The University of Hong Kong

Faculty of Engineering

Department of Computer Science

CSIS7702 4-Module Project

Deploying Enterprise Web Services by DJVM Approach

Submitted in partial fulfillment of the requirements for the admission to the degree of Master of Science in Computer Science

By
Lam King Tin
1998029183

Supervisor's title and name: Associate Professor Dr. C. L. Wang

Date of submission: 5 Sep 2006

Declarations

I hereby declare that the dissertation entitled “Deploying Enterprise Web Services by DJVM Approach” represents my own work and has not been previously submitted to this or any other institution for a degree, diploma and other qualifications.

_____

Lam King Tin
2006

Acknowledgements

Give thanks to the LORD, for he is good; his love endures forever. The LORD is my shepherd, I shall not be in want … Even though I walk through the valley of the shadow of death, I will fear no evil, for you are with me … Surely goodness and love will follow me all the days of my life, and I will dwell in the house of the LORD forever. (Pslam 23 and 107)

Originally, this dissertation work was not quite successful - the software items were far from stable and speedup was poor despite long time of efforts. It was really the same experience in the quote: "I walk through the valley of the shadow of death." However, deep thanks to my Lord God Almighty for his wonderful grace, I have got the valley through! For God helps me with his mighty hand. It was an unforgettable moment to work out useful data in this project which is seemingly impossible to me. Throughout the past three years of my master study, I have been holding this key verse in the bible to live by faith: "*But seek first his kingdom and his righteousness, and all these things will be given to you as well.*" (Matthew 6:33) When I decided to follow Jesus and preach the gospel, God has been faithfully giving me "all these things" including this dissertation sponsored by my kind professor and supported by my research fellows! I believe all these helps are truly prepared by my Lord. ☺ May He be glorified.

I would like to thank my supervisor, Dr. Cho-Li Wang for his endless support, patient guidance, encouragement, many invaluable advices and financial support throughout this project. His professional teaching and genuine attitude in researching a problem has inspired me a lot during my master study. I also want to thank my second examiner, Dr. Anthony Tam for his precious comments in my presentation and his help on inspecting this report.

Special thanks to Dr. Zhu Wenzhang, James Luo and Alan Li for their expertise and assistance in fixing JESSICA2 issues. They have been very supportive and patient to me. I am full of gratitude to their help and useful suggestions.

I would also thank Benny Cheung and Roy Ho for their assistance in securing the Ostrich cluster environment for my development and experimental work.

Next is to thank my brothers and sisters in our church. Without their earnest prayer and care, it would not have been possible for me to complete this study.

Finally, I want to express my deepest gratitude to my wife and my parents. My wife has offered me her full support in whatever she can do – she always pray and cook for me ☺. And my parents also do. They are my truly beloved ones.

Abstract

Cluster has been commonly employed to support high-performance web server applications. However most existing server-side clustering software tools fall short in transparency and scalability. Recent advances in distributed Java virtual machine (DJVM) researches could help to provide more transparent and scalable clustering. This dissertation explores the applicability of DJVM to speedup application servers by clustering at the JVM-level. To account for the effectiveness of this approach, we ported a version of Apache Tomcat, a popular web application server, onto our DJVM prototype JESSICA2. During the integration process of Tomcat and JESSICA2, we identified and addressed various compatibility and performance issues which are insightful to how application server and DJVM should interface in an effective way. We successively modified both systems accordingly and evaluated the performance of the integrated server with several application benchmarks. Experimental results have showed that this approach can give a much better speedup and scalability than web server-based dispatching in relatively compute-intensive web applications. We have done a ground-breaking work in DJVM systems by extending their practical use to the web application domain.

Table of Content

List of Figures

Chapter 1. Introduction

1.1 Background

In the recent decade, many advances in server-side technologies have revolutionized the nature of web applications in terms of their sophistication. The scope of services that can be supported online is ever widening – shopping, stock trading, bill payment and businesses can all be done online. However, the workload demands on servers also grow drastically with service needs. Web requests are no longer simple webpage retrieval but tend to be increasingly resource-intensive. A single request may trigger a database search, a transaction, a complex business program and also dynamic content generation. The server bottleneck hence becomes more critical than the network bottleneck and limits the scalability of servers in processing large numbers of simultaneous requests. Researches on Internet performance also show almost 40% client latency is causing on the server side [7]. Therefore high-performance servers are vital to service providers for presenting services of excellence to clients.

Clustering has became a common approach to solve the server bottleneck problem. *Google*, the most popular Internet search engine portal, employs a cluster of over 8000 machines to cope with the enormous daily search requests [8]. Their choice of using clusters instead of powerful mainframes is motivated by lower cost along with greater I/O device bandwidth and better scalability for ever surging demands.

However, without a scalable software support, the goal of high performance is still impossible no matter how many machines are being used. Therefore, introducing scalable clustering support to server systems has become a hot research topic. The cluster computing community takes an active look at Java. With the power of "write once run anywhere", Java is among a top choice for web application development. Many Java application servers like IBM WebSphere, BEA Weblogic, JRun, JBoss and Apache Tomcat had emerged quickly in the market and attracted many enterprises as their clients. With the abovementioned performance concern, they are progressively built or upgraded with clustering ability.

Clustering a web application means two things: request load balancing and service availability maintenance. It can be done at different levels of the system hierarchy, ranging from the operating system to the application itself. In this dissertation, we would propose clustering at a middleware level; more exactly, it is at the Java virtual machine (JVM) level that is below the server. The title "DJVM approach" may sound new to many web developers. Indeed, we apply a distributed Java virtual machine (DJVM) to cluster an application server to speed up performance. Clustering at the JVM level rather than at the application or at the server layer has a number of advantages to be explained in section 1.3 to follow.

1.2 General Approaches of Server Clustering

Clustering helps to balance the workload on all servers in the cluster and maintains service availability even if any one server suffers from failure. We would summarize the common approaches to accomplish both the functions below.

1.2.1 Load Balancing

As a cluster is generally with a single entry point, the load balancing mechanism is naturally done at the entrance by request dispatching. Typically, a switch or a web server is situated at front-end to dispatch incoming requests to server instances based on some scheduling policies like round-robin. Switch-based dispatching works at TCP/IP level and is said to be *content-blind* (OSI layer-4 service). On the other hand, dispatching by a web server is *content-aware* (OSI layer-7 service): the server can look at the requested URL, cookie header, etc to determine where the request should be dispatched, thus attaining better web content cache affinity. Therefore, this is a more popular option. Server-based dispatching is usually done by web server connectors (e.g. mod_jk and mod_proxy) which have built-in scheduling algorithms to distribute requests evenly. Of course, load balancing can also be done by using distributed computing models like Java RMI, CORBA and Servlet's forward ability. However, the problem is that most application servers just support them and leave the usage of these instruments to application developers who need to master these technologies rather than working on their business logic programming.

1.2.2 High Availability

Service availability is more difficult to achieve as it needs to consider the different scenarios that could be resulted when a server failure does occur. We may need to take care of data integrity issues if a client session is broken at the meantime. Some mechanisms periodically save active client session objects to a shared database or file system to let other server instances take up the request when one of them crashes. A more direct approach is to replicate session data of every request to the memory of one or more servers by some messaging services. However, scalability will be a great concern here because replicating sessions may involve intensive object serializations.

We will go through the existing clustering solutions with more technical details in Chapter 2. The major drawbacks of most of the general approaches are limited scalability, lack of transparency and sometimes interoperability.

1.3 Our Approach

1.3.1 Distributed Java Virtual Machine (DJVM)

A Distributed Java Virtual Machine (DJVM) is a cluster-wide virtual machine (i.e. a group of cooperative JVMs) that supports the parallel execution of threads inside a multithreaded Java application with single-system image (SSI) illusion on clusters [3]. In this way, the multithreaded Java application runs on a cluster as if it ran on a single machine with improved computation power. A DJVM inherits Java's portability and hence provides a more portable and more user-friendly parallel environment than many other existing parallel programming languages such as MPI.

Our approach makes use of the multithreading feature of a Java application server to perform load balancing. Threads inside the server are distributed to the JVMs over the cluster in a direction to balance the workload due to the incoming requests.

1.3.2 Advantages of Using DJVM

DJVM is a novel approach in server-side load balancing. Depending on the DJVM design, a number of possible advantages can be offered:

1. **Transparent Clustering:**

   DJVM has already taken care of the clustering aspect so that web developers can cluster their applications with virtually no coding or setup effort. In contrast, many clustering facilities shipped with application servers require complicated setup, configuration and performance tuning to achieve a targeted scalability.

   Also, DJVM can allow JVM instances to join at runtime to scale up performance without reconfiguration and service interruption. In mission-critical applications, one-minute down time can lead to very serious impacts. General approaches, however, usually require change of configuration files and server restart which could be risky to service availability.

   DJVM is basically a shared memory programming paradigm. Therefore session data objects can be shared transparently among all server instances. If one cluster node fails, other nodes can use their cached copies to serve subsequent requests belonging to the same session.

2. **Better Speedup and Scalability:**

   DJVM is by nature a good infrastructure to scalability. First, clustering at JVM level should be faster than at server level or application level using technologies like RMI and CORBA because it is closer to the machine code level.

   Secondly, DJVM could achieve more dynamic load balancing in the following sense. Consider the case of round-robin web server connectors. If the processing time of each request is largely uneven, then some nodes could be overloaded with long running threads while some are idle for their threads have finished processing the assigned short-lived requests. There is no way to retune the workload after requests have been dispatch. However, this unbalanced situation can be avoided by DJVM. If the DJVM has dynamic thread migration ability like the case of our JESSICA2, then workload can be readjusted by moving out some intensive threads to the idle nodes. Besides resulting in better speedup, this also suggests that DJVM can provide a more suitable runtime environment to support irregularly structured applications on server platforms.

3. **Cooperative Caching Support:**

   Cooperative caching makes use of the remote memory of other cluster nodes to avoid excessive disk accesses. Consider the current overhead ratio of over 10:1 for a 4-KB page fetch via disk access and via Fast Ethernet; the cached objects in a DJVM can be utilized to maximize web content affinity and greatly improve server performance.

1.3.3 This Dissertation Work

To testify the benefits of the DJVM approach, we made a great effort to port the popular application server Apache Tomcat (version 3.2.4) onto our DJVM prototype called JESSICA2. In effect, we arrive at a Java application server clustered at the JVM level. Application benchmarking and performance analyses were carried out to investigate the scalability and the possible bottlenecks in the overall system. Finally, we tried to compare the speedups obtained by web server dispatching and our system.

Our work is important because a successful outcome will not only help to solve web application server scalability problems in a transparent manner but also motivate the public acceptance of DJVM systems in wider application areas. However, we have been facing great technical challenges. Porting a full-fledged application server onto a DJVM prototype is highly difficult because of the huge rift in their design goals and software robustness. JESSICA2 DJVM was designed to support compute-intensive applications mostly in the scientific area. Its implementation was not optimized for I/O intensive server-side applications. Therefore careful analyses and modifications are necessary to address the system-wide conflicts between the two kinds of systems.

1.4 Organization of this Dissertation

The rest of this dissertation is organized as follows. Chapter 2 reviews the previous work related to Tomcat clustering and other DJVM systems which can support web applications. Chapter 3 gives a study on the architecture and characteristics of Tomcat and JESSICA2, followed by their integration. Chapter 4 explains the implementation details of how the two systems are integrated and enhanced. Chapter 5 presents the experimental results obtained from performance evaluation on our system. Chapter 6 discusses the various issues with the DJVM approach. Finally, in Chapter 7, we would draw a few conclusions from our findings and suggest possible future work.

Chapter 2. Related Work

In this chapter, we will first have an overview on existing Tomcat clustering solutions. Secondly, we will also review a few clustered JVM systems that have been used to run server-side applications.

2.1 Tomcat Clustering

2.1.1 Load Balancing

Tomcat and Apache web server are usual coworkers to support load balancing for a web site. Actually, the load balancing function is not handled by Apache itself but by the server connectors pluggable to it. There are many connector implementations such as mod_proxy, mod_jk, mod_rewrite and mod_backhand. Besides basic scheduling algorithms, some of them can support "sticky sessions" which means they can memorize which node a request was dispatched to and subsequent requests belonging to the same session will be assigned to the same node for attaining *session affinity* and hence a good cache hit rate.

2.1.2 In-memory Session Replication

Service availability in Tomcat is achieved by *session replication*. There are many implementations of session replication. It can be done by using a shared database or shared file system to make sessions available to other server instances. However this approach is not scalable. A more popular and better alternative is to use *in-memory* session replication which relies on a specific messaging protocol or a distributed shared memory for sharing session objects across the cluster. Several implementations of this approach are introduced as follows.

2.1.2.1 Tomcat Clustering by JavaGroups

JavaGroups is a Java-based toolkit for reliable group communication. It can ensure each group member receives the same sequence of messages in the same well-defined order. [15] built an in-memory session replication plug-in for Tomcat 4 based on JavaGroups. However, in order for the replication to work correctly, any attribute value that is stored in the session has to implement the java.io.Serializable interface. Object serialization poses great impact on the scalability in this kind of system.

2.1.2.2 Tomcat Clustering by JavaSpaces

JavaSpaces is a core Jini service and can be used to design a clustering solution in the distributed shared memory model. [12] proposed a space paradigm approach by which a request is fulfilled by having an object move from one machine to another, carrying with it the present state of execution and everything else needed, including the bytecode, if needed, using an associative, distributed, shared memory. Figure 2.1 shows the architecture of such a system. The Cluster Server Connector receives the requests from the clients, and the Cluster Server Processor encapsulates the requests into RequestEntry objects and writes them into the JavaSpace. The Cluster Worker Connector then takes these requests from the space and passes to the Cluster Worker Processor to fulfill them. Load balancing, request-level and session-level failovers are

naturally supported by this approach. In fact, this architecture has some similarity with our final-version system in that wrapped request objects are passed to workers through a shared memory space. However, their work is modified on Tomcat 4 and there is no experimental results published, so we cannot compare with this system.

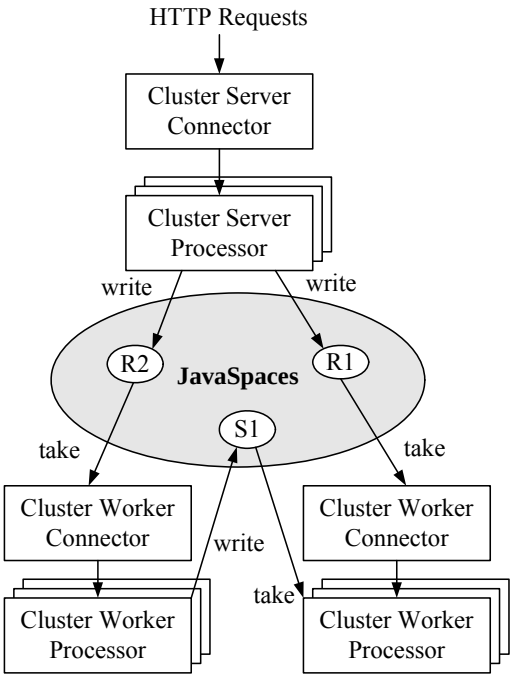

Figure 2-1: Architecture of Tomcat 4 clustering solution by JavaSpaces

2.1.2.3 Tomcat 5 Built-in Clustering

Starting from Tomcat 5 series, built-in clustering is provided by a proprietary protocol. Developers can use the SimpleTcpCluster and SimpleTcpClusterManager classes that are shipped with Tomcat 5 installation. Session replication in the current version is an all-to-all replication of session state, meaning the session attributes are propagated to all cluster members all the time. This algorithm is only efficient when the clusters are small. For large clusters, the next Tomcat release will support primary-secondary session replication, where the session will only be stored at one or maybe two backup servers.

2.2 Related Clustered JVMs

There are a number of DJVM or clustered JVM projects pioneered by different institutes and universities. Below is a list of these works including our JESSICA2.

- Java/DSM, Rice, 1997
- JavaParty, University of Karlsruhe, 1997
- cJVM, IBM Haifa, 1999
- Jackal, Vrije University, 2000
- Hyperion, ENS Lyon, U. New Hampshire, 2000
- JSDM, Tokyo Institute of Technology, 2001
- Kaffemik, Trinity College, Dublin, INRIA, 2001
- J/Orchestra, Georgia Tech, 2002
- JESSICA2, University of Hong Kong, 2002
- dJVM, Australian National University, 2002
- JavaSplit, IBM Haifa, Israel Inst. of Tech, 2003
- Terracotta Clustered JVM, Terracotta, Inc., 2006

However, most of these JVM systems are still at the research stage and far from being applied to support server-side applications except Terracotta Clustered JVM which will be discussed below. Another work which may be relevant to ours is cJVM which has successfully run a Java server application benchmark with proven scalability. We will review these two JVM systems and try to compare them with JESSICA2.

2.2.1 Terracotta Clustered JVM

Terracotta Clustered JVM has emerged on the market not far ago as of this writing. It is believed to be the only one production-ready clustered JVM up to now that can support realistic application servers like Weblogic and Tomcat. It ships with several packages namely Terracotta DSO, JDBC and Sessions for clustering web applications. Figure 2.2 depicts their system architecture. Terracotta relies on a centralized server connected to all the clustered JVMs to replicate state across application servers.

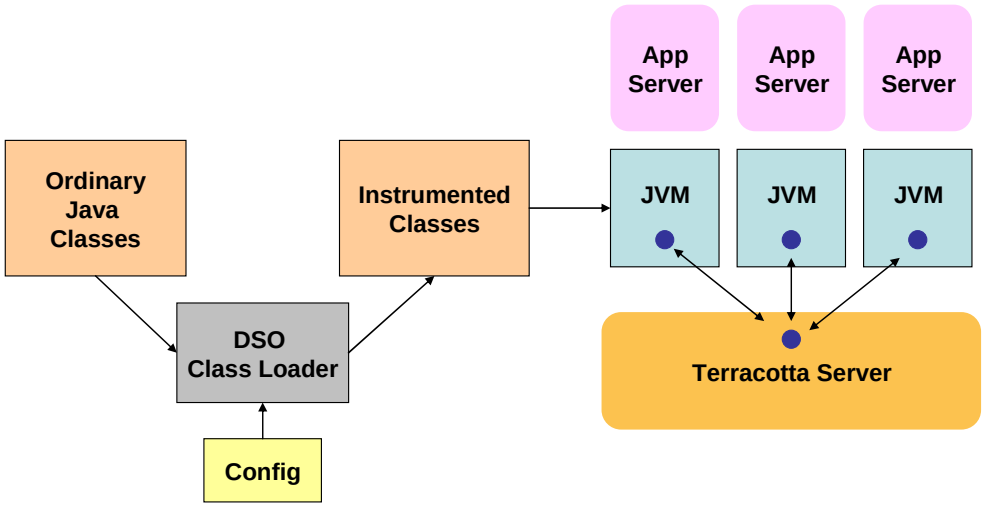

Figure 2-2: Architecture of Terracotta Cluster JVM

Terracotta DSO (Distributed Shared Objects)

Terracotta DSO is the core technology for clustering JVMs. It works by bytecode instrumentation of application classes [17]. Hooks are injected to the bytecode when Java classes are loaded in the JVM. These hooks detect at runtime field changes on object instances of the wanted classes (by user configuration) and generate messages to central server to update the replicated state of the object. Likewise, the hooks may receive asynchronous but transactional notifications from the server when other nodes perform update to the shared objects.

Terracotta is doing exactly what we present in this dissertation - JVM-level clustering approach that can help web developers to transparently cluster their web applications. However, complete transparency is still not fully supported in some cases. For example, nested lock cannot be used.

We could compare Terracotta with JESSICA2 in some aspects:

- Terracotta replicates and ensures consistency of only those user-specified objects. On the other hand, JESSICA2 applies the consistency protocol to all objects which are accessed and cached remotely. Thus, in terms of clustering efficiency, JESSICA2 could lag behind. However, in terms of transparency, we do better since we do not require users to configure the concurrency control semantics in the distributed context as in Terracotta.
- Terracotta uses bytecode instrumentation at the time of class loading to insert hooks to check and synchronize object states. JESSICA2 uses JIT compiler to generate native code for object state checks; also object header is extended to distinguish master/cached copies and to maintain the state of object.
- Both systems employed weak Java Memory Model (JMM) that resembles lazy release consistency, write updates only propagate at memory boundaries, i.e. lock/unlock. So both systems should run practically fast. However, JESSICA2 implemented various adaptive optimization techniques that can save or aggregate messages further.
- Terracotta does not provide load balancing solutions while JESSICA2 achieves dynamic load balancing by thread migration.
- Terracotta is not fully SSI-compliant (e.g. it does not have a global I/O space); JESSICA2 implements SSI extensively (e.g. it has I/O redirection features).
- Terracotta uses a centralized approach for memory consistency while JESSICA2 does it in a more distributed manner. For example, the Terracotta Server is always the lock manager of all concerned objects for enforcing concurrency restrictions and it communicates with hooks in the shared objects. In JESSICA2, the lock manager of an object is the JVM that owns the master copy of the object. Likewise, updates are flushed to the centralized server in Terracotta while in JESSICA2, updates are flushed to the object homes which can be different machines. Theoretically, JESSICA2 should see less bottleneck issues.
- Class library of Terracotta is already JDK 1.5; however JESSICA2 is still mainly of JDK 1.1 (and some 1.2 classes). Therefore, JESSICA2 supports up to Tomcat 3.2.4 only but Terracotta can run Tomcat 5 and other application servers.

2.2.2 cJVM

cJVM is a quite early clustered JVM project held by the IBM Haifa Research Labs. Its purpose is to enable large multithreaded Java server applications such as Jigsaw to run transparently on a cluster and to leverage the full power of a cluster, attaining high scalability [9].

First of all, threads are distributed in a load balancing direction over the cluster nodes when they are created. This will offer the application with enlarged computing power. Secondly, cJVM applies a *master-proxy* object model and a technique called *method shipping* to support transparent object accesses by the distributed threads. In cJVM, when an object is created in a node, the object is called the master object. Other nodes can remotely access the object via a proxy object which is created in their own heaps. Field access and method invocation of the proxy object will be shipped to the node where the master object resides for execution. In effect, all heaps cooperate to present a universal heap to the threads and no cache consistency issue is involved.

cJVM achieves about 80% efficiency on 4 nodes connected by Myrinet for the pBOB application benchmark (its modified version was adopted as SPECjbb2000). However, this cannot fully reflect the scalability. SPECjbb2000 tends to be much simpler than a full-fledged application server like Tomcat. Also it does not support JIT compilation mode, limiting its practical use in high performance server applications.

We could compare cJVM with JESSICA2 in certain aspects below:

- Both cJVM and JESSICA2 support thread initial placement. cJVM places threads by a dynamic load balancing function while JESSICA2 simply does it in a round-robin manner.
- JESSICA2 supports JIT compilation, dynamic thread migration and single I/O space support which are all absent in cJVM's implementation.
- cJVM's master-proxy model fixes the location of the master objects while JESSICA2 uses an adaptive object home migration protocol.

Chapter 3. System Analysis and Integration

In order to cluster Tomcat over our JESSICA2 DJVM in an effective way, we must correctly recognize their system architectures and runtime characteristics, followed by proper fixes on their possible misfits in the integrated system. In this chapter, we will first give a detailed study on Tomcat and JESSICA2. Then we will illustrate how their integrated version operates in a cluster-wide JVM environment.

3.1 Apache Tomcat

3.1.1 Overview

Apache Tomcat is the official reference implementation for the Java Servlet and JavaServer Page (JSP) technologies [18]. Various surveys reveal Tomcat is the most widely used open-source servlet engine and has been downloaded more than 10 million times [19], showing its popularity in the web community. Tomcat currently has three version series from 3 to 5. Tomcat 4 has switched to a new servlet engine core, namely Catalina, which has a very different architecture and threadpool design from version 3. We can only support up to Tomcat 3.2.4 due to limited class library of JESSICA2. Also, Tomcat is 100% pure Java which is a crucial requirement for us to realize its Java thread migration using JESSICA2.

Tomcat itself is a multithreaded application. Multithreading helps filling up processor idle time in the event of I/O blocking and is suitable for I/O intensive applications like Tomcat. When the current thread blocks on I/O, another thread can be scheduled to process other requests at a little cost of thread context switching.

3.1.2 System Architecture

The overall architecture of Tomcat 3.2.4 is depicted in Figure 3.1 below.

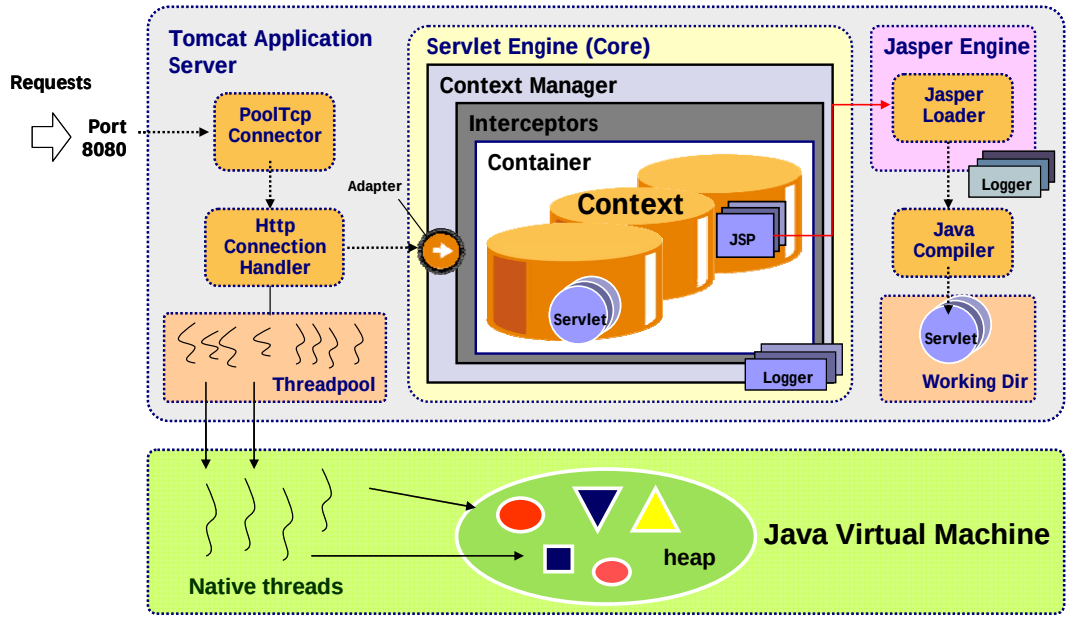

Figure 3-1: Architecture of Tomcat 3 Application Server

Tomcat consists of two major subsystems:

1. **Servlet Engine:** This is the main subsystem of Tomcat implementing the core semantics for servlet handling. It consists of many well-defined components responsible for different internal tasks which include locating the servlet context, loading servlet classes and calling the servlet's service() method to fulfill an incoming servlet request.

2. **Jasper Engine:** This part is the implementation of the JSP specification. A JSP is an HTML page with embedded java code, which can be compiled on demand when the page is requested. In short, Jasper engine is responsible for parsing the body of the page (through the Jasper loader) and compiling it into a servlet class. Then the execution follows as if a normal servlet request is received. The compilation cost is one-off, only happening at the first visit of the page. So JSP technology generally runs faster over interpreter languages like Perl.

Inside Tomcat, a threadpool is created for serving requests. The use of threadpool has two great advantages: (1) it effectively reduces the expensive cost of spawning a new Java thread for each request which might be short-lived; (2) it puts a control on the system resources so that the system will not be overloaded by too many simultaneous requests. Each thread in the pool is listening to a common socket. When a connection arrives, a thread will wake up from blocked state and call an handler to start the servlet processing inside the context manager. When there are simultaneous requests more than the number of threads in the pool, a new thread can be created. If the maximum number of threads defined in Tomcat is reached, no more connections will be accepted and are lined up in the backlog queue.

For a deeper account on the major components and their functionalities in Tomcat 3 servlet engine, please refer to Table A.1 in the Appendices. Our focus here is on the Tomcat runtime behavior so that we can think of tailoring to support it on JESSICA2.

3.1.3 Flow of Operations

We would present some more low-level description on what has happened inside Tomcat when a request is being processed (Table A.1 may be useful here). Tomcat serves the incoming requests in a multithreaded manner as follows:

- When a socket connection is accepted (done inside the PoolTcpEndpoint object), the processConnection() method of HttpConnectionHandler will be called.
- Depending on the "reuse" flag, the processConnection call will either create a new HttpRequestAdapter object or arbitrarily take a previously created object from the HttpRequestAdapter pool, for wrapping up the socket connection. A HttpResponseAdapter will be created in a similar manner for this request.
- Then the readNextRequest() method of the HttpRequestAdapter object is called to read the HTTP request line and HTTP headers. The request URL is parsed accordingly.
- Next, the service() method of the ContextManger will be called with the pair of HttpRequestAdapter and HttpResponseAdapter passed to it.

- The context manager will go through a chain of interceptors (hooks) such as AccessInterceptor, SimpleMapper1 and SessionInterceptor on the request which perform functions like authentication, context mapping and session handling.
- Then the request will enter the core of the servlet engine. A ServletWrapper will be called to locate the requested servlet. The servlet class will be loaded and initialized (at the first time).
- Then the doService() inside the ServletWrapper will be called, which invokes the requested servlet's service() method
- The doGet() or doPost() method implemented by the application servlet is called accordingly.
- The application servlet's business logic now executes, it may call getParameter(), getSession(), getAttribute(), etc to read the data in the request and the session, if any. It can also call setAttribute() to write data to the session.
- When the servlet business logic completes, any result could be written to the output stream of the HttpServletResponse which is bound to the accepted socket.

The processing flow presented here looks complicated already but indeed it has been much simplified from the actual details. The request processing calls to the inner core layer by layer passing through connector, handler, context manager, interceptors, container, context and lastly reaching the servlet. Therefore, we could expect the stack of Tomcat threads will not be small. Many java frames are put on the stack because of this interceptor-based (or hook-based) design of Tomcat.

3.1.4 System Characteristics

In this section, we would highlight some runtime characteristics of Tomcat. Later, we will see all these could have serious impacts on compatibility and performance when running on top of JESSICA2.

3.2.3.1 Synchronized Blocks in Tomcat

Tomcat's internal has quite many synchronized blocks of code. This is mainly because Tomcat will need to access various shared objects in each request processing cycle. We summarize below three critical sections in Tomcat's processing.

Threadpool Entering

Before a connection can be accepted, Tomcat needs to acquire an idle thread from the shared threadpool. Being more specific, it needs to get a ControlRunnable object from the ThreadPool vector via some synchronized methods. ControlRunnable is linked to a Thread object which has started the run() method but is waiting for notification to be up to service. Threadpool synchronization overhead is on per-request basis.

Session Management

If a servlet application needs to use sessions, the following methods will be called:

- In StandardManager: findSessions(), getNewSession()
- In SessionIdGenerator: getIdentifier(), generateID()

These methods are also synchronized because Tomcat uses a single hash table to store session objects which is shared by all threads. Depending on the web application, session handling can be a per-request cost.

Extensive Use of Object Pooling

Tomcat applies the object pooling technique to minimize the overhead of creating short-lived objects and result in less garbage collections. Below is a list of some frequently accessed object pools in Tomcat 3:

- **Connection cache**: a pool of TcpConnection objects used by PoolTcpEndpoint
- **HttpRequestAdapter pool**: used by HttpConnectionHandler
- **Recycled sessions vector**: a pool caching expired session objects which can be reused by the StandardManager for saving new session creations.

Object pools are again shared and accessed by all threads via pool locking and unlocking. In a single JVM runtime, synchronization cost is not apparent. Tomcat can enjoy great performance speedup from object pooling which is indeed a common server optimization technique.

3.2.3.2 Large Number of Objects

Figure 3.2 shows the startup memory footprint of Tomcat provided by GNU project debugger. It can be seen that there were over 200 thousands objects created in the heap when Tomcat is just brought up. This number appears to be huge to us. But it is in fact very common in enterprise-scale application servers. Some of them, e.g. IBM WebSphere have an even larger memory footprint that require a high-configuration machine for smooth running. Tomcat belongs to a kind of server on enterprise scale.

We can imagine when threads in Tomcat are distributed over the cluster, there will be large amount of remote object accesses requested from workers because most of the objects have been created in the master JVM. Also object state checking overhead would become enormous. This is a pressed challenge to our JESSICA2 prototype.

Memory statistics

j.l.String:	Nr	15648	Mem	733K	other-nowalk:	Nr	66	Mem	30K
obj-no-final:	Nr	**143372**	Mem	5626K	**prim-arrays:**	Nr	**8999**	Mem	582K
ref-arrays:	Nr	**3907**	Mem	877K	**j.l.Class:**	Nr	**464**	Mem	72K
obj-final:	Nr	249	Mem	10K	java-bytecode:	Nr	3051	Mem	180K
exc-table:	Nr	277	Mem	17K	jitcode:	Nr	1468	Mem	975K
static-data:	Nr	147	Mem	4K	constants:	Nr	429	Mem	328K
other-fixed:	Nr	22015	Mem	1613K	dtable:	Nr	398	Mem	33K
methods:	Nr	425	Mem	544K	fields:	Nr	270	Mem	31K
utf8consts:	Nr	7114	Mem	366K	interfaces:	Nr	199	Mem	3K
locks:	Nr	0	Mem	0K	thread-ctxts:	Nr	74	Mem	1114K
gc-refs:	Nr	573	Mem	13K	jit-temp-data:	Nr	12	Mem	379K

Figure 3-2: Memory footprint of Tomcat 3 in a single-node JVM

3.2.3.3 Routine Daemon Threads

In Tomcat, besides the threads for request handling, there are also daemon threads for some routine tasks. These threads are scheduled to wake up from sleep regularly at a predefined interval. Below are the examples describing their natures.

- **LogDaemon (Flusher)**

TomcatLogger is a component responsible for logging server runtime information such as failed requests. LogDaemon is a thread running the TomcatLogger class to look into a queue and will writes out everything of there to the sink (e.g. log file).

- **MonitorRunnable Daemon**

This daemon monitors and cleans up the threadpool from too many spare threads at regular interval (default per minute). If the idle thread count in the pool is greater then the defined maximum spare thread count, then the excessive threads which were spawned at high server load will be terminated.

- **StandardManger Demaons**

Tomcat gives each context (i.e. web application) a StandardManager daemon for session management. These background threads will reap old session data (or put them to the recycling vector) from the shared hash table.

3.2 JESSICA2

3.2.1 Overview

JESSICA2 (Java-Enabled Single-System-Image Computing Architecture version 2) is a distributed Java Virtual Machine (DJVM) developed at the University of Hong Kong. It is designed to support parallel execution of multithreaded Java applications over a cluster. With JESSICA2, a single Java program can span over multiple machines, and enjoy the combined computing power, memory and I/O capacity, as if it is running on a single powerful machine.

JESSICA2 is the first DJVM featuring a lightweight Java thread migration mechanism operating at Just-in-time (JIT) compilation mode. By this sound feature, Java threads can freely move across node boundaries to make better use of computing resources. JESSICA2 also offers user-friendly transparent clustering that requires no source code modification and bytecode preprocessing. It will automatically take care of data consistency of the shared objects, thread distribution and I/O redirection so that the program will see a single-system image (SSI).

JESSICA2 was developed from Kaffe JVM 1.0.6 (class library JDK 1.1 to 1.2). It is a proven successful implementation that achieves scalable speedup in most scientific benchmarking experiments. The success is attributable to the many advanced features and optimizations all over the system that are to be explained.

In the coming sections, we will review the system architecture and main features of JESSICA2. We will try to make it concise here since our focus is to explore its useful aspects and possible limitations for supporting Tomcat.

3.2.2 System Architecture

The overall architecture of JESSICA2 is depicted in Figure 3.3 below.

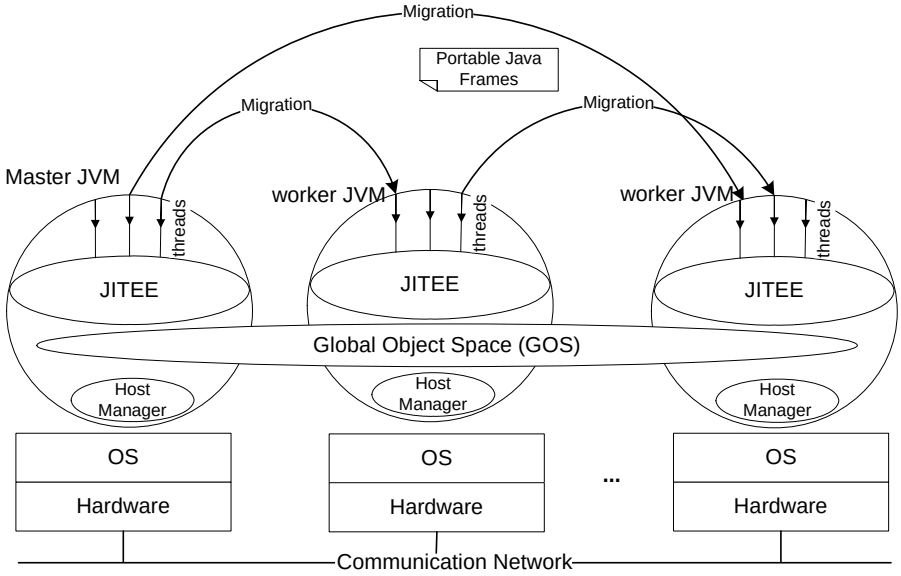

Figure 3-3: Architecture of JESSICA2

In JESSICA2, the node where the application starts is called the *master node*. All the other nodes in the cluster can join as *worker nodes* by starting JVM processes which connect to the port open at the master JVM. In effect, a number of nodes participate in the execution of a Java application. The Java threads in the application will be initially placed among the nodes in a round-robin manner. Threads can also be dynamically migrated from one node to another if a thread scheduler receives an migration request from the load monitoring daemon in the JVM.

Each thread of the application runs in a JIT execution engine (JITEE) which is extended to instrument the Java thread stacks on request for migration, and also to add checking code for maintaining global object accesses.

The *Global Object Space* (GOS) layer is embedded inside the JVM to make the shared objects created by the Java application visible to all the threads running on different machines. There is also an I/O server daemon resided in the master JVM for processing I/O redirection requests sent from the workers.

The *host manager* is a daemon running inside each JVM that manages the cluster nodes and provides TCP-based communication supports for the GOS layer and I/O redirections.

3.2.3 Main Features

3.2.3.1 Transparent Java Thread Migration

To support thread migration, we need some mechanisms to capture the thread's execution state and restore it onto the target machine. In an JIT-enabled environment, Java threads are running in a native context, we call it a *raw thread context* (RTC), which is usually unrecognizable an another machine. JESSICA2 designs a *bytecode-oriented thread context* (BTC) for portable thread migration. BTC is derived from the RTC of a suspended thread for migration and is sent to the target machine to aid the thread state restoration. BTC-RTC transformation however faces two challenges: (1) the native PC in the RTC may situate at the middle of the native code block compiled from a bytecode instruction. (2) the types of the stack variables can only be known at runtime. There are two mechanisms employed in JESSICA2 to overcome them:

- **Dynamic Native Code Instrumentation (DNCI)**: instrument lightweight native code to support RTC-BTC transformation when a Java method is first compiled by the JIT compiler during execution. Migration points are added between some bytecode boundaries chosen by heuristics, e.g. before a method call or a loop. At these points, register and stack variable type spilling back to memory are done.

- **JIT Recompilation (JITR)**: re-run the JIT compiler, trace the steps of the compiler to the thread stop points, and collect the bytecode PC, the stack pointer, the operand types and values during the recompilation. The complete process of this mechanism consists of totally seven steps: stack walk, frame segmentation, bytecode PC positioning, breakpoint selection, type derivation, translation, and native code patching. Their detailed explanations can be found in [4].

JESSICA2 uses JITR by default because it charges instrumentation cost only when migration does occur. Therefore, JESSICA2 runs at full speed most of the time during execution. DNCI is suitable for irregular applications that make frequent migrations.

3.2.3.2 Global Object Space (GOS)

When threads move to different machines, they see different memory spaces. The Global Object Space (GOS) layer in JESSICA2 leverages a software DSM-like service to support remote object accesses from all threads across different nodes. Using page-based DSM systems to support distributed object sharing in Java would suffer from serious false-sharing, so JESSICA2 extends the heap in JVM to enforce object access states through software checks. This design also allows the GOS to exploit the runtime information in the JVM kernel to reduce communication costs.

Figure 3.4 below shows the overall structure of the GOS. The heap in each JVM is logically divided into two areas, namely the *master heap area* and the *cache heap area*. The master heap area is storing ordinary Java objects when they are first created in the heap. When they are accessed by some remote threads, cached copies will be left in the cache heap area in the remote JVMs to reduce unnecessary network traffic caused by subsequent accesses. The cached object is similar to the original object (we called it the master object) except that it has different flags like status and timestamp

in the object header for maintaining its consistency. Each thread is logically given a private area in the cache heap and a hash table for quick lookup of its cached objects.

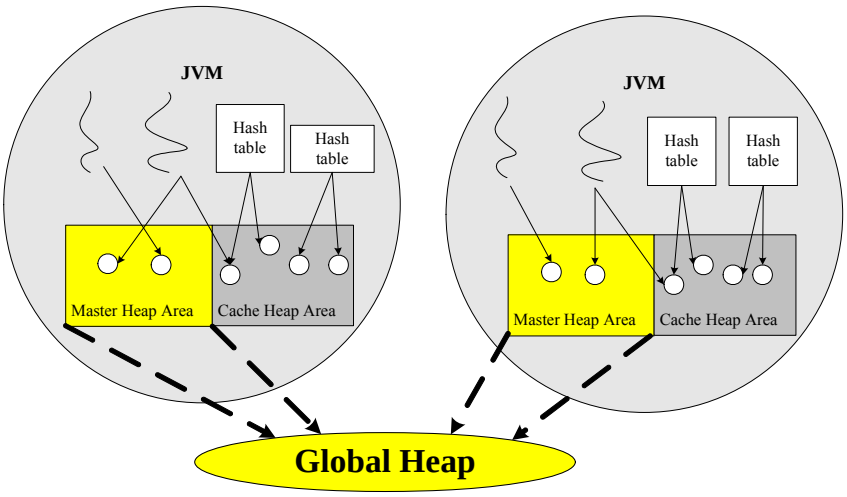

Figure 3-4: Overview of the GOS distributed object heap

Figure 3.5 shows the more detailed internal data structures which are used to organize the cached objects. When a new object is to be cached, a cache header will be created for the object and is indexed through the hash table of the caching thread. The cache header is shared by other threads in the node if they also want to cache the same object. But they will keep their own private copies of the cache object. The JVM internal representation of Java thread is also extended to carry a list of "host caches" which is designed to speed up the search of objects for flushing.

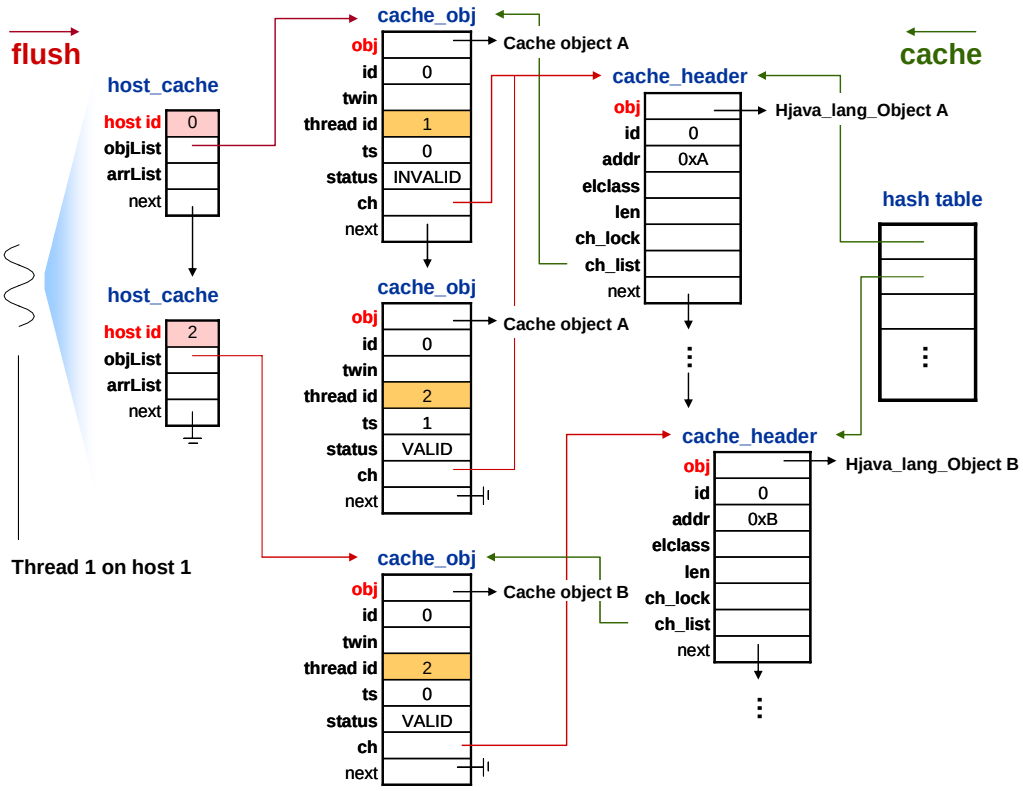

Figure 3-5: Internal data structures of cache area in the GOS

Synchronization – Cluster-wide JMM

The cache consistency protocol in the GOS is said to be a cluster-wide Java Memory Model (JMM) which can be visualized in Figure 3.6. In JMM, a shared object access is protected by a synchronized block. When entering a monitor (i.e. lock), it needs to flush all objects cached by the current thread. The flush operation is to invalidate all the cached objects and to write back diff of any dirty objects to homes. Home nodes will apply the diff updates the master objects. Later, when the thread uses an invalidated cached object, it will fault-in the most up-to-date copy from the home node and the cached copy becomes valid again. After the thread completes its job and exits the monitor (i.e. unlock), it must flush (write back diff) all dirty objects back to the homes for update. (Note: Invalid object access can fault in the fresh copy because we tweak the bytecode `GETFIELD`, `GETSTATIC`, `AALOAD` etc to call our GOS interface function to contact the home node.)

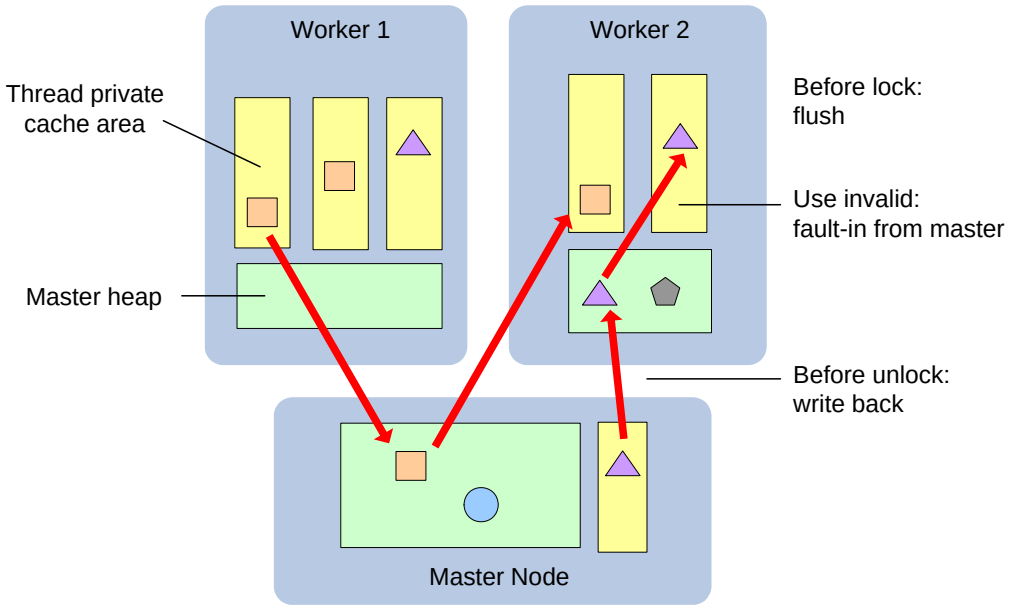

Figure 3-6: Cluster-wide Java Memory Model (JMM)

Lastly, there are three optimization techniques implemented in the GOS to make it run more efficiently, namely:

- **Adaptive object home migration**: This is a featured technique which detects a dominant writer in the cluster on an shared object and migrates the object home to it so as to eliminate excessive remote write backs.
- **Object pushing**: This is a pre-fetching technique that exploits the connectivity of a Java object. It scans through the field definitions in an object and aggregates also the objects it refers into one message to save communications.
- **Fast object state checking**: This is to use the JIT compiler to generate native code for the object state checking instead of simply directing it to the GOS interface functions.

3.2.3.3 Global I/O Redirection

I/O redirection provides a SSI view for Java threads all over the cluster to perform I/O operations as if they were running on a single JVM. Java I/O library in JESSICA2 is extended to fulfill this SSI requirement. The modifications include file and network I/O. The internal file handle in the JVM is extended to use the first half word denotes the host id of the machine where the file is first opened (we also call it master node). Remote read/write operations will be redirected to the node with host id extracted from the file handle. The I/O server daemon inside the master JVM is responsible for handling the redirection requests. A new daemon thread will be spawn to process socket connect, accept, read and datagram receive requests to avoid blocking.

Some strategies are used to save redirection cost. A read-only open operation of file system I/O first checks the local disk before redirecting to the master node. Other file I/O operations will always be redirected. For network I/O operations, connectionless open (such as UDP) will be done locally. The other operations such as TCP open will be redirected to the master JVM.

3.3 System Integration

In this part, we will present how Tomcat is running over JESSICA2. We will also point out some performance issues of the integrated server due to the mismatched runtime characteristics of both systems.

3.3.1 Tomcat-on-JESSICA2 Architecture

Figure 3.7 shows the overall architecture of the Tomcat-JESSICA2 application server. Basically it is of no architectural difference from Tomcat running on an ordinary JVM. But with JESSICA2, Java threads in Tomcat have been mapping to native threads created in all the participating JVMs.

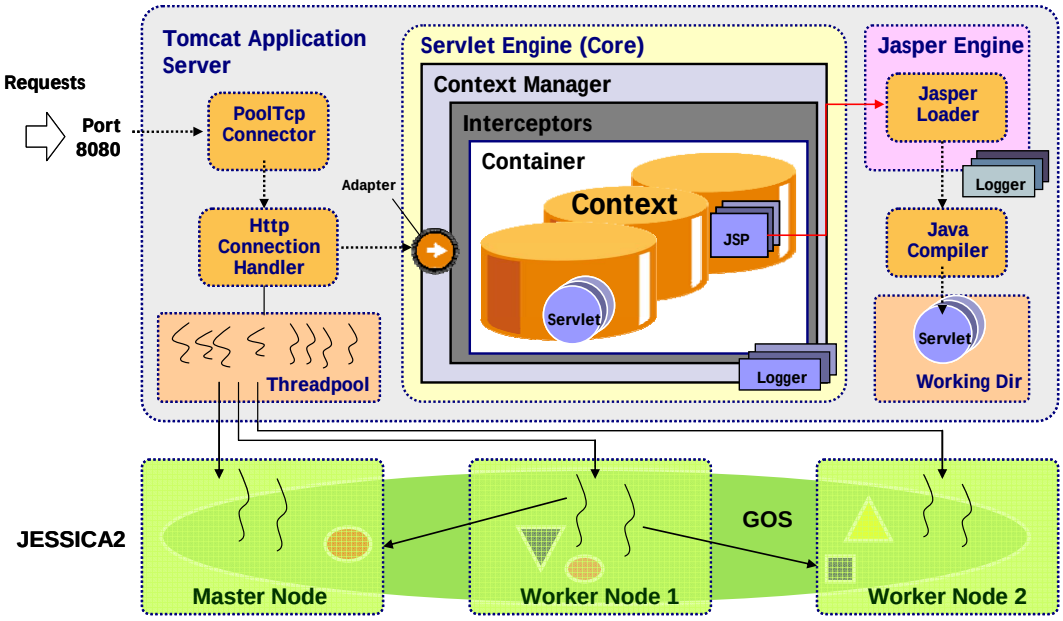

Figure 3-7: Architecture of Tomcat-on-JESSICA2 application server

3.3.2 Cluster-wide Tomcat Operations

Tomcat is now running in a distributed manner, utilizing the thread migration, global object space and I/O redirection features provided by JESSICA2. We will elaborate how these mechanisms are actually used by Tomcat below.

3.3.2.1 Distributed Thread Running

Figure 3.8 shows a typical Java thread stack of a request processing thread in Tomcat. Since Tomcat is not linked to any native library, the whole stack of Java frames can be migrated to worker nodes through JIT recompilation. Figure 3.9 shows the code snippet of the TcpWorkerThread.runIt() method which calls endpoint.acceptSocket(). Therefore, after migration, the thread will accept incoming socket connections in a remote manner via I/O redirections back to the master JVM.

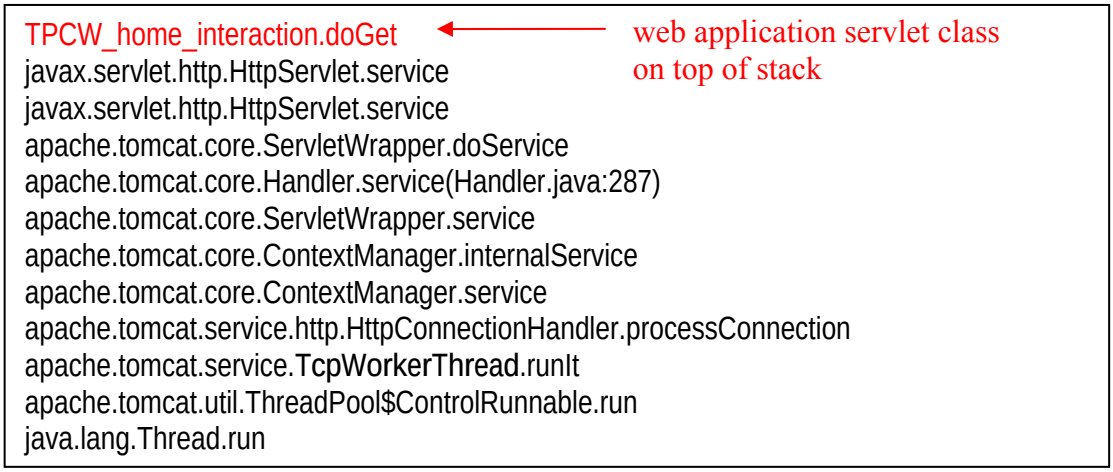

Figure 3-8: A typical thread stack of a Tomcat request servicing thread

```
while(endpoint.running) {
    Socket s = endpoint.acceptSocket();
        if (null != s) {
            // Continue accepting on another thread...
            endpoint.tp.runIt(this);

            try {
                if( usePool ) {
                    con=(TcpConnection)connectionCache.get();
                    if( con == null )
                        con = new TcpConnection();
                                                    }
                con.setEndpoint(endpoint);
                con.setSocket(s);
                endpoint.getConnectionHandler().processConnection(con, perThrData);
            } finally {
                con.recycle();
                if( usePool && con != null ) connectionCache.put(con);
            }
            break;
        }
    }
}
```

Figure 3-9: Source code snippet of TcpWorkerThread Runnable

3.3.2.2 Shared Object Access via the GOS

Refer to Figure 3.9 again. We can see migrated threads will do get() and put() on the connectionCache object remotely via the GOS. Both are synchronized methods, that means all cached objects will be flushed at these calls, including the array used to pool all the TcpConnection objects.

Session clustering via the GOS

In particular, the GOS can achieve the effect of session replication as in the common Tomcat clustering approaches. Figure 3.10 shows a shopping cart example which helps to visualize how a session is clustered among the JVMs. When a fresh request arrives at Tomcat, a shopping cart session object will be created, say, in Master. If the next request from the client is dispatched to another thread running on Worker 1. Then the created session will be fetched to Worker 1 and remains in the thread's cache area. In this way, all cluster nodes can be able to process requests of this client through the session. This aims to achieve high-availability and load-balanced service.

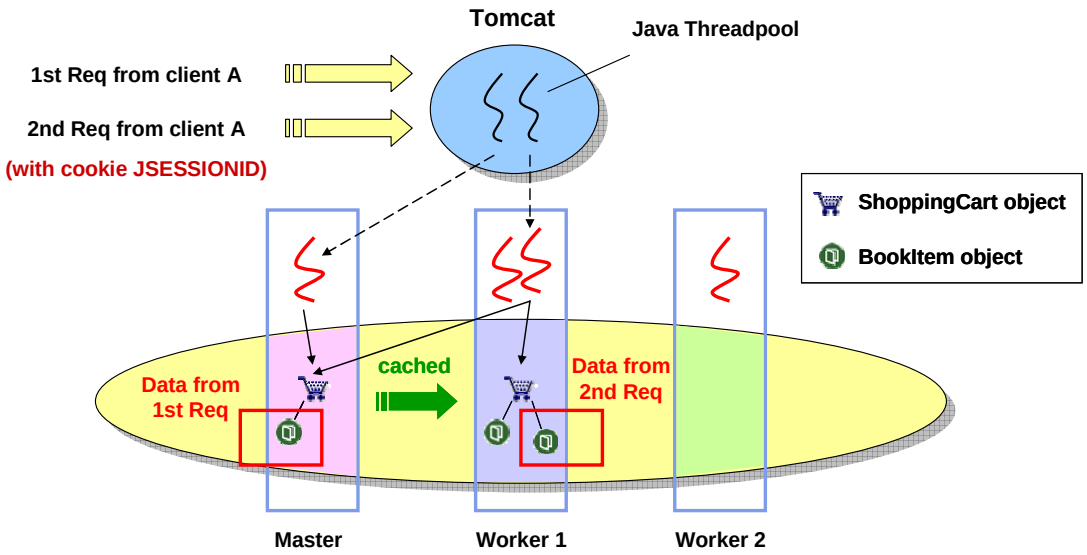

Figure 3-10: Session clustering via the GOS

We wish also to point out a batch of sessions are actually clustered on every request touching on a session. Tomcat uses a Java hash table to store sessions. The Hashtable class in Kaffe implementation is just a wrapper of HashMap that uses the Entry[] array to store the objects. In the GOS, a huge array (larger than 64K size) will not totally cached on a node. Instead, only a range of it will be cached. The cache array will have additional fields in its header to define the accessible range. That means some elements in Entry[] will go to a worker at each synchronized access of the hash table. Therefore, we are doing a coarse-grained session clustering.

3.3.2.3 Socket Read/Write via I/O Redirection

Lastly, Tomcat threads on worker nodes need to redirect most of the I/O operations back to the master since the accepted socket is opened there. Figure 3.11 below shows a sequence of redirections that will occur for a single request. In particular, the accept and read redirections sent to the master will make it spawn a separate I/O daemon.

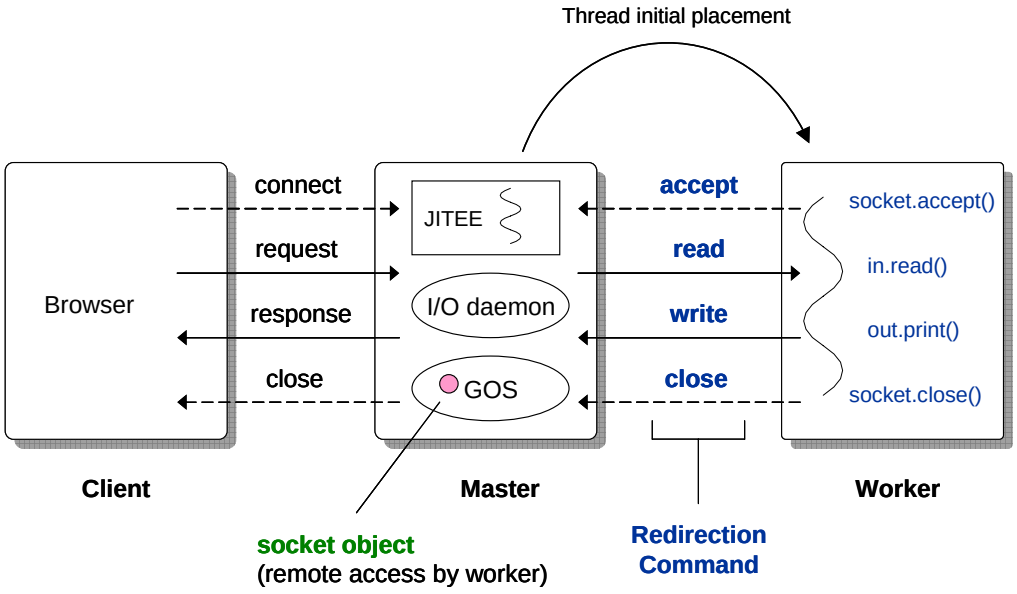

Figure 3-11: Socket handling via global I/O redirection

3.3.3 Problems of Direct Integration

After understanding the above execution manners, we may already have ideas on the resulted performance of Tomcat over JESSICA2.

3.3.3.1 Heavy Synchronization

As we have seen, due to intensive object pooling in Tomcat, synchronized resource accesses are frequent. This imposes serious impact to JESSICA2 because all objects cached by a remote running thread will need to be flushed and faulted in again. If this operation happens in each request-response cycle, then it is a large factor limiting the speedup achievable by JESSICA2. We will see in the experimental results later this is a true performance issue.

3.3.3.2 High Traffic in the GOS

Recall one characteristic of Tomcat is its huge number of objects. We have also seen a typical Tomcat thread stack is not small, if all objects being referenced in all methods on the stack need to be fetched to remote nodes, the GOS will be stressed with these traffics. Flush operation would be slow if the cache heaps become large. Another factor mounting up the GOS traffic is the transfer of large arrays of object pools and byte messages.

3.3.3.3 Master Being the I/O Bottleneck

Although our I/O redirection mechanism is good to meet SSI requirements, the master will inevitably become the I/O bottleneck. To make matter worse, intensive spawning of I/O daemon will create another performance concern. We will see not all web applications will hit this bottleneck but those with high data volume.

3.3.3.4 Undesirable Migration of Tomcat Daemons

Recall that Tomcat has several kinds of daemon threads for routine tasks. These threads are touching the threadpool, session hash table and writing log files. If they are migrated to worker nodes, they will add more remote access and synchronization overheads to the GOS as well as more I/O redirections which are undesirable. This is a limitation of JESSICA2 that it is not aware of the task nature of threads and has no idea to decide which thread should not be migrated.

3.3.3.5 Loss of Session Data Locality

Although we can do session clustering via the distributed heap, this mechanism is however not quite efficient. The primary factor is the need of transferring and maintaining many cached portions of a possibly large hash table among all cluster nodes. Secondly, since every thread may access a session in the table's Entry[] in an arbitrary or random manner, whether partial array caching can save or induce more overhead becomes a question. Finally, compared to sticky-session clustering solutions, we are in fact losing data locality because we dispatch a request to a thread without memory of which of them would have the cached session. Overhead of remote access and synchronization on the hash table will be resulted if the request is dispatched to a thread which did not cache the required session before.

3.3.4 Summary of Tomcat-JESSICA2 Overheads

- Synchronization of threadpool and object pools
- Remote object fetching in general execution
- I/O redirections and I/O daemon spawning
- Additional GOS traffic caused by migrated Tomcat daemon threads
- Fetching initialized static data from master when a worker loads a class
- Dynamic thread migration overhead (stack capturing and restoration)
- Exchange of computing resource statistics (e.g. CPU %) among cluster nodes

Combining all the above problems and overheads, the clustered version of Tomcat over JESSICA2 performs even poorer than a single-JVM Tomcat server. We need to modify Tomcat and JESSICA2 to make them have a better interfacing.

Chapter 4. Implementation

In this chapter, we will go through the implementation details of this dissertation. First, the porting methodology we adopted is briefly introduced. Next we will review some program fixes and modifications done on JESSICA2. Finally, Tomcat modifications will be explained in details.

4.1 Porting Methodology

- Do compatibility tests to find out the highest version of Tomcat that can run on Kaffe JVM 1.0.6 (Tomcat 3.2.4 is found).
- Then test the Tomcat version with JESSICA2 on a single machine.
- Repeat the test over a few number of cluster nodes.
- Fix the inherent problems of JESSICA2 to support Tomcat functionally.
- Perform stress testing on Tomcat over JESSICA2 to enhance its stability.
- Do performance analysis with various application benchmarks. Then modify and tune up Tomcat over JESSICA2.
- Do performance tests with different combinations of JESSICA2's optimizations.
- Compare the scalability with web-server based dispatching solutions.

4.2 JESSICA2 Fixes and Modifications

4.2.1 Error Fixes

Originally, JESSICA2 was unable to run Tomcat functionally although Kaffe can do. This was due to the inherent program bugs inside various parts of JESSICA2. Bug fixing is the most tedious process in this work. However, with the concerted effort of our research team members over the past 18 months, we overcome this daunting task and see a progressively stable Tomcat running on JESSICA2.

Table A-2 in Appendices lists out all the major error logs and bug fixes done on JESSICA2. We would highlight a couple of educative examples here to appreciate how errors could happen in a DJVM system.

Log 7: JSP Class Loading Problem

Tomcat has its own custom classloaders such as Webapp Classloader and Common classloader to load the applications. Different classloaders will create different name spaces for the Java classes, i.e. classes of equal name loaded by different loaders will be considered as different classes. But JESSICA2 does not support namespace in the GOS to distinguish classes loaded by different loaders, i.e. all classes will be shared by different loaders. Since the classloader for loading certain JSP applications is changed for some classes, worker nodes will fail to locate and load them.

This is a limitation of JESSICA2. To work around this problem, we skip the checking of the classloader name in the class entry lookup function in JESSICA2 so that worker JVMs can be able to look up the classes loaded by Tomcat's custom classloaders over the GOS.

Log 12: Throwable Packing Error

We found that JESSICA2 would suffer from segmentation fault when it tried to pack the class signature of the inner class LogEntry of TomcatLogger. Later we found that this happened because the GOS failed to pack a Throwable object. The root cause is tricky. Throwable has an instance field called backtrace. The Kaffe implementation of the Throwable class uses the buildStackTrace function to construct a C structure called "stackinfo" and sets it to be the backtrace field. Although it is casting as a Hjava_lang_Object pointer, it is actually not a Hjava_lang_Object structure at all and appears as an alien memory block to the GOS. Therefore segmentation fault will occur at packing this field.

This problem log reveals that an open-source class library may have certain parts of implementation that has assumed to work only in a single JVM environment. Similar problems are found in an application which uses BigInteger to do RSA encryption. The BigInteger class in Kaffe is linked to a native GNU math library (gmp) and this will cause segmentation fault in the GOS. Robust object packing over a distributed heap would need an extensive regression test on the class library which is hard to perform due to the lack of such testing tools. Our workaround for this problem is to comment out the use of the buildStackTrace function in Exception.c and Throwable.c and set backtrace to null so that the GOS can pack it without problem. However, if Java Throwable is really thrown out, its back trace along the call stack will not be able to see.

4.2.2 Modifications

Besides program fixes, we also rectify JESSICA2 to support Tomcat with better functionality and efficiency.

4.2.2.1 Apply Patches to the Java Timezone Class

The original implementation of the Java Timezone class has some inherent problems. When a Timezone object is created, it will open and read all time zone configuration files in the operating system recursively and save the data into a hash map. First, this will make Tomcat over JESSICA2 run slowly at startup because all workers need to perform this step. Another more serious problem is that Tomcat cannot be started up with a heap size larger than 64MB. This is because with a larger heap size, garbage collection does not occur and the files to open exceeds 1024 which is the maximum limit of file descriptors, fds, supported on Linux. This also results in invalid host id error which is extracted from the first half word of the wrong fd value. To solve such problems, we port a later Kaffe implementation of TimeZone and UNIXTimeZone which open the necessary time zone files only on demand.

4.2.2.2 Support MySQL JDBC and Apache SOAP

Using Tomcat alone is not sufficient to support useful applications. Thus, we ported a couple of useful Java packages - MySQL JDBC driver and Apache SOAP engine - on the Tomcat-JESSICA2 server to make it able to support database operations and web services. They need porting because the worker nodes failed to initialize some classes

in these packages. In JESSICA2, when a worker dynamically loads a class, it will contact the master to load the class too and fetch the initialized static fields from the master. However, the GOS seems unable to pack static fields with control characters and of java.lang.Class correctly, therefore the worker JVM suffers segmentation fault. Our workaround is to relax the protocol to let worker JVMs initialize the classes on their own rather than fetching the master copies.

4.2.2.3 Exclude Daemons in Thread Initial Placement

As mentioned in last chapter, it is not desirable to initially place or migrate Tomcat daemon threads to worker nodes. We modify the startThread function and make use of the daemon flag in the Hjava_lang_Thread structure to do the tweak. For a thread with daemon flag set on, the thread will always be started in the local JVM.

4.3 Tomcat Modifications

In this section, we will present the modifications done on Tomcat. We modify Tomcat with the objective to make it come up to the DJVM programming paradigm, e.g.:

- Trim down the number of synchronization blocks wherever possible.
- Minimize the number of objects accessed and updated in a synchronized block.
- Let object be created locally in threads whenever possible.

4.3.1 Threadpool Restructuring

The first item we need to revamp is the threadpool design of Tomcat. The original Threadpool class causes most of the performance issues because it is under intensive synchronization whenever Tomcat harvests an idle thread from it, resulting in a heavy per-request communication cost in the cluster-wide environment.

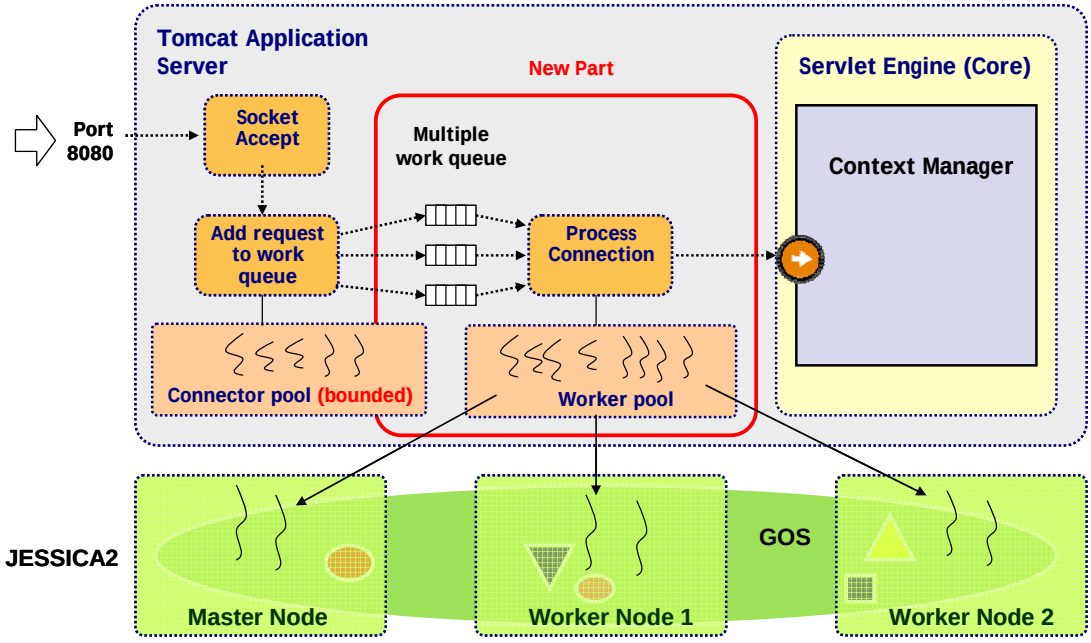

Figure 4-1: Architecture of modified Tomcat-on-JESSICA2

Using Multiple Work Queues

While synchronization cannot be totally eliminated, we can reduce locking on a single resource to minimize the waiting time of the other threads when the current thread which gained the lock is flushing and fetching objects. Also, synchronization can be done in a way to affect as fewer objects as possible. By these principles, we designed a threadpool with multiple work queues as its entrance. Figure 4.1 shows the modified architecture of Tomcat. In our design, we introduce a new threadpool called the *worker pool* which allocates a private queue to each thread in it. The threads in this pool are distributed to all the cluster nodes. The queues are for buffering up accepted socket connections. The original threadpool is not retired but kept for accepting connections in a multithreaded manner. The threads in this pool which we call the *connector pool* are set as daemon threads so that they will not be migrated. Each accepted socket will be dispatched to one of the queues in a round-robin schedule. Then a worker thread will pick up the socket object, process it and execute the target servlet. In this way, only two threads – one connector and one worker – will compete the same lock every moment. In contrast to the original threadpool which is competed by all threads, this implementation is more efficient. The synchronization blocks in the new threadpool are also kept as thin as possible to minimize access faults which cause fetching of the master objects. Any output result is still redirected to the master node. However, we have eliminated the redirection cost of accept() since connections are now always accepted by the connector threads bounded in the master node.

4.3.2 Dissolve Intensively Shared Object Pools

As the overhead of competing shared object pools become serious in a cluster-wide environment, we decided to dissolve some of them, that means we would avoid their use along the critical path. For example, we do not use the connectionCache and the HttpRequestAdapter pool anymore. Instead, we let each thread create the objects they need at startup. The objects will not be garbage collected until the threads end. In this way, we are not losing the merit of object pooling. We also bypassed the use of an intensively shared object called RecycleBufferedInputStream in the new system.

4.3.3 Add a JSP Compiler Plug-in

This modification is not related to performance but to make JSP web applications to be able to run on JESSICA2. By default, the JSP compiler in Tomcat is the Sun Java Compiler which is absent in JESSICA2. Our solution is to add the KjcJavaCompiler plugin [23] to Tomcat, set it to be the default JSP compiler in the WebXmlReader source and rebuild Tomcat.

4.3.4 Tomcat Startup Script

The Tomcat startup script is modified to start with JESSICA2. A new configuration file called JHosts is added to let users specify the host names or IP addresses of the worker nodes to be used. One important note is that all web application classes, third party class libraries and the working directory for compiled JSP classes must be added to the classpath in the script for both master and worker JVMs to locate them.

Chapter 5. Performance Evaluation

To evaluate the combined Tomcat-JESSICA2 package, a set of measurements and experiments was conducted in a cluster environment. Besides using web applications to assess the scalability obtained, specific overhead studies and comparisons were also included to explore the underlying system behavior.

5.1 Performance Metrics

Recall the definition of speedup below for clarity of our measurements:

$$S_p = \frac{T_1}{T_p} = \frac{W_p}{W_1}$$

where T_1 is the execution time using 1 node (W_1 is throughput obtained by 1 node);
T_p is the execution time using p nodes (W_p is throughput obtained by p nodes);

Note: In all the following experiments, we have taken *absolute speedup* - we use the original version of Tomcat which is supposed the best-known program for measuring the single-node performance.

A constant workload is injected to Tomcat with varying number of nodes in each test case and the total execution time and throughput are measured. We use 8 threads for most experiments and scale the cluster up to 8 nodes to assess the scalability.

5.2 Experimental Platform

All experiments were conducted on the HKU Ostrich Cluster with the following configurations.

Hardware Configuration

- CPU: PIII 733 MHz
- RAM: 512MB
- Interconnect: one 8-port Gigabit Ethernet backbone + four 24-port Fast Ethernet switches

Software Platform and Tools

- Fedora Core 1 (Linux kernel 2.4.22)
- HKU-SLIM with NFS shared file system
- MySQL database server 4.0.24
- MySQL Connector/J 3.0.16 (GA release)
- Apache web server 2.0.53 (with mod_jk connector 1.2.10)

In particular, Apache JMeter [20] is a very useful stress/volume testing tool. We used it to simulate various levels of workload of simultaneous client requests.

5.3 Application Benchmarks

This section will describe the applications used to evaluate the performance of our DJVM-clustered Tomcat server. DJVM researches usually evaluate the performance by solving scientific problems such as π-calculation, Successive Over Relaxation (SOR) and the Traveling Salesman Problem (TSP). However, these benchmarks are rather primitive and are not common server-side applications. Therefore, they are not suitable to benchmark our server. Worse still, there are very limited servlet-based application benchmarks publicly available for our experiments except the TPC-W bookstore benchmark. Therefore we need to implement by our own some more benchmarks which should model possible and realistic web application scenarios.

We deployed totally four application benchmarks, namely:

1 TPC-W Bookstore
2 Online Bible Quote/Search Tool
3 Stock Price Data Feed Service
4 SOAP Securities Order Processing

Their characteristics and particular testing parameters will be explained as follows.

5.3.1 TPC-W Bookstore

The TPC-W benchmark [21] has been developed by the Transaction Processing Performance Council (TPC) in response to the rise of e-commerce systems. It models the behavior of an on-line bookstore, including many elements commonly found in e-commerce applications: a web-site supported by a web serving component which can present both static and dynamic web pages; a relational database which is accessed from the web server to provide transaction processing and decision support. It also intensively uses sessions to model the shopping cart scenario.

We will deploy the servlet version developed by ObjectWeb [22]. A MySQL database with size of around 250 MB representing 144,000 customers and 10,000 book items is used to simulate a business case in reality. There are also 20,000 book images (random pixels generated by the gd graphics library) for client browsing. A screen capture of the application homepage is shown in Figure 5.1 below for reference.

In short, this application has the following characteristics: short-lived requests, large number of sessions and quite I/O intensive due to the download of graphic images. We speculate the speedup obtained through JESSICA2 for this kind of application is limited because the extra JVM-level overhead will dominate for short-lived requests.

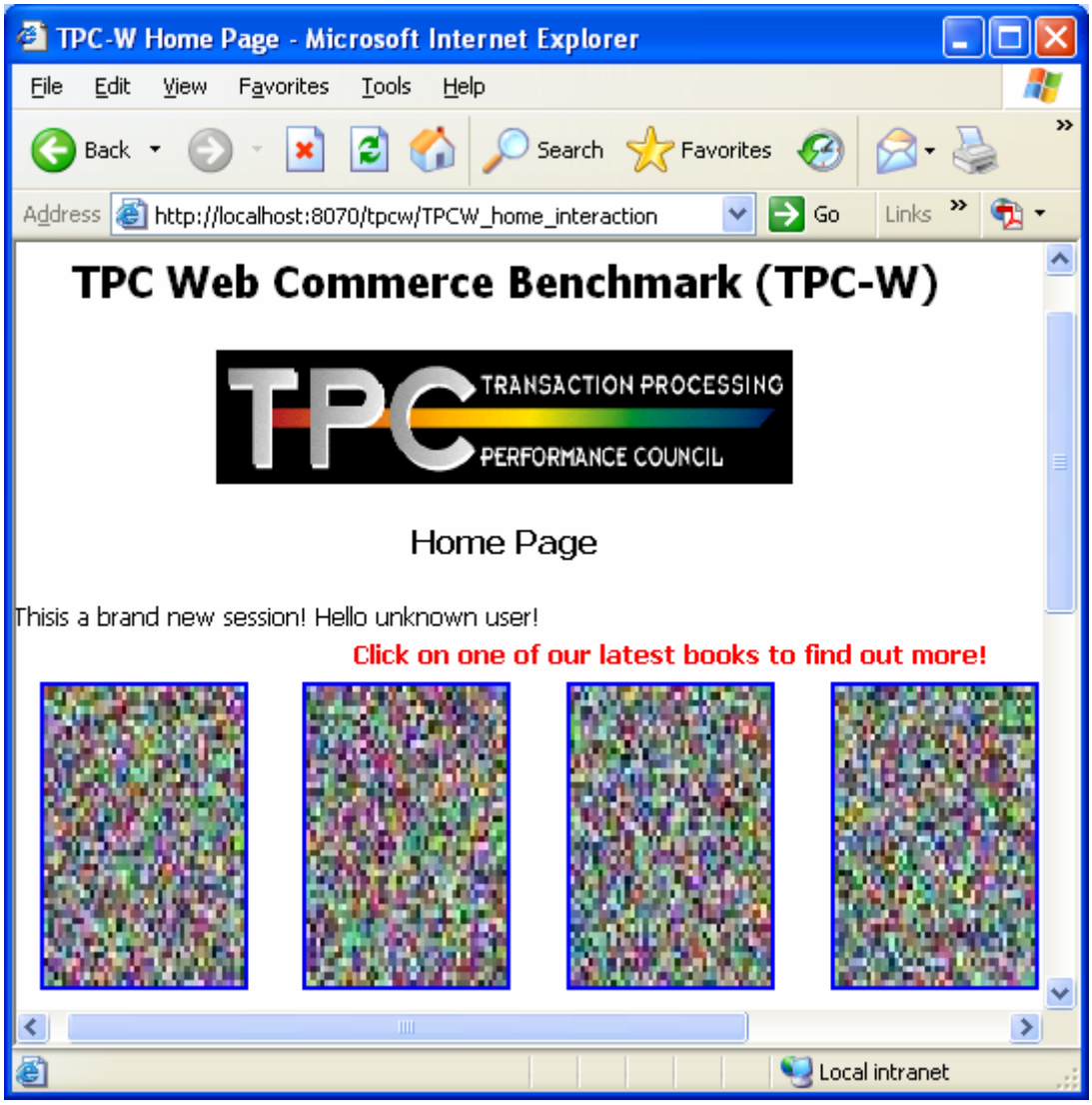

Figure 5-1: Home page of TPC-W benchmark

5.3.2 Online Bible Quote/Search Tool

This application models web applications having frequent file accesses. It represents many common online document retrieval applications (like government press releases, news archives and company catalogs). It is common for these applications be with some advanced text search facilities.

The implementation actually takes reference to BibleGateway.com which is a portal providing online services for searching and quoting verses in the Holy Bible. Each of 66 books in the bible is saved a text file in the shared file system of the cluster. The processing time of each request depends on the input parameter of the number of verses in query.

With JESSICA2, this servlet application essentially acts like a parallel file server. We expect this kind of application could attain good speedup because file accesses become distributed or parallel. Also the requested file content is read from file line by line, causing many string concatenations which are compute intensive although many people may have an illusion that they are lightweight.

5.3.3 Stock Price Data Feed Service

This application is constructed to model a stock market data provider. This is an imperative B2B (or B2C) service – securities firms, brokers, banks and general retail customers rely on such services for real-time or historical price quotation. In B2B scenario, data consumers are connected to such service with price feed link running on a specific messaging protocol. In hot market seasons, price fluctuations happen to be frequent and will cause heavy workload on the data provider server.

Due to the difficulty of implementing real-time quotation, our implementation is a historical quote service. We follow the trend of using XML messages to deliver the price data. When the servlet receives a request, it will randomly query one instance of a 4-node MySQL database cluster and format the data into an XML message sending back to the client. We prepared the database by downloading real stock price data from Yahoo Finance website for modeling true data size. Each request carries the parameters: stock code, start date and end date. The processing time of each request depends on the number of days in quotation.

5.3.4 SOAP Purchase Order Processing

Lastly, we also write a SOAP-based application to test the effectiveness of running web services over our Tomcat-JESSICA2 system. It is common in B2B sector to exchange purchase order messages which are in batch format containing a number of transactions. Stock Initial Public Offer (IPO) is such a scenario. SOAP protocol is XML-based and serves as a platform for handy service invocation. XML parsing is however an intensive operation that will burden the server when simultaneous SOAP requests are received.

We implemented a Java class that has a method for processing a batch of orders. First the received SOAP message is parsed by calling the methods provided by the SOAP engine (which internally parses the XML message in DOM model). Then each order is validated against the customer database and then updated to the transaction database. A report listing successful and failed orders is created at the end of the processing. In our testing, the number of orders in each request ranges from a few tens to a few hundreds.

5.4 Experimental Results

Various performance evaluations were conducted and their results are presented as follows.

5.4.1 Scalability Study

Figure 5.2 depicts the scalability achieved by the modified Tomcat on JESSICA2 in each application.

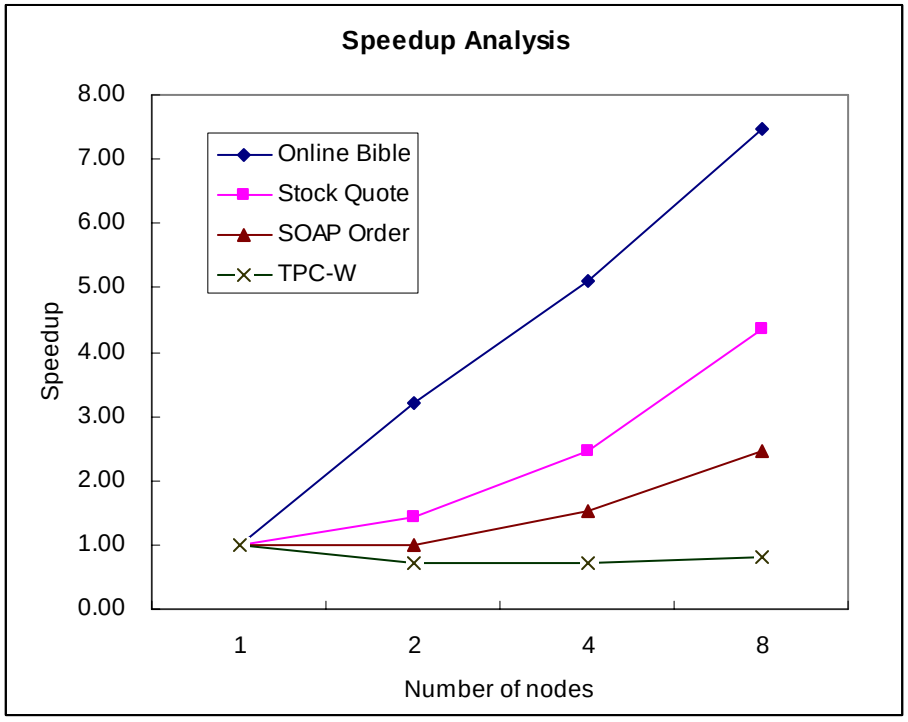

Figure 5-2: Scalability curve of various application benchmarks

Several phenomena can be observed from this set of results. First, for the online bible application, we obtained a very linear scalability. We even got superlinear speedup in the 2- and 4-node cases. Such superlinear speedup is attributable to (1) the additional computing resources (caches/memory) for doing string concentration; (2) parallel file accesses; both are provided by JESSICA2 and are absent in the single-node setting.

Stock quote data service achieves average speedup, at around 50% efficiency. The reason is due to the large response size. Creating XML document from the database is resource-intensive and so this application should benefit from JESSICA2, however the return of such long XML messages to the clients is redirected to the master node and gets it congested. The augmented resource advantage is nearly half compensated.

SOAP order processing scales poorly due to heavy synchronization inside the SOAP engine. The GOS traffic log revealed many objects belonging to the SOAP engine like MessageRouterServlet, DeploymentDescriptor and SOAPMappingRegistry are being exchanged every request-response cycle. Some of these objects are large in size as SOAP tends to use Hashtable for mapping objects frequently. Although Tomcat has been tuned well on JESSICA2, SOAP is not. To attain good speedup for SOAP-enabled web services, we need another porting effort customized for SOAP, although this case should be much simpler than Tomcat. This also suggests whenever a new application library is added on top of Tomcat, it may not scale well without performance tuning. The servlet application itself should also follow certain rules (e.g. doing less synchronization) which make it favor on the DJVM paradigm.

Finally, TPC-W attains negative speedup, confirming our speculation. This application is merely I/O intensive; it does not take advantage of the resources augmented by JESSICA2 to do computation. And because of its short-lived request

characteristics, the per-request GOS traffic and I/O redirection overhead become dominating, hence having a negative effect to this kind of application.

The next study supplements the above discussion. We aim to find out the dependence of speedup on the average request processing amount. The application used in this experiment is the online bible benchmark. Data size means the average number of bytes returned in the response which depends on the requested number of verses, and is proportional to the number of compute cycles in this application. We wish to clarify larger response size may not necessarily mean more processing but in this application this is true. Figure 5.3 shows the scalability curves corresponding to three different data sizes – small (< 4KB), medium (~ 12KB) and large (~ 30KB). It is clear that the larger the data size, the better the scalability. The reason is that larger data size requires more compute cycles which make effective use of the CPU cycles offered by JESSICA2, thus pushing up the whole system's efficiency.

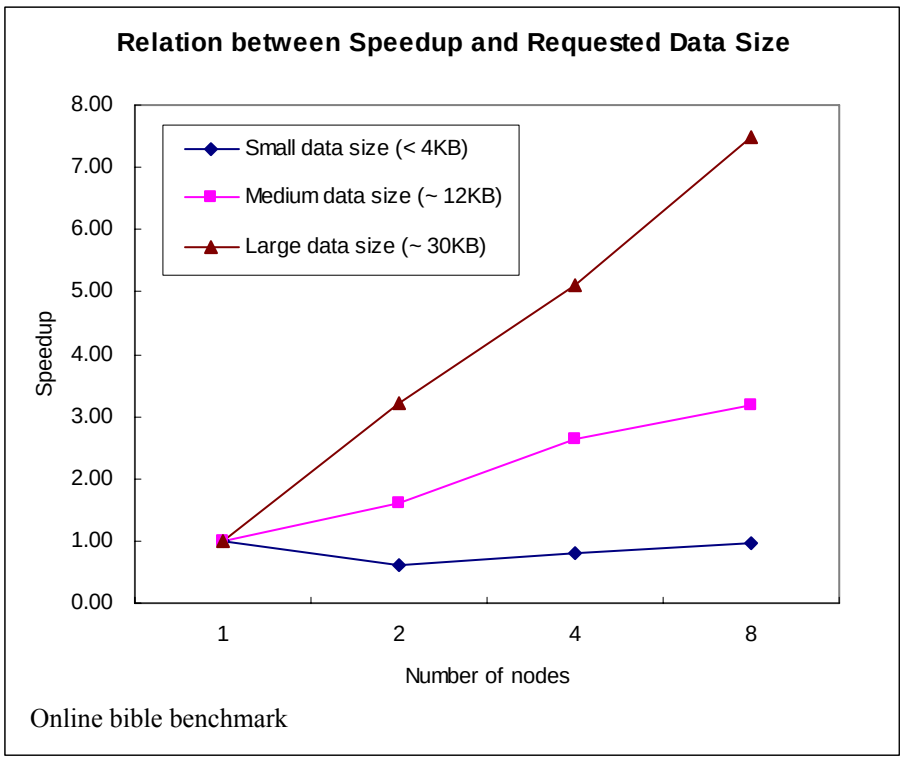

Figure 5-3: Relation between speedup and amount of data processing

5.4.2 Evaluation of Tomcat Modifications

Next, we will evaluate the effectiveness of the modifications applied to Tomcat. Unless specified otherwise, all comparison experiments and overhead studies in the context below is using the online bible benchmark (we skip other benchmarks for there is not much difference in the result in most cases).

First, we compare their scalability attained. The result is depicted in Figure 5.4. In all node combinations, the modified version attains a throughput over the double of the original version. Such an improvement is mainly caused by successful reduction of synchronizations and hence saving a lot of GOS traffic. Another reason, but less

significant, is the modified Tomcat accepts sockets locally at the master; this avoids one unit of I/O redirection overhead per request-response cycle. More overhead comparisons between both versions will be presented in the sections to come.

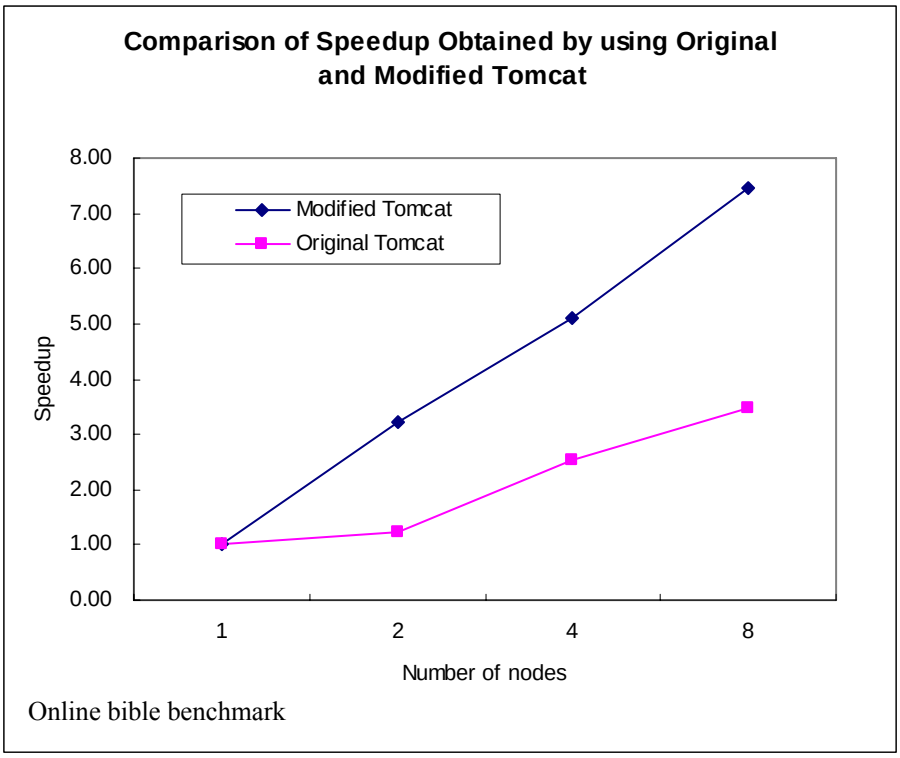

Figure 5-4: Comparison of speedup between the original and modified Tomcat

5.4.3 GOS Overhead Study

In this section, we will present several insightful figures about the running condition of the Global Object Space.

5.4.3.1 Cache Heap Size

As mentioned before, Tomcat has a large memory footprint with over 200 thousands objects. In this study, we aim to investigate how large the cache heap would be and its size variation along the execution. We use the average number of cached objects to represent the heap size for easier experimental tracking. Figure 5.5 shows the average number of cached objects of a thread in the original and modified Tomcat along with different lengths of execution time.

First, we can see the average per-thread cache area is not too big, around hundreds of objects. That means the number of objects that are being used for serving requests during a typical thread execution in Tomcat is limited. So this is supportable on JESSICA2.

Secondly, we can notice the size of per-thread cache area in original version is several times larger than the modified. The original one scales around several hundreds of objects. When using 8 threads, the total cache heap size will be several thousands. Synchronization on such a big cache heap will generate a high GOS traffic rate and

slow flushing over the space. This is a dominant factor limiting its scalability. On the other hand, the new version cached less objects and suffer less synchronizations due to rectified flow of execution. This contributes to its scalability rise as we have seen.

However, we can also observe that the cached heap size in the new version increases with the length of execution linearly while the original tends to saturate over a period of time. This an undesirable property in a server system and could be caused by lack of proper garbage collection over the cached heap area in current JESSICA2. More future analysis and enhancement would be needed to solve this problem.

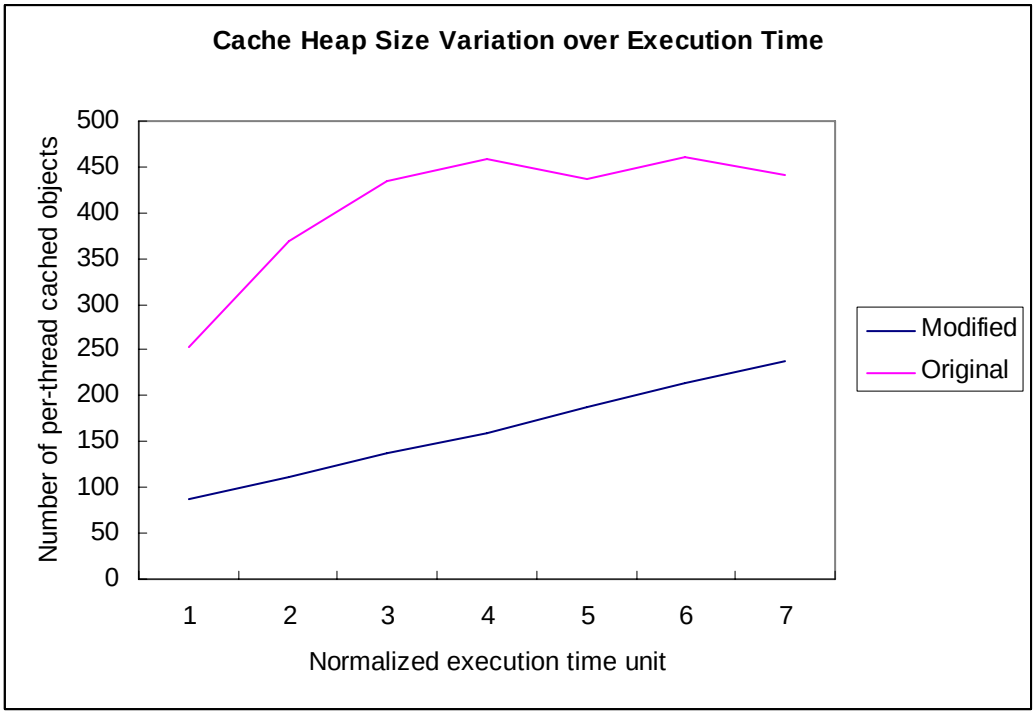

Figure 5-5: Cache heap size variation

5.4.3.2 GOS Traffic

Figure 5.6 shows the number of objects exchanged over GOS is huge in the original Tomcat – an average rate of over 7,000 GOS requests per second is recorded in the 8-node case for the online bible benchmark. Indeed in all the benchmarks, a huge figure is observed. The root cause for this high traffic volume is due to synchronizations which happen at each request processing. In contrast, the GOS traffic in the modified version is much smaller. Such enormous saving of the communication overhead reflects effectiveness of using multiple-work-queue threadpool which avoids several synchronization blocks that happen in the original threadpool coding. Skipping the use of some shared object pools in the HTTP request handler also contributes to this success because of less locking. This result shows that our modification is quite effective though it is slight (less than 2% source code change).

Figure 5.7 shows the distribution of GOS traffic over the cluster in the 8-node case. We can see in both versions the master node is loaded with most of the GOS requests. This is expected because most objects were created on it and worker nodes need to fetch them remotely. In the original version, objects are flushed back to the master

node or to other workers for object state consistency according to the JMM. We can see from Figure 5.8 that the inter-worker communication in the original version is intensive exactly due to this reason. In the modified version, this situation is much improved – we can revisit Figure 5.7 to see there are basically no GOS requests sent to worker nodes for object fetching. Our code restructuring lets workers create objects locally and effectively bypasses synchronizations; this saves workers from the need to send back updates to remote object homes as in the original version.

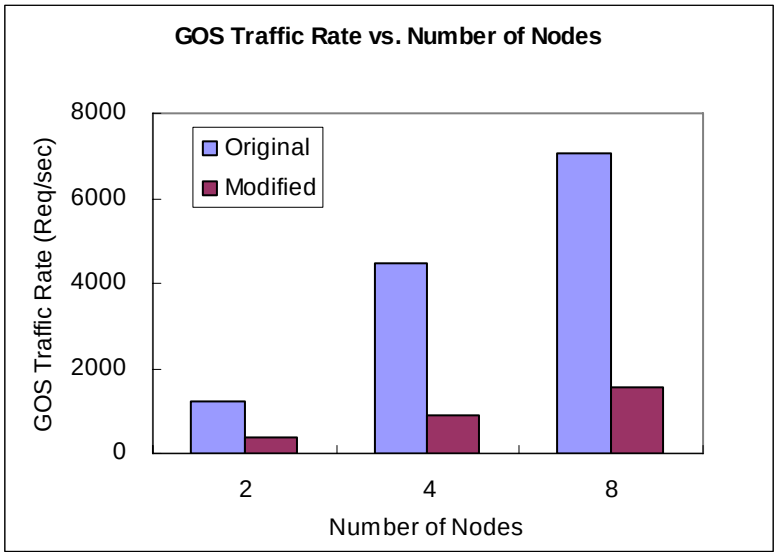

Figure 5-6: Average GOS traffic rate of the original and modified Tomcat

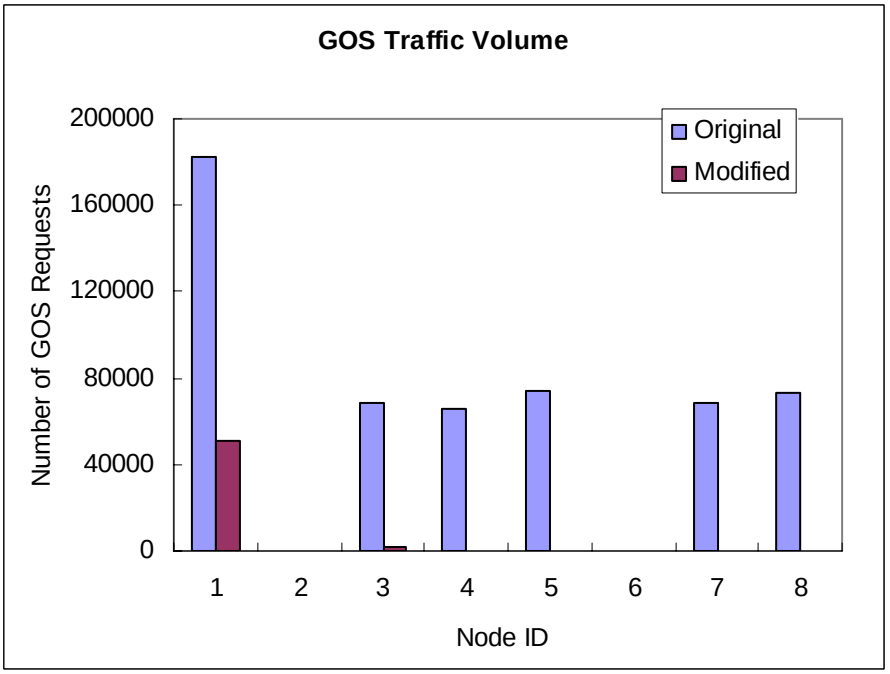

Figure 5-7: GOS traffic volume distribution over nodes

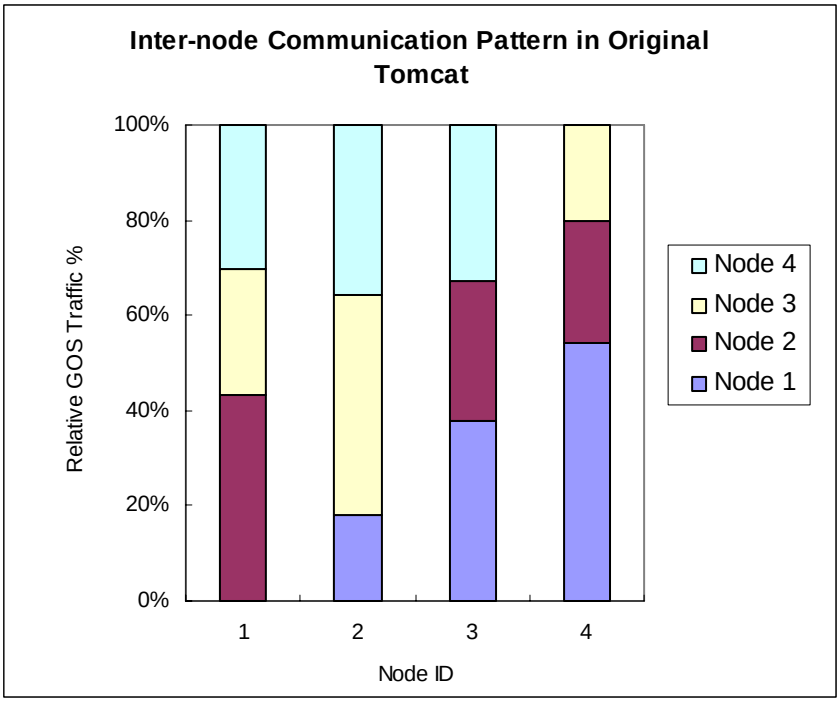

Figure 5-8: Inter-node communication distribution in the original Tomcat

5.4.3.3 GOS Traffic Breakdown

Next, we will take a closer view on the breakdown of the GOS requests. Figure 5.9 below shows GOS request type distribution on the master. We can see remote object fetching (GetObj) is the most frequent operation; second is remote object locking; followed by array fetching and then flush operation. Remote object locking is severe in the original Tomcat but is much alleviated in the modified version.

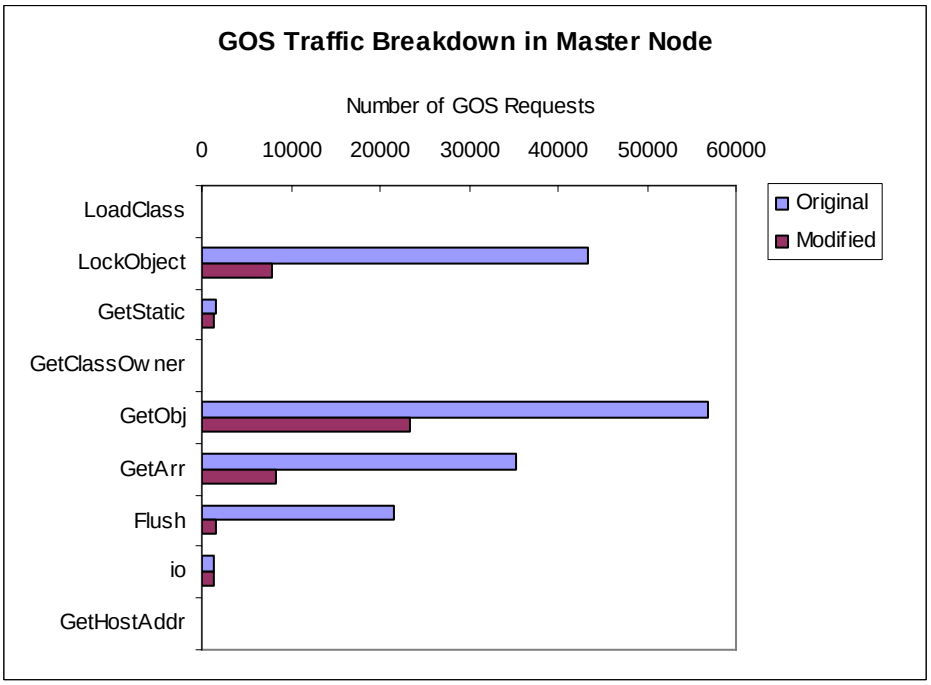

Figure 5-9: GOS Traffic breakdowns of request types (in master node)

Figure 5.10 below shows the average breakdown of GOS requests on workers. As explained before, in the modified version has no demand at all except sending back lock acknowledgements on the work queue object to the master.

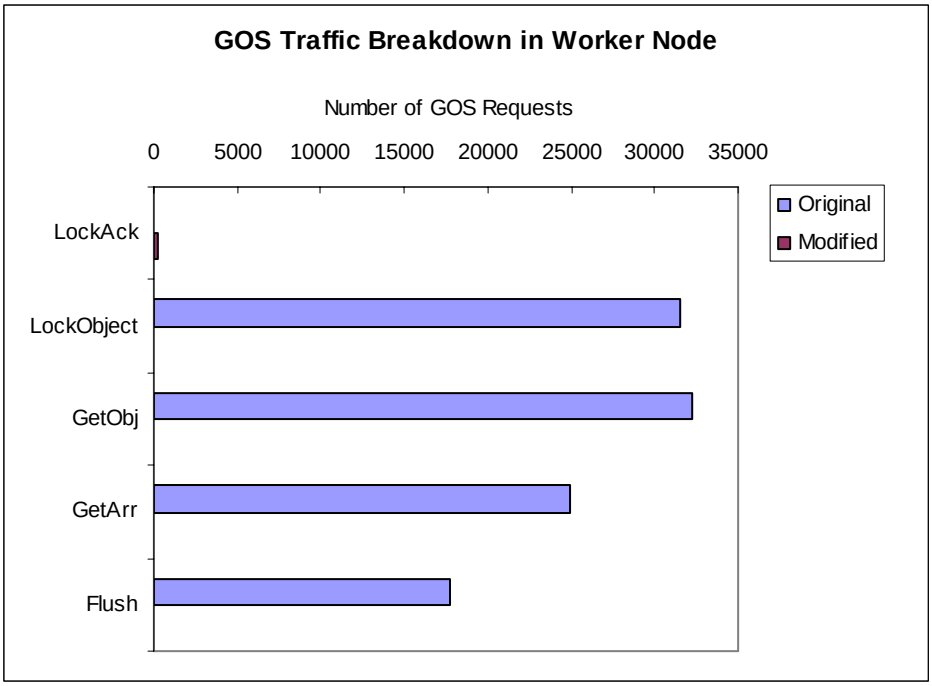

Figure 5-10: GOS Traffic breakdowns of request types (in worker node)

5.4.3.4 Hot Objects

This section presents and compares the top ten frequently accessed objects via GOS in the original and modified version. This study facilitates design of right optimization strategies in Tomcat or JESSICA2 enhancements. As we can see in Figure 5.11, the most frequently packed object in original Tomcat is RecycleBufferedInputStream which was designed to save garbage collections by reusing the stream object. However, when Tomcat runs in DJVM environment, this simply causes severe contention of this single resource. Figure 5.12 shows the modified Tomcat has a more diverse distribution on the objects to pack and eases possible contentions. We can also see Tomcat uses vector classes quite intensively. We can devise specific optimization techniques at the DJVM level in future enhancements.

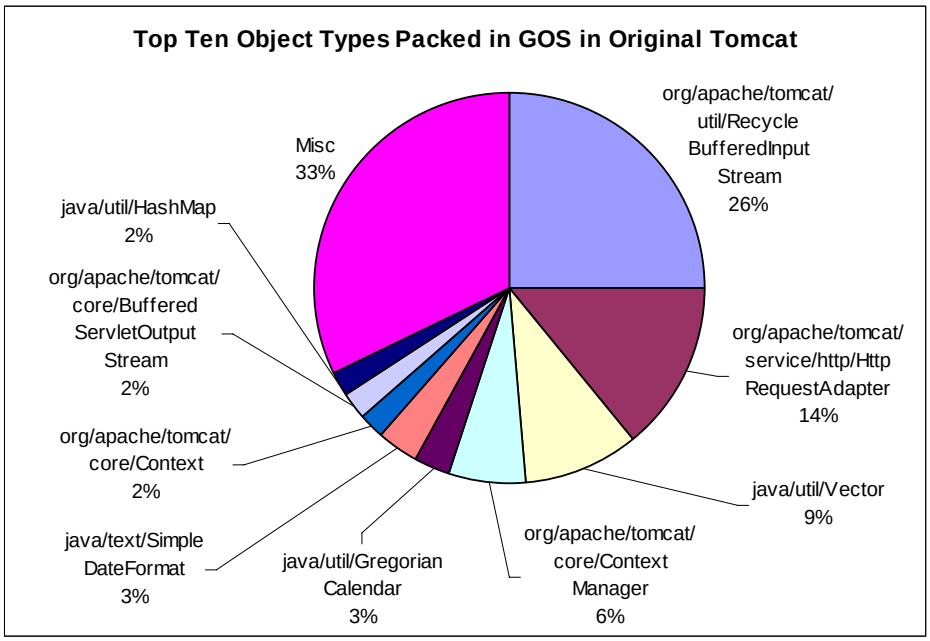

Figure 5-11: Top-ten hot objects packed over GOS in the original Tomcat

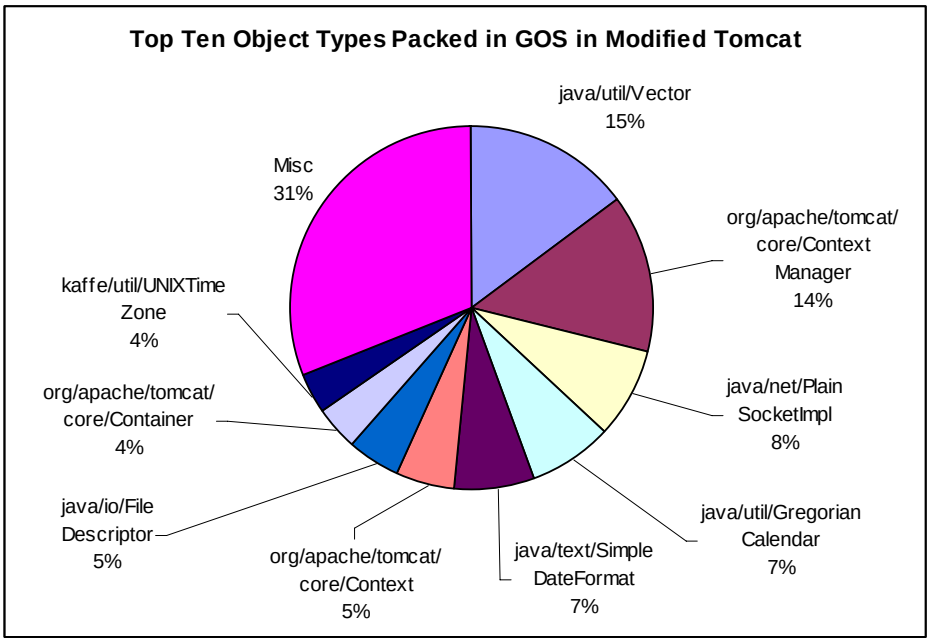

Figure 5-12 Top-ten hot objects packed over GOS in the modified Tomcat

5.4.3.5 GOS Optimization Effects

We would evaluate the effectiveness of two GOS optimization techniques – object home migration and object pushing in this section.

Object Home Migration

For the original Tomcat, this feature was disabled in our experiments due to the resulted instability. It was found that after migration of some byte arrays (which belong to the RecycleBufferedInputStream object) from the master, later update on the arrays performed by the requesting worker will fail and dump the JVM.

However, in the modified version, this error was not encountered. This is because we does not have such byte arrays being actively written by worker nodes and therefore their migrations do not occur throughout the execution. In the online bible benchmark, there is virtually no migration occurred except one or two migrations of the java.util.GregorianCalendar object from the master node to a worker. However, after this migration, the object needs to be successively packed back to the master node which is still referring it. Therefore, one object migration resulted in more remote object accesses, violating the original goal of this optimization. In the SOAP order processing benchmark, there are frequent migrations of java.io.FileDescriptor objects from the master to all workers. However, the measured throughput with and without home migration has no noticeable difference. The benefits offered by home migration here tend to be offset by some other limiting factors in the application such as packing of large-size arrays and hash maps in the SOAP engine.

In Tomcat, single-writer pattern seems to be seldom because every worker threads are having equal chance in reading and writing a shared object like a pool. Enabling home migration in this kind of application may even have a negative effect to cause object homes bouncing back and forth. Therefore, this posterior pattern adaptation strategy does not help under the roughly random access dynamics in Tomcat

Object Pushing (Pre-fetching)

By default, the object pushing optimization is enabled in JESSICA2. This option can be switched off by running JESSICA2 with the –Jnoprefetch option. In Figure 5.13, we can see enabling object pushing causes a slight decrease in speedup. The reason for this can be explained by the strong ramification among objects in Tomcat - one object has many cross-references to other objects. For example, a HttpRequestAdapter object has many fields like Socket, HttpServletRequest, Response, Hashtable, Context, Container, ContextManager and many string objects. However not of them are needed in serving a request. With this tangled field-referencing nature of Tomcat, object pushing will transfer more unnecessary objects and lead to a poorer speedup.

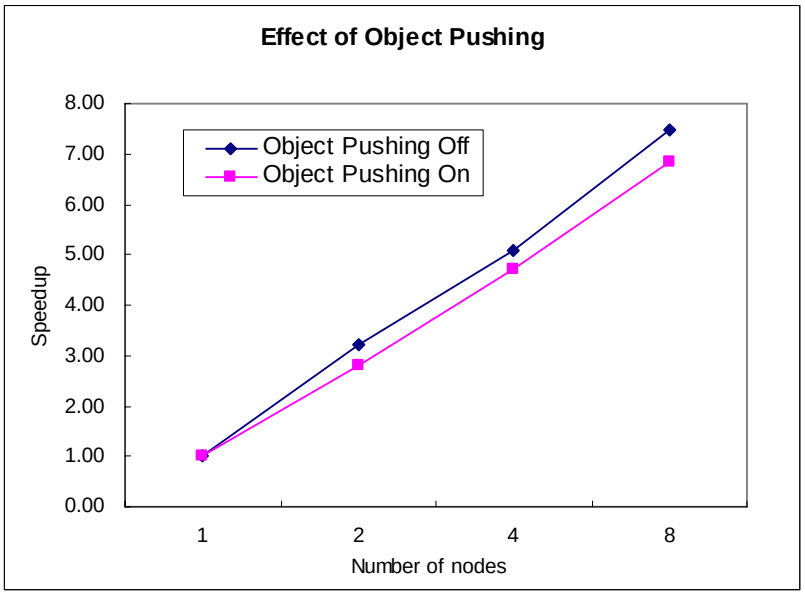

Figure 5-13: Effectiveness of object pushing optimization

5.4.4 GIO Overhead Study

Figure 5.14 below shows the breakdown of the types of I/O redirection requests in both the original and modified versions. Socket write predominates here simply because the online bible benchmark returns lengthy data to clients. The significance here is that I/O daemon spawning overhead in modified Tomcat is lowered by half because it is restructured to accept sockets locally at the master node.

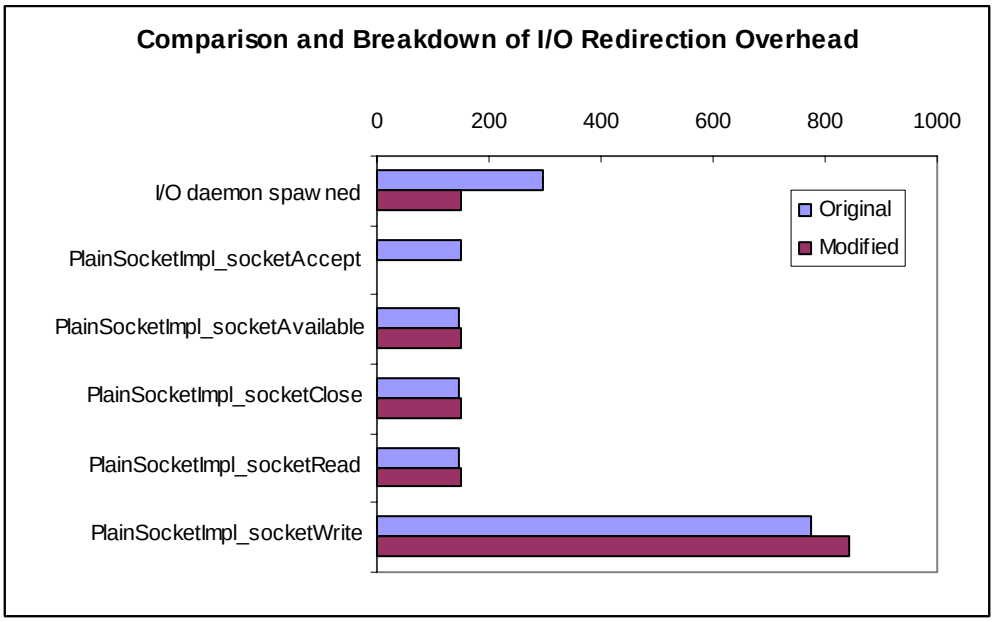

Figure 5-14: Breakdowns of I/O redirection overhead

5.4.5 Thread Migration and Initial Placement

This study aims to investigate the load balancing effectiveness of thread migration and initial placement.

Thread Migration

No. of node	Execution Time (mm:ss)	Throughput (req/min)	Speedup	Migrated Threads
1	05:20.4	46.7	1.00	0
4	02:32.2	97.5	2.11	3
8	02:32.8	97.3	2.10	6

Table 5-1: Performance results of dynamic thread migration

Thread Initial Placement

No. of node	Execution Time (mm:ss)	Throughput (req/min)	Speedup
1	05:20.8	46.7	1.00
4	01:00.8	243	5.27
8	00:41.8	345.7	7.68

Table 5-2: Performance results of thread initial placement

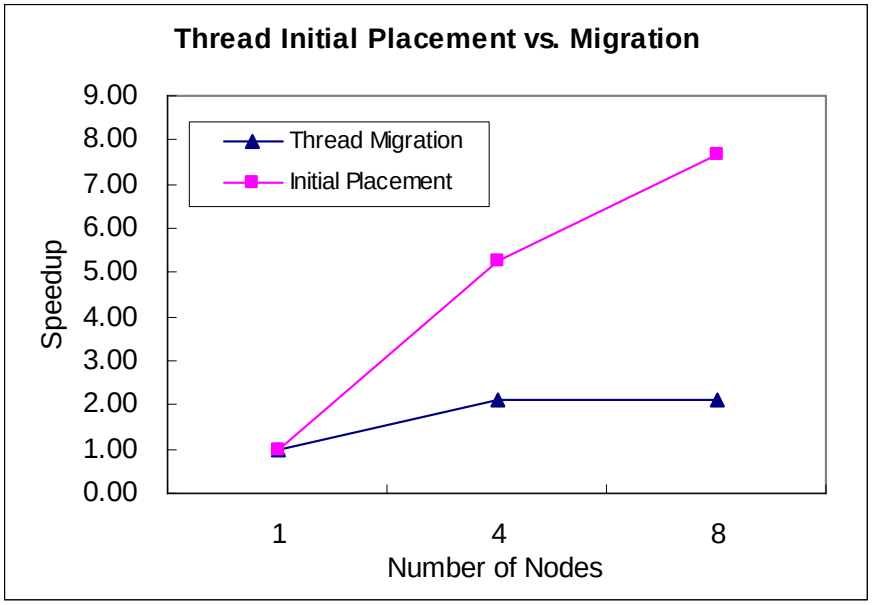

Figure 5-15: Speedup comparison of thread initial placement and thread migration

Figure 5.15 shows that thread initial placement outperforms dynamic thread migration several times. This phenomenon is quite the contrary to the case of irregular multithreaded applications.

The reason for the poor speedup achieved by dynamic thread migration is multi-folded: (1) We use 8 threads in Tomcat but only 6 of them were migrated. There is hence one idle machine. (2) Some threads are migrated near the end of the stress testing. Experiments show that it usually takes 3/4 to half of the testing interval for all threads to be migrated out to each worker. Thus the additional computing resources provided by the worker processors are indeed largely out of reach from the application. This could imply either the thread scheduler has some intrinsic problems or the current *work stealing* load balancing mechanism in JESSICA2 is too slow to react to dynamic workload changes in the server. (3) After migration completes, it is also found that the throughput immediately drops by around 15% to 20% followed by a rise again. This is because the migrated thread is busying the master node for fetching all its referenced objects created at the master. On the other hand, thread initial placement does not suffer from this overhead because all workers have fetched most of the necessary objects at startup; the "working set" of the threads is also smaller. (4) Finally, we would point out the parameter -JDelay which is used to control the time interval of resource statistics exchange between the master and the workers. Since dynamic thread migration is found to be too inactive, we are forced to shorten this parameter to 1 second in order to see reasonable thread migrations. Such frequent statistics exchange overhead limits the scalability of the overall cluster.

Combining all above, the speedup that can be offered by thread migration is much lower than that offered by thread initial placement.

5.4.6 Comparison with Web Server-Based Dispatching

In this final experiment, we would compare our DJVM approach with a web server-based dispatching solution. Apache server with mod_jk connector is a very common option in server load balancing. Therefore, we choose it in this experiment.

First, we tried the TPC-W benchmark but our system gives negative speedup. Apache with mod_jk achieved an average speedup but the scalability is far from linear. The possible reason is that the bottleneck happens at the single-node database tier. Scaling out Tomcat instances hence does not help much on the overall performance. In this kind of application, DJVM approach is not as effective as common solutions.

Then, we performed the study using the online bible benchmark and obtained the result in Figure 5.16 below. Clustering by JESSICA2 achieved twice the performance obtained by mod_jk.

The reasons for better performance are mainly due to the more load balanced state maintained by JESSICA2 in the cluster. Figure 5.17 shows the distribution of average CPU utilization of all the cluster nodes during the test. It can be seen that mod_jk failed to provide a load balanced situation: node 3 and 8 are much more loaded than node 4. There could be possible internal problems in mod_jk connector so that it cannot achieve a fair round-robin scheduling. On the other hand, JESSICA2 levels off the load more evenly via thread scheduling at the JVM level.

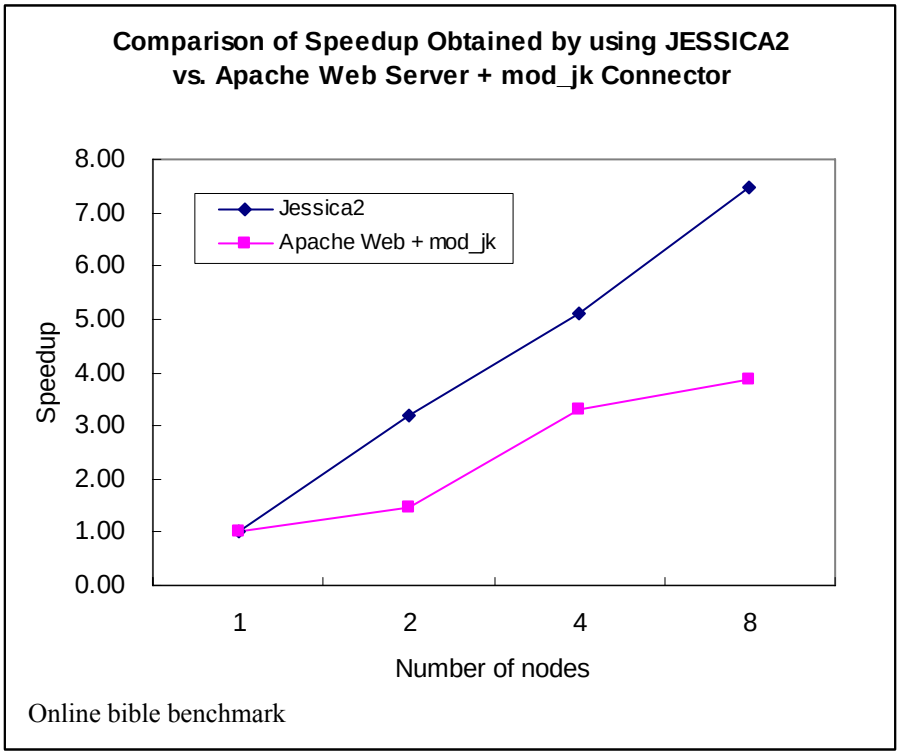

Figure 5-16: Comparison of speedup by Tomcat-JESSICA2 and Apache mod_jk

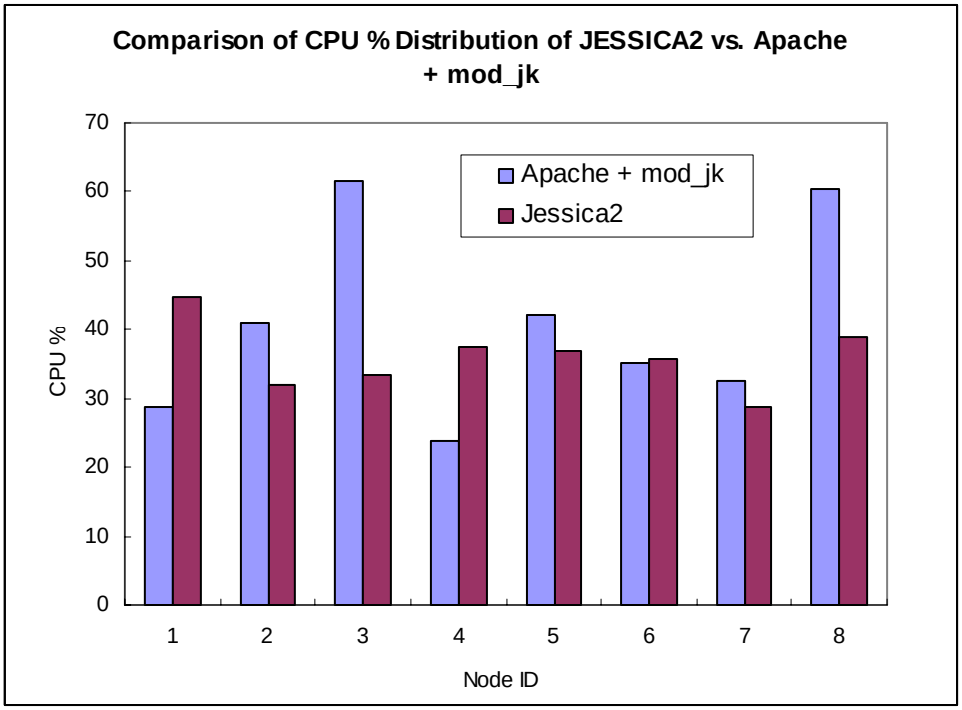

Figure 5-17: CPU usage distribution of Tomcat-JESSICA2 and Apache mod_jk

Chapter 6. Discussion

In this chapter, we will discuss various issues of the DJVM approach proposed by this dissertation. In particular, we would account for the limitations of current JESSICA2 implementation in supporting application servers.

6.1 Poor Speedup in Fine-grained Work

From our experiments, we can see that the computation/communication ratio is the dominant factor which limits the scalability of multithreaded Java server running on DJVM. This confirms with the finding in [1]. Better design and implementation of the GOS layer is vital to reduce the communication cost and hence to allow a wider scope of applications with smaller computation/communication ratio.

6.2 Call for Better Consistency Model

The major limiting factor in our scalability is caused by intensive GOS traffic for maintaining cached object consistency. Although we have adopted lazy release consistency at memory boundaries, the GOS traffic is still high due to large number of objects in cache heap areas resulted from typical enterprise-scale servers.

Our GOS implementation is following the original JMM. However, a more efficient implementation should be viable but is currently lacking. For example, there should be no need to check and flush immutable objects (marked with Java keyword `final`) in the cache heap because these objects are read-only. So their state should always be kept valid and we can save the cost of enforcing our consistency protocol on these objects. However, our current handling is to invalidate all cached objects and compare timestamps between the master copy and cached copy. Perhaps, we can allocate a partition in the cache heap area to hold read-only objects which will not be affected during synchronization.

Furthermore, the JMM is currently undergoing revision through the Java Community Process (JCP) as some researchers argue that the original JMM is not well designed and prohibits some common compiler optimizations. Hopefully, a future version of JESSICA2 can implement a better designed JMM efficiently.

6.3 Effectiveness of Object Home Migration

General advice in the web developer community is to avoid storing large objects and to limit the number of objects saved in a HTTP session for reducing the impact on the system's available memory and the cost of object serialization. Therefore, the effectiveness of our home migration protocol is being challenged by web applications in which single-writer patterns are rarely detected. However, we could argue that with object home migration, we can support more and larger objects in a HTTP session. For example, in a B2B application scenario, a session can be used to accumulate a list of transactions which are successively processed by a single writer in a loop. Then home migration will be activated and reduce the number of remote updates.

6.4 Dynamic Thread Migration

We have seen from the experimental result that the good feature of dynamic thread migration is greatly hampered when it is used in server domain due to the various issues we explained. However, if more embarrassingly parallel threads in Tomcat and more reactive migrations can be achieved, this mechanism will be capable to provide some very useful aspects we mentioned in Chapter one – e.g. server resources can be integrated in a zero-downtime fashion by plugging worker JVMs, web applications of irregular workload can enjoy more speedup from thread migrations.

6.5 Array Checking Overhead

Originally, the stock price data feed benchmark has implemented DES encryption that can be enabled for secure service. However, after testing, we found that a negative speedup was resulted and we need to disable this feature. The reason is due to heavy array accessing in the encryption algorithm, causing a lot of array checking overhead. This result also confirms with the slow down of the compress benchmark in [1] which bears similar runtime nature consisting of intense array processing. To make it worse, partially cached arrays cannot even enjoy the fast state checking optimization. We need more advanced compiler analysis technologies for reducing array checking overheads to support these important kinds of server-side applications.

6.6 Lack of High Availability Support

We have discussed session clustering via a JVM-level distributed heap. However, it cannot be used to support high-availability service yet because the JVM heaps are tightly coupled with communication messages. When one node failed, the distributed heap will be partially corrupted. Future researches in developing fault tolerant DJVMs could be the direction to cope with this limitation.

6.7 Need of Porting

Before a mature DJVM system exists, we still need efforts of porting the applications onto the DJVM in order to let users enjoy transparent clustering. This is currently a limitation forbidding DJVMs to support wider scope of web applications. However, the maturing of DJVM systems could eventually transform the way we do clustering in the future. This could also bring both advantages and impacts to the server-side community. Perhaps small and medium enterprises would like the benefits provided by DJVM systems. However, application server developers would be afraid of the possible sale dropdown of their licenses if DJVM has done a multiplicity effect to cluster their servers.

Chapter 7. Conclusions and Future Work

7.1 Conclusions

The goal of our proposed DJVM approach for web application clustering is to achieve better scalability in a transparent manner. Based on our experimental results, we can appreciate the very first success in testifying this goal.

We would make several conclusions in this dissertation below:

1. We have gained a practical experience in how to analyze the cluster-wide runtime behavior of an application server over a DJVM system and to make their interface be more compatible. Successful porting of Tomcat on JESSICA2 has been a breakthrough in the DJVM research community.
2. Our Tomcat-JESSICA2 server has achieved encouraging performance result in applications of B2B service nature.
3. DJVM clustering is suitable for web applications of more compute cycles.
4. Session clustering can be done via a JVM-level distributed heap. However, it is not efficient enough and not yet already to support high availability.
5. We spotted a number of limitations in the current version of JESSICA2. Better implementation in the consistency protocol, dynamic thread migration and array checking optimization are important for realizing good performance in common web applications.

7.2 Future Work

7.2.1 On the Application Server Layer

7.2.1.1 Co-design of DJVM-tailored Application Servers

We have briefly noticed the effective programming style in the DJVM model. Indeed, we can try to co-design an application server with thread nature, shared object access pattern and workload scheduling all being cooperative to the DJVM layer. This could be more flexible and promising than porting an existing server system. Of course, we have to study and maintain the benefits of common server optimizations in our work.

7.2.1.2 Further Enhancements on Tomcat

We can further revise Tomcat's locking granularity and its thread nature to reduce the GOS overhead. Also, we can build stick-session scheduling inside Tomcat. This can help increase session data locality and hence reduce the number of remote object accesses. This scheduling should be however done at thread level rather than node level as in the common solutions because threads could have migrated and offset the chance of obtaining data locality.

7.2.1.3 Support for EJB Containers

Nowadays, enterprises tend to use the Enterprise JavaBeans (EJB) technology to program their business logic rather than using Java servlets. Therefore the heavy workload should have concentrated on the EJB container which is also multithreaded. When a higher version of JESSICA2 is ready, we can try this kind of porting.

7.2.2 High-Availability Tomcat on JESSICA2

As discussed, to support high availability, we must implement some fault tolerant mechanisms in a distributed shared heap to prevent a single node failure from collapsing the whole server. However, this is a very challenging DSM research in that the server events may not be replayed, performance will also be much degraded due to any extra backup action of heap areas. Software transactional memory solutions [10] may give us a hint to develop this kind of DSM layer for future JESSICA2.

7.2.3 Wish List of New JESSICA2 Implementations

- **Upgrade to Latest GNU Classpath**: This is a necessary step to support later versions of Tomcat, EJB containers and many common Java packages.

- **Fast Checking for Arrays**: Currently we can partially cache a large array, but fast inline object checking would be disabled, causing great slowdown for data-intensive applications. This limitation can be solved by integrating array index bound checking and cache range checking.

- **Debugging Thread Migration**: Current JESSICA2 has a limitation that forbids debugging on migrated threads and makes development hard. Thread migration will see unexpected errors if GNU project debugger (GDB) is enabled. There is assertion in the setupFrame and pack_frame functions in the migration module for double checking the stack content inferred by JIT recompilation. When the stack states are inconsistent, the assertions will fail. As GDB may insert dummy frames onto the program stack during debugging, this will make frame packing being confused during thread migration. We may need to study how to solve this problem in later JESSICA2 implementation.

- **Inter-Worker Thread Migration**: Due to some reasons, thread migration from a worker to another worker is currently disabled. We may need to study the problems behind and resume this feature to realize free movement of threads across node boundaries.

- **More Sensitive Thread Migration**: The dynamic load balancing policy should be carefully reviewed to make more responsive migrations to happen. A more intelligent cost model can be implemented to determine which thread should be or should not be migrated.

- **Distributed Garbage Collection (DGC)**: Current JESSICA2 will never garbage collect the cached objects on the remote machines and lead to wasteful memory consumption. Some efficient incremental DGC algorithms should be considered in future JESSICA2 upgrades.

- **Fixing Memory Leaks throughout the System**: At a high server workload, JESSICA2 would sometimes throw OutOfMemory exceptions. According to our research team members, there should be quite many places in JESSICA2 having memory leakages and leading to this problem. We can use some memory checkers like Valgrind [24] to locate these subtle errors.

- **More Advanced Global I/O Implementation**: Instead of I/O daemon spawning, we can revamp the global I/O redirection code to be an I/O multiplexed version for saving thread creation cost. Furthermore, we can look at mechanisms to allow worker nodes to deal with socket connections without going to the master. The socket cloning mechanisms in [6] or some migrating socket solutions could give us some ideas on how to make this transparently in the JVM layer.

- **Customizable Level of SSI**: So far the design of JESSICA2 has done very well in transparency and the SSI compliance: users generally need not to configure the JVM before usage. However customizable transparency could be attractive if we allow users to sacrifice some transparency for better performance based on their application needs. For example, it is usually not desirable to redirect every System.out.print() to the master. With a NFS-enabled cluster, file write redirections are not necessary if asynchrony is not a concern. Therefore, future JESSICA2 should be planned with configurable SSI level. We may borrow some ideas from Terracotta in their configurable clustering semantics.

References

[1] Wenzhang Zhu, Cho-Li Wang, and F.C.M. Lau. “JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support”. IEEE Fourth International Conference on Cluster Computing, Chicago, September 2002.

[2] Weijian Fang, Cho-Li Wang, Francis C.M. Lau, “On the Design of Global Object Space for Efficient Multi-threading Java Computing on Clusters”, Special Issue on Parallel and Distributed Scientific and Engineering Computing in the Parallel Computing, Vol.29, pp. 1563-1587, 2003.

[3] Wenzhang Zhu, Weijian Fang, Cho-Li Wang, and Francis C.M. Lau, “High Performance Computing on Clusters : The Distributed JVM Approach”, in High Performance Computing: Paradigm and Infrastructure, John Wiley & Sons, Inc. 2004.

[4] Wenzhang Zhu, Weijian Fang, Cho-Li Wang, and Francis C.M. Lau, “A New Transparent Java Thread Migration System Using Just-in-Time Recompilation”, The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), pp. 766-771, MIT Cambridge, MA, USA, November 9-11, 2004.

[5] Ge Chen, Francis C.M. Lau, and Cho-Li Wang, “Building a Scalable Web Server with Global Object Space Support on Heterogeneous Clusters”, Cluster Computing 2001, pp.313-320.

[6] Yiu-Fai Sit, Cho-Li Wang, Francis Lau, “Socket Cloning for Cluster-Based Web Server”, IEEE Fourth International Conference on Cluster Computing (CLUSTER 2002) Chicago, USA - September 23-26, 2002, pp. 333-340.

[7] Leszek Borzemski and Krzysztof Zatwarnicki, “A Fuzzy Adaptive Request Distribution Algorithm for Cluster-based Web Systems”, Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, February, 5-7, 2003, p.119.

[8] Athanasios E. Papathanasiou and Eric Van Hensbergen, “KNITS: Switch-based Connection Hand-off”, Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE Vol. 1, 2002, p. 332- 341.

[9] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In Proc. of International Conference on Parallel Processing, 1999.

[10] Kaloian Manassiev, Madalin Mihailescu, Cristiana Amza, "Exploiting distributed version concurrency in a transactional memory cluster", Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, 2006, p.198 – 208.

[11] Greg Barish, “Building Scalable and High-Performance Java Web Applications Using J2EE Technology”, Addison Wesley, 2001.

[12] Shyam Kumar Doddavula, “Clustering with Tomcat”, Article published on O’Reilly ONJava.com, 2002.
URL: http://www.onjava.com/pub/a/onjava/2002/07/17/tomcluster.html

[13] Srini Penchikala, “Clustering and Load Balancing in Tomcat 5, Part 1”, Article published on O’Reilly ONJava.com, 2004.
URL: http://www.onjava.com/pub/a/onjava/2004/03/31/clustering.html

[14] Srini Penchikala, “Clustering and Load Balancing in Tomcat 5, Part 2”, Article published on O’Reilly ONJava.com, 2004.
URL: http://www.onjava.com/pub/a/onjava/2004/04/14/clustering.html

[15] Filip Hanik, “Clustering Technologies – In Memory Session Replication in Tomcat 4”, Article on TheServerSide.COM, 2002.
URL: http://www.theserverside.com/tt/articles/article.tss?l=Tomcat

[16] T. Wilkinson, “Kaffe - A Free Virtual Machine to run Java Code”, 1998.
URL: http://www.kaffe.org

[17] Terracotta Clustered JVM, 2006
URL: http://www.terracottatech.com

[18] Apache Tomcat official web site.
URL: http://tomcat.apache.org/

[19] Apache Tomcat Wiki web site.
URL: http://wiki.apache.org/tomcat/PoweredBy

[20] Apache JMeter official web site.
URL: http://jakarta.apache.org/jmeter

[21] TPC-W benchmark specification, Transaction Processing Council.
URL: http://www.tpc.org/tpcw/

[22] JMOB - TPC-W Benchmark, Java Servlets version by ObjectWeb.
URL: http://jmob.objectweb.org/tpcw.html

[23] KjcJavaCompiler download web site; plug-in developed by Takashi Okamoto.
URL: http://www.koders.com (search “KjcJavaCompiler”)

[24] Valgrind memory debugging tool.
URL: http://valgrind.org

Appendices

Major Components of Tomcat 3 Servlet Engine

Component	Description and Function	Relevant Classes
Context Manager	The main entry point for tomcat execution. It coordinates the activity of most other functional modules.	ContextManager
Context	Represent a web application. It encapsulates all the properties defined in the web application descriptor (web.xml) and in the <Context> tag in server configuration file (server.xml). Context is associated with a Request after the contextMap() callback completes. By default, this mapping is done by the SimpleMapper interceptor.	Context
Container	Represent a group of URLs sharing a common set of properties. Container is associated with a Request after the requestMap() hook completes. SimpleMapper is again the core implementation of this hook, providing support for prefix, exact and extension mappings. Other interceptors can provide optimized mappings for particular subsets (like JspInterceptor) or implement custom mapping schemes.	Container
Interceptor	Represent the building blocks and extension mechanism for Tomcat. Most of the Tomcat functionality is implemented using modules. Modules operate on Tomcat's core objects and can hook in and extend Tomcat. Using Interceptor, one can control all aspects of request processing - parsing, authentication, authorization, sessions, response commit (before headers are sent), buffer commit (before any buffer is sent - it can be used to support HTTP1.1 for example).	BaseInterceptor ContextInterceptor RequestInterceptor
Servlet Wrapper	An object that wraps and invokes a servlet. It is responsible for loading and creating the servlet instance.	ServletWrapper
Standard Manager	Responsible for session management. Active sessions are stored in a hash table keyed by sessions identifiers. Expired sessions will be purged periodically.	StandardManager StandardSession StandardSessionInterceptor SessionSerializer

Request and Response	Contain all operations delegated to modules to call for processing requests and responses. Tomcat 3.2 exposes the internal buffers instead of using the Stream/Writer interfaces; core components will have direct control over the buffering and char/byte translation. HttpRequestAdapter / HttpResponseAdapter is the major implementation of the Request and Response interfaces.	Request RequestImpl Response ResponseImpl HttpRequestAdapter HttpResponseAdapter
Connector and End Point	Handle all the details related with TCP server functionality - thread management, socket accept policy, etc. As soon as it gets a socket, it just handles the stream to a handler by calling the handler's processConnection method. The major connector/endpoint is PoolTcpEndpoint / PoolTcpConnector which maintains a pool of threads for accepting incoming connections.	TcpEndpoint TcpConnection PoolTcpEndpoint PoolTcpConnector
Handler	Interface to enter the Context Manager for mapping to and calling the target servlet. It calls the readNextRequest method to read and parse the request URL and HTTP headers. Then it calls ContextManager.service() with a HttpRequestAdapter / HttpResponseAdapter pair which encapsulate the socket connection for response writing. The major handler being used is HttpConnectionHandler.	HttpConnectionHandler Ajp12ConnectionHandler Ajp13ConnectionHandler
Helper and Utility Classes	There are many utility classes in Tomcat for easing its development. Some of them serve the purpose of object pooling to avoid unwanted garbage collection overhead, e.g. MimeHeaderField, RecycleBufferedInputStream. The most important class in this category is Threadpool which is used by PoolTcpEndpoint for simultaneous socket accepting.	ByteBuffer MimeHeaders MimeHeaderField MessageBytes MessageChars PrefixMapper RecycleBufferedInputStream StringManager ThreadPool ThreadPoolRunnable

Table A-1: Major components of Tomcat 3 servlet engine

Tomcat-JESSICA2 Error Logs

Note: The logs are presented in their chronological order and classified by severity.

Log	Aspect	Description		
1.	**Problem:**	Segmentation fault will occur if JESSICA2 is compiled with gcc higher than 2.95.3.		
	Severity:	Low	**Dump:**	./runj2.sh: line 44: 31848 Segmentation fault $KAFFE $JOPT –Jport … $*
	Explanation:	This problem is due to missing memory protection.		
	Solution/ Workaround:	Workaround is to use gcc 2.95.3 to compile JESSICA2. Later Kaffe fixes on Linux memory protection (mprotect) are applied, now it can be compiled using gcc 3.x without causing runtime error.		
2.	**Problem:**	Illegal monitor state exception		
	Severity:	Critical	**Dump:**	java/lang/IllegalMonitorStateException Kaffe: gos.c:1976: getMasterObj: Assertion ‘buf’ failed.
	Explanation:	The problem is caused by the handling of Object.wait() with timeout specified. The original JESSICA2 doesn't acquire the lock again when the wait(timeout) is timeout. This results in an inconsistent lock state and causes the error. Since the master will terminate after it gets an illegal locking exception, the GetMasterObject() in the worker node will get nothing from the socket that was closed by the master. Then it returns wrong data and terminates too.		
	Solution/ Workaround:	lock.c is fixed accordingly.		
3.	**Problem:**	Zero-size array problem in GOS		
	Severity:	Critical	**Dump:**	"Assertion `fidx < nrTypes && size 0' failed."
	Explanation:	This is caused by allocating zero size memory in gos.c for array cache. Java accepts zero-sized array. But a mistake in handling of zero-sized array causes this error. The original code will skip the trailing word "-1" when it meets a zero-sized array.		
	Solution/ Workaround:	gos.c is fixed accordingly.		
4.	**Problem:**	Function to disable interrupts failed.		
	Severity:	Critical	**Dump:**	Kaffe: exception.c:372: dispatchException: Assertion `!intsDisabled()' failed. Kaffe: gos.c:1982: getMasterObj: Assertion `buf' failed.
	Explanation:	The getMasterObj error is caused by the failure of the master node. The first error message "initsDisabled()" is the root cause. It is probably caused by some unmatched lock/unlock.		
	Solution/ Workaround:	gos.c is fixed accordingly.		

5.	**Problem:**	Garbage collector encounters invalid object pointers.		
	Severity:	High	**Dump:**	"Kaffe: mem/gc-incremental.c:873: gcMalloc: Assertion `fidx < nrTypes && size 0' failed."
	Explanation:	The error is caused when many requests are sent to Tomcat. It is a bug in original Kaffe which was fixed in its later version in creating daemon threads. They assign the function pointers in some fields of the thread object for temporary use, which GC collector assumes to be some valid Java objects. In normal cases, the problem won't happen since the daemon threads are not created often and the GC collector won't happen to run in the middle of the thread creation. In our case, when more requests arrived at tomcat, the GC will be triggered during thread creation, which will encounter invalid object pointers.		
	Solution/ Workaround:	Fixes in later Kaffe version are applied to several source files – the thread creation function is modified.		
6.	**Problem:**	MySQL Connector/J 3.0 JDBC Driver cannot run on JESSICA2.		
	Severity:	Low	**Dump:**	"Kaffe: gos.c:481: updateObjData: Assertion 'class' failed."
	Explanation:	This is because the worker node cannot locate the class files of the JDBC driver. Explicit classpath setting has to be in place.		
	Solution/ Workaround:	Add the driver's JAR file to the classpath environment variable on the worker node.		
7.	**Problem:**	Tomcat cannot compile JSP when running on JESSICA2.		
	Severity:	Medium	**Dump:**	java.lang.NoClassDefFoundError: sun/tools/javac/Main
	Explanation:	JSP is compiled into servlet before execution. By default, JSP compiler in Tomcat is Sun Java Compiler. Sun Java Compiler is NOT supported by JESSICA2.		
	Solution/ Workaround:	Solving this problem needs upgrade of both Tomcat and JESSICA2. We use Kjc, an open source compiler to be the JSP compiler in Tomcat. The steps below should be followed: • Search, download KjcJavaCompiler.java from www.koders.com. • Put it in the folder share\org\apache\jasper\compiler. • Modify WebXmlReader.java in share\org\apache\tomcat\context to set Kjc compiler as the default JSP compiler. • Rebuild Tomcat using Ant; • Add the Kjc compiler JAR file (version: kopi-1.5B) to the classpath. After this fix, JESSICA2 could still fail in some JSP applications due to two more reasons. (1) JESSICA2 does not support namespace in the GOS to distinguish classes loaded by different loaders, i.e. all classes will be shared by different loaders. Since the classloader for loading certain JSP applications is changed for some classes, worker nodes will fail to locate and load them. Fix on classPool.c to bypass classloader checking is needed to let workers load classes correctly. (2) The classpath at worker node is not updated to include the compiled JSP classes if the compilation takes place at another node. This problem can be worked around by including the working directory (usually $TOMCAT/work/localhost_8080%2f[context-name].) which contains the compiled JSP classes.		

8.	**Problem:**	When the host manager calls handleGetHostAddr, segmentation fault will happen.		
	Severity:	Critical	**Dump:**	Program received signal SIGSEGV, Segmentation fault. handleGetHostAddr (hid=1, fd=46, buf=0xa42f5b0 "") at hostman.c:1211 1211 id = *(hid_t*)(buf+sizeof(int));
	Explanation:	This is because of incorrect free up of buf in the function.		
	Solution/ Workaround:	hostman.c is fixed.		
9.	**Problem:**	Reported by our research teammate, SOR benchmark suffered from slowdown of execution in remote thread by over 10 times.		
	Severity:	Critical	**Dump:**	N/A
	Explanation:	This is a small coding mistake of enableFastCheck() in gos.c.		
	Solution/ Workaround:	Add back the mistakenly commented line in enableFastCheck(): CLR_RD_CACHE(ch->obj->cache)		
10.	**Problem:**	Worker nodes calling updatePrimArr and updateRefArr will fail.		
	Severity:	High	**Dump:**	Program received signal SIGSEGV, Segmentation fault. 0x005a0827 in updateRefArr (obj=0x8, lower=7, upper=8, ptr=0x9156c77 "", elclass=0x85af680) at gos.c:633 assert(lower <= upper && ca->lower <= ca->upper);
	Explanation:	This problem will happen if object home migration is enabled. The GOS may not pack byte array properly.		
	Solution/ Workaround:	Tomcat is modified and successfully bypassed this error. Future review on the packing functions in the GOS is needed.		
11.	**Problem:**	The GOS failed in packing array with a null element.		
	Severity:	Critical	**Dump:**	Program received signal SIGSEGV, Segmentation fault. pack_string_data (commbuf=0xaed1010, obj=0x0) at gos.c:99 assert(!IS_CACHEOBJ(obj) && OBJECT_CLASS(obj) == StringClass);
	Explanation:	This is due to missing protocol implementation for null string. The pack_string_data function has not catered null representation.		
	Solution/ Workaround:	Modify pack_string_data and unpack_string_data in gos.c. Use "-1,$" to represent a null string over the GOS.		
12.	**Problem:**	The GOS failed to pack TomcatLogger$LogEntry.		
	Severity:	High	**Dump:**	Program received signal SIGSEGV, Segmentation fault. pack_class_sig (commbuf=0xa948010, class=0x104689ff) at gos.c:169

	Explanation:	LogEntry class contains a Throwable field. So the GOS is trying to pack a Throwable object which has an instance field called backtrace which is constructed using Kaffe native function buildStackTrace which returns a C structure “stackinfo”. Kaffe just casts it to be Hjava_lang_Object but the allocated memory is not Java object at all.		
	Solution/ Workaround:	Comment out the use of buildStackTrace in Throwable.c and Exception.c and set the backtrace field to null.		
13.	**Problem:**	When Tomcat is run with a larger heap size (-ms128m -mx256m), Tomcat cannot start up as error occurs at the method addZoneFiles in the class Java.util.TimeZone. In some cases, invalid host id is resulted.		
	Severity:	Medium	**Dump:**	Program received signal SIGSEGV, Segmentation fault. addZoneFiles__Q34java4util8TimeZonePQ34java4lang6StringPQ34java2io4File () at TimeZone.java:97
	Explanation:	This is due to the Kaffe implementation of TimeZone.java will recursively open all the found time zone files on Linux. There are over a thousand of such files. If we change the heap from default 64MB to a larger size, no garbage collection occurs and the maximum 1024 fd’s that can be used will be exceeded. Wrong fd of deliberately large value will be returned and causes invalid host id which is extracted from the 1st half word of fd.		
	Solution/ Workaround:	Port later Kaffe implementation of TimeZone and UNIXTimeZone to JESSICA2.		
14.	**Problem:**	MySQL JDBC driver failed at send buffer command.		
	Severity:	Medium	**Dump:**	Program received signal SIGSEGV, At MysqlIO.java:1762
	Explanation:	Problem has occurs during ChartoByte conversion in MySQL. GOS seems failed to pack a byte array in SingleByteCharsetConverter in MySQL driver. The charToByteMap contains all ‘?’ and make SQL statements cannot be executed.		
	Solution/ Workaround:	Tweak the setClassOwner function in classMethod.c. If we see the class name starts with ‘com/mysql’, then we skip remoteGetStatic.		
15.	**Problem:**	Apache SOAP engine failed to run on Tomcat-JESSICA2.		
	Severity:	Medium	**Dump:**	Program received signal SIGSEGV, Segmentation fault. 0x005fd170 in utf8ConstUniLength (utf8=0x0) at utf8const.c:295 const char *const end = ptr + strlen(utf8->data);
	Explanation:	The GOS seems failed to pack java.lang.Class. The class entry name is null and causes segmentation fault.		
	Solution/ Workaround:	The workaround is also to Tweak the setClassOwner function in classMethod.c. If we see ‘org/apache/soap’, ‘com/sun/activation’, ‘com/sun/mail’, ‘javax/mail’ and ‘javax/activation’, skip calling remoteGetStatic.		

16.	**Problem:**	Invalid fd is resulted when running Apache SOAP.		
	Severity:	Medium	**Dump:**	Error was: java.io.IOException: Unknown error 4294967295
	Explanation:	SOAP is trying to open /soap-2.3.1/webapps/soap/soap.xml which is not present in the directory. SOAP will catch any IO Exception and use a default configuration. The java_io_FileInputStream_open function should raise a java.io.IOException when file cannot be found. But this is missing in the code.		
	Solution/ Workaround:	Fix java_io_FileInputStream_open to throw java.io.IOException if return code, rc > 0.		
17.	**Problem:**	When a SOAP application writes a random access file, the JVM will be aborted due to allocation of zero memory size in heap.		
	Severity:	Medium	**Dump:**	Program received signal SIGABRT, Aborted. #3 0x002c12f8 in __assert_fail () from /lib/tls/libc.so.6 #4 0x008b1e91 in gcMalloc (gcif=0x92d960, size=0, fidx=12) at mem/gc-incremental.c:873 #5 0x008b27a2 in jmalloc (sz=0) at gc.c:21
	Explanation:	The failed function is java_io_RandomAccessFile_writeBytes. In some cases, zero-length byte will written and this results in allocating a zero-size memory in the heap which is not permitted.		
	Solution/ Workaround:	This function should check if len > 0 before executing buf = KMALLOC(len). java_io_FileOutputStream_writeBytes is fixed alike.		
18.	**Problem:**	High stress to Tomcat will exception related to Threadpool.		
	Severity:	High	**Dump:**	N/A
	Explanation:	Unknown		
	Solution/ Workaround:	Tomcat is reengineered to use multiple work queue thread pool and this stability problem is bypassed.		
19.	**Problem:**	Thread migration is problematic in debug mode.		
	Severity:	Medium	**Dump:**	__assert_fail () from /lib/tls/libc.so.6 pack_frame (commbuf=0xa5f6010, btx=0xa3a8710) at migration.c:1905 start_migration () at migration.c:2029
	Explanation:	GDB could contaminate the stack to migrate. Assertion failed in the setupFrame and pack_frame functions used to double-check the stack content inferred by JIT recompilation because the stack states are inconsistent. This could be due to GDB may could insert dummy frames onto the stack for debugging and confuse frame packing during thread migration.		
	Solution/ Workaround:	Now the only workaround is not to use debug mode when dynamic thread migration is being used. Future support for debugging thread migration can be considered.		

Table A-2: List of Tomcat-JESSICA2 error logs