CS–2001–02

Scaling Java-based Dynamic Web Services

Sara E. Sprenkle Jeffrey S. Chase

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

May 2001

Scaling Java-based Dynamic Web Services

Sara E. Sprenkle and Jeffrey S. Chase
Dept. of Computer Science
Duke University
Durham, N. C. 27708–0129
{sprenkle, chase}@cs.duke.edu

May 2001

Abstract

Managing distributed state is a difficult challenge for building scalable, distributed, wide-area applications. This project presents the design of an infrastructure, called Ivory, to simplify construction of distributed applications by automatically caching and replicating data structures and code. We illustrate the use of our infrastructure in service caches that replicate Web service code and data used to generate dynamic content. The service cache relies on Ivory to maintain consistency of cached data as a basis for scalable dynamic Web services.

Ivory is designed to automate key aspects of state management in a flexible, efficient, and scalable way. A key element to our approach is the use of bytecode transformers that automatically adapt Java applications to the Ivory infrastructure. Bytecode transformers insert new code into compiled applications to notify Ivory of data structure modifications and to invoke operations for maintaining consistency; the transformation is powerful but requires only minimal application programmer involvement. Furthermore, to reduce the space and communication overhead necessary for maintaining data, we use *conits*—groups of application-defined related objects—as the granularity for caching, consistency, and synchronization. Using bytecode transformers and the conit granularity in the infrastructure design allows authors to choose application-appropriate data management and consistency policies.

1 Introduction

The web has evolved from being an archive of distributed, static information to an interactive, wide-area application service provider. This evolution requires new techniques for improving web performance, as measured by client-perceived latency. One approach to maintaining low latency is to scale these applications. Ivory is a project that attempts to answer the problems and questions posed by scaling Java-based dynamic web services, a large subset of these wide-area applications.

Ivory is an infrastructure that replicates and caches data structures and code for distributed services and applications over the wide area. Ivory provides generic support for managing distributed state for applications to implement data management policies. Since Ivory handles data consistency, application authors can set aside issues of data management and, therefore, can concentrate on building wide-area services.

The primary goal of Ivory is to automatically make dynamic services scalable, as measured by throughput and client-perceived latency. To achieve this goal, we need to minimize the storage and communication costs of maintaining state for consistent data and code replication. Another goal is for the infrastructure to be general for use by many different applications; we need to divide functionality between the system and the application so that the system does not restrict policy decisions that the application should make.

A key element to our approach is the use of bytecode transformers that automatically adapt Java applications to the Ivory infrastructure. Bytecode transformation [8] is a powerful tool that injects code into compiled Java applications. With little application programmer involvement, application code is transformed to notify the system of changes to the data structures and to call methods to receive and propagate object updates.

For high performance, we need to minimize the overhead of maintaining consistent data structures. Other systems—such as Thor and object-oriented data bases—group data in clusters or "crystals". Ivory maintains data based on application-defined groups of related objects, called *conits*. Based on the assumption that related objects are more likely to be read and modified within the same period of time, using conits improves the performance of Ivory in three important ways:

- **Consistency** – maintain versions based on conits instead of objects and amortize cost of propagating updates,
- **Synchronization** – lock objects with one shared lock, and
- **Caching** – amortize cost of faulting objects.

Our primary goal is to create a general infrastructure for caching and replicating data that is automatic, yet is also appropriate for the wide range of applications that could use it. We want an automated infrastructure so that replication is fast and efficient. However, applications can make better—and therefore, more efficient—decisions about consistency, synchronization, and caching of their data. Bytecode transformers and conits are used to strike a balance between the desire for automation and application-appropriate behavior.

In the next section, we further discuss the motivation for Ivory. In Section 3, we outline the design decisions we made to achieve our goals. An explanation of the architecture is in Section 4. The implementation of the data management infrastructure and its use in service cache prototype are in Sections 5 and 6, respectively. In Section 7, we evaluate our prototype and discuss future work. In Section 8, we discuss some work related to this project. Finally, in Section 9, we conclude.

2 Motivating Applications

Dynamic services have changed the basic function of the web. Previously, people used the Internet to access a large, distributed library of static documents. To handle the demand for the documents, web proxy caches stored frequently-accessed documents and served requests for those documents. Today, the web is used to access dynamic content, such as personalized web pages and on-line customer accounts.

The demand for dynamic services is growing faster than web servers can handle it. To generate dynamic content, servers execute code that can operate on user-request parameters, server state, and databases. To illustrate the need for scaling dynamic services, consider that the CPU cycles that a dynamic service uses to generate one page could instead serve many static documents. To ensure low client-perceived latency, we can offload some of the burden from the web servers and push content closer to the end user. Using intermediate servers also improves service throughput.

The traditional techniques for improving network performance cannot be applied to dynamic content. For example, web proxy caches cannot handle dynamic content. Unlike static documents, dynamic content cannot be cached because it depends on underlying code and data that might have changed since the last response was generated. Web proxy caches use expiration times to determine when content should be thrown out of caches; however, the granularity of expiration times is pages or documents, which is too large a granularity for dynamic content.

We propose service caches to improve performance of dynamic services. To make dynamic services scalable, the code and data used to generate dynamic content are replicated on other nodes, and we monitor changes to the data. By replicating the content at remote sites, the service has better *fault tolerance* (if one site fails, requests can be redirected to another site), *load balancing* (if one site is overloaded, requests can be redirected), *incremental scalability* (adding new replica sites or adding resources to one replica has low cost), and *client latency* (a replica closer to the client can handle the content request). Caching service programs introduces security, trust, and resource allocation issues that will not be addressed by this project but are being researched by related efforts [3, 4, 9, 18, 19].

Utilizing the network's computational and storage resources improves service performance at the cost of maintaining distributed state within the replicas. When data changes on the server or replicas, we record the change and notify the replicas or primary server, respectively. The service cache does not specify the amount data and code that should be replicated or the degree to which data should be kept consistent; without those restrictions, we can create a range of caches with varying degrees of consistency, depending on the requirements of the application.

Our service cache replicates Java code and data used to generate dynamic content. Using Ivory, the code and data is replicated such that the code generates the same response as the server, within some application-appropriate margin of error. To achieve this goal, we need to minimize the storage and communication costs of maintaining state for consistent data and code replication. Other ideal properties for a service cache include deployment with a common framework for replicating services on the Web and dynamic setup and teardown. Using Ivory can help achieve these goals, as we discuss in more detail in Section 6.

3 Design Principles and Goals

Our main focus for Ivory is caching and replicating data structures in a scalable, consistent way. By automating state management, we make progress towards automatically converting unscalable applications into scable ones. To achieve automatic scalability, we need to reduce the time and space overhead of maintaining consistency; these overhead considerations influenced all design decisions.

The infrastructure must provide general consistency support so that it can be used by a wide-range of applications. Rather than implementing a consistency policy to which all applications would have to conform, the application should make consistency decisions.

Furthermore, the choice for replication granularity can affect efficiency. A fixed granularity may be suited for only a few applications; however, application authors can choose a granularity that is appropriate for the application and thus more efficient. Applications should guide the choice of a suitable replication granularity.

Finally, we want to limit the cost of state management if there is no replication. A server should not be slowed down when no replicas are communicating with the server.

4 Architecture

In this section, we discuss the decisions that guided the design of Ivory's architecture. Ivory is designed to replicate and cache Java data structures, as illustrated in Figure 1. Ivory maintains the data's consistency so that clients can contact either the primary or the replicas to access the data.

4.1 The Case for Conits

To replicate data efficiently and automatically, we must have some knowledge of its structure. Java data structures are objects linked together by references. Related objects have temporal locality in that they are read and written in the same block of time. We call a group of related objects *conits*, i.e., consistency units. Using conits adds flexibility into the system and improves system performance in several important respects.

We want to choose the granularity for efficient data replication and synchronization. Replicating all data objects when only a subset of the objects is needed wastes computational and network resources. Our system can create partial replicas or caches of distributed data based on conits.

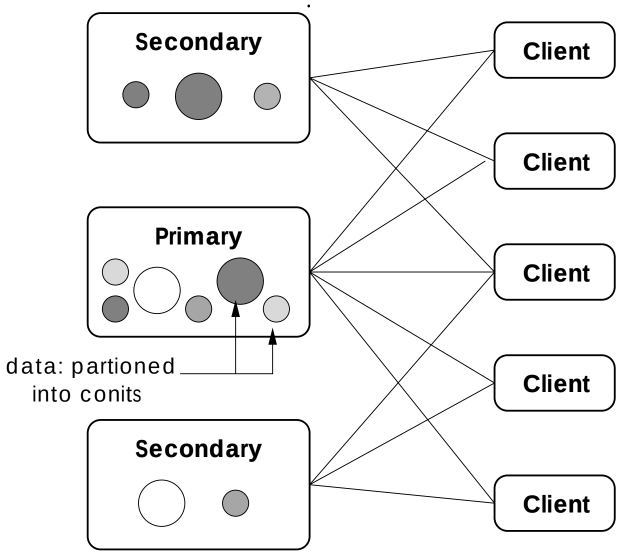

Figure 1: Architecture of Ivory: subsets of primary's data store are replicated on secondaries.

Caching by conits reduces state management and is more space efficient. A subset of all the conits reside on a replica; replicas can evict and fault in conits as needed, depending on storage constraints and the replica's replacement policy. State savings are proportional to the number of objects contained in one conit; rather than maintaining information about each object, we maintain information about groups of related objects.

The tradeoffs in conit size are similar to the tradeoffs in page sizes: a conit that contains many objects can result in false sharing and too many faults, while smaller conits require more state management overhead.

Ideally, granularity is determined by the application because the application knows how to group related objects to minimize false sharing and the number of faults. Objects can be grouped such that writes affect the majority of the conit's members— thus reducing false sharing—and subsequent read requests are for objects contained in the recently-faulted conit, i.e., we prefetched objects for future requests, which may increase performance. However, that solution puts a heavy burden on application programmers to specify how data objects should be grouped. Using Ivory to determine the granularity would eliminate the burden on the programmer but would not necessarily yield the most efficient groupings.

Our solution is a compromise between the two extremes. Applications specify conit membership for a subset of objects; other objects are lazily added to conits by the system.

Consistency and Synchronization

When an object is updated, the changes must be propagated to all copies of the object to maintain consistency. The transmission should not occur when actions are in progress on objects, i.e., update propagation is atomic with respect to actions so that objects are self-consistent when changes are applied or transmitted. An action may affect more than one object; therefore, amortizing the cost of transmitting all the changes for the conit rather than transmitting each individual object's changes may improve performance.

Replicas must also handle conflicting updates; updates to an object are *conflicting* if different sites apply changes to the object before seeing another site's changes to the same object. Solutions for handling conflicting updates range from pessimistic (preventing conflicts before they happen) to optimistic (reconciling them after they happen.)

For stronger consistency, we could use one lock, distributed over all replicas, for each object. Consistency is guaranteed because the lock owner knows he has the most recent copy of an object. The overhead for state management, locking, and conflict detection with this approach is high. Using conits decreases the overhead but can also reduce concurrency—and, therefore, availability and performance—unnecessarily.

At the other end of the spectrum, an optimistic approach is to abort one of the updates when a conflict is detected. In this case, larger conits may produce false conflicts, i.e., there is no conflict because the changes apply to two different objects that belong to the same object cluster, and the update cancellation is unnecessary. If the conflicts are reconciled instead, the overhead of reconciling the conflicts in an object cluster is higher than reconciling conflicts to a single object.

The ideal approach to consistency differs depending on the application. Projects like Bayou [10, 11, 17] and Coda [16] investigated optimistic consistency approaches and their validity in database applications and file systems, respectively. The TACT [21] (Tunable Availability/Consistency Tradeoffs) project explores the tradeoffs between consistency and availability and performance in more depth.

Ivory is designed to be flexible and allow for pluggable policies for consistency. Applications should decide when to get updates from the primary. An application may not require tight consistency bounds; therefore, we can improve application performance by handling requests from the replica without contacting the server for updates.

An application should also decide if replicas will act as a write-back or a write-through cache and how to resolve conflicts. For better performance, groups of writes should be buffered and then pushed as a group to the primary. However, waiting to write updates increases the probability that the update conflicts with the primary's data.

The desire for a non-restrictive infrastructure motivates us to separate the needs of data replication and management from those of systems that use the replicated data.

4.2 Division of Function

We want to create a general infrastructure that can be used by a variety of applications. To achieve that goal, we need to separate the concerns of applications. We discussed some of the concerns of the application and the system in the last section. In this section, we outline applications' expectations for the infrastructure and the responsibilities of the application.

While Ivory determines how data is replicated and kept consistent, the application guides how much data is replicated and to what degree the data is kept consistent. Ivory maintains consistent data among replicas and provides an interface for choosing how to manage the distributed data. Applications call into this interface to implement consistency policies—such as if replicas should behave as write-back or write-through caches. Ivory relies on the application to provide guidance on choosing the most appropriate granularity for the application. The details of Ivory's assumptions about applications and how applications hook into the interface and guide granularity decisions are in the next section.

5 Design and Implementation

In this section, we discuss the assumptions about the applications that will use the infrastructure and present the implementation of Ivory. Figure 2 illustrates a high-level view of Ivory's implementation, which is written in Java 1.3 and run in the Java environment [13].

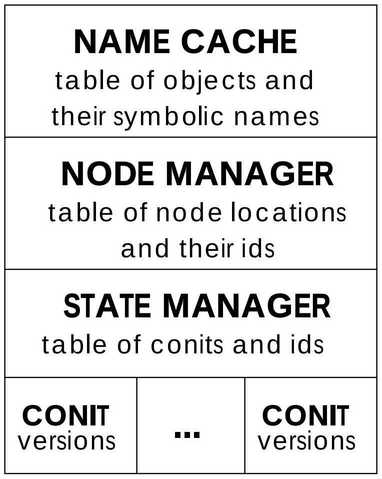

Figure 2: The Ivory Infrastructure

5.1 Assumptions about Applications

To build our first prototype, we make some simplifying assumptions about the applications that will use Ivory. These constraints may be loosened in future prototypes, but they did not restrict the usefulness of the system.

Applications using Ivory are written using Java technology. We also assume that applications are properly synchronized. Our system will not decrease the performance of the application by doing unnecessary additional synchronization, since it should be handled by the application.

Restricting applications to Java is a reasonable decision for implementing our first prototype. Since Java code and data can be migrated, we do not have to consider the heterogeneous platforms of the wide-area environment for which Ivory is designed. Furthermore, a large number of Web applications are built and run on Java technology—such as servlets [1] and Java Server Pages (JSPs) [2]; therefore, we are not severely limiting Ivory's potential users.

Using Java introduced many opportunities for building Ivory's automated state management. Java objects are convenient units for caching and replication. Since Java is a strongly-typed, object-oriented language with entry points to the object and the invocation boundaries clearly defined, the system can automatically monitor changes to data objects more easily with the use of bytecode transformers.

Bytecode transformation

Bytecode transformation [8] is a powerful technique for automatically adding functionality to Java programs and other software packaged to run in the Java environment [13]. In this technique, transformer programs modify existing software by automatically parsing and manipulating compiled classes represented as classfiles. Program transformation is valuable primarily for implementing general features and adaptations that may be specified independently of the application functionality. The new features are decoupled from the application programs and implemented in the transformer once rather than reimplemented in each program. Because they operate on the compiled program, bytecode transformers allow late adaptations or extensions of packaged software, even if source code is not available.

For Ivory, bytecode transformers are used to monitor data structures and to add hooks from applications into the Ivory interface. Specifically, the transformers inject code to

- make objects serializable by adding the Serializable interface to all objects,
- mark an object as dirty if a method makes changes to the object,
- call Ivory's consistency mechanisms for implementing application's consistency policies,
- check that conits are resident, when referenced in public accessor methods; if a conit is not resident, the conit is faulted,
- determine if a cross-conit reference is to a conit that originates on this node; if the conit originates from another node, fetch the updates to this conit,
- and nullify the references from an object when the object is thrown out of the cache.

With little knowledge of Ivory's infrastructure, a programmer can transform his application to use Ivory. Programmers must be aware of the concept of conits and grouping related objects to use the bytecode transformer. We plan to provide a toolkit to support application integration with Ivory, but that is beyond the scope of this project.

5.2 State Management

One of the goals of this project is to minimize the amount of state overhead required to manage data. In this section, we discuss how we manage state and what optimizations we used to reduce overhead.

We begin by defining some of the managed information.

- Node Identification: Nodes must have globally unique identifiers; we assume that we have unique node ids.

- Conit Identification: Conits have globally unique identifiers, which are assigned by the node that creates the conit and are qualified by the node's id.

- Object Identification: Objects must have globally unique identifiers. A fully-qualified object id is its conit id and its conit-unique identifier.

- Class Identification: Class ids are unique per service because only the primary server assigns ids. This constraint may change when we add peer communication.

Ivory is made up of three main components—a name cache, a node manager, and a state manager. The name cache is a table of symbolic names that refer to objects. The node manager maintains information about peer nodes; this information is used to send and receive updates. The most important part of Ivory is the state manager, which manages the conit state and initiates receiving and propagating updates.

For every conit, the server or replica keeps two hashtables—one mapping objects to ids, the other mapping ids to objects. If an object has not been sent or has been modified since the last time it was sent, the server sends the object's conit id, its id, and its class information. We discuss update propagation in more detail in Section 5.4.

We optimize sending class information by assigning service-unique ids to classes. When an object is sent, we send the id of the object's class. If a class has not been sent to the secondary already, we also send the name of the class. The receiving node looks up the class by its id, which is stored in a hash table. If the id is not already in the table, the receiver uses the class name to create a new class object. Thus, the class object is created only once per service per node. This optimization requires the server to store information about which classes it has sent to the node for this service already.

5.3 Conit Management

Using conits as the granularity for caching, replication, and synchronization reduces the state overhead and resources required by Ivory. In this section, we discuss how we manage conits.

Conit membership is partially application-defined, partially automatic. A conit, as shown in Figure 3, consists of Java objects and specially-defined "conit roots". The application defines conit membership for the conit roots. A conit root can either start a new conit or attach to an existing conit. The roots are ingress points to the conit. A reference into a conit must come through one of these conit roots, but any object can make references to any other object whether or not the object is in the same conit. Examples of references are a symbolic name for an object in a conit and a reference from an object in another conit. We make the restriction that conit roots must belong to a conit before propagating updates.

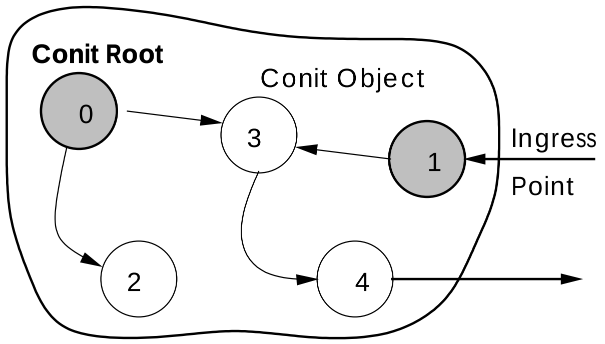

Figure 3: An example of a conit: a conit consists of conit objects and conit roots, which are ingress points into the conit.

Other objects are dynamically added to conits only when a conit is propagated to another node. When a conit is transmitted to another node, all objects reachable from the root—except objects that are known to belong to another conit—are added to the conit. Lazy addition of objects to conits reduces state and time overhead when a conit is not replicated. Until an object is added to a conit, we can ignore modifications to the object; we do not waste resources maintaining information about short-lived objects that are never replicated.

Since the application specifies the conit membership of only a subset of the objects, the burden on the application programmer is reduced, and the replication granularity should be appropriate for the application.

We make some assumptions about the structure of conits and the objects that they contain.

Conit roots have no public fields, only public accessor methods. We have this constraint because we cannot easily track accesses to fields using bytecode transformers.

Accesses to a conit start by invoking an ingress object, i.e., the only way to get into a conit from another conit is through a conit root. This restriction simplifies the system because it does not have to check for faults on every object access—only accesses to conit roots. Furthermore, we can throw conits out of a replica and keep only information about conit roots; otherwise, it would be much more difficult to determine if a cross-conit reference is null or to a conit that is not resident.

Every object in a conit is on a reference path, i.e., in the transitive closure, from one of the conit roots. This is enforced by our policy of lazy addition to conits. We can add objects not assigned to a conit already because other conits will not try to claim them.

When an object is faulted, the faulter receives all information about its conit, i.e., the whole conit. This constraint has the side effect of potentially improving performance; by prefetching all objects in the conit, we can reduce the number of future faults for objects within the same conit.

We also require that updates to conit objects are through methods of the `Consistent` interface and that the objects only affect one conit. The bytecode transformer injects calls into the Ivory interface from methods that implement the Consistent interface. Updates to multiple conits introduces issues of transactions and synchronization that are topics of future research.

5.4 Versioning

While our versioning model is not particularly novel, it is worth noting how versioning is affected by the use of conits.

Object updates are monitored at the conit-level. Rather than keeping track of each replica's current version, conits keep a list of dirty object hampers, as shown in Figure 4. The state required therefore grows with the number of objects in the conit instead of the number of replicas.

Without loss of generality, we will describe the server's policy for maintaining versions. Versions are maintained in dirty lists that are labeled by a logical timestamp. The server knows the timestamp of its current version of each conit. As objects are dirtied, references to the objects are added to the most recent dirty list. If an object is in a previous dirty list, it is removed from the older list. If the object was the last object in the dirty list, the dirty list is deleted. A new version list is created each time a node receives a request for updates. To reduce space requirements, if the most recent dirty list contains no objects, the list is simply assigned the next logical time stamp.

When the server receives a request for a conit, the replica also sends the timestamp t of its current version. The server sends the objects that have been

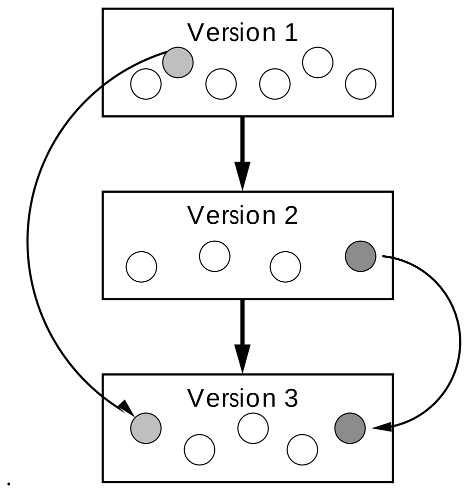

Figure 4: Tracking versions within a conit

dirtied since t.

The state required by this algorithm is linear with the number of objects in a conit. However, the server also stores information about each replica's current version of the conit so that the dirty lists can be pruned. In future implementations, we may explore other pruning techniques that do not require this information and other versioning algorithms.

5.5 API

Several operations on the data are available to applications.

Touch: Get updates for this object and its conit if the object could be stale.

Fault: When an application on a secondary node follows a cross-conit reference to a conit that is not resident on that node, the node faults and fetches that conit from the primary node, which is usually the originating node.

Commit: Push a conit's updates to the primary. If there is a conflict with the primary, the conit is *evicted*. When the conit is next accessed, the full conit is fetched from the primary.

Evict: A conit can be thrown out of a cache. Only information about conit roots is saved because other conits may only refer into a conit through these objects.

These few operations allow applications to implement many different policies for maintaining consistency.

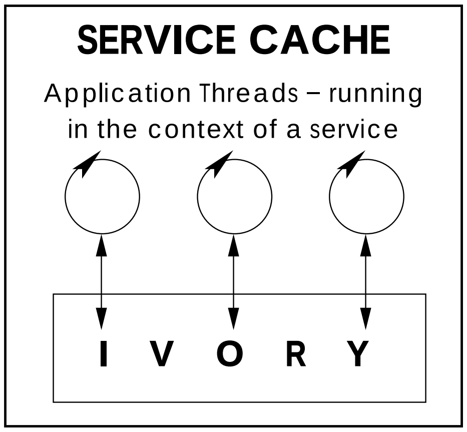

Figure 5: Service Cache using Ivory

6 Service Cache Framework

Service caches are used to scale dynamic services by replicating the Java code and data used to generate dynamic content. These caches should produce correct results, as would be produced by the server–within some application-appropriate margin of error. We want consistent copies of the code and data, while keeping the client-perceived latency low, and we want to minimize the cost of managing state. We will now discuss how we implemented a simplified service cache using Ivory.

A service cache is associated with a service. For our purposes, a service is a group of applications that execute on the same set of data objects. As shown in Figure 5, application threads, which run in the context of a service, are transformed to make calls into the Ivory infrastructure to access and modify data objects.

The implemented service cache is simply a thin layer around Ivory. The service cache is not fully-functional and lacks features such as restricting the amount of space available for the cache or a replacement policy for evicting conits. However, the service cache is a good example application that uses Ivory and is sufficient for testing and evaluating Ivory.

7 Experiments and Results

To test Ivory's scalability, we ran a series of tests using the service cache as our test application. We ran the service cache server and replicas on Sparc Ultra 1s running Java's WebServer on top of Solaris 2.8. The server has 256 MB of RAM, while replicas have either 128 or 256 MB.

The tests are designed to evaluate the performance and scalability of Ivory. The server is primary for a service that maintains a set of four data structures, each of which contains 16 objects. The data structure is a linked list but not Java's LinkedList. Although the list is synthetic, it simplifies evaluating Ivory. Each data structure is referenced by a symbolic name in the server's name cache.

Two servlets run in the context of the service; one servlet requests to read the named data structure, i.e., printing the data structure, while the other modifies each object in the data structure. The invalidation time for data is three seconds, i.e, if the object is referenced at least three seconds after the object was last updated, the replica requests updates from the primary.

We used SimClient [12] to run our tests. SimClient generates a workload for Web servers from a list of URLs; in this case, the URLs are servlets. SimClient also records the latency and throughput for each request.

We created two kinds of trace files: one for reading, one for writing. The tracefile was generated by randomly choosing a data structure from the four available data structures to read or write. SimClient generates a constant write load per second, e.g., sending 5 write requests per second, as defined by our input parameters. The read load is constant along another dimension: there are at most eight outstanding read requests at any time.

For this set of experiments, we concentrated on evaluating the effect of varying writes and adding replicas on service performance. We varied the rate of write requests from 5 to 45 requests per second, and added up to three replicas. A write rate of 45 requests per second means that one tree is modified on average about 11 times per second.

After the data structures were initially created and named, each test ran for five minutes. The results of our experiments are in Figures 6 through 9.

7.1 Replication Overhead

To evaluate the overhead of evaluation, we need to compare the results from having no replicas to having one inactive replica. The inactive replica requested each data structure from the server before the tests were run; the server then must maintain version information about the conit.

The two lines on all four graphs are nearly identical. Read and write latency are only slightly increased by replication. We believe the smaller throughput of the zero replica case at 25 write requests per second in Figure 7 is an anomaly.

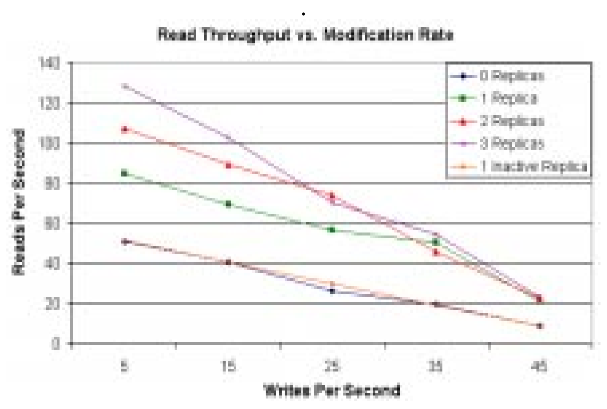

Figure 6: Read Throughput

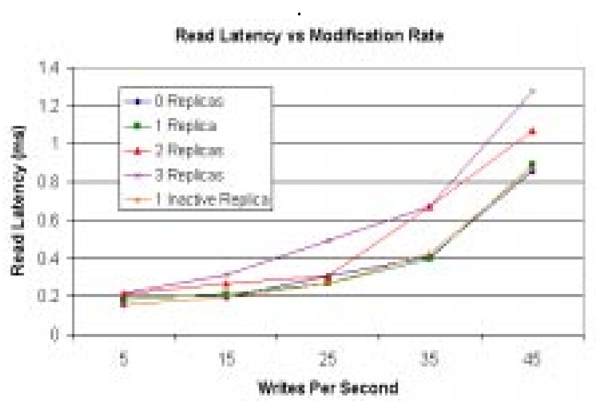

Figure 8: Average Read Latency

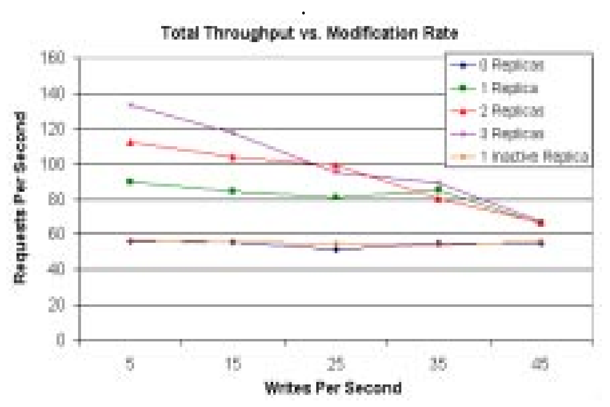

Figure 7: Total Throughput

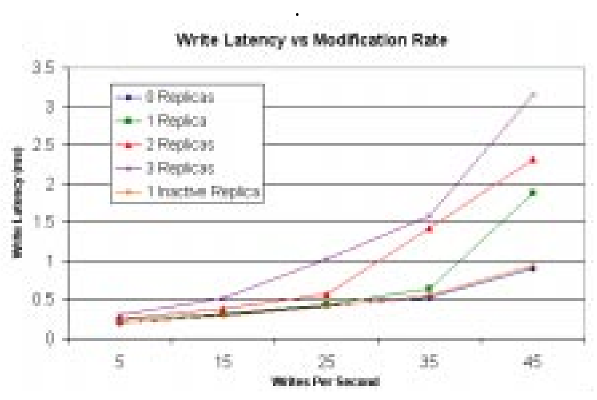

Figure 9: Average Write Latency

7.2 The Effects of Replication

The graphs show that Ivory's replication provides scalability and increased performance for service caches. At low write rates, the system performance is greatly improved by adding replicas; throughput of the service is much higher, while latency is about equal.

As expected, increasing the modification rate decreases the performance of the system and replication. Write requests starve out reader requests at the server and require more update propagation to replicas.

7.3 Discussion and Future Experiments

These results are promising and encourage further testing and analysis of Ivory. We would like to test Ivory using different sized conits to determine the effect of conit size on state management and overhead. While we believe that Ivory will be used in applications where the rate of modification is relatively low, we will look into reducing the impact that writes have on the overall performance of the system. We would also like to add more replicas and to test Ivory using real-world data structures and other applications.

8 Related Work

There are many systems that have goals similar to our data management system or to our proposed applications for this system—such as the service cache; we have learned about many of these other systems. In this section, we highlight a few of them.

Distributed Data Structures (DDS) [14] have similar properties to Ivory's data management layer, but they are not as general as Ivory. Instead of restricting application authors to a few data structures, Ivory automatically distributes and maintains the consistency of any data structure.

Emerald [15] is similar to our proposed service cache in that it is designed to improve the performance of distributed programs. However, objects are relocated—rather than replicated—to other nodes.

Cao's Active Caches [5] is another Java-based approach to caching dynamic content. The Active Cache implementation utilizes Java's security features to perform computation on cached documents. A cache applet is attached to a document so that proxies, without server interaction, can perform the necessary processing on a cached document to make it current.

IBM's trigger monitor [6, 7] pre-generates the dynamic responses to user requests. The responses' data dependencies on underlying data are used to determine when a response should be regenerated, i.e., when data changes occur, the responses that depend on the changed data are regenerated. Pre-generating pages greatly decreases the workload on the server. Since disk space is now inexpensive, pre-generating large numbers of pages is an option. However, if data changes at a much higher rate than pages containing the data are requested or the data affects many pages that probably will not be requested, pre-generating the pages wastes resources.

TACT [21, 20] first introduced the term "conits" for specifying flexible, application-specific consistency and availablity requirements. We believe that TACT can use Ivory for more efficient data replication and management. Instead of replicating entire data stores, as is currently implemented, Ivory can manage partial replicas.

9 Conclusion and Future Work

Ivory is designed to simplify the construction of wide area applications by Ivory providing general support for automatically replicating and caching Java data structures. Ivory replicates data based on conits, which are the granularity for caching, synchronization, and consistency. Conits add flexibility and improve the efficiency of replication. Bytecode transformers automatically adapt applications to Ivory's framework. Finally, we showed that, in the case of service caches, Ivory scales when data is modified at a relatively low rate.

We have mentioned many possible areas for future research. First, we want to further evaluate the current Ivory prototype and analyze the effects of conits and conit size on performance and scalability. We would also like to test other applications on Ivory. Providing a toolkit for transforming applications for use with Ivory would simplify that process.

More general research issues include allowing updates to affect multiple conits, introducing peer-to-peer communication to reduce the server bottleneck, and experimenting with hierarchical conits or other conit structures.

For more information and the source code, please see `http://www.cs.duke.edu/~sprenkle/acad/project`.

Acknowledgments

Dejan Kostic collaborated on the design of Ivory as part of the class project for CPS 212, Distributed Systems. Syam Gadde and Darrell Anderson provided valuable assistance, suggestions, and feedback in evaluating Ivory.

References

[1] *Java Servlet Technology.* http://java.sun.com/products/servlet/.

[2] *JavaServer Pages.* http://java.sun.com/products/jsp/.

[3] G. Banga, P. Drushel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. *Third Symposium on Operating Systems Design and Implementation*, February 1999.

[4] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS wide area security architecture. In *USENIX Security Symposium*, January 1998.

[5] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, September 1998.

[6] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic data. In *IEEE INFOCOM '99*, March 1999.

[7] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A publishing system for efficiently creating dynamic web content. In *IEEE INFOCOM 2000 Conference*, Tel-Aviv, Israel, March 2000.

[8] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, 1998.

[9] G. Czajkowski and T. von Eicken. Jres: A resource accounting interface for Java. In *1998 ACM OOPSLA Conference*, October 1998.

[10] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Workshop on Mobile Computing Systems and Applications*, December 1994.

[11] T. Douglas. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th Symposium on Operating Systems Principles*, December 1995.

[12] Syam Gadde. *Proxycizer Documentation.* http://www.cs.duke.edu/ari/cisi/Proxycizer/.

[13] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley Publishing Company, Reading, Massachusetts, 1996.

[14] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In *Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[15] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[16] L. B. Mummert and M. Satyanarayanan. Large granularity cache coherence in the coda file system. In *USENIX Summer 1994 Conference*, Boston, U.S., 1994.

[17] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: Replicated database services for worldwide applications. In *In Proceedings 7th SIGOPS European Workshop*, pages 275–280, September 1996.

[18] Amin Vahdat. Toward wide-area resource allocation. In *Parallel and Distributed Processing Techniques and Applications*, June 1999.

[19] D.S. Wallach, D. Balfanz, D. Dean, and E.W. Felten. Extensible security architectures for Java. In *16th ACM Symposium on Operating Systems Principles*, pages 116–128, October 1997.

[20] H. Yu and A. Vahdat. Combining generality and practicality in a conit-based continuous consistency model for wide-area replication. In *Operating Systems Design and Implementation*, October 2000.

[21] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *21st International Conference on Distributed Computing Systems (ICDCS)*, April 2001.