WebSphere Application Server development. He received an M.S. degree in computer science from the University of North Carolina at Charlotte. He joined IBM in 1999 and has worked on host integration products and edge services components before his current assignment in WebSphere Application Server development.

Dan Shupp *IBM Software Group, 4205 S. Miami Blvd., Durham, NC 27703 (shupp@us.ibm.com).* Mr. Shupp is a staff software engineer with IBM's WebSphere Application Server development group in Research Triangle Park, NC. He received a B.A. degree in computer science from Johns Hopkins University in 2000. Since joining IBM, he has worked on emerging Internet technologies for the WebSphere Performance Group and has worked on dynamic caching since it was incorporated into WebSphere in Version 3.5.3. He is currently on assignment in Beijing, working with IBM China Research Labs on grid computing projects for WebSphere.