A Performance Study of Clustering Web Application Servers with Distributed JVM †

King Tin Lam, Yang Luo, Cho-Li Wang
Department of Computer Science, The University of Hong Kong
{ktlam, yluo, clwang}@cs.hku.hk

Abstract

A Distributed Java Virtual Machine (DJVM) is a cluster-wide set of extended JVMs that enables parallel execution of a multithreaded Java application. It has proven effectiveness for scaling scientific applications. However, leveraging DJVMs to cluster real-life web applications with commercial server workloads has not been well studied. This paper presents a new generic clustering approach based on DJVMs that promote user transparency and global object sharing for web application servers. We port Apache Tomcat to our JESSICA2 DJVM and study the performance of a wide range of web applications running on the server. Our experimental results show that this approach can scale better than the traditional clustering approach, particularly for cache-centric web applications.

1. Introduction

While application servers have become standard infrastructure for supporting web-based enterprise applications, their clustering solutions are generally laborious and error-prone. Distributed Java technology has contributed a wealth of APIs for application clustering. Mastering these APIs is however practically daunting. A holistic solution would be turning to a generic clustering middleware platform that eliminates the need of application code changes and restrictive design patterns imposed by the existing clustering technologies.

In this work, we propose a *Distributed Java Virtual Machine (DJVM)* approach to web application clustering. A DJVM is a set of coupled JVM instances spanning multiple cluster nodes to enable parallel execution of a multithreaded Java application as if it was running on a single powerful machine. As the design of a DJVM adheres to the standard JVM specification, it hides the cluster from the application by presenting a *single-system image (SSI)*, making application design orthogonal to the low-level clustering decisions. So web applications that follow the original Java multithreaded programming model on a single machine can be easily clustered without rewriting application code. Besides, DJVMs enable global sharing of cluster-wide resources among distributed threads. This provides a platform for cooperative computations and data caching among server nodes that can be exploited to outperform the existing clustering approaches in the web community.

DJVM research efforts like [1, 2, 8] have proven successful for scientific benchmarks. However, clustering real-life commercial applications with server workloads on top of DJVMs is more challenging and has not been well-studied. This work bridges the gap by presenting performance results of running Apache Tomcat on the JESSICA2 DJVM. Tomcat, typifying object-based servers, has some unique runtime properties including intensive threading, high read/write ratios, extensive object sharing via collections framework and fine-grained irregular object access patterns. We discuss their potential impacts on the DJVM performance through testing with a diversity of web applications. We show that for I/O-centric applications, the DJVM approach can scale out transparently, yet equally well as traditional clustering and for cache-centric applications, the throughput gets much better due to global cache hits given by our distributed shared heap design. Other contributions of this work include a taxonomy of existing web application clustering solutions with their common drawbacks identified, characterization of DJVM-level overheads for server applications, followed by suggestions on the next-generation DJVM design and optimization.

For the rest of this paper, Section 2 surveys the existing web application clustering solutions. Section 3 describes the JESSICA2 DJVM. In Section 4, we explain Tomcat running on JESSICA2. Section 5 evaluates the performance of the DJVM clustering approach. Section 6 discusses the related work. Our conclusions and suggested future work are given in Section 7.

† This research is supported by Hong Kong RGC grant HKU7176/06E and China 863 grant 2006AA01A111.

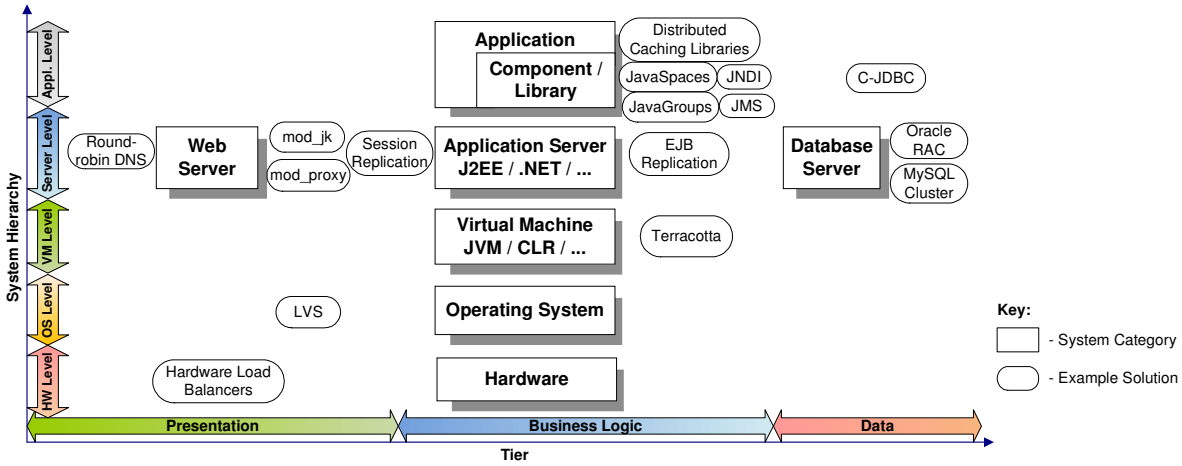

Figure 1. Taxonomy of existing server clustering technologies

2. Existing Server Clustering Solutions

In the web community, clustering is broadly viewed as server load balancing and failover. Figure 1 shows a comprehensive survey on the existing clustering solutions. Along one dimension of the taxonomy, clustering solutions vary in scalability, flexibility and maintenance cost when implementing at the level of hardware, OS, virtual machine, server or application. In the other dimension, they differ in clustering granularity (requests, sessions, components, objects or connections) along the tier of processing.

Hardware load balancers are easy to use but expensive and inflexible for growth. OS-level solutions such as Linux Virtual Server (LVS) [3] provide the appearance of an SSI for all applications at the cost of complicated setups and OS kernel upgrades. Round-robin DNS, although easy to implement, has no knowledge of user sessions, thus causing integrity problems. Web server plug-ins like Apache mod_jk connector [4] are cost-effective and session-aware, but they may create load hotspots due to *session stickiness*.

Advanced clustering technology, spawned chiefly in the J2EE world, supports state sharing across application servers. HTTP sessions, stateful Enterprise JavaBeans (EJBs) and plain old Java objects (POJOs) embedding business data are commonly shared application states. Some application server products ship with clustering support for HTTP sessions and stateful session beans. Conventional approaches to shared-database and shared-file state persistence scale poorly. In-memory session or EJB replication is an improved technique that serializes objects into byte streams for sending to peer servers over communication services like Java Messaging Service (JMS) and JavaGroups [5]. But this involves group-based synchronous replications (generally all-to-all replications) that are only efficient in very small-sized clusters. EJB clustering requires complex setup of a cluster-wide shared JNDI tree for lookup of clustered objects. Clustering POJOs that conform to no standard interface needs application code retrofit using extra APIs such as JavaSpaces [6] or distributed caching libraries [7] to share objects among the JVMs.

There are also data-tier clustering solutions like MySQL Cluster, Oracle Real Application Clusters (RAC) and C-JDBC which support connection load balancing and synchronous replication of data updates over the cluster. Database cluster size is however often limited due to more expensive hardware and licensing costs.

We note that most existing (Java-based) clustering solutions have poor user transparency and suboptimal performance. Some of the common drawbacks can be summarized as follows:

(1) *Imposing burdens and restrictions on application designs*: complex setup, application rework with extra APIs and restrictive design patterns (e.g. can't share non-serializable objects) burden developers.

(2) *Breaking referential integrity*: object clones made by Java serialization lose the original object identity when deserialized and may cause consistency problems among objects with cross-references.

(3) *Imposing costly communication*: Java serialization significantly degrades performance, since it has to trace and clone many objects even for one field change on a shared object.

(4) *Lacking global signaling or coordination support*: Without cluster-wide synchronization, subtle consistency issues arise when some design patterns or event-based services (e.g. timers) are migrated from standalone platforms to clusters.

(5) *Lacking global resource sharing*: Most clustering solutions lack global information for wisely managing the aggregated resources of the whole cluster.

3. The JESSICA2 Distributed JVM

JESSICA2 [8] is a DJVM designed for transparent parallel execution of multithreaded Java applications in a cluster environment. It provides the illusion of an SSI when connecting Java with clusters such that applications are clustered transparently with no burden of source code modification and bytecode preprocessing. It automatically handles thread distribution, data consistency of the shared objects and I/O redirection so that the program sees only a single system, with the aggregated computing power, memory and I/O capacity of the entire cluster.

JESSICA2 has bundled several salient features for SSI realization. First the class loader of JESSICA2 is extended with a remote class loading capability. When a worker JVM cannot find a class file locally, it can request the class bytecode on demand and fetch the initialized static data from the master JVM through network communication. This feature greatly simplifies cluster-wide deployment of Java applications.

Second, JESSICA2 incorporates a cluster-aware JIT compiler to support lightweight Java thread migration across node boundaries to assist global thread scheduling. Besides an initial thread placement for striking a raw load balance, dynamic load balancing is possible at runtime by migrating Java threads that are running into computation hotspots to the less loaded nodes.

For seamless object views from migrated threads, JESSICA2 provides a heap-level service called *Global Object Space (GOS)* [10] to support location-transparent object access. Distributed threads share objects in the GOS as if they were in a single heap. The GOS implements packing functions to ship object data to the requesting nodes. Received objects are cached locally to improve data access locality. Cache coherence across reads/writes on shared objects is guaranteed by a home-based release-consistent memory model.

JESSICA2 offers a global I/O space via a transparent I/O redirection mechanism built in the native class library so that I/O requests (file and socket accesses) can be served, virtually, by any node without strict reliance on shared file systems or virtual IP. To exploit I/O parallelism atop transparency, connectionless network I/O and read-only file accesses, if local replicas exist, are done locally without redirection.

4. Running Apache Tomcat on JESSICA2

Apache Tomcat is the official reference implementation of the Java Servlet and JavaServer Page (JSP) specifications. It is also the world's most widely used open-source Java web application server [9].

Tomcat characterizes real-life Java applications that are usually more complex, data-centric, object-oriented and extensive in Java library usage than scientific applications. We now summarize Tomcat's runtime properties and their potential impacts on the DJVM performance:

(1) *I/O-intensive and highly-threaded*: most web server workloads are I/O-bound and composed of short-lived requests of small computation-communication ratios. Application servers usually configure a large thread count to hide I/O blocking latency. This implies longer wait time if lock contention occurs.

(2) *High read/write ratios*: reads typically dominate in web applications owing to user browsing behaviors. DJVM design can make use of this trait to optimize the cache coherence protocols.

(3) *High utilization of collections framework*: Tomcat makes extensive use of Java collection classes like Hashtable and Vector to store information (e.g. web contexts, sessions, MIME types, status codes, etc). Thread-safe operations over these objects will cause excessive synchronizations, thus erecting a barrier to scalability in networked cluster environments.

(4) *Fine-grained object access with irregular reference locality*: By Tomcat's object-oriented design, object accesses are very frequent; object graphs are complex with ramified connectivity. Frequent hash table accesses induce irregular reference locality, contrasting with consecutive memory access pattern in most scientific (SPMD) applications. This complexity demands smart object prefetching techniques to avoid excessive fine-grained communications.

Figure 2 depicts the execution of Tomcat on top of JESSICA2 in a 4-node cluster. The threads created at startup will migrate to the worker nodes to balance workload. The threads then load the classes of the Java library, Tomcat and the web applications deployed to it dynamically through JESSICA2's cluster-aware class loader. The distributed threads continuously pull workloads from the master node by accepting and handling incoming connections via transparent I/O redirections.

When a client request is accepted, the context manager of Tomcat matches it to the target web context. If the request carries session state, such as a cookie, the standard manager will search for the allocated session object from the session's hash table. The underlying GOS allows all Tomcat container objects allocated in the master JVM (e.g. web contexts, and sessions hash table) to be transparently shared among the distributed threads. With object state checks injected by the JIT compiler, access faults on a non-local object reference will cause the up-to-date object to be fetched from its home node. Cluster-wide consistency is then enforced on the home copy and all cache copies derived from it.

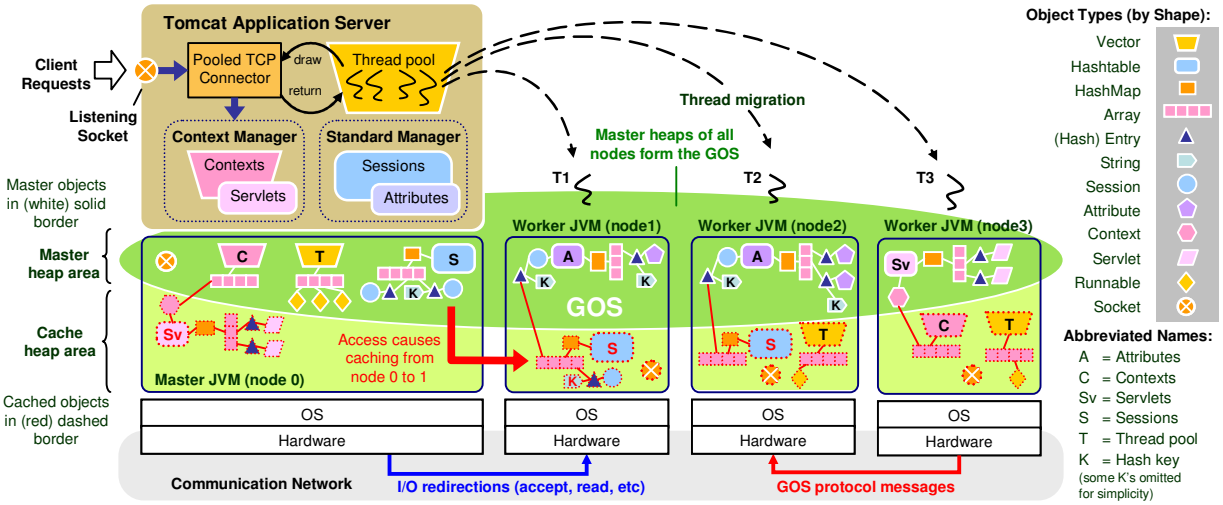

Figure 2. Execution of Tomcat on JESSICA2 DJVM

JESSICA2 extends the existing Java Memory Model (JMM) to provide a consistent unified heap over a cluster. We call the portion of heap storing usual unshared objects and home object copies the *master heap area*. Each thread is given a local memory work area, called the *cache heap area,* for keeping copies of remote objects. On entering an object monitor (corresponding to a lock), the thread invalidates its cache heap area so that later uses will fault in their up-to-date objects. On exiting the monitor, updated cache objects are flushed to their homes so the next acquiring thread can see the changes. In this way, the GOS creates a helpful cache effect that we call *implicit cooperative caching* among the threads.

This effect is illustrated by Figure 2. The thread T1 faults in the object graph under S and caches a copy of S in its local heap. When T1 serves a new client session, a new hash entry will be put into S. This corresponds to building a local reference from the cached S to the new entry. Upon synchronization events, this update is propagated back to the home of S so that the thread T2 can see and access it by remote fetching from node 2. This global cache effect transparently shifts the duty of managing session data consistency across servers to the GOS layer and makes every node eligible to handle requests belonging to any client session.

JESSICA2 strives to address the characteristics of Tomcat discussed above. We optimize the coherence protocol (based on property 2) by attaching per-object timestamp checks to lock-acquire requests, avoiding invalidation of cache copies that are still valid (particularly those read-only) to minimize access faults and hence the lengths of critical sections. The GOS also supports various optimizations such as *object pushing* (a prefetching technique that can address property 4). More details on the GOS design can be found in [10].

5. Performance Analysis

5.1. Experimental Setup

Our experimental platform consists of three tiers: (1) *web tier*: a 2-way Xeon SMP server with 4GB RAM for running the master JVM of JESSICA2 with Apache Tomcat 3.2.4 started on it; (2) *application tier*: a cluster of eight x86-based PCs with 512 MB RAM acting as the DJVM worker nodes; (3) *data tier*: a cluster of four x86-based PCs with 2GB RAM supporting MySQL Database Server 5.0.45. All nodes run under Fedora Core 1 (kernel 2.4.22). The three tiers are connected up by Gigabit Ethernet, while nodes within the same tier are linked by Fast Ethernet networks.

The initial and maximum heap sizes of each worker JVM are set to 128MB and 256MB respectively. Each database node has the same dataset replica. MySQL replication is enabled to synchronize the replicas at nearly real time. Jakarta JMeter 2.2 is used to synthesize varying workloads to stress the testing platform.

Table 1 shows the application benchmark suite used to evaluate our clustering approach using the DJVM. They are designed to model real-life web applications of diverse workload characteristics.

Table 1. Application benchmark suite

Application	Object Sharing	Workload Nature	I/O
Bible-quote	No sharing	I/O-intensive	Text files
Stock-quote		Compute-intensive	Database
Stock-quote/RSA			
SOAP-order	HTTP session	I/O-intensive	
TPC-W			Database / image files
Bulletin-search	Cached query results	Memory-intensive	Database

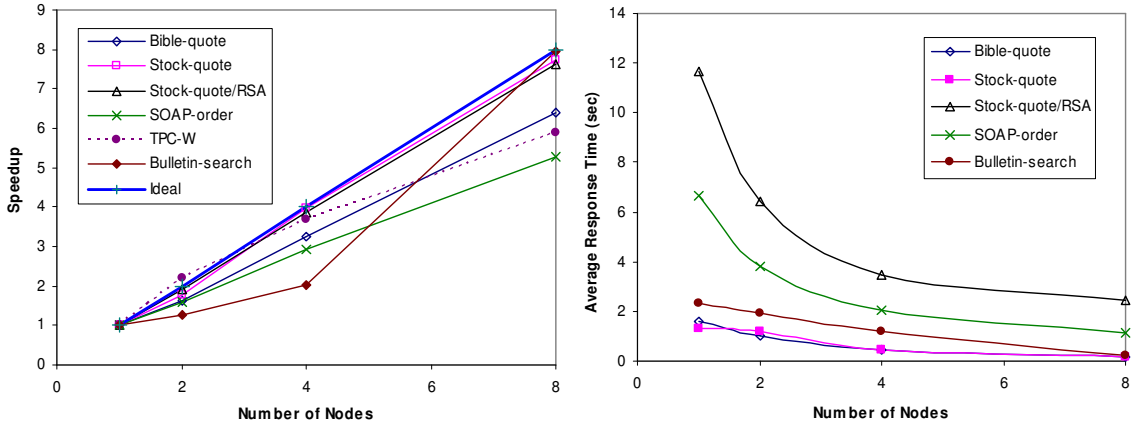

Figure 3. Scalability and average response time obtain by Tomcat on JESSICA2

(1) *Bible-quote* models applications such as text search engines, news archives and company catalogs. This servlet is I/O-intensive, serving document retrievals and search requests over a set of text files of books.

(2) *Stock-quote* simulates stock market data providers. The application reads stock price data matching the input date range from the database and formats the query result into an XML response.

(3) *Stock-quote/RSA* is secure version of Stock-quote involving compute-intensive operations of 1024-bit RSA encryption on the price data.

(4) *SOAP-order* characterizes a B2B e-commerce web service. The application parses SOAP messages (by Apache SOAP 2.3.1) of securities order placements, checks the user accounts and order details and then puts the successful transactions into the database.

(5) *TPC-W* is a standard transactional web benchmark specification. It models an online bookstore with session-based workloads and a mix of static and dynamic web interactions. We use the Java servlet implementation from [11], but with pooled database connections cached in thread-local storage [12].

(6) *Bulletin-search* emulates a search engine in a bulletin board or web forum system. We take the data dump from the RUBBoS benchmark [11] to populate the database. The application maintains a hash-based LRU-cache map of the results of the costly database searches, and is thus memory-intensive.

The original Tomcat is ported to JESSICA2 with a few customizations: (1) We replace the original thread pool by a simpler implementation that spawns a static count of non-pooled threads based on the server configuration file; (2) several shared object pools (e.g. static mapping tables for MIME types and status codes) are disintegrated into thread-local caches. The modifications are non-intrusive (only about 370 lines of codes including the new class, corresponding to 0.76% of the Tomcat source base).

5.2. Scalability Study

In this experiment, we measure the peak throughputs and average response times obtained by scaling worker nodes from two to eight. The speedup is calculated by dividing the baseline runtime of Tomcat on Kaffe JVM 1.0.7 by the parallel runtime of Tomcat on JESSICA2. Figure 3 shows the results obtained for each benchmark. We see that most applications scale well, achieving efficiencies ranging from 66% (SOAP-order) to 96.7% (Stock-quote). Bible-quote, Stock-quote and Stock-quote/RSA show almost linear speedup because they belong to the class of stateless applications, yielding true parallelism without any GOS communications between the JVMs. In particular, Stock-quote and Stock-quote/RSA involve operations of coarser work granularity, such as string manipulations and RSA encryptions, and are hence more able to attain nearly perfect scalability. Smaller speedups are obtained for stateful applications like SOAP-order and TPC-W, since they involve GOS overheads when synchronizing HTTP session and some other objects across the JVM heaps. Bulletin-search shows a nonlinear but steepening curve in speedup when the number of worker nodes scales out due to the implicit cooperative cache effect given by the GOS.

Table 2 shows the cluster-wide thread count used in each application and the overall protocol messaging overheads of JESSICA2 in the 8-node configuration. The count of I/O redirections is proportional to the request throughput and generally does not impact scalability. The higher number of GOS protocol messages explains the poorer scalability obtained by the application as shown in Figure 3. Bulletin-search is an exception because its performance is more determined by its cooperative caching benefits which could supersede the cost of GOS communications.

Table 2. JESSICA2 messaging overheads

Application	# Threads	# GOS messages / sec	# I/O redirections / sec
Bible-quote	80	0	2006
Stock-quote	80	0	1791
Stock-quote/RSA	80	0	275
SOAP-order	16	979	146
TPC-W	40	351	1413
Bulletin-search	16	483	297

5.3. Comparison with Existing Solutions

We compare the DJVM approach with a common clustering method for Tomcat using web load balancing plug-ins. We run an instance of Apache web server 2.0.53 on the web tier and eight standalone Tomcat servers on the application tier of our platform. The web server is connected to the Tomcat servers via the mod_jk connector 1.2.18 with sticky-session enabled[1]. The cluster-wide thread count and heap size setting in this experiment are the same as in the DJVM approach.

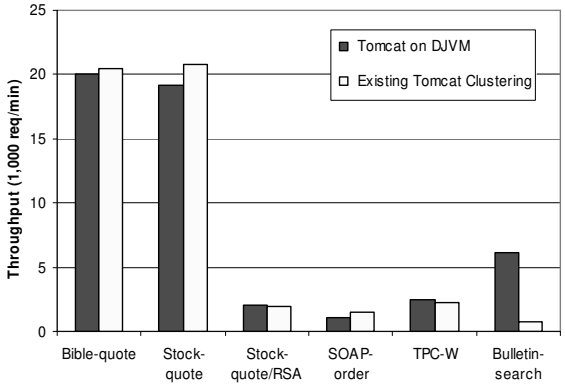

Figure 4. Comparison of Tomcat on DJVM and existing Tomcat clustering

Figure 4 shows the throughputs obtained by the two clustering approaches on eight nodes. We can see that both solutions achieve similar performance (within ±8%) for those stateless web applications (Bible-quote, Stock-quote and Stock-quote/RSA). These applications exhibit embarrassing parallelism and do not gain much advantage from the GOS. Both solutions perform more or less the same because our transparent I/O redirection and mod_jk's socket forwarding are functionally alike for dispatching requests and collecting responses.

TPC-W performs about 11% better on JESSICA2 than with mod_jk. One reason is that servers sharing sessions over the GOS are no longer restricted to handle requests bounded to their sticky sessions while load hotspots can happen intermittently when using mod_jk.

[1] In-memory session replication is not supported in the comparison.

On the other hand, SOAP-order performs 26% poorer on JESSICA2 than with mod_jk. The main factor that pulls down the throughput is that the SOAP library has some code performing fairly intensive synchronizations when processing every request. Bulletin-search performs 8.5 times better on the DJVM due to secondary application cache hits contributed by the GOS.

5.4. Effect of Implicit Cooperative Caching

Bulletin-search exemplifies the class of applications that can exploit the cache effect of GOS to virtualize a large heap for sharing application data. Table 3 shows the application cache hits obtained by Bulletin-search when the number of nodes scales from one to eight. With the GOS, the capacity setting of the cache map can be increased proportional to the node count beyond the single-node limit for different portions of the map that are stored under different heaps. A newly created cache entry will have its object reference written to the shared hash map. Threads can exploit *indirect (global) cache hits* in case the desired object is not in the local heap, easing the database bottleneck.

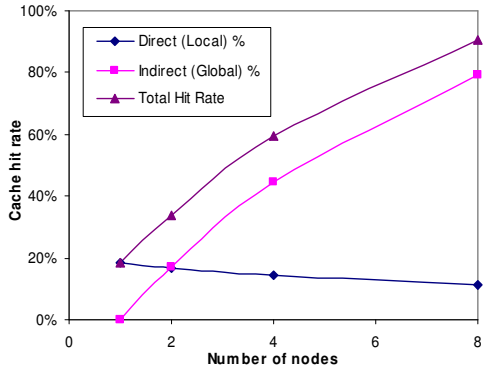

Figure 5. Bulletin-search's cache hit analysis

We can see from Figure 5 that the overall hit rate keeps rising along with the scaling of worker nodes of the DJVM and that most of the cache hits are caused by indirect hits when the single-node capacity has been exceeded. Here we define a term called *relative cache size (RCS)* to refer to the percentage of the aggregated cache size (combining all nodes) relative to the total size of the data set. In the 4-node case, when the RCS is below 50%, the achievable cache hit rate is only around 60%. Since the other 40% gets no improvement, the speedup is merely a factor of two. But when the RCS exceeds a certain level (e.g. 90% in the 8-node case), most of the requests are fulfilled by the global cache instead of going through the database tier. This explains the non-uniform scalability curve of this application in Figure 3.

Table 3. Bulletin-search's cache size setting and hit rates augmented by GOS

No. of Nodes	Cache Size (#Cache Entries)	Relative Cache Size	Total Hit Rate	Indirect Hit Latency (ms)	Cost Ratio of Miss : Indirect Hit	Throughput Speedup
1	512	12.5%	18.6%	N/A	N/A	N/A
2	931	22.7%	33.9%	9.07	40.79	1.26
4	1862	45.5%	59.3%	8.18	45.23	2.02
8	3724	90.9%	90.7%	11.74	31.52	7.96

Table 4. GOS overhead breakdown

GOS Message Type	# Messages / Sec		
	SOAP-order	TPC-W	Bulletin-search
Lock acquire	198	48	61
Lock release	198	48	61
Flush	217	70	92
Static data fetch	18	10	0
Object fault-in	197	99	160
Array fault-in	79	50	105

Table 5. Cluster-wide locking overheads

Application	#Local locks / sec	#Remote locks / sec	%Contended remote locks	Local : remote lock ratio
SOAP-order	232631	198	35%	1175:1
TPC-W	240470	48	45%	5010:1
Bulletin-search	27380	61	6.5%	449:1

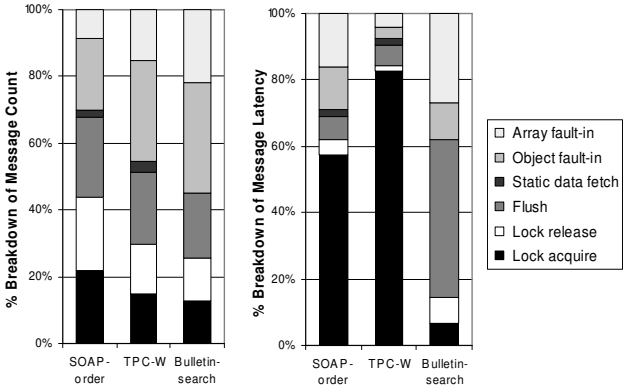

Figure 6. GOS percentage overhead

5.5. GOS Overhead Breakdowns

Table 4 shows the GOS overhead breakdowns in terms of messaging rates for the three stateful applications. Figure 6 supplements this with a percentage breakdown of message count and latency. Lock acquire and release messages are issued upon locking a remote object. Flush messages are sent at lock releases (there are slightly more flush messages than lock releases since updates may flush to more than one home). Other overheads are related to access faults. SOAP-order clearly has higher remote lock rate than other applications. On closer investigation, we found that one utility class of the SOAP library would induce, for each request, 5-6 remote locks on a hash table in the web context and four remote locks on ServletContextFacade due to Tomcat's facade design pattern. Such heavy remote locking explains the relatively poorer scalability of SOAP-order.

Table 5 presents the local and remote locking rates for each application. We can see local locks are much more than remote. The reason is that Java-based servers perform thread-safe reads/writes on I/O stream objects, issuing enormous local locks. While local lock latency is very short (about 0.2us in our study), remote lock latency is however several thousands times longer in commodity clusters; remote locks are yet practically much fewer in most web applications. Table 5 also tells us that SOAP-order and TPC-W have about 35% to 45% remote locks under cluster-wide contention, thus prolonging the wait time before locks are granted. This is why lock acquire has been the dominant part in the message latency for these two applications in Figure 6.

6. Related Work

Shasta [14] is a fine-grained software DSM system using binary code instrumentation to transparently give cluster-wide semantics to memory accesses. Oracle 7.3 database server is ported on Shasta running on SMP clusters. Running TPC-B (OLTP) and TPC-D (decision support) on the system showed only a slight speedup of 1.1 on three servers even with a low-latency Memory Channel Network. Their experience reveals severe restrictions of OS-level SSI solutions, compared to our JVM-level approach, since correctness of binary applications relies on the much stricter consistency model imposed by hardware. Being able to adopt a relaxed memory model as in our case is essential to server applications that may need frequent synchronization.

One of the earliest DJVM designed to transparently run multithreaded server applications, such as Jigsaw, on a cluster is cJVM [1]. cJVM operates in interpreter-mode and uses method shipping to realize a *master-proxy* model enforcing sequential consistency on distributed shared objects. In contrast, JESSICA2 runs in JIT-compilation mode and conforms to release consistency. In [15], cJVM is evaluated with pBOB (Portable Business Object Benchmark), a business benchmark inspired by TPC-C, on a 4-node cluster with non-commodity Myrinet. They obtained an efficiency of around 80%. However such a high efficiency may not be achievable if their protocol runs with JIT enabled and commodity Ethernet as in our case.

Terracotta [16] is a recent JVM-level clustering product. It uses bytecode instrumentation techniques

similar to JavaSplit [2] except that it works within an aspect-oriented programming (AOP) framework and has to instrument product-specific classes. Users need to manually specify shared classes as *distributed shared objects (DSOs)* and their cluster-aware concurrency semantics. This configuration-driven approach is "translucent" (less transparent than our SSI-oriented approach) and liable to subtle semantic violations. Terracotta relies on a central server to store all DSOs and to coordinate shared updates across JVMs, which will likely create hotspots as the cluster size increases. In contrast, our home-based coherence protocol is decentralized and immune to such bottlenecks.

7. Conclusion and Future Work

We have presented a new transparent clustering approach using distributed JVMs (DJVMs) for web application servers. DJVMs enhance the ease of application clustering and global resource integration – both of which have been problems for existing web-domain clustering solutions. Our performance study of running Apache Tomcat on JESSICA2 demonstrates scalable speedups for a variety of web applications, particularly those session-based and cache-centric ones. Our overhead analysis shows that scalability hinges on the number of remote locks (under contention) due to fine-grained object sharing.

We accordingly suggest some design guidelines for next-generation DJVMs. First, one should maintain high execution concurrency so that the helpful cache effect of global object sharing will not be offset by the cluster-wide locking overheads. We find that thread-safe Java collection classes, being the usual containers for shared objects, are the major source of remote locks. In fact, more scalable containers have been recently developed under the latest Java concurrent utility package [13]. However, these concurrent data structures are not immediately portable to cluster environments, in terms of performance. For example, they make heavy use of volatile fields for lightweight synchronization, but these are treated as locks and handled inefficiently by most DJVMs. Advanced coherence protocol designs such as concurrent-read-exclusive-write support for volatile fields will be vital for continued success in using the DJVM technology to scale the latest Java applications.

The importance of optimizations that save or aggregate fine-grained remote operations in DJVMs is next to concurrency. Some solutions like method shipping, thread migration, object home migration and prefetching have been proposed [1, 8, 10]. But these items of work focus on mechanisms, rather than how to best use these mechanisms for a balanced effect of data locality, message aggregation and load distribution. An adaptive hybrid use of these runtime techniques, guided by lightweight profiling, is a challenging research problem for future study.

References

[1] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[2] M. Factor, A. Schuster, and K. Shagin. JavaSplit: a runtime for execution of monolithic Java programs on heterogeneous collections of commodity workstations. In *Int. Conf. on Cluster Comput.*, pages 110-117, 2003.

[3] W. Zhang. The Linux Virtual Server Project. http://www.linuxvirtualserver.org/.

[4] The Apache Software Foundation. The Apache Tomcat Connector. http://tomcat.apache.org/connectors-doc/.

[5] B. Ban. JavaGroups multicast communication toolkit. http://www.sourceforge.net/projects/javagroups.

[6] Q. H. Mamoud. Getting started with JavaSpaces technology: Beyond conventional distributed programming paradigms. http://java.sun.com/developer/technical Articles/tools/JavaSpaces/.

[7] C. E. Perez. Open source distributed cache solutions written in Java. http://www.manageability.org/blog/stuff/distributed-cache-java.

[8] W. Zhu, C. L. Wang, and F. C. M. Lau. JESSICA2: A distributed Java virtual machine with transparent thread migration support. In *Proc. IEEE 4th Int. Conf. Cluster Comput.*, Chicago, Sep. 2002, pp. 381–388.

[9] A. Zeichick. Tomcat, Eclipse named the most popular in SDTimes study. http://www.sdtimes.com/content/article.aspx?ArticleID=31882.

[10] W. Fang, C. L. Wang, and F. C. M. Lau. Efficient global object space support for distributed JVM on cluster. In *Proc. Int. Conf. on Parallel Processing*, pages 371–378, British Columbia, Canada 2002.

[11] ObjectWeb. JMOB: Java Middleware Open Benchmarking. http://jmob.objectweb.org/.

[12] B. Goetz. Threading lightly, Part 3: Sometimes it's best not to share - exploiting ThreadLocal to enhance scalability. http://www.ibm.com/developerworks/java/library/j-threads3.html.

[13] D. Lea. Java concurrent utility package. http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html.

[14] D. J. Scales, and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, Saint Malo, France, 1997.

[15] Y. Aridor, M. Factor, A. Teperman et al. Transparently obtaining scalability for Java applications on a cluster. *Journal of Parallel and Distributed Computing*, v.60 n.10, p.1159-1193, Oct 2000

[16] A. Zilka. Terracotta - JVM clustering, scalability and reliability for Java. http://www.terracotta.org.