Evaluation of Edge Caching/Offloading for Dynamic Content Delivery

Chun Yuan
Microsoft Research Asia
3F Sigma Center, #49 Zhichun Road
Beijing 100080, China
86-10-62617711
t-cyuan@microsoft.com

Yu Chen
Microsoft Research Asia
3F Sigma Center, #49 Zhichun Road
Beijing 100080, China
86-10-62617711
i-yuchen@microsoft.com

Zheng Zhang
Microsoft Research Asia
3F Sigma Center, #49 Zhichun Road
Beijing 100080, China
86-10-62617711
zzhang@microsoft.com

ABSTRACT
As dynamic content becomes increasingly dominant, it becomes an important research topic as how the edge resources such as client-side proxies, which are otherwise underutilized for such content, can be put into use. However, it is unclear what will be the best strategy and the design/deployment tradeoffs lie therein. In this paper, using one representative e-commerce benchmark, we report our experience of an extensive investigation of different offloading and caching options. Our results point out that, while great benefits can be reached in general, advanced offloading strategies can be overly complex and even counter-productive. In contrast, simple augmentation at proxies to enable fragment caching and page composition achieves most of the benefit without compromising important considerations such as security.

Categories and Subject Descriptors
H.3.4 [**Information Storage and Retrieval**] Systems and Software – *distributed systems, performance evaluation (efficiency and effectiveness)*; H.3.5 [**Information Storage and Retrieval**] Online Information Services – *Web-based services*

General Terms
Measurement, Performance

Keywords
Edge caching, offloading, dynamic content

1. INTRODUCTION
Dynamic pages will dominate the Web of tomorrow. Indeed, one should stop talking about dynamic *pages* but, instead, dynamic *content*. This necessitates architectural change in tandem. In particular, resources that are already deployed near the client such as the proxies that are otherwise underutilized for such content should be employed.

Legitimate strategies include offloading some of the processing to the proxy, or simply enhancing its cache abilities to cache fragments of the dynamic pages and perform page composition. While performance benefits including latency and server load reduction are important factors to consider, issues such as engineering complexity as well as security implication are of even higher priority. Although there have been extensive researches on the subject of optimizations for dynamic content processing and caching, we still lack the insight on what will be the best offloading and caching strategies and their design/deployment tradeoffs.

In this paper, using a representative e-commerce benchmark, we have extensively studied many partitioning strategies. We found that offloading and caching at edge proxy servers achieves significant advantages without pulling database out near the client. Our results show that, under typical user browsing patterns and network conditions, 2~3 folds of latency reduction can be achieved. Furthermore, over 70% server requests are filtered at the proxies, resulting significant server load reduction. Interestingly, this benefit can be achieved largely by simply caching dynamic page fragments and composing the page at the proxy. In fact, advanced offloading strategies can be overly complex and even counter-productive performance-wise if not done carefully. Our investigation essentially boils down to one simple recommendation: if end-to-end security is in place for a particular application, then offload all the way up to the database; otherwise augment the proxy with page fragmentation caching and page composition. While our results are obtained under the .NET framework, we believe they are generic enough to be applicable to other platforms.

The rest of the paper is organized as follows. Section 2 covers related work. Various offloading and caching options are introduced in Section 3, which also discusses important design metrics. Section 4 examines the benchmark used and also some of the most important .NET features employed. Detailed implementations of the offloading/caching options are discussed in Section 5. Section 6 describes the experiment environment. Results and analysis are offered in Section 7. Finally, we summarize and conclude in Section 8.

2. RELATED WORK
Optimizing dynamic content generation and delivery has been widely studied. The main objectives are to reduce client response time, network traffic and server load caused by surges of high volume of requests over wide-area links. Most works focus on how to support dynamic content caching on server side [8][9][17]. Some others also extend their cache to the network edge and show better performance result [10]. Fragment caching [3][4] is an effective technique to accelerate current Web applications which usually generate heterogeneous contents with complex layout. It is provided by today's common application server product like Microsoft ASP.NET [13] and IBM WebSphere Application Server [6]. ESI [5] proposes to cache fragments at the CDN stations to further reduce network traffic and response time.

Application offloading is another way to improve performance. In Active Cache [2], it is proposed that a piece of code be associated with a resource and be able to be cached too. The cache will execute the code over the cached object on behalf of the server and return the result to the client directly when the object is requested at a later time. With the blurring of application and data on current Web, this scheme becomes less effective. To do more aggressive application offloading, WebSphere Edge Services Architecture [7] suggests that portions of the application such as presentation tier and business logic tier be pushed to the edge server and communicate with the remaining application at the origin server when necessary via the application offload runtime engine. An extreme case of offloading is given by [1]. The full application is replicated on the edge server and database accesses are handled by a data cache which can cache query results and fulfill subsequent queries by means of query containment analysis without going to the back-end.

We focus on the proxies that are already installed near clients. We also examine exclusively on offloading and caching of anything other than the database content, as we believe mature technologies to manage hard states in a scalable fashion across wide-area are yet to be developed. To the best of our knowledge, we are the first to report design and implementation tradeoffs involved in devising partitioning and offloading strategies, along with detailed evaluations. There also has been no work evaluating offloading versus advanced caching mechanisms. Finally, this is the first work we know of that experiments with the .NET framework in this aspect.

3. OFFLOADING AND CACHING OPTIONS ENUMERATED

There are a number of issues to be considered for distributing, offloading and caching dynamic content processing and delivery, they are: 1) available resources and their characteristics, 2) the nature of these applications and 3) a set of design criteria and guidelines. In this section, we discuss these issues in turn.

3.1 Resources Where Offload can be Done

Figure 1 shows graphically various resources involved.

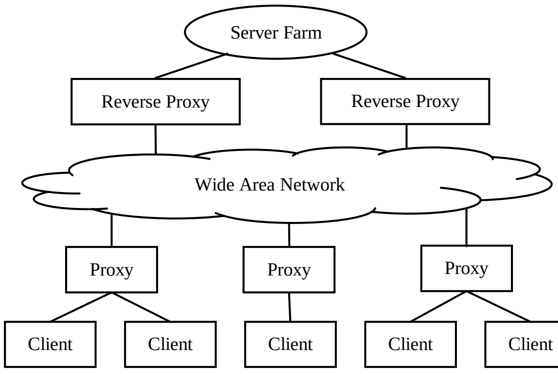


Figure 1. Resources available for offloading and caching

Client. As a user-side agent, client – typically a browser – is responsible for some of the presentation tasks, it can also cache some static contents such as images, logos etc. The number of clients is potentially many; however they usually have limited capacities and are (generally speaking) not trusted.

Proxies. In terms of scale, proxies come second. Proxies are placed near the clients and are thus far from the server end. The typical functionalities of proxies include firewall, and caching of static contents. They are usually shared by many clients and are reasonably powerful and stable. However, except the case of intranet applications, content providers do not have much control over them.

Reverse Proxies. Reverse proxies are placed near the back end server farm and act as an agent of the application provider. They serve the Web request on behalf of the back end servers. Content providers can fully control their behaviors. However, the scale of reverse proxies only goes as far as a content provider's network bandwidth allows. In this paper, we consider them as part of the server farm.

Server. Servers are where the content provider has the full control. In the context of this section, we speak of "server" as one logical entity. However, as it shall be clear later, "server" itself is a tiered architecture comprised of many machines and hosting the various tiers of the Web application.

As far as dynamic content is concerned, typically only the servers and clients are involved. Proxies, as of today, are incapable of caching and processing dynamic contents. In this discussion, we have also omitted CDN stations as we believe they can be logically considered as an extension of either proxies or reverse proxies. Some of the more recent progresses have been discussed in the Section 2.

3.2 Application Architecture and Offloading Options

Logically, most of the Web applications can be roughly partitioned into three tiers: presentation, business logic, and back end database. The presentation tier collects users' input and generates Web pages to display results. The business logic tier is in charge of performing the business procedure to complete users' requests. The database tier usually manages the application data in a relational database.

Based on the 3-tier architecture, N-tier architecture is also possible. The most complex tier in a Web application is the business logic tier. This tier performs application-specific processing and enforces business rules and policies. Because of its complexity, the business tier logic tier itself may be partitioned into smaller tiers, evolving into the N-tier architecture.

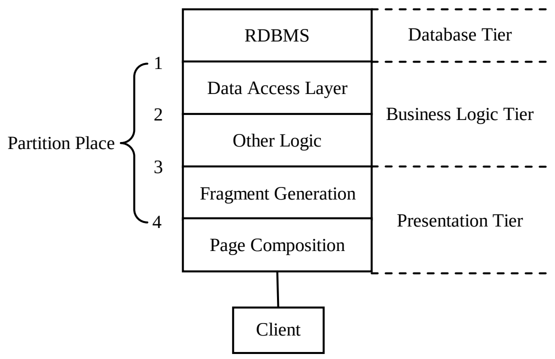


Figure 2. The 3-tier architecture and partition places

Application partitioning and offloading can be applied based on the tier structure of the Web application. Without loss of generality, we only consider Web browser as the application client here. From the back end database to browsers, we can find several candidate partition places as shown in Figure 2.

The first partition place is the database access interface. The ODBC, JDBC, ADO etc. are this kind of interfaces. Current applications use connection strings to specify the database server to be accessed. It is possible to point to a specific remote machine in the connection string.

The second partition place is at the data access layer inside the business logic tier. Because of the complexity of the business logic, it is a common practice to develop a set of database access objects to shield the detail inside the database. Other business logic objects can access data through these objects using simple function calls. The clear-cut boundary at this layer makes it a good candidate of partition point for offloading.

The third partition place is between the presentation tier and the business logic tier. The presentation tier gathers the user input and translates the user request into a processing action at the business layer. The business tier usually provides a single-call interface for each type of requests. The clearly defined interface here provides strong clues for partition.

The fourth partition place is inside the presentation tier. The Web pages generated by the application are structuralized and split into fragments each of which has consistent semantic meaning and life time. The back end servers provide page fragments and composition frameworks. The entire Web page is assembled at the offloading destination. ESI [5] is a good example of this strategy.

Of course, what we have enumerated here is only a starting point. Specifically, within the business logic tier there can be multiple logically legitimate offloading points. However, as we should discuss later, advanced offloading strategies often risk high complexity without clear benefit in return.

3.3 Important Factors to be Considered When Offloading

Having discussed various resources upon which offloading and caching can be performed, and various partitioning strategies, the actual implementation and deployment must consider a number of important factors. In our opinion, the following three are the most important ones: *security*, *complexity* and *performance*.

Security. Sensitivity of data as well as processing that are to be offloaded may vary. A given piece of data and processing can be distributed as far as its security perimeter permits. This is one reason we are concerned with who controls what in the resource distribution earlier. Enforcing security end-to-end only applies to certain Web applications (e.g. intranet) and pays a cost (e.g. VPN overhead) in return.

Complexity. Another factor that should be considered is the engineering cost. Although Web applications are developed according to 3-tier or N-tier architecture, the tier boundaries are usually not clear. This problem is obvious for the tiers that are part of the business logic in an N-tier application.

Even if the tier boundaries are clear, the implementation still cannot be fully automated. For example, application partitioning usually requires transforming some of the LPCs (local procedure call) into RPCs (remote procedure call). Because most of the runtime systems do not support migrating LPC to RPC transparently, source code modification, recompilation and subsequent testing are necessary. If synchronous procedures calls are to be changed to asynchronous calls, the implementation efforts would be even greater.

Performance. Even when resources such as proxies are freely available, distributing the processing and caching must bring significant benefits to justify the additional complexity involved. End user's latency as well as improvement of scalability are the primary metrics. On this, the network condition is the first critical factor to be considered. Generally speaking, the communication quantity across the partition should be minimized on low bandwidth networks. Likewise, for high latency networks, the frequency of synchronous communication should be reduced. In general, a useful guideline to start with is that communication channel over wide area network should be light weight and stateless.

4. THE PET SHOP BENCHMARK

In order to evaluate different offloading options, we use Microsoft .NET Pet Shop as our benchmark. It comes from Sun's primary J2EE blueprint application, the Sun Java Pet Store [16] and models a typical e-commerce application, an online pet store. E-commerce sites like this are among the most common Web applications.

Pet Shop is implemented using ASP.NET, and the source code is freely available at [11]. ASP.NET brings several important optimizations and the two of them, *stored procedure* and *output caching* will be discussed in the following sections.

4.1 Pet Shop Architecture

The complete 3-tier architecture of Pet Shop is described in the whitepaper at [11]. To illustrate the design, we will look at an example of the interaction between the three tiers as shown in Figure 3.

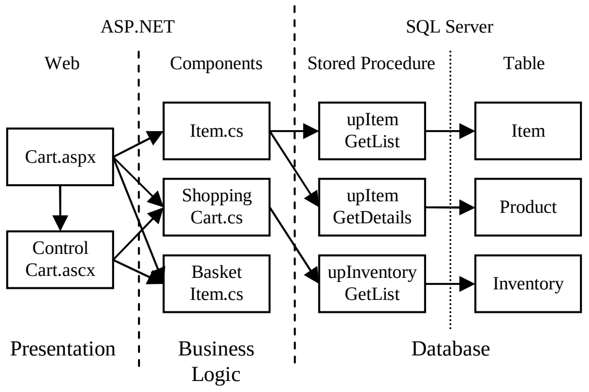


Figure 3. The Pet Shop architecture (portion)

The *presentation tier* communicates with browsers directly. It contains *Web Forms pages* (*aspx* files), *Web Forms user controls* (*ascx* files) and their *code-behind* classes (in namespace `PetShop.Web`). Similar to the ASP and JSP page, Web Forms pages represent dynamic pages. The Web Forms user controls represent portions of Web Forms pages and thus can not be requested independently. While the aspx and ascx files contain the visual representation, the code-behind classes contain processing logics. When a request arrives, the specified Web Forms page and Web Forms user controls are loaded. The corresponding code-behind objects responsible for generating responses will initiate calls to the business logic tier for request processing (arrows in Figure 3).

The objects in the *business logic tier* (in namespace `PetShop.Components`) accept invocations from the presentation tier. If the processing does not require database interaction, for instance displaying shopping cart content, results are returned right away. Otherwise, the business logic objects will generate database queries through a specific database access class (`PetShop.Components.Database`). Instances of this class set up database connections, pass database queries through ADO.NET interfaces and return query results to upstream.

The *database tier* consists of application data and *stored procedures*. A *stored procedure* is used to encapsulate a sequence of SQL queries which complete a single task. Using *stored procedures*, interactions between the business logic tier and the database tier can be reduced, thus increasing performance. For instance, placing an order normally requires several calls between the business tier and the back end database. With stored procedure, an order can be encoded into a string and transferred to the database, where the string is decoded and multiple SQL statements are issued to complete the order. From this perspective, most of the Pet Shop *stored procedures* are essentially part of the business logic tier. They are included in the database tier simply because they are stored and are executed in the SQL server. This is one example where the boundaries of tiers get blurred.

4.2 ASP.NET Output Caching

The.NET Pet Shop leverages ASP.NET output caching to increase throughput and reduce server load [12]. Similar function is also provided by other products such as IBM WebSphere's *response cache* [6][7]. When a page is requested repeatedly, the output caching allows subsequent requests to be satisfied from the cache so the code that initially creates the page does not have to be run again. Besides caching the entire page, ASP.NET allows Web Forms user controls to be cached separately. As we will explain in detail in Section 5.3, this feature of fragment caching is what we employ to enhance caching capability at the proxy side.

ASP.NET provides duration and versioning control for each cached entities (*Web Form* and *Web Form user control*). Duration specifies the life time of a cached page. Versioning allows caching multiple result pages or page fragments for a single form or control. For example, `Product.aspx` produces different result pages for different products. Storing a single result page in output cache can hardly gain any benefit since users tend to browse different products. By keeping multiple result pages, the most frequently accessed pages will be cached eventually, saving large amounts of processing time.

5. EXPERIMENT PREPARATION

In this section, we discuss in detail how different offloading and caching strategies are implemented in Pet Shop.

According to the partition points in section 3.3, the following offloading options are investigated: F_0, F_{db}, $F_{remoting}$ and F_{proxy}. They are shown in Figure 4; the legends are:

- B: Browser
- A: Page Assembling and Fragment Caching
- G: Fragment Generation
- P: Presentation
- L': Business Logic except Data Access Layer
- DA: Data Access Layer
- L: Business Logic
- DB: Database
- Cloud: Wide Area Network

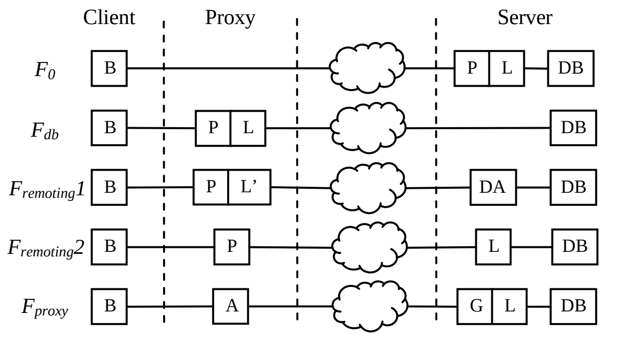


Figure 4. Implementation of offloading options in Pet Shop

The base line is F_0 which leverages neither processing nor caching abilities of proxies. By pushing fragment caching and page assembly to the proxy, we get F_{proxy} which corresponds to partition place 4. By offloading the presentation tier to proxies, $F_{remoting}2$ implements partition point 3. $F_{remoting}1$ and F_{db} are similar except that $F_{remoting}1$ leaves the data access layer at back end servers while F_{db} offloads the complete business logic tier. Therefore, they correspond to partition point 2 and 1, respectively.

We use proxy and front end, server and back end interchangeably in this paper for all configurations other than F_0.

5.1 Implementing F_{db}

The implementation of F_{db} is trivial: all we need to do is modify the connection string of the database access interface. The connection string is changed from the default value "`server=localhost;…`" to "`server=some other machine; …`" so that the front end is forced to access the remote machine hosting the SQL server.

5.2 Implementing $F_{remoting}$

The $F_{remoting}$ option investigates different ways of offloading inside the business logic tier, in particular the partition point 2 and 3 (see Figure 2). We employ the .NET Remoting feature to accomplish this task, which we will discuss first.

5.2.1 .NET Remoting

Microsoft .NET Remoting provides a rich and extensible framework for objects living in different application domains, in different processes, and in different machines to communicate with each other seamlessly. The framework considers a number of matters, including *object passing*, *remote object hosting strategy*, *communication channel* and *data encoding*.

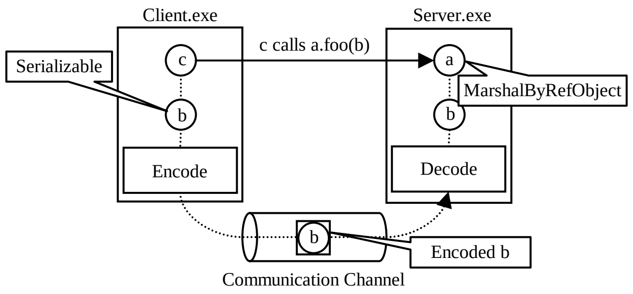


Figure 5. RPC using .NET Remoting machanism

Objects can be passed either by reference or by value:

- **By value**. Objects that would cross application domain boundary, such as the object `b` in Figure 5, can be passed by value. In .NET Remoting, pass-by-value objects are all marked with *Serializable* attribute.
- **By reference**. Objects that reside in only one application domain and provide interfaces to other applications are passed by reference (such as the object `a` in Figure 5). In .NET Remoting, pass-by-reference objects should be derived from a system class *MarshalByRefObject*.

For a pass-by-reference object, .NET Remoting provides three hosting strategies to support object activation and lifetime management.

- *SingleCall* objects' activation and lifetime are determined by server. They service one and only one request coming in, i.e. different client requests are services by different objects.
- *Singleton* objects' activation and lifetime are also determined by server. Unlike the *SingleCall* objects, *Singleton* objects service multiple clients and share data by storing state information between client invocations. There is only one singleton object instance of a given class at the server side.
- *Client-activated* objects' (*CAO*) activation and lifetime are determined by client. The server creates an object upon an activation message from client. The object services for the client until the client allows it to be released. If the communication between server and client is stateful, *CAO* should be used.

For a more in-depth treatment of these hosting strategies, please refer to [14].

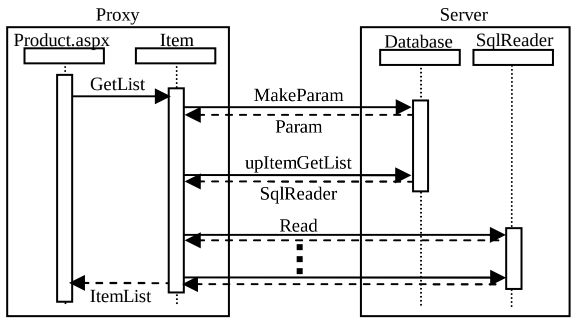
RPC requests and responses are encoded into formatted messages and transferred over a communication channel. In Figure 5, when object `c` makes a call to object `a`, the request (including object `b` as the parameter) is encoded and transferred to the server side. At the server side, the message is decoded and an actual call to object `a` is made.

5.2.2 Detailed Implementation

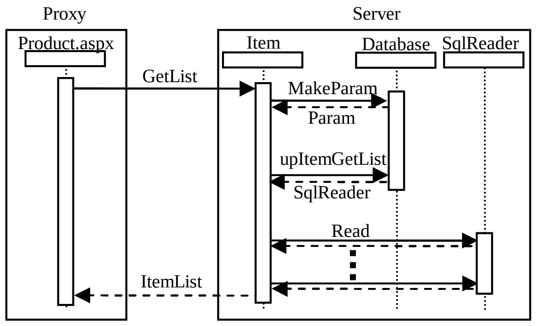
As explained earlier, in $F_{remoting}$, we try to partition the application in the logic tier. While there maybe many different options, as an extensive exercise we investigate how to partition at point 2 which separates the data access layer from other business logic layers, and point 3 which is located between the presentation tier and the business logic tier (see Figure 2 and Figure 4).

Regardless of the specific partition strategy, the task in this configuration is always to replace LPC (Local Procedure Call) with RPC (Remote Procedure Call). This entails a few steps. The first is to determine the locations of classes to be run as mandated by a given partitioning strategy (server or proxy) and from there derive the RPC boundaries. The second step is to modify the application source code so that RPC can take effect. Finally, the hosting strategy for objects at server side and communication channel between server and proxy are decided.

Partition place 2 requires the least amount of engineering efforts in that there is only one class to be modified — `Database` in namespace `PetShop.Components`. The `Database` objects run at server side and provide interfaces for the other logical tier objects to access information in the back end database. Thus, they are pass-by-reference objects. For each user request, the responsible logic tier object at proxy side needs to issue multiple procedure calls to server (as shown in Figure 6(a)). Because these calls are related to each other, the state information along the call sequence should be maintained. On the other hand, different user requests need exclusive objects to provide services. Therefore, the only hosting strategy is *CAO*. This strategy turns out to have a dramatic performance impact: our test runs reveal that the benchmark now performs much worse than not offloading *at all*. The reason is that multiple RPCs corresponding to a single user request results in multiple round trips between proxy and server. Consequently, partition point 2 is not a good offloading option for Pet Shop. The lesson we learned here is that, under a given partition strategy, the logical constraint and performance constraint may very much be in conflict with each other. What makes this matter even more complex is that this is also application dependent.



(a) $F_{remoting}1$



(b) $F_{remoting}2$

Figure 6. Flow examples of partition place 2 (a) and 3 (b)

Partition point 3 works but with a rather involved manual examination of all candidates. Specifically, all classes in namespace `PetShop.Components` except `Error` and `Database` are affected[1]. Some of them will reside at server side only and act as pass-by-reference remote objects, such as `Item`, `Profile`, `Order`, `Customer` and `Profile`. The others are pass-by-value objects that will travel between proxy and server, such as `ShoppingCart`[2], `BasketItem`, `ItemResults`, `ProductRe-sults` and `SearchResults` (Actually they are all RPC parameters or responses.). With this strategy, the proxy needs to issue only one RPC for each user request in most cases and no state information is going to be shared among different RPCs (as shown in Figure 6(b)). Moreover, because the objects of `Item`, `Profile`, `Order`, `Customer` and `Profile` do not store state

[1] `Error` objects are responsible to report local errors. Although `Database` objects reside at server side, they do not provide interfaces for proxy any more in this option.

[2] Actually `ShoppingCart` will access database in `Cart.aspx` when updating the shopping cart information. To deal with it, we modify `Cart.aspx` and `Order` to ensure that the update would be executed on server.

data[3], there is no difference whether distinct user requests are processed by a shared object or exclusive and different ones. Therefore, all the three hosting strategies are eligible. We conducted an experiment to compare the performance of all of them under the same test condition (as shown in Table 1) and found that *SingleCall* and *Singleton* outperforms *CAO* significantly. In the experiment, we notice that under the same conditions the memory consumption and CPU load on server for *CAO* is higher than that for *SingleCall* and *Singleton*. This may be due to the fact that the server no longer can perform aggressive garbage collection as it can in the other two options.

Table 1. Comparison of *Singleton*, *SingleCall* and *CAO*

	Requests/sec	Response time (ms)
Singleton	649.07	24.66
SingleCall	649.17	24.46
CAO	552.31	43.75

In the following sections, $F_{remoting}$ will refer to $F_{remoting}2$ using *Singleton* only. While we did arrive at an adequate partition strategy in the business logic tier for Pet Shop (see performance results in Section 7), our experience pointed out that this is a rather complex process and, even though it is possible to derive a consistent set of guidelines, it will be quite a challenge to perform automatic partitioning.

5.3 Implementing F_{proxy}

As we described earlier, the goal of F_{proxy} is much more modest: augment the proxy cache so that it can cache fragments and perform dynamic page assembly. Recall that the objects in the *presentation tier* of the original Pet Shop are divided into two parts: container pages (*Web Forms*) and fragments (*Web From user controls*). Each container page includes placeholders of some fragments.[4] Their contents, either obtained from the output cache or generated afresh upon a miss, are to be inserted into the container at runtime to compose a complete Web page.

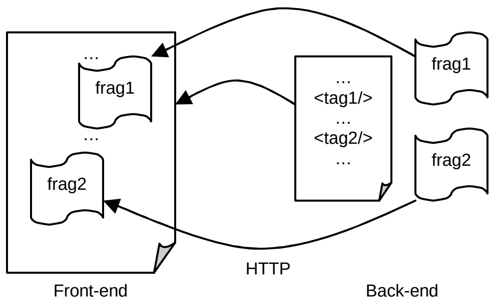


Figure 7. Page fragment composition

The flow of F_{proxy} is shown in Figure 7. In order to accomplish this task, we need to 1) replicate the tier responsible for output caching and page assembly at the front-end and 2) make a back-end version of Pet Shop which is the real generator of content.

The back-end version is implemented in two steps. First we make each fragment be able to be separately retrieved with a URL like a common Web page. Then in their container pages, we replace the fragment's placeholder with a special tag that will not be interpreted by ASP.NET but indicate to the front-end that it is to be expanded with the content of a fragment.

We create another application to play the role of output caching and page assembly at the front-end. This application has the same set of pages and fragments as Pet Shop and their containment relationships are also maintained, but no actual content is included. When a page is loaded, the container page and the nested fragments will be loaded in turn. What they actually do is retrieving their corresponding "page" from the output cache in case of cache hits, or from the back-end application otherwise. After the real content has arrived, the page composition begins. The special tags in the fetched container page are replaced with the content of its subordinate fragments. The process will be recursively performed if there are nested containments, until the page is finally composed.

In summary, here are the steps that will occur at runtime:

1. client sends request to proxy
2. proxy fetches container and fragments either from the output cache (now hosted at proxy) or request from backend
3. proxy assemble the page and returns to the client

We should point out that this version of implementation is for a quick evaluation of the F_{proxy} option and is much more like a hack: there are no modifications to either proxy or .NET framework anywhere, we simply replicate some of the .NET functionality at the proxy side. As a result, it incurs additional overhead that would otherwise absent in a more solid and application-independent implementation, which is one of our ongoing works.

6. EXPERIMENT SETUP

6.1 The Test Bed

There are totally five machines in our experiment: two servers and three clients (as shown in Figure 8). Both servers are powerful machines each with two Pentium4 2.2GHz CPUs, 2GB RAM and two 73GB SCSI disks. The clients are PCs with sufficient power so that they never become bottlenecks in test runs. The three clients are connected to the front end server through a 100Mbps Ethernet switch. In order to avoid contentions between front-end–client communication and front-end–back-end communication, two network adapters are installed on the front end server. The two portions of traffic will go through different network adapters and switches.

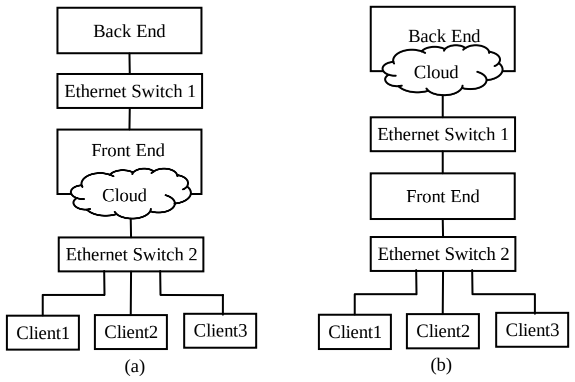


Figure 8. Network configuration in the experiment

In the test runs, F_0 uses Figure 8(a) while the other three configurations use Figure 8(b).

[3] In the C# source code, these five classes have no member variables.

[4] The container object of a fragment can be a fragment too. In Pet Shop, all fragments are contained in a page.

- For F_0, the front-end server runs the original Pet Shop application and the back-end server runs database.
- For $F_{remoting}$, the presentation tier is hosted at the front-end and all the rest, including the business logic tier and the database, are at the back-end.
- For F_{db}, the original Pet Shop runs at the front-end which accesses the database at the back-end server.
- For F_{proxy}, the page assembly and fragment caching tier runs at the front-end and the modified Pet Shop runs at the back-end accessing a local database.

The software environment of our test bed is shown in Table 2.

Table 2. Software configuration in the experiment

Web Server	IIS 5.0[5] + ASP.NET[6]
DBMS	SQL Server 2000
Operating System	Windows 2000 Advance Server SP 3
Network Emulator	Shunra\Cloud

The Shunra\Cloud (version 3.1) [15] is used to emulate network latency in our experiment. It is attached to a network adapter and affects all the IP packets through it. In Figure 8(a), Cloud resides in the front-end server to emulate WAN conditions between clients and Web server. In (b), Cloud is associated with the back-end server to emulate WAN conditions between proxies and content provider's servers. Cloud only imposes minor additional loads on the server it attaches and its effect is negligible.

6.2 Client Emulation

We use Microsoft Application Center Test (ACT) to emulate surges of clients. For each test, ACT distributes test load to the client machines. Each client creates enough threads to simulate a number of concurrent Web browsers visiting the Web application under test. The actual behavior of the threads is controlled by a test script. In each test, a thread repeats the following steps until the test duration is over. It first opens a persistent HTTP connection to the Web server. Then it chooses a request, sends it to the Web server. After receiving the response, it waits for some thinking time (50 milliseconds in our tests) and chooses the next request and so forth. The selection of the request to send is determined by a state transition diagram, which defines the probability of going to the next request from the previous one. If the thread chooses to exit, it stops sending requests and closes the connection.

Table 3. Distribution of the test workload

Activity	Percentage
Category Browsing	18%
Product Detail	16%
Search	18%
Home Page	18%
Shopping Cart	7%
Order	1%
Account/Authentication	22%

[5] In order to obtain reasonable performance, the application protection mode is set to medium.

[6] Come with .NET Framework V. 1.0.3705

The workload generated by the test script roughly follows the distribution shown in Table 3, which corresponds to typical user browsing patterns for such Web site:

6.3 Tuning

Through our experiments, we found it is necessary to fine-tune the configurations so as to eliminate as many side-effects as possible. The major tunings are reported in this sub-section.

6.3.1 Queuing and Threading

ASP.NET assigns processing and I/O jobs (via .NET Common Language Runtime (CLR)) to worker threads and I/O threads in a thread pool with a specified maximum size to process the requests from an incoming queue for the application requested. For some configurations in our experiment, satisfying a request at the front-end server may require accessing external resources (i.e. object, database, page fragment) from the back-end server with a network delay of hundreds of milliseconds. The worker thread has to be blocked and wait for I/O to complete. Therefore a high number of concurrent requests will lead to many blocked threads in the pool as well as pending requests in the queue. To minimize this kind of effect, we make the following adjustments:

1) We set every application's request queue long enough to guarantee that requests be normally served instead of being rejected with an HTTP message indicating server error when server-side congestion occurs. In reality Web servers limit the length of request queues (for dynamic page) to prevent lengthy response time that is not acceptable to users and therefore save resources for incoming requests. In our experiment we need to measure the normal response time for each request, so we raise the default queue length limit from 100 to 1000.
2) Thread pool size affects performance significantly in a subtle way. Too few threads can render the system under-utilized because there may be many blocked threads. While increasing thread pool size might accelerate processing and improve utilization, it will cause more thread context switch overhead. In our experiment, we tune the thread pool size at the front-end server for each configuration under each test condition (various network latencies and output cache hit ratio) to produce the optimal result (throughput and utilization). In general, a configuration with longer processing time caused by network latency needs a larger thread pool size. For example, when the network latency is 200ms and the output cache hit ratio is 71%, the optimal thread pool size of F_0, F_{db}, $F_{remoting}$ and F_{proxy} is 25, 30, 30 and 50 respectively.

6.3.2 Connection Pooling

In the three configurations other than F_0, the front-end server needs to make heavy communications with the back-end server via TCP. On a network with high latency, the connection overhead becomes prominent due to TCP handshake and slow start effect if no connection pooling is used. In order to prevent this kind of performance degradation, we set appropriate pooling parameters for each configuration according to the connection mechanisms used between the front-end and the back-end.

In F_{db} configuration, each time the front-end wants to create a connection to the back-end database, the underlying data access component (ADO.NET) will pick a usable matching connection from a pool of a certain size. If there is no usable connection and the size limit is not reached, a new connection will be created.

Otherwise the request is queued before a timeout error occurs. When the network latency is high and the work load is heavy, the number of concurrent connections in use would constantly exceed the limit, i.e. the connection pool becomes the bottleneck. Therefore, we set the connection pool size large enough (300 rather than the default 100).

In $F_{remoting}$ configuration, front-end objects invoke back-end objects through .NET Remoting TcpChannels, which will open connections as many as needed and cache them for later use before close them after 15-20 seconds of inactivity. Since each test run has a warm-up period, there is no negative impact on performance from this setting.

In F_{proxy} configuration, the front-end needs to retrieve Web objects from the back-end. We set the maximum number of persistent connections high enough (1000) to avoid congestions.

6.4 Measurement

Our experiment consists of two parts. The first part measures the performance of the four configurations under three fixed network latencies (50ms, 200ms and 400ms). The second part measures the performance under several cache hit ratios.

For each test configuration, we vary the number of concurrent connections and run ACT to stress the application and measure the throughput (in terms of requests per second, or RPS), response time (in terms of the time to the last byte of a response) and utilization of the servers (front-end and back-end). Each run starts with a warm-up period (3 minutes), after which ACT begins to collect data about the system status every one second by reading performance counters of processors, memory, network interfaces, ASP.NET, .NET CLR and SQL Server. Data collection traffic only consumes a slight portion of bandwidth compared to client requests and server responses so it interferes little with the network utilization of test load. The collection process will last for 3 minutes and then ACT stops running. Before each test run, we reset the Web server (to flush the cache) and the database tables (to restore the data load) so that the results from different runs are independent.

7. RESULTS AND EVALUATION

In this section, we offer detailed experiment results and analysis. Due to the large number of configuration combinations, we can not present all the data. Thus, we report response time and resource utilization with one representative network latency in Section 7.1, and briefly go over more advanced variations in Section 7.2.

7.1 Basic Results

7.1.1 Response Time

Figure 9 shows the average response time versus concurrent connections number of the four configurations. The cache hit ratio is 71%, and the network latency is 200ms. The loads on the front end server in the three offloading options are higher relative to that of F_0, either due to more processing or higher overhead. This is the reason of the climbing of the latency curves corresponding to the three optimizations. This is an artifact of our test-bed where there is only one proxy and is generally not a concern because proxies are many in real deployment.

When there are less than 300 connections, F_{db}, $F_{remoting}$ and F_{proxy} offer a response time better than F_0. The reduction depends on the network delay. When connection is 15, the reduction is 76%, 75% and 64%, for F_{db}, $F_{remoting}$ and F_{proxy} respectively. In F_0, every time a request is issued from the client, it travels through the delayed link and so does its response. Therefore the response time is always above the network roundtrip time (400ms). In the other three configurations, many requests can be satisfied at the front-end right away. F_{db} and $F_{remoting}$ achieve this by replicating the application logic fully or partially from the back-end. F_{proxy} caches dynamically generated output such as pages and fragments and serves subsequent matching requests from the cache directly or by composing fragments in the cache, thus reducing the frequency of accessing the back-end significantly.

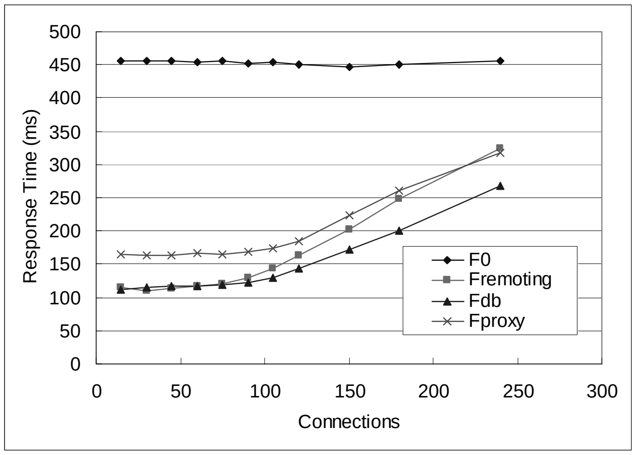


Figure 9. Response time versus number of connections

The response time of F_{db} and $F_{remoting}$ are very close under small number of connections. This is because their partition points afford them the same back-end access rate and the traffic incurred by each access is enough to be transferred within one roundtrip for both configurations.

Compared to F_{proxy}, F_{db} and $F_{remoting}$ do more than just caching: they are capable of offloading some of logical processing as well. For example, the responses of some requests, such as `GET CreateNewAccount.aspx` and `GET SignIn.aspx`, are not cacheable, so they will always go to the back-end in F_{proxy}. But neither F_{db} nor $F_{remoting}$ requires backend access for these requests.

The fact that the cacheable pages occupy a large portion of the test loads is one reason that F_{proxy} does not lag too far behind. However, there is one more subtle and interesting case where F_{proxy} actually wins out.

F_{proxy} is optimized for retrieving Web pages and fragments from the back-end and performing page assembly. When it encounters a request for a page containing fragments, all objects including the container page and the fragments inside that are not cached will be fetched from the back-end asynchronously, allowing them to be downloaded in parallel. After all objects arrive (in any order), they are composed together and a complete page will be returned. While in F_{db} or $F_{remoting}$, due to the fact that the partition points are inside the application logic, *all* accesses to the back-end such as database query and remote object invocation are blocking. For example in $F_{remoting}$, if completing a request needs to call the back-end twice for generating two fragments in the page, the two invocations must be performed sequentially even though the two fragments are independent of each other. The asynchronous retrieving nature of F_{proxy} saves significant time for processing such pages compared to F_{db} and $F_{remoting}$. For instance, the page `Cart.aspx` (uncacheable, used for adding, updating and removing items in a shopping cart) contains one cacheable fragment and two uncacheable ones which show a favorite list and a banner respectively. The favorite list and the banner need to query the back-end database and the container page needs too in case of adding an item to the shopping cart. When the network latency is 200ms and the work load is light, it takes F_{db} and $F_{remoting}$ 1245ms and 1285ms respectively (both over three roundtrips) but takes F_{proxy} 784ms to generate the page.

Therefore, we can classify requests into three distinctive classes (with their ratios):

- Class A (71%): cacheable response
- Class B (14%): uncacheable response and may need to access database
- Class C (15%): also uncacheable response but there is no need to access database

Figure 10 compares the average response time of the requests in each class for $F_{remoting}$, F_{db} and F_{proxy}. All of them return responses for requests of class A in negligible time. F_{proxy} wins in class B due to its asynchronous optimization mentioned above. $F_{remoting}$ responds slower than F_{db} because it introduces overhead when doing remote object invocations. In class C, both F_{db} and $F_{remoting}$ can return responses immediately, while F_{proxy} has to fetch content from the back-end with significant delay.

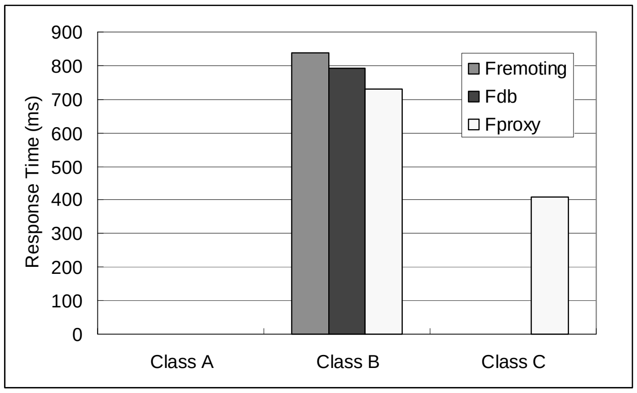


Figure 10. Average response time of each class of pages

7.1.2 *Scalability and Server Load Reduction*

The other functionality of offloading and caching at proxies is reducing server load so as to achieve better scalability. Since our test-bed consists of only one proxy and one database server, we can only infer from the load distribution among resources. However, all offloading configurations filter at least 70% server requests through proxy for this test script due to the output cache hits.

Figure 11 plots the aggregated server loads for all four configurations. Theses are loads that will still remain at the server side regardless in real deployment. The curve of F_0 adds up loads of both the front end (which runs the application) and the backend (the database). Curves of all three offload configurations are the loads on the backend server only and they are:

- F_{db}: the database
- $F_{remoting}$: the remaining of application logic and the database
- F_{proxy}: modified Pet Shop application and the database

As we explained earlier, the front end server in these configurations all have higher loads than in F_0, either because of more processing or higher overhead. Consequently, the three curves of the offload configurations do not extend to as high throughput as in F_0 because our front end proxy becomes bottleneck.

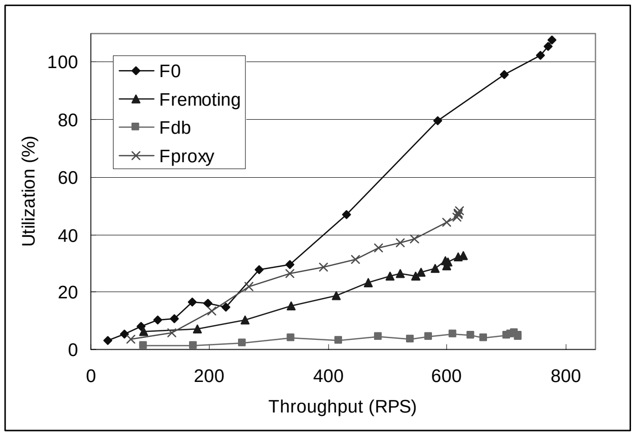


Figure 11. CPU utilization in server of all configurations

Note: The three curves of the offload configurations stop at throughput points beyond which our front end proxy becomes saturated.

As can be seen, in our test environment, the backend database is not the bottleneck. The largest load reduction is achieved with F_{db} where the functions of application server are taken entirely by the proxies distributed near the client. Because there is still some processing remaining at the back-end in $F_{remoting}$, it can not achieve the same level of server load reduction as F_{db}. As expected, F_{proxy} is the third and achieves reasonable load reduction comparing to F_0.

In reality, a server complex is made up of a tier of machines running application servers backed by the database machines. The differences between the load curves of F_0, $F_{remoting}$, F_{proxy} and F_{db} are what will be run on the machines hosting the application servers. It is evident that to achieve identical throughput, F_{db} will require the least resources inside server complex, followed by $F_{remoting}$ and F_{proxy}.

7.2 Advanced Evaluation

7.2.1 *Vary Network Latency*

We repeat the experiment under two other network latencies: 50ms and 400ms with light server load. The results are shown in the following figure.

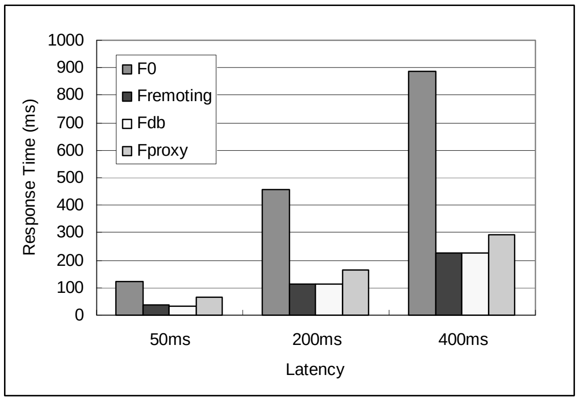


Figure 12. Response time with different network latencies

There is no major surprise: the response time of F_0 is always over one roundtrip time. Both F_{db} and $F_{remoting}$ offer comparable response time and F_{proxy} is slower but still competitive. As expected, the benefit of offloading/caching at proxy increases with network latency.

7.2.2 *Vary Cache Hit Ratio*

We repeat the experiment under two other output cache hit ratios: 52% and 30%. The network latency is fixed at 200ms. The response time of the configurations under different hit ratios is shown in Figure 13.

This result is also straightforward: increased cache hit ratio benefits offloading/caching at proxy. We note again that the caching is a more significant factor, though this is specific to this application and the test scripts we used for the experiments.

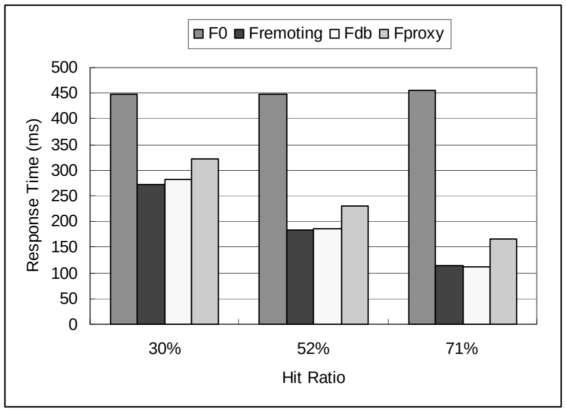


Figure 13. Response time with different cache hit ratios

8. SUMMARY AND CONCLUSION

We believe that Web page as a unit will disappear over time and be replaced by dynamic content that is deeply personalized. Utilizing the proxy servers located near the client to distribute and offload the processing and caching of dynamic content is entirely reasonable. This is all the more so because these edge devices are already deployed and at the same time underutilized as far as dynamic contents are concerned. However, such deployment requires not only the re-engineering of applications themselves, but also the considerations of security issues and careful evaluation of performance benefits.

In this paper, using a representative e-commerce benchmark, we have enumerated extensively many partitioning strategies. Without going into specifics, the following conclusions can be drawn: 1) offloading and caching at edge proxy servers achieves significant advantages without pulling database out near the client. Our results show that, under typical user browsing patterns and network conditions, 2~3 folds of latency reduction can be achieved. Furthermore, over 70% server requests are filtered at the proxies, resulting significant server load reduction. 2) This benefit can be achieved largely by simply caching dynamic page fragments and composing the page at the proxy, whereas advanced offload options are often overly complex and can be counter-productive if not done carefully. 3) Many progresses in the most recent Web programming platform play important roles for this to happen. In the .NET framework, the output caching capability, stored procedures and, to a less extent, the Remoting mechanisms all made significant contributions. 4) The 3-tier Web architecture gives a general guideline but is not as helpful when actual partitioning strategy is devised.

Specifically, we have evaluated the following three partitioning strategies and our findings and recommendations are as follows:

- Replicating all application components except database to the edge provides the best average response time and the highest reduction of back-end server load. It is also the easiest to implement. However, this would be otherwise impossible if not for a highly efficient implementation of database interaction. That is to say, this option will give only disappointing offloading performance if not for the fact that database *stored procedure* has encapsulated multiple SQL queries within one request. The restriction is that since the complete business logic is pushed to the proxy, this option is suitable only for intranet applications or other situations where end-to-end security is already in place.
- Partitioning the application components and moving some of them on the edge offer similar response time as well as server load reduction. The prerequisites are that the application is carefully partitioned and appropriate RPC mechanism is used. However, it would require considerable engineering cost for a complex Web application that was not designed to be run in a distributed fashion to begin with. Furthermore, if it is impossible to determine the sensitivity of the processing offloaded to the proxy, then this option will require end-to-end security as well. Thus, in view of the simplicity of the database offloading strategy, this option, while theoretically interesting, is unlikely to be justified.
- We find that simply augmenting the capability of proxy today to cache dynamic fragment and compose the complete page is also very effective in terms of latency and server load reduction. The security requirement is minimal because what get cached at proxy are contents, *not* logics. This option requires change to the original application, but the process can be reasonably automated or comes for free if standard guidelines such as ESI are followed.

Our investigation thus essentially boils down to one simple recommendation: if end-to-end security is in place for a particular application, then offload all the way up to the database; otherwise augment the proxy with page fragment caching and page composition.

Although our experiment was carried out within the .NET framework, the conclusion should be general enough to be valid on other platforms, especially for e-commerce type of applications. Many of the .NET framework and ASP.NET features are shared by other competing platforms. Furthermore, we found that network latencies dominate the performance anyway, and the .NET framework itself contributes negligible overhead in comparison.

The TTL-bound consistency enforcement as is used in this benchmark prevails in today's Web deployment, largely due to its simplicity. If more advanced, server-driven consistency maintenances are to be taken in the future, we still believe our observation to hold true in general because cache hit ratio will remain the same for all these options. What makes the difference is mainly the amount of processing being offloaded, which is orthogonal to consistency mechanism.

Our future works include a number of directions. We believe while e-commerce application is interesting, they will represent only a small portion in the future given that Web services will grow to cover more Web usage scenarios. Thus, we are actively seeking new applications and repeat our investigations. We are also working on a more robust implementation of caching fragments and page assembly on top of the proxy architecture. Last, but not least, we believe it is important to extend this work to include underprivileged users with slow and narrow connections to the proxies. In such cases, caching at proxy is no longer sufficient: we need such functionality to move to the client side. Our preliminary experiments have indicated that this can bring about significant benefits, even though one no longer enjoys the high caching hit ratio at proxies because of sharing among different users.

9. ACKNOWLEDGMENTS

We thank members of the Media Management Group for their feedback and support. Wei-Ying Ma has been instrumental for this work to get started. We also thank the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] Amiri, K., Park, S., Tewari, R. and Padmanabhan, S. DBProxy: A Self-Managing Edge-of-Network Data Cache. IBM Research Report, RC 22419, April, 2002.

[2] Cao, P., Zhang, J. and Beach, K. Active Cache: Caching Dynamic Contents on the Web. In: Proc. of IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), pp. 373-388.

[3] Challenger, J., Dantzig, P. and Iyengar, A. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In: Proc. of ACM/IEEE Supercomputing'98 (SC98), Orlando, Florida, November, 1998.

[4] Datta, A., Dutta, K., Thomas, H., VanderMeer, D., Suresha and Ramamritham, K. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In: Proc. of ACM SIGMOD Intl. Conf. on Management of Data, Madison, Wisconsin, USA, June, 2002, pp. 97-108.

[5] Edge Side Includes. http://www.esi.org

[6] IBM WebSphere Application Server. http://www-3.ibm.com/software/webservers/appserv/

[7] IBM WebSphere Edge Server. http://www-3.ibm.com/software/webservers/edgeserver/

[8] Iyengar, A. and Challenger, J. Improving Web Server Performance by Caching Dynamic Data. In: Proc. of the USENIX 1997 Symposium on Internet Technologies and Systems (USTIS'97), Monterey, CA, December 1997.

[9] Labrinidis, A. and Roussopoulos, N. WebView Materialization. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Dallas, Texas, USA, May 2000, pp. 367-378.

[10] Li, W.S., Hsuing, W.P., Kalashnikov, D.V., Sion, R., Po, O., Agrawal, D. and Candan, K.S. Issues and Evaluations of Caching Solutions for Web Application Acceleration. In: The 28th Int. Conf. on Very Large Data Bases (VLDB 2002), Hong Kong, China, 20-23 August, 2002.

[11] Microsoft .NET Pet Shop. http://www.gotdotnet.com/team/compare/petshop.aspx.

[12] Microsoft ASP.NET Caching Features. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconaspcachingfeatures.asp

[13] Microsoft ASP.NET Site. http://www.asp.net/

[14] MSDN: An Introduction to Microsoft .NET Remoting. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/introremoting.asp

[15] Shunra\Cloud. http://www.shunra.com/cloud.htm

[16] Sun Java Pet Store. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications/sample_application/functionality/index.html

[17] Yagoub, K., Florescu, D., Valduriez, P. and Issarny, V. Caching Strategies for Data-Intensive Web Sites. In: Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Cairo, Egypt, 10-14 September, 2000.