Toward Automatic State Management for Dynamic Web Services [*]

Geoff C. Berry, Jeffrey S. Chase, Geoff A. Cohen, Landon P. Cox, and Amin Vahdat
Department of Computer Science
Duke University
{gcb,chase,gac,lpc1,vahdat}@cs.duke.edu

Abstract

A key challenge in the development of the Internet is to simplify construction of scalable wide-area services. One approach to scaling wide-area services is to deploy generic computing power and storage in the network, and use it to absorb service load through dynamic resource recruitment, active caching, or dynamic service replication. Each of these approaches introduces distributed state and an accompanying burden on the programmer to manage that state.

This paper develops an approach to automatic state management for replicated services, a key step toward the goal of automatically converting unscalable service implementations into scalable ones. We demonstrate a prototype implementation of automatic state management, called Ivory. Ivory transforms the bytecodes of a Java-based service to trap updates to its data structures and propagate modified objects among to peer replicas. We demonstrate our approach in the context of a *service caching* framework that replicates service code and data on demand, and present measurements of an example Web portal application that shows the overhead and scalability benefits of service replication using Ivory.

1 Introduction

Dynamic Web content is becoming increasingly important as Internet services continue to evolve. A *dynamic* Web service generates custom documents on-the-fly by executing code over internal service state often extracted from a database. The generated response may depend on arguments in the request, session history, or user information accessed through a cookie, and the service may update its internal state as a side effect of the request. The Common Gateway Interface (CGI), Microsoft's Active Server Pages (ASP), Java servlets, JavaServer Pages (JSP), and other technologies for dynamic Web content are central to the continuing evolution of "Web servers" into "Web application servers" supporting personalized content (e.g., *my.*.com*), electronic commerce and auctions, online financial services, and communication services such as Web-based mail. The technology is also gaining popularity as a vehicle for delivering outsourced application services, e.g., billing or personnel management for small businesses, and to associate code with static content, e.g., to track user access patterns.

Scaling these dynamic Web services is a key challenge for the continuing development of the Internet. Increasing bandwidths enable more advanced dynamic applications, but these applications are interactive and must respond quickly. Faster networks alone cannot overcome latencies imposed by server load or the speed of light, or outages caused by server failures or network glitches. The solution is to use caching and replication to push application data and processing out into the network and closer to the end users. Unfortunately, dynamic content defeats current Web proxy caches. Dynamically generated documents are not cacheable because they may change each time they are accessed. This presents a fundamental limit to the effectiveness of Web document caching. For dynamic services to benefit, caching and replication strategies must be extended to replicate some or all of the *service* itself — its code and internal state — rather than merely caching or replicating the *documents* that it generates.

Several frameworks exist for managing service replication in the Web, including research systems [VAD$^+$98, RRRA99] and emerging Web hosting providers (e.g., Akamai and Sandpiper Net-

[*]This work is supported by the National Science Foundation under grants CCR-96-24857 and CDA-95-12356. Geoff Cohen was supported in part by an IBM Cooperative Graduate Fellowship.

works). One difficulty with replicating service state is that the service must maintain consistency among its replicas; updates originating at any site must propagate to all replicas using the modified state. The specific way to address the consistency problem depends on the representation of the data, its internal consistency requirements, and the nature of the updates. For example, some or all of the server state may be stored in a collection of files, a relational database, or data structures generated by programs. Each requires a different level of system support to maintain consistency.

This paper presents techniques for automatically managing replica consistency, taking an important step toward transparent replication of dynamic Web services. We focus specifically on services based on server-side Java technology, as one example technology for producing dynamic Web content. Our approach is based on a toolkit, called Ivory, that leverages Java binary rewriting tools to instrument the compiled bytecodes for the service implementation, adding new instructions to capture and propagate object updates.

While the general problem of replica consistency is extremely difficult, our solution is promising for state represented as Java object structures with no concurrent write sharing of any individual object among the replicas. Our system provides no means to order updates from multiple replicas. This is adequate for the large class of dynamic Web services in which content updates disseminate from the primary server and user updates are limited to state associated with a particular user (e.g., user profiles, shopping carts, mail boxes, account information) or with a group of clients bound to a single replica (e.g., a business using an outsourced application).

We illustrate our approach as the core of a *service caching* framework for dynamic Web services. This framework extends the Web proxy caching infrastructure to allow on-demand partial replication of services in *Web application proxies*. Service caching leverages the transportability of Java code and Java objects, and the partitioning of Java-based services into discrete code units (*servlets*). Service caching is superficially similar to Java applets in that application services are delivered by server-supplied code without the need for clients to install, administer, or maintain the application software. However, service caching differs in several fundamental respects.

- There is little or no burden on the programmer to use service caching; bytecode rewriting transparently adapts the service code to run outside of the server.
- The transformed service code can make use of generic processing power and storage in the network. In particular, it is easy to expand capacity by adding more application proxies.
- Service caching with Ivory provides for regular, incremental, transparent, and consistent propagation of updates in both directions between each proxy and the primary server.
- Web application proxies can exploit sharing of content within a client population, in a manner analogous to static Web proxy caching.

We demonstrate service caching with a dynamic portal application intended to be representative of *my.yahoo.com* and other commercial Web services providing personalized views of news, stock quotes, sports scores, weather, etc. In this paper, service caching and the portal example serve to illustrate the Ivory approach to automatic state management. In particular, this paper does not address important security, access control, and resource management issues for service migration and on-demand replication. For example, it would not be useful for a large search engine to download its entire index to a proxy. We view service caching as a specific instance of a general vision of migratable service code, which is addressed more comprehensively by recent research on WebOS [VAD$^+$98] and Active Names [VDAA99]. Similarly, the Ivory approach to automatic state management is applicable within these more general frameworks.

This paper is organized as follows. Section 2 presents the Ivory approach to state management and its prototype implementation. Section 3 illustrates the use of Ivory in the service caching architecture. Section 4 sets our approach in context with related work. Section 5 presents experimental results showing the overhead of automatic state management and the performance benefits of service caching. Section 6 concludes.

2 Ivory Architecture

This section describes the structure of the Ivory system, the techniques and mechanisms used to manage replicated service state, and the issues raised by our approach. While the implementation described here is specific to Java, the underlying principles extend to any similar language. A Java-based service may instantiate its data from some external storage, e.g., files or a database, but Ivory deals only with the data's representation as Java data structures.

Ivory consists of a *state manager*, a *serializer*, and a *transformer* program built using JOIE, a programmable bytecode rewriting toolkit [CCK98]. The transformer instruments the compiled service code with write barriers that capture object updates and notify the state manager, which propagates the modified object values. The serializer is an extended object serialization package used by the state manager to propagate updates incrementally. The following subsections explore the Ivory architecture in more detail.

2.1 Replicas and Views

To instantiate a new replica, a server serializes some set of objects into a TCP stream. The receiver unpacks the serialized objects to create a complete or partial replica of the service. The policies for selecting replica sites or objects to replicate are left unspecified; the service caching framework outlined in Section 3 illustrates one useful set of policies.

To simplify the exposition, we describe how the system manages state shared among a single pair of replicas. Generalization to multiple replicas is straightforward. eralization to multiple replicas is straightforward.

The set of objects shared by a pair of replicas is called the pair's *view*. Each replica's state manager maintains a table of references to objects shared with its peer, called a *view table*. The serializer uses the view table to propagate object updates incrementally, as explained in Section 2.4. If either replica loses state in a failure, it may be reestablished from the survivor's view table.

Connected replicas must agree on the set of objects contained within the shared view. Views are not static; either peer may create new objects or add objects to the view. In our current prototype each view contains a closed set of objects; any object that becomes reachable from other objects in the view is automatically added to the view and propagated to the peer, as described in Section 2.4.

2.2 View Consistency

Replicated data may include arbitrarily linked data structures with strong internal consistency requirements. Therefore, Ivory must propagate updates in such a way that each replica observes only internally consistent states. Our solution is to borrow the notion of an atomic commit from ACID transaction systems. The state manager records new object values only at well-defined *commit points* occurring at the completion of sequences of updates that transition the modified structures from one consistent state to another. When a service thread reaches a commit point, all objects modified since its last commit point are committed to the state manager. The state managers observe the following constraints, which are both necessary and sufficient to preserve consistent views: (1) the state manager never propagates an object that is dirty but uncommitted, (2) if the state manager sends to a peer any object modified in a given commit, then it sends all objects in the peer's view that were modified in that commit or in a preceding commit, and (3) the receiving replica applies updates from a given commit as an atomic unit, and never interleaves them with processing for a request that might access the modified objects.

Our approach leads to the following consistency guarantee for each replica's view. For each peer P, consider the set of locally replicated objects whose latest update originated at P. The objects in this set have the same state that they had at some commit point recently occurring on P. Thus this state is internally consistent, but it is permitted to be stale, i.e., subsequent commits may have established a more recent state not yet reflected in the replica. In general, a client bound to a single replica of a dynamic Web service cannot detect that its replica's data is stale, since any state presented to the client in a response could change before the client submits the next request. However, stricter session guarantees are needed if clients migrate between replicas. Also, if multiple sites may generate conflicting updates then the system must impose some safe ordering on these updates. Safely replicating this class of service requires an external concurrency control scheme [FCNyML94] or restricted data representations that can tolerate multiple update orderings [PST^{+}97].

Rather than attempt to determine appropriate commit points automatically, we currently require the programmer to direct the placement of commits. Ivory defines a null interface called *Consistent*; methods of interfaces marked by the programmer as extending the *Consistent* interface are assumed to transform the data from one consistent state to another. The bytecode transformer instruments these methods to commit updates before returning.

Our Ivory prototype imposes coarse-grained concurrency control on the service to ensure that updates and requests do not interleave within any replica. Our approach establishes a global reader/writer lock in the state manager. *Consistent* methods, which update replicated state, are instrumented to include a prologue that acquires the global lock in write mode. Request handlers acquire the lock in read mode, and may promote to write mode if the handler encounters a consistent method. The serializer holds the lock in read mode when it is propagating updates, and in write mode when it is applying received updates. The locking rules prevent any thread from observing a possibly inconsistent state while it is processing a request.

2.3 Propagating Updates

The state manager is responsible for propagating the new object values to replicas as needed, as determined by the lists of dirty objects passed to its commit method. Each site must track object membership in the views of connected replicas, so that it may propagate any modified object to all views that contain it. To meet this need, the state manager maintains a *copyset table* mapping each object to a list of views that contain the object. The copyset table is updated when the state manager adds or removes objects from a view.

Ivory accommodates both *push* and *pull* policies for propagating updates to peer replicas. To implement a *push*, the state manager simply serializes all committed objects into the stream(s) bound to each containing view at commit time. The *pull* state manager retains records of dirty objects until the peer asks for them; it maintains with each view a list of objects that are *dirty in the view*, i.e., objects that have not been propagated since they were last modified. Each object that is dirty in the view is marked with a dirty bit in the object's view table record. On commit, modified objects that are not already dirty in the view are added to the view's dirty list. When the replica requests updates, the state manager serializes the view's dirty objects into the stream, clears the view's dirty bits for the objects in the list, and empties the list. The pull (lazy) model may impose a round-trip latency on some requests, but it is less consumptive of network bandwidth, it matches the request/response structure of HTTP, and it supports the browser "refresh" button.

The state managers require state proportional to the sum of the number of objects in all the views. There are three hash table entries for each $(object, view)$ pair, one in the copyset table and two in the view table (OID-to-reference and reference-to-OID). Each entry contains at most an object reference and a dirty bit. State management overhead for a dirty object is proportional to the number of views containing the object.

2.4 The Serializer

The state manager uses a new serialization package to propagate modified object values. The Ivory serializer is similar to Java Serialization in that it packs and unpacks object values into and out of a stream, in this case a network connection between a replica pair. However, the Ivory serializer differs in three key respects:

- It is *incremental*: the serializer transmits only the objects that have been modified or added to the view, rather than reserializing the entire data structure. This minimizes the overhead to propagate updates, and it allows different sites to concurrently update different portions of a connected data structure.
- It is *iterative* rather than recursive, so it is not vulnerable to stack overflows for deeply nested data structures.
- It uses efficient class-specific serialization methods generated and installed in each class by the bytecode transformer, rather than using reflection to discover each object's internal structure at runtime.

The view tables are the basis for the Ivory serialization scheme. Each view table maintains a mapping between object references to integer object IDs (OIDs) that are unique within the view. The state manager initializes the mappings when it creates the view, and updates it by assigning new OIDs as objects are added to the view. View tables allow the peers on either side of a connection to agree on a common space of OIDs. OIDs enable incremental and iterative serialization because they provide a means to represent objects *by reference* in the serialized stream.

One role of the serializer is to automatically add objects to the view when those objects become reachable from other objects in the view. This can occur only when an object A already in the view is modified to include a reference to an object B that is outside of the view, causing B and its descendents to become reachable from the view. The serializer discovers and handles all such cases while serializing the updated A, when it queries the view table for an OID to represent the referenced object B. The sending serializer allocates an OID for B, adds the mapping to the view table, and transmits

```
ALOAD 0                              // this
GETFIELD <Ivory_dirty>
IFNE @END                            // if true goto end
ALOAD_0                              // this
ICONST_1                             // true
PUTFIELD <Ivory_dirty>
ALOAD_0                              // this
INVOKESTATIC <Hamper::setDirty>
END
```

Table 1: Bytecode transformer splice for write barrier on *putfield* operations.

B together with its new OID. The receiver instantiates a new copy of B and installs the new mapping.

2.5 The Role of Bytecode Rewriting

Ivory uses bytecode rewriting to make state management automatic. The replication system is a general-purpose Java package; a service implementor could design the service to use the replication package by making explicit calls to the state manager. Our goal is to automate this process, providing a means to automatically convert unscalable service implementation into scalable ones. Bytecode transformation is a promising approach to injecting new system functionality into Java programs without modifying the application source code or the Java system itself (the language, compiler, and JVM).

The Ivory bytecode rewriter is built using JOIE [CCK98], a bytecode rewriting toolkit. JOIE bytecode rewriters are *transformer* classes written in Java using JOIE primitives for deconstructing and instrumenting compiled Java classes. JOIE transformers can be used to transform stored classfiles after compilation, or they may be applied on-the-fly by a transformer-enabled JOIE ClassLoader as it loads service classes into the JVM.

Ivory includes three JOIE transformers that illustrate simple tasks easily achieved with bytecode transformation. First, a serialization transformer injects code used by the incremental serializer for efficient, transparent serialization, as described in Section 2.4. Second, a consistency transformer identifies methods that implement the interface *Consistent* (Section 2.2), and injects a prologue and epilogue to synchronize the consistent method and to commit dirty objects on completion. Finally, a write barrier transformer installs write barriers to track and record object updates.

The write barrier transformer modifies each class to add a dirty bit to each instantiated object. It injects new instructions (a *splice*) into the bytecode to set an object's dirty bit each time one of its fields is modified with a `putfield` instruction. To capture all needed updates, the transformer must rewrite any class that updates any replicated objects, including classes that set a public field rather than calling a method. The spliced code sequence, shown in Table 1, collects modified objects on a dirty list (a globally shared *hamper*) as the dirty bit transitions from clean to dirty. The epilogue for *Consistent* methods commits the dirty list to the state manager, which records the dirty objects, resets their dirty bits, and empties the hamper.

For automatic state management to be practical, it is critical to minimize state management overheads in the transformed bytecodes. The simplest write barrier transformer instruments every `putfield` instruction, which is wasteful when multiple writes occur within the same code path. The JOIE toolkit provides primitives to partition the bytecode into basic blocks and to perform simple control flow analysis. We have used these primitives to implement two refinements of the simple write barrier transformer. The *BasicBlock* transformer updates the dirty bit at most once per basic block. The *Dominator* transformers performs a dominating path analysis to reduce dirty bit updates for basic blocks accessed through the same code path. The more sophisticated transformers trade off transformation speed for more efficient runtime performance. Section 5 presents performance results from these transformers.

3 Service Caching

This section outlines the structure of the service caching framework used in our experiments. The framework is designed to illustrate use of Ivory in conjunction with a simple scheme for replica creation and request routing. It allows on-demand caching of service code and data in *application proxy servers*, which extend static proxy caching to support local execution of cached Java classes and data.

Our prototype enables transparent service caching for services implemented using the JavaServer Pages (JSP) standard [PLC99], a recent extension of the earlier Java Servlet standard [Dav99]. The JSP standard supports construction of dynamic Web services from static templates (e.g., HTML) containing embedded scripting code invoked at page-fetch time to fill in dynamic content. The JSP script code is written in Java, although future releases may support other scripting languages. JSP scripts access the service data using a registry that allows them to retrieve some subset of the underlying service objects by symbolic name. JSPs are compiled to generate Java servlets, which are Java classes implementing standard methods for handling HTTP requests.

The service caching framework uses JSP servlets and the retrieved objects as the granularity of caching. The proxies cache servlets by the URL name prefix (not including arguments). The proxy can then service any URL operation with a matching prefix by executing the cached JSP servlet locally. As the servlet retrieves objects by name, the Ivory-enabled proxy server contacts the home site to retrieve the objects and add them to its view. The proxy maintains a local cache over the name registry, so that it can serve repeat references by symbolic name from its object cache. Of course, the retrieved objects may reference other objects directly; the serializer in the home server automatically adds these descendent objects to the view as described in Section 2.4. Thus the local object cache may contain service objects that are not named symbolically.

This replication scheme results in a simple hierarchy in which each proxy has a connection for exchanging updates with a parent, e.g., the home site. Updates may flow in either direction. For example, in the portal application described in Section 5, the server notifies the proxies of updates on the content provider (e.g., news and stock quotes). Our prototype application proxy pulls updates from the server on each client request. A weaker but more efficient implementation could limit the pull rate according to some policy, e.g., pull at most once a minute unless the client hits the "refresh" button. In the other direction, the proxy may notify its parent of updates to user-specific information (e.g., a user profile). For security reasons a server may refuse to accept updates to service state from a proxy, but our prototype has no access control.

Service caching in application proxies can improve the scalability, availability, and response time of a dynamic Web service. The benefits accrue from several factors:

- It offloads the processing cost of generating dynamic content. This allows the home server to support a larger community of clients before it saturates.

- It exposes to the application proxies the "assembly" of dynamic documents from static and dynamic components, allowing the proxy to cache the static content. Requests that hit in the cache retrieve at most the objects needed to generate the dynamic components. In many (but not all) services this will reduce network bandwidth consumption to satisfy the request.

- For requests that hit in the cache, a pull to refresh the cache state will transfer only the objects that were modified. This can substantially reduce overhead and network bandwidth demand for services with sufficiently low update rates. On lower bandwidth links the smaller transfer size can substantially reduce response time.

- Many services can tolerate delays in the dissemination of content updates from the server. This can reduce the propagation frequency and the overhead of round-trips to query for updates, reducing latency and bandwidth demands. It also improves service availability, since a client may continue interacting with a replica if the home server fails or is unreachable.

- Like static proxy caches, application proxies can deliver caching benefits from shared data brought into the cache by multiple users accessing the same Web sites.

4 Related Work

Replication for improving server performance is not a new technique, having analogs in file servers, databases, and web servers. Many of these systems propagate updates to replicated state incrementally. For example, Delis and Roussopoulos explore a log-based approach for updating client caches in a relational database system [DR92]. Updates are centralized at the server; before a client accesses a data item in its cache, it first contacts

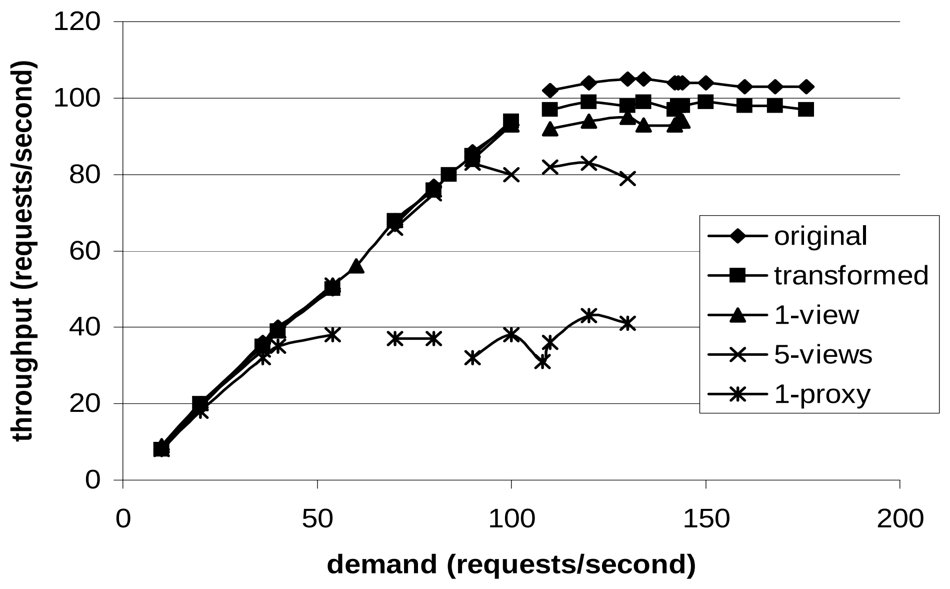

Figure 1: Request throughput for the portal application on a single server.

the server to retrieve log records generated since the cached copy of the item was last updated. Relative to this large body of work, our contribution lies in: (1) system support for incremental updates to replicated Java data structures, (2) our use of bytecode rewriting to make state management automatic for Java-based services, and (3) our use of automatic state management to extend the Active Cache idea [CZB98] to handle Java-based dynamic content.

Our pull-based incremental update propagation is also similar to delta encoding of updates to web pages [MDFK97]. HPP [DHR97] preprocesses web pages to identify static versus dynamic portions. HPP could be used for our simulated web portal application to cache portions of the web page. Ivory extends these ideas to handle the Java state used to generate dynamic content. In this sense, these systems are complementary to Ivory and can be used together for efficient delivery of web content.

Software write detection has been used previously for distributed shared memory, fault isolation, garbage collection, and dynamic data race detection [ZSB94, HM93, SG97]. It has been used for Java in the PSE persistent storage engine from Object Design Inc. [Obj98], which includes a utility that transforms Java classes to be storable in their persistence storage infrastructure, but does not detect if the instances have become dirty. PJama [ADJ+96], earlier called PJava, uses a modified JVM to supply orthogonal persistence to user classes. Our approach using transformation allows the key elements of this functionality on a standard JVM.

The Ivory prototype is applicable to pure Java-based services. However, techniques similar to our own, for example, leveraging related work in the database community on materialized views [GM95] are more generally applicable to other kinds of services.

5 Experience

This section presents experimental performance results to illustrate the benefits and quantify the costs of automatic state management using Ivory. We experimented with a simple portal application to demonstrate the potential of the service caching framework. The portal site is implemented as a JSP servlet that generates "personalized" HTML viewing a selection of content objects that are randomly generated and regularly updated. The content mimics common information such as news categories and headlines, stock quotes, weather, and sports scores. The content is stored in a variety of Java data structures and referenced by randomly generated user customization profiles.

Personalized Web portals are illustrative of a growing class of Web services that generate documents dynamically based on user preferences and changing underlying content. We show how to improve the scalability of these services by using

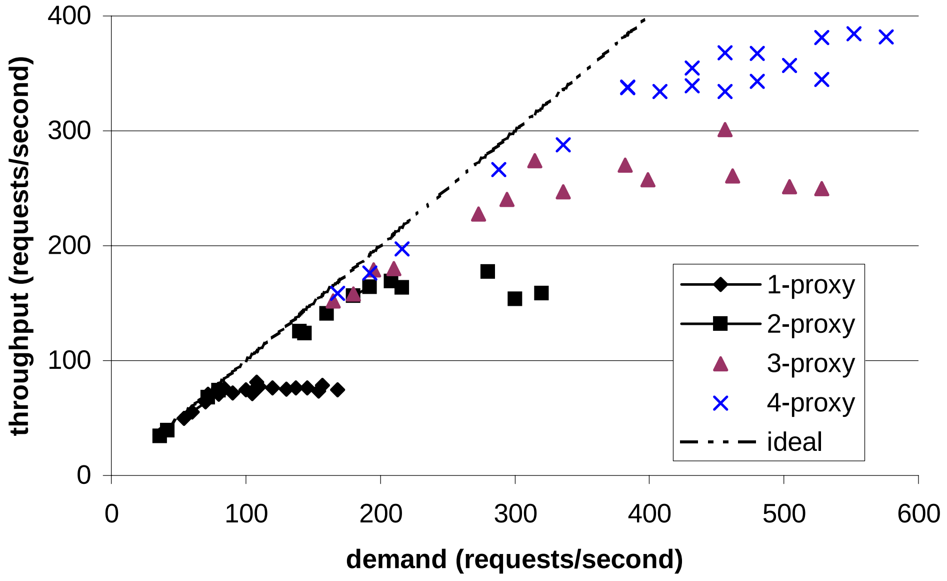

Figure 2: Request throughput for the portal application using Ivory-enabled Web application proxies.

Ivory to cache subsets of the underlying content objects at application proxies; this allows the proxies to generate the documents locally, contacting the server only to receive updates. In this way, proxies can service requests for dynamically generated data, acting as logical extensions of the service.

The simulated web portal consists of a server (the portal site) and a varying number of proxies and clients. The server and proxies are based on TinyServer, a simple Java-based servlet engine developed for a distributed systems class. A collection of servlets and associated classes implements the application proxy cache engine, the server interface to the state manager, the naming registries for the JSP service caching framework, and the portal application. URL requests to the portal servlet specify a user by name, demand loading the user profile and any referenced content objects if they are not already resident in the cache.

Figures 1 and 2 show the overhead costs and scaling benefits of Web application proxies using Ivory for the portal application with representative parameter settings. In these experiments a community of client processes generates a stream of page view requests, with each process using a different user profile. These are closed loop experiments in which each client process issues a request, awaits the response, then sleeps for five seconds before issuing the next request. The figures give delivered throughput as a function of *demand*, the request arrival rate for an ideal server that responds to each request with zero latency. The number of user profiles scales with demand; in these experiments 500 profiles generate a demand of 100 requests per second. Each profile references 60 items randomly selected from a universe of 2500 content objects. While the total data size is less than a megabyte, we stress the system by aggressively updating the data: 15% of the objects are updated each second. All proxies and servers are 167 MHz Sun Ultra 140 workstations running Solaris 2.6 and JDK 1.1.5, interconnected with the clients by a switched 100 Mb/s network.

5.1 Service Overhead

Figure 1 quantifies the overhead of our prototype by showing the saturation points of a single server in various configurations. The top two lines show that transforming the service code to track object updates degrades its saturation throughput by about 6%; Section 5.3 explores this cost in more detail. In addition to the fixed cost of tracking object updates, servers incur an additional cost to track the objects that are dirty in the view of each peer replica. This cost scales with the number of peers and with the number of updates recorded for the objects in each peer view. The next two lines in Figure 1 show that this cost is significant in the prototype: maintaining copy sets and dirty sets for each complete replica degrades request throughput by about 3% to 4% in this experiment, primarily due to the aggressive update rate.

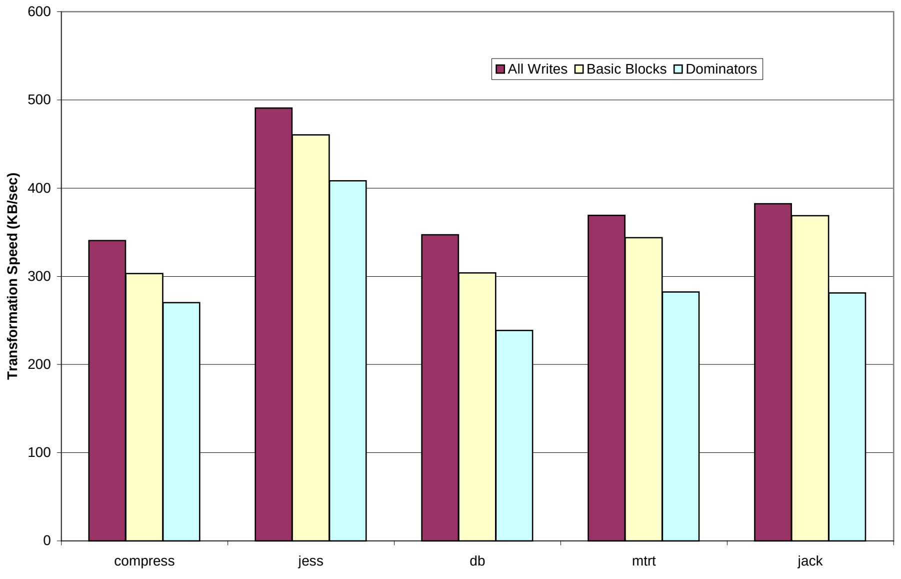

Figure 3: Transformation speed for Ivory bytecode rewriting.

The lowest line in Figure 1 shows the steady-state request throughput through a single proxy using the pull-based state manager with a two-second update window. This gives a pessimistic estimate of the effect of update propagation on request throughput. Once the proxy's cache has been loaded in the first few seconds of the experiment, the proxy satisfies each request from the cache unless the update window expires. On the first request after the update window expires, the proxy queries the primary server for updates to its view, which in this experiment returns new values for an average of 30% of the objects in the cache. The proxy cannot execute any more requests until it has applied these updates, leading to a significant drop in request throughput. This is a pessimistic test for two reasons. First, a practical configuration would use a larger update window. Second, the proxy in this experiment is serving requests for a single service, and these requests cannot be overlapped with update propagation for the service. In practice, each proxy would serve requests for multiple services, and would overlap update propagation with request processing for other services.

5.2 Scaling Benefits

Figure 2 illustrates the scalability benefits of Web application proxies using Ivory. Like Figure 1, Figure 2 gives request throughput as a function of request demand. In these experiments, each client process is bound to a Web application proxy, with the clients evenly distributed among varying numbers of proxies backed by a single primary server. The proxies use a five-second update window.

Figure 2 shows that serving the portal application from Web application proxies allows it to scale to larger numbers of clients. Aggregate request throughputs at saturation scale almost linearly as proxies are added. In principle, proxies can be added and will deliver linear scalability up to the point at which the primary server saturates in delivering updates to the proxies.

This experiment is conservative in that it does not reflect the cost to fetch server data over a wide-area link, which often carries higher latency and lower bandwidth than the link to the proxy. Of course, the performance delivered in practice will also depend on application parameters including the size of the generated content, the size of the objects used to generate the content, the update rate for those objects, the processing cost to generate the content, client bandwidths to the proxy and the server, and the degree of sharing among multiple clients bound to the same proxy. We leave a more complete exploration of the parameter space to future work.

5.3 Write Barrier Overhead

We ran a second set of experiments to better approximate the cost of bytecode transformation for tracking updates. We transformed five programs from the SpecJVM98 suite (*compress*, *jess*, *db*,

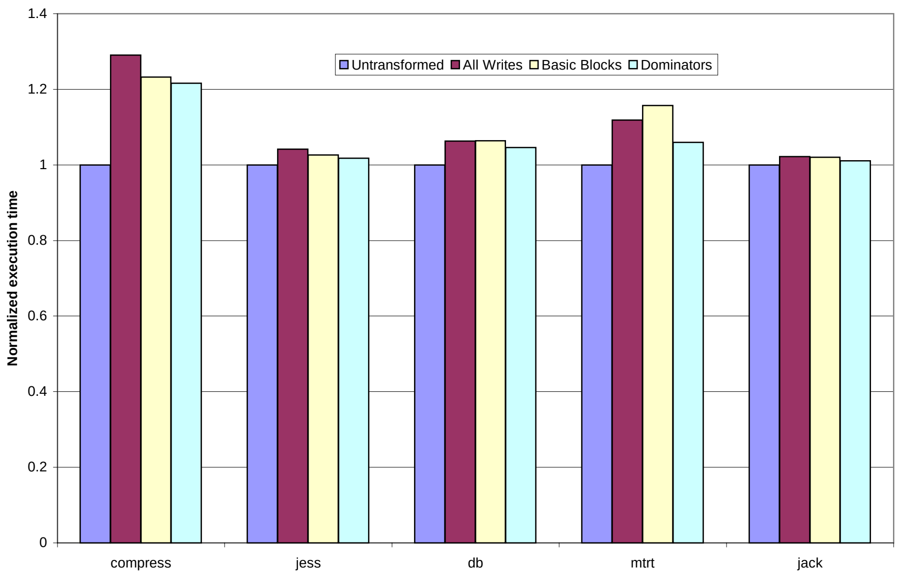

Figure 4: Runtime overhead for code instrumented with write barriers.

mtrt, and *jack*) using the three versions of the write barrier transformer outlined in Section 2.5. We measured both the speed of the transformation and the slowdown of the transformed bytecode. Figure 3 and Figure 4 show results from a 300 MHz UltraSPARC-IIi processor running Solaris 5.7 and JDK 1.2. The transformers process and rewrite bytecode at between 280 KB/s and 500 KB/s, with the more sophisticated transformers running slightly slower but producing more efficient transformed code. The slowdown of the transformed code is under 10% (using the *Dominator* transformer) for three of the five benchmarks, with only *compress* showing a significant slowdown of 22%. The benefit of control flow analysis is typically modest but is significant for some applications. For *mtrt*, *Dominator* reduces the slowdown from 12% (with *AllWrites*) to 6%.

6 Conclusion

Caching and replication are key techniques for scaling Web services. Unfortunately, state replication introduces a difficult state management problem, since service state must be kept consistent across replicas. This is a challenging and problem for services with dynamically generated content, which is increasingly prominent.

This paper describes the design and implementation of Ivory, a system that automates state management for dynamic services based on server-side Java technology. Ivory uses bytecode rewriting to instrument compiled service code with hooks into a state management package, taking automatically converting centralized service implementations into scalable, replication-aware, wide-area applications. We illustrate use of Ivory in a service caching framework for dynamic services based on JavaServer Pages (JSPs). The JSP standard is well-suited to service caching because it imposes a partitioning on the service code and data, and its naming infrastructure is easily extended to demand-fault service objects referenced by symbolic name. Our approach takes a significant step toward generalizing Web caching and replication infrastructures to handle dynamic content. This can significantly improve scalability, response times, and consumed wide-area bandwidth for dynamic Web services.

References

[ADJ+96] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, December 1996.

[CCK98] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic Program Transformation with JOIE. In *USENIX 1998 Annual Technical Conference*, pages 167–178, June 1998.

[CZB98] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware*, 1998.

[Dav99] James Duncan Davidson. Java Servlet API: Version 2.2. Technical report, Sun Microsystems, June 1999.

[DHR97] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[DR92] A. Delis and N. Roussopoulos. Performance and Scalability of Client-Server Database Architectures. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 610–623, August 1992.

[FCNyML94] Michael J. Feeley, Jeffrey S. Chase, Vivek R. Narazayya, and Henr y M. Levy. Integrating coherency and recoverability in distributed systems. In *Proceedings of the First Symposium on Operating System Design and I mplementation*, pages 215–227, November 1994.

[GM95] Ahish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *Data Engineering Bulletin*, June 1995.

[HM93] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-or iented language. In *SOSP93*, pages 106–119, December 1993.

[MDFK97] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of ACM SIGCOMM*, pages 181–194, August 1997.

[Obj98] Object Design Inc. ObjectStore PSE Resource Center, 1998. `http://www.odi.com/content/products/PSEHome.html`.

[PLC99] Eduardo Pelegri-Llopart and Larry Cable. JavaServer Pages Specification: Version 1.1. Technical report, Sun Microsystems, August 1999.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, pages 288–299, October 1997.

[RRRA99] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service. In *IEEE Int. Conf. on Distributed Computing Systems*, May 1999.

[SG97] Daniel J. Scales and Kourosh Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 157–169, October 1997.

[VAD+98] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services for Wide-Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems*, Chicago, Illinois, July 1998.

[VDAA99] Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, October 1999.

[ZSB94] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software Write Detection for Distributed Shared Memory. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 87–100, November 1994.