# JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support

Wenzhang Zhu, Cho-Li Wang and Francis C.M.Lau
*Department of Computer Science and Information Systems*
*The University of Hong Kong*
*Pokfulam, Hong Kong*
*{wzzhu+clwang+fcmlau}@csis.hku.hk*

## Abstract

JESSICA stands for "Java-Enabled Single System Image Computing Architecture". It is a distributed Java Virtual Machine designed to provide a high-performance parallel execution environment for multithreaded Java applications on clusters. JESSICA achieves *single system image* by hiding all the distributed aspects of cluster computing with a *global object space* for the user's Java program, thus makes the user do not need to care which node any Java thread of the program is running on. Because the physical location of thread execution is transparent to the Java program, a thread migration mechanism was built within JESSICA to enable dynamic load balancing at runtime.

In this paper, we present JESSICA2, which is a new distributed Java Virtual Machine developed based on our previous JESSICA system. JESSICA2 implemented a *cluster-aware Java execution engine* modified from the latest Kaffe JVM 1.0.6, which allows Java thread to be migrated when it runs in Just-in-Time compiler (JIT) mode. We replace the DSM-based global object space of JESSICA with a more efficient object-oriented solution, which is tightly coupled with the JESSICA2 thread migration system in the JVM layer. This makes the new thread migration mechanism able to distribute the Java threads among cluster nodes without using DSM to share the Java thread execution context. In addition, the new global object space implementation is featured by its *non-blocking* object access support that can effectively avoid the common blocking nature in traditional DSM. Several micro-benchmarks and Java applications have been tested on JESSICA2. Significant performance improvement was observed.

# 1. Introduction

A *distributed Java Virtual Machine* (DJVM) is able to execute multithreaded Java applications on parallel or distributed platforms, while providing the *Single System Image* (SSI) [1] illusion to the Java threads. DJVM has been identified as an ideal platform for the execution of parallel or distributed applications [2,3,5,6,15].

Existing prototypes of DJVMs fall into two catalogs. One approach fully relies on the common cluster infrastructure such as *Distributed Share Memory* (DSM) system to support the parallel execution of Java threads on clusters. The original JVM needs only minor changes to turn it into a DJVM. Java/DSM [3], Hyperion [5], Jackal [15] and our previous project JESSICA [2] are examples of such type. This approach simplifies the design and implementation of DJVM as all distributed Java threads can transparently access objects on the common object space created by the underlying DSM. Besides, thread synchronization and object consistency are managed by the DSM's locking/unlocking mechanisms and data consistency protocols. However, such a layered design has put constrains on the DJVM to develop further optimizing techniques to support efficient object sharing among distributed Java threads, mainly due to the mismatching of memory models between Java and the underlying DSM [19].

The other type of DJVM modifies JVM in order to enable its distributed computing power, i.e. this type of DJVM is *cluster-aware*. The famous example is cJVM [6] developed by IBM Research & Development Labs in Haifa. This kind of approach needs careful considerations on different aspects of the DJVM such as distributed class loading, shared object placement and access, distributed thread management and synchronization, etc.. The major advantage of such approach is that it can make use of the Java semantics to optimize the execution of Java threads on cluster environments. For example, efficient shared object access, flexible class loading and thread management can be built inside the DJVM. However, to our best knowledge, the cJVM prototype was implemented by modifying Java interpreter. Such approach has the major performance weakness inheriting from the slow Java interpreter execution and may not be efficient enough for solving computation-intensive problems.

On the other hand, as the physical boundaries between cluster nodes have disappeared among the distributed Java threads through the support of DJVM, there is a need of a thread migration mechanism built within the DJVM to enable dynamic load balancing by migrating Java threads between cluster nodes at runtime without programmers' involvement. A DJVM having this thread migration feature can handle the thread distribution more flexibly than other means like the static thread allocation. The transparent migration mechanism also provides a fine-grained mechanism to distribute the computation among the cluster nodes to help optimize the resource utilization in the cluster.

Among various DJVMs, JESSICA [2] is the only existing DJVM that can support Java thread migration. JESSICA can execute multithreaded Java applications on top of clusters without any modification of Java application source codes or bytecodes. JESSICA was built on top of a page-based DSM Treadmark [7] to provide an SSI view to Java applications. A thread migration mechanism called *Delta Execution* was implemented based on Kaffe JVM 0.9.1 [8] to support the preemptive migration of Java threads [4]. However, JESSICA can only migrate thread while executing in the interpretation mode.

The JESSICA2 is our new DJVM that goes far beyond the previous JESSICA and tries to improve the capabilities of DJVM in many aspects. At this stage of research, JESSICA2 aims at achieving the following goals:

1. **High performance**. Our DJVM aims to provide a high-performance computing platform for running multithreaded Java programs. Thus, the DJVM should be able to

execute Java applications in JIT compiler mode to gain the better performance than executing in interpretation mode. Moreover, it should be able to leverage the collective computing power of clusters for the execution of the multithreaded Java programs as compared to the single-node JVM.

2. **Transparency**. The multithreaded Java program can be run on the DJVM without any modification. The cluster environment is totally transparent to the Java programs. The threads created by the Java program will neither know nor need to know the location it is running on during its execution. Any need to modify the existing Java application source programs will violate the transparency requirement and discourage the use of proposed DJVM.

3. **Efficient Thread Migration**. Our DJVM will provide a mechanism to support the dynamic distribution of Java threads among the nodes of clusters at runtime. The feature can efficiently handle the workload imbalance problem while running multithreaded Java applications on clusters.

Our prototype uses the latest Kaffe JVM 1.0.6 [8] and runs in a Linux cluster. We introduce a new cluster-aware Java execution engine, *JITEE*, which supports the execution of distributed Java threads in JIT complier mode. This results in a major improvement in performance than the old interpreter-based implementation. New thread migration mechanism is implemented in JITEE to distribute the Java threads among cluster nodes without using DSM to share the Java thread execution context. Rather the migrated Java thread execution context is carefully captured at the bytecode boundary and it is then translated into a machine-independent text format and can be restored by a remote node. A new *global object space* (GOS) layer using portable object format for exchanging object data is implemented to support the access of shared objects between the distributed Java threads.

The rest part of the paper is organized as followed. Section 2 presents the overall architecture of JESSICA2. We discuss the details of Java thread migration mechanism and the shared object access in section 3 and section 4 respectively. Section 5 shows the results of our prototype system. The related works are discussed in Section 6 and a short conclusion is given in Section 7.

## 2. JESSICA2 Architecture

Figure 1 shows the overall architecture of JESSICA2. JESSICA2 runs on a cluster environment and it consists of a collection of modified JVMs that run on different cluster nodes. We distinguish a node that starts the Java program as the *master node* and the JVM running on it as the *master* JVM. The main thread starts execution on the *master* JVM and terminates on the same JVM. All the other nodes in the cluster are the *worker nodes* running a *worker* JVM to participate in the execution of a Java application. The *load monitor* is an independent process that runs on any node of the cluster. The *load monitor* is responsible for monitoring the system load of the cluster and scheduling Java thread migration. In JESSICA2, all the nodes can serve both as a migration source and a migration target. The Java threads in the application can migrate from one node to another node upon receiving requests from the *load monitor*.

Each modified JVM on the cluster node uses the modified Java bytecode execution engine, JITEE, to execute the threads. JITEE is aware of the cluster environment and it is the core of the JESSICA2 for achieving a single system illusion to the running threads. The Java thread migration mechanism is built inside JITEE to support the mobility of Java threads at the bytecode boundary. The details of this mechanism will be discussed in section 3.

The *global object space* layer is embedded inside the JVM to enable the shared objects access among the Java threads running on different nodes of the cluster. Two main kinds of operations are included: the data access and thread synchronization operations. The data access operations correspond to the bytecode instructions that access the class static data (GETSTATIC/PUTSTATIC), object fields (GETFIELD/PUTFIELD), and the array components (XALOAD/XASTORE). The thread synchronization operations implement the Java thread synchronization primitives such as *lock()*, *unlock()*, *wait()*, *notify()* and *notifall()* on remote Java objects. The details of the design and implementation will be discussed in section 4.

A daemon thread called *host manager* is running inside the JVM to manage the cluster nodes and provides basic communication supports for GOS and the JITEE. The *host mangers* in different JVMs communicate with each other through TCP connections.

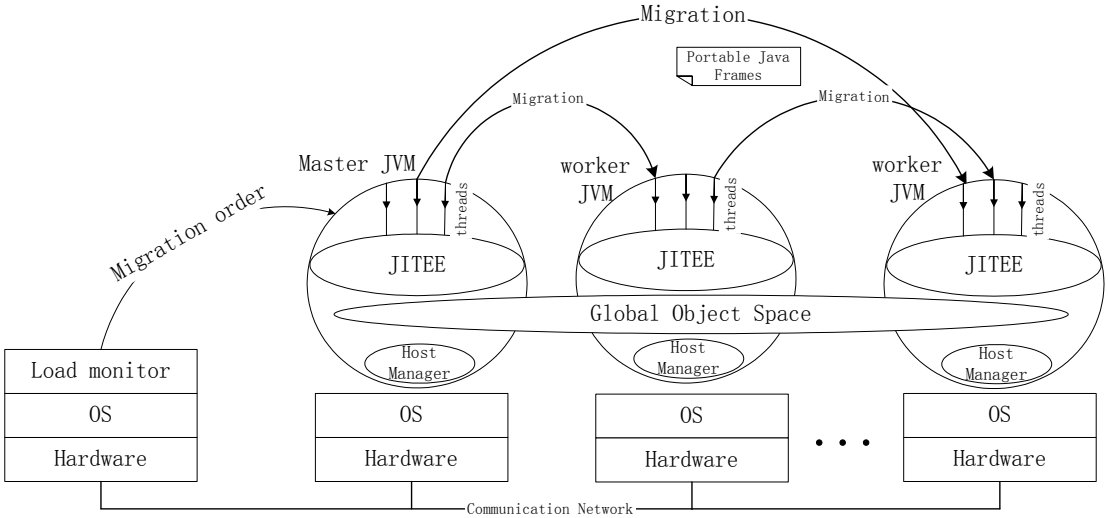


**Figure 1. An Overview of JESSICA2 System Architecture**

# 3. Transparent Java Thread Migration in JIT Compiler Mode

## 3.1 Overview

The JVM specification [9] defines various runtime data structures that are used during execution of a Java program which together form the execution context of a Java thread. The data structures include: the heap, the method area, and JVM stack.

In our new design, we have abandoned the use of existing infrastructures like a page-based DSM because of the performance and portability reasons. Rather, we implement the GOS to store only the shared Java objects in the heap. Under the new environment without a DSM, we should apply different strategies to different types of Java thread context.

As the global heap for the DJVM will be realized by the embedded GOS, this relaxes the complication by shipping only the Java object references (together with its host id to form a global id) without actually moving the object data to the remote node during the Java thread migration stage. The GOS will handle these object data movements once the migrated thread restarts its execution on the remote node and need to access the objects.

For method area, we need some special treatments on the class loading to preserve the correctness of rebinding the class structures on the remote machine. Class loading in Java is a complicated issue in DJVM. [6], we need to classify the properties of different steps during the class loading. In our DJVM, we follow the similar class loading of common DJVM prototypes such as cJVM and Hyperion JVM, by partially loading each class independently on each JVM but preserve a single copy of static data for non-system classes.

The remaining problem is the JVM stack for Java threads. It is the key to supporting thread migration in Java. The thread migration mechanism involves two main operations, i.e., capturing and restoring JVM stacks of a Java thread. We use JITEE to generate efficient native codes for managing the thread execution context at runtime and extend the JVM thread system to support stack capturing and restoring. To be portable, the thread context captured is translated into a machine-independent text format and is able to be restored by the target JVM through its *host manager*.

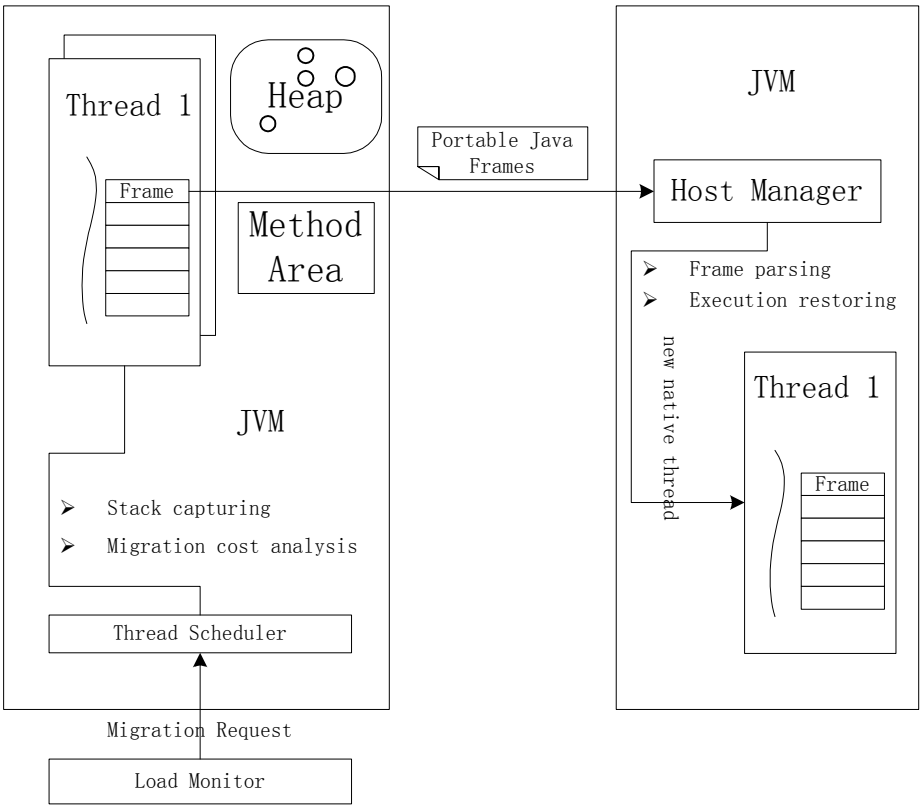


**Figure 2. Transparent Java Thread Migration Process**

Figure 2 shows the control flow of the Java thread migration mechanism. The migration journey starts from the request sending from the *load monitor* through TCP connection to the overloaded JVM. Upon receiving the migration request, the thread scheduler in the execution engine will analyze the stack of the running threads. The stack analysis function will scan the stack of each running thread, and estimate the cost of the migration for each thread. In our current prototype implementation, we use a simple heuristic function to evaluate the cost based on the number of frames and number of Java object references in the stack frame. The thread with lowest cost will be chosen as the candidate. The candidate thread will be suspended and the thread stack will be captured which will then be translated into a machine-independent text format. The portable stack frames will be shipped to the remote *host manager* of the target node. Upon receiving such frames, the host manager in the target JVM will call the migration

manager to parse the frames into the intermediate format after resolving the class and variable types. A new native thread will be created and associated with the migrated Java thread object. This thread will be given an argument pointing to the intermediate frames and bootstrap the thread execution.

### 3.2 Stack capturing

The JIT compiler makes the execution context of Java thread more complicated to capture compared to the interpreter, although the definition of the execution context of a Java thread is the same for both interpreters and JIT compilers. We identified the following issues to be solved within JIT compilers:

(1) In JIT mode, the thread runs its native codes generated by the JIT compiler. The PC register in an interpreter will point to the Java bytecode of the corresponding method while in a JIT compiler it points to the native codes instead. We need to capture the Java thread context at the boundary of bytecodes instead of native codes so that the PC is the pointer to the bytecode. But when a thread is chosen by the thread scheduler as the candidate to migrate, it is most likely running at some point of native codes that is not in the bytecode boundary. We should be able to "slide" the execution to the point of bytecode boundary.

(2) As JVM is a stack-oriented machine, one of the important tasks of a JIT compiler is to allocate registers for Java variables in the register-based machines [10,11]. As a result, the local variables or stack variables in a method may be loaded into specific registers during the execution of native codes generated by a JIT compiler. Moreover, the subsequent operations on the variable may take place in the allocated registers, which never happens in the case of an interpreter. The problem has the same characteristics as the optimized code debugger. But it is impractical for us to store heavy data structures to support full debugging of the bytecodes in the execution engine. We should be able to tackle the register information in JIT mode.

(3) As the stack variables are dynamically pushed into or popped from the thread stack, the types of specific stack slots cannot be determined in advance. To encode the variable in a machine-independent format, it is required that the types of the variables are known at the time of thread migration. In [12], it is proposed to use a separated type stack operated synchronously for interpreter during thread execution. Although such method can be used in the case of JIT compilers, it doubles the operation time to access the stack variable. New efficient methods suitable for processing stack variables in JIT compiler mode are needed.

(4) Lastly, the JVM stack in an interpreter is defined explicitly in the internal data structures of a JVM. But for a JIT compiler, the stack is implicitly managed by the native codes generated. Moreover, one single running stack for a Java thread is often shared among the Java methods, the native Java methods, and the internal JVM functions (including the JIT compiler). Therefore, an efficient management of the JVM stack in JIT mode should be called for.

To address the first two problems, we limit the migration to take place at some specific points. During the native codes generation for Java methods in the JIT compiler, we insert codes that *spill* the machine registers to the memory slots of the variables and check the migration request at such points. The candidate threads to migrate, when running to such points will find out that they are requested to migrate, and will call the appropriate functions to do the stack capturing and migration. The resulting effect is like that the thread slides to the safe point before migration.

However, it needs careful plan to insert such checkpoints. More checkpoints will increase the migration response time. In addition, it will expand the native code size and slow down the program execution. In our design, we choose two types of points as the migration checkpoints in the native codes: (1) the start of an invocation to a method of application classes. (2) the start of a *basic block* [16] that pointed by a back edge. The reason for first choice is mandatory as the migration request may happen in the called method in which case the memory slots of the variables of previous stack frames must have the latest values. The reasons for the second choice are: firstly, we want the size of register spilling codes to be minimized because there are only rather limited global registers to spill at the start of a basic block. Secondly a basic block pointed by a back edge often leads to a loop in the program. Therefore, it is able to stop the running threads at a proper time, e.g., before entering a long loop. Also note that the checkpoints are not inserted in the classes in Java standard libraries.

To tackle the third problem, we choose to do the *type spilling* at the migration checkpoints discussed above. The type information of stack variables at such points will be gathered at the time of bytecode verifying before compiling the Java methods. We use one single type to encode the reference type of stack variable as we can deduce the real type of Java object from the object reference. We choose one encoding for each of primitive types except that sub-integer types like byte are treated as integer. Therefore, we can compress one type into 4-bit data. Eight compressed types will be bound in a word, and an instruction to store this 32-bit machine word will be generated to spill the information to appropriate location in the current method frame. For typical Java methods, only a few instructions are needed to spill the type information of stack variables in a method, which results in better performance improvement than the synchronous type stack method.

The fourth problem is solved by generating native codes that link the Java thread stack dynamically upon method invocations. The execution stack of a Java thread is interleaved with Java method frame and the internal JVM functions frames we call *C frames*. In our thread migration, we choose the consecutive Java frames to be migrated to the remote machine. Upon completion of such Java frames, the control will return back to the source machine to complete the C frame execution.

### 3.3 Stack restoring

The stack restoring needs to recover the machine registers in the migration target node. Figure 3 shows the procedures of restoring a thread stack. The steps are shown with the numbers before them as the order.

The first step is to parse the text frames into the target memory data structures. Incorrect format will be rejected during this step. In our implementation, a parser for the stack frames written in YACC program is used. The next step is the Java method compilation for each stack frame. As the execution engine only automatically invokes the JIT compiler upon method invocations, we need to call the compiler manually as the frame has already been called on migration source node. During this compilation, extra information about the register allocation at the restoring point will be extracted. For those methods that have already been compiled, a lightweight partial compilation function will be called to get the register allocation information.

The third step is just to manipulate the machine call stack so that when all the migrated frames are finished, the control can be returned to the completion handler. The fourth step will setup the Java stack frames according to the machine architecture. During the stack frame setup, for each Java frame, a small code stub to recover the register based on the register allocation information extracted in step 2 at the restoring point will be inserted just before control is returned to the current frame. In step 5, we set the right stack point and jump to the entry point the register

recovering code stub of first Java frame and start the thread execution. When control returns from last Java frame, the completion handler will pack the return results and call the migration manager function in the *host manager* to send back the data to the migration source node of the Java thread.

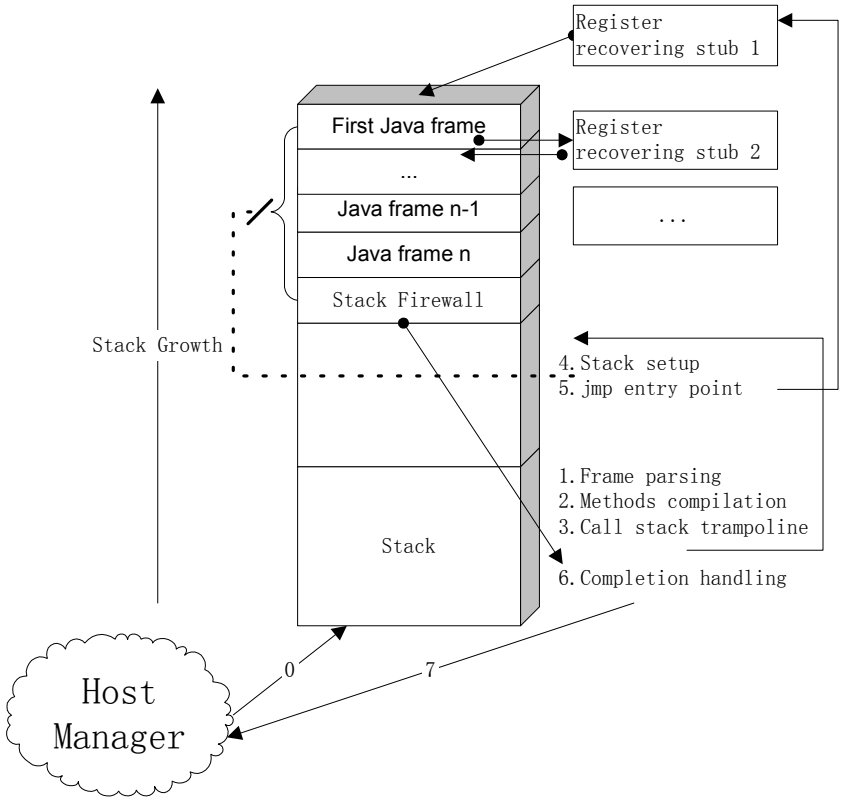


**Figure 3. Java Thread Execution Restoring in JESSICA2.**

# 4. Global Object Space

As we mentioned before, our previous project JESSICA uses a page-based DSM as the global heap. However, the page-based DSM can't be tightly coupled with JESSICA's thread system. For example, the page-based DSM uses the hardware page fault mechanism to activate the remote object access. When a page fault happens to fetch the remote data, the whole DSM system will block all the threads on the current node. Such case will result in great performance loss. On the other hand, the paged-based DSM can't be tightly coupled with Java language. This is because Java is an object-oriented programming language. The access unit in a Java program is an object or more precisely, an object field. A paged-based DSM will inevitably suffer from false sharing when multiple irrelevant objects are allocated in the same memory page.

To support the access of shared object between the JVMs, we built a new GOS layer using portable object format for exchanging object data. This layer is embedded in the JVM and provides a single heap illusion to the Java programs.

## 4.1 Memory model and object access

The JVM specification [9] defines the Java Memory Model (JMM) to constrain the memory behavior of Java threads. The JMM has been criticized [13] and the new JMM is still under

peer review [20]. Based on the common understanding of the JMM, multithreaded Java programs assume that there is a single heap visible to all the threads. The heap stores all the master copies of objects. Each thread has a local working memory to keep the data of objects in the heap that it must access. In a single-node JVM implementation, this working area can be regarded as the machine registers. When the thread starts execution, it operates on the data in its local working memory. Java threads use *monitor* to synchronize the concurrent thread execution in a critical section. When entering a monitor, the thread must flush its working memory to the heap to ensure that Java thread can access the latest object data in the critical region. When exiting the monitor, the modifications of objects inside the working memory must be reflected in the heap.

In JESSICA2, we follow the above understanding of JMM to implement our GOS layer as the global heap. Each JVM in the DJVM will contribute a portion of its heap, the *master heap area*, to the global heap area as shown in Figure 4. Another portion of heap, the *cache heap area*, in a JVM's heap is used for storing cached remote objects. For all the threads inside one JVM, the access to objects in *master heap area* is just the same as the simple-node JVM implementation, i.e., all the objects in the area are the master copies and can be loaded directly into the thread's working memory. The access to the objects in *cache heap area* will be handled by the GOS layer. The *cache heap area* is slightly different to thread working memory in the JMM definition. However, by making this area to be per-thread data structure, it can be viewed as an extension to the local machine registers and therefore behaves just the same as the definition of JMM.

Upon entering a monitor, the working memory of the running thread will be flushed. The modifications of objects will be sent to the nodes where the master copies reside and the status of the object will be set invalid. Later access to invalidate cache object will need to request from the master copy. Upon exiting a monitor, the modifications of objects will also be sent to the master copy site.

To provide quick access to an object, we didn't use an opaque handle to represent an object. Instead, an additional pointer pointing to the global definition of the object is added to the header of an object and it is used to distinguish the master copy and the cached copy. An object with null global definition pointer stands for a master copy. A simple check instruction on this pointer field in the native codes, generated by the JIT compiler, yields the type of object immediately. With the pointer, it is faster to determine the global object address than the searching in a centralized object table.

The quick flushing of the per-thread working memory is done through a per-thread hash table. We use per-thread cache instead of letting all the local threads share a whole cache. The advantage of former can be explained in the following points:

1. It is closest to the definition of JMM by viewing the per-thread cache area as the extension of machine registers.
2. The access to the per-thread cache is free of contention. As thus, the per-thread cache need not be guarded by locks.
3. Using the per-thread caching, when there are multiple Java threads executing at the same JVM, one thread will not invalidate other threads' fresh cache copies. Therefore, it won’t interfere other threads’ execution.
4. The size of per-thread structure will be smaller and it is faster to scan the whole data structure at the time of flushing.

In JESSICA2 when an object is created in a thread, the JVM that runs this thread will be considered as the *home* of the object. The object will be allocated in the *global heap area*. The later exposure of the object to other threads executing in other JVMs will require a translation of this object reference into a global address pair <host id, native address>. The remote threads

when seeing this reference for the first time will allocate a cache copy of this object in its *cache object area* and associates a pointer to a global definition of this object. For array objects, the dimension and the element type of the array are also transferred.

We use portable format to transfer our object data. All field data in an object together are transferred with their unique identifications. The unique field identification is in a way similar to the class constant pool index and it is more portable than using the memory offset. The primitive types data use little-endian encoding and the object reference uses the <host id, native address> as the global identifier during the data exchange.

To hide the communication latency, we use the threaded-IO interface inside the JESSICA2 to transfer the object data. When one thread is blocked in sending object data, the thread will yield the CPU and let other thread in the local JVM continue the execution.

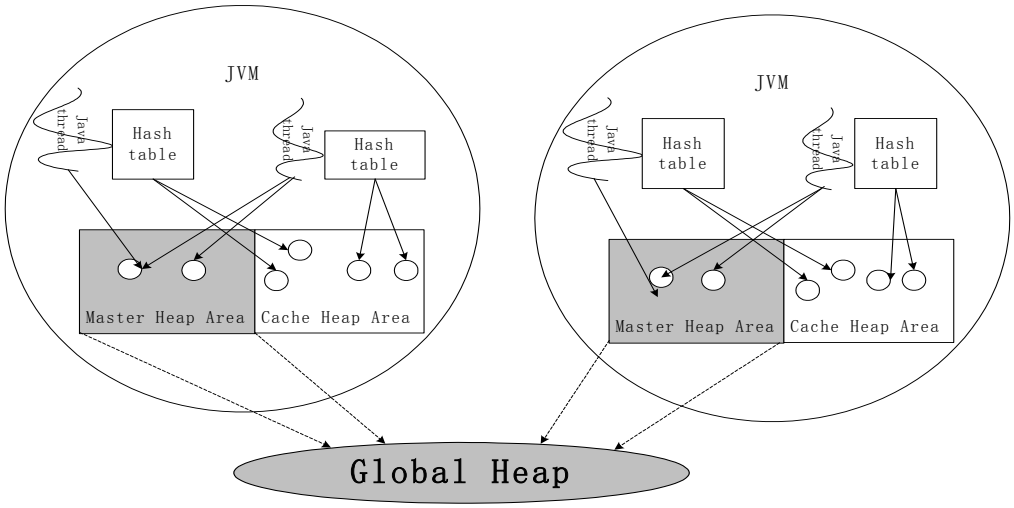


**Figure 4. The Internal Architecture of the Global Object Space**

## 4.2 Thread Synchronization

The thread synchronization in the global object space implements in a distributed manner the Java synchronization primitives including *lock()*, *unlock()*, *wait()*, *notify()* and *notifyAll()*. As Java synchronization takes place on a Java object, we fix the JVM that owns the master copy to be the synchronization handler of this object. The JVM will be the *home* of the lock. The threads in the home JVM will do the synchronization in the normal way on the object. The threads in the remote JVM, when trying to perform synchronization on a cache object, will send the synchronization request to the home JVM. The home JVM will place the request of remote threads in the proper queue of the synchronized object and return the synchronization result to the remote thread.

## 4.3 Garbage Collection

At the current stage, we use conservative strategies to handle the garbage collection. We don't employ global garbage collection to clean up un-used objects in the global heap. Instead we have each JVM do the garbage collection in its own heap. In a local JVM heap, the objects in the *cache heap area* will not be garbage collected. Also the objects in the *master heap area* that has been exposed to remote nodes will not be scanned by the original garbage collector. The rest of the objects in the *master heap area* will be garbage collected in the normal way.

In our assumption, the number of objects shared by different threads will be quite limited compared to the normal objects that are not thread-escaped. Therefore, by using our garbage collection strategies, we won't greatly affect the memory saving resulted from garbage collection. And the advantage of our strategies is that we don't need to introduce global synchronization among the JVMs to do the garbage collection, which otherwise will slow down the whole system.

# 5. Performance Results

Our DJVM has been implemented based on Kaffe JVM 1.0.6 on 540MHz Pentium-II clusters running Linux 2.2.14 kernel connected by Fast Ethernet.

## 5.1 Microbenchmarks on Java Thread Migration

We perform microbenchmarks to measure the cost of the transparent Java thread migration by timing the migration process through i386 real time-stamp counter. The cost will be divided into three parts. The first part is the constant cost to create the native thread. In JESSICA2, we use Kaffe-1.0.6 green thread system. It takes about 432 us to create a new native thread.

The second part of the cost is related to the length of the frames. This part includes the stack capturing, frame parsing, class and method resolution and the frame setup. Table 1 shows the measurements to these operations for different size frames.

| Time (in us) | Stack capturing | Frame parsing | Class and method resolution | Frame setup |
|---|---|---|---|---|
| 1 frame (12 variables, length=475 bytes) | 232 | 166.89 | 3,431 | 9.6 |
| 2 frames (17 variables, length=482 bytes) | 437 | 328 | 13,747 | 13 |
| 11 frames (92 variables, length=3049 bytes) | 12,993 | 1,383 | 227,587 | 49 |

**Table 1. Timing Breakdowns on JESSICA2 Thread Migration Operations**

The third part of the cost is caused by the network communication to transfer the frames. In JESSICA2, we use TCP connection to access the remote object and in our measurements. It takes totally about 5ms to setup a TCP connection, send a 1KB frames, and receive an ACK message.

## 5.2 JESSICA2 versus Kaffe 1.0.6

The migration mechanism will place the checkpointing instructions in the native codes generated. However, this kind of native code instrument occurs only on the methods of application classes. According to our measurements, the increase in code size is less than 1%. However, the object checking has increased the total native code size by nearly 50% for typical applications. The object checking uses two machine instructions, i.e. CMP and JNE in i386, to determine whether the home of the object is in local node. However, these branch instructions also cause some registers to spill. For local objects the control flow will follow the static branch prediction algorithm in Pentium II [18].

We compared the performance of original Kaffe JVM and JESSICA2 using Java Grande Forum Benchmark Suite Thread Version 1.0 [17]. During the tests, the JIT compiler mode is enabled. The first three programs test the performance of several low-level operations such as thread

fork/join, barriers and synchronized methods/blocks by creating multiple threads within a specified time interval. The remaining programs carry out specific operations frequently used in Grande applications. In the last 5 test programs, only 1 thread is created. For the purpose of comparison, we also include the results of all test programs running under the Kaffe 1.0.6 interpreter mode. Table 2 shows the results.

| Time (In Seconds) | Kaffe 1.0.6 JIT | JESSICA2 | Slowdown | Kaffe 1.0.6 interpreter |
|---|---|---|---|---|
| JGFBarrierBench | 22.58 | 26.79 | **18.64%** | 77.1 |
| JGFForkJoinBench | 6.91 | 7.2 | **4.20%** | 29.95 |
| JGFSyncBench | 71.65 | 52.18 | **-27.17%** | 50.39 |
| JGFCryptBenchSizeA (1 thread) | 9.85 | 11.98 | **21.62%** | 129.44 |
| JGFLUFactBenchSizeA (1 thread) | 9.29 | 10.97 | **18.08%** | 79.60 |
| JGFSORBenchSizeA (1 thread) | 46.01 | 36.52 | **-20.63%** | 413.85 |
| JGFSeriesBenchSizeA (1 thread) | 36.91 | 36.38 | **-1.44%** | 208.47 |
| JGFSparseMatmultBenchSizeA (1 thread) | 26.17 | 31.92 | **21.97%** | 352.29 |

**Table 2. Single Node Performance Benchmark using Java Grande Forum Benchmark Suite - Thread Version 1.0.**

From the above table, we can see that the worst slowdown is the JGFCryptBenchSizeA program, which performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of 3,000,000 bytes. Due to the large number of iterations on the checking array object, the slowdown has reached 21.62%. In some cases, such as the JGFSORBenchSizeA and JGFSyncBench (see those with negative slowdown), JESSICA2 may even outperform Kaffe in JIT compiler mode. This is due to the changes of our new lock native codes to the original Kaffe locks and there are lots of synchronization operations in these two programs even when there is no contention for the locks with one thread running. For all programs except the JGFSyncBench, the original Kaffe running in the interpreter mode performs much slower than the JESSICA2 and original Kaffe running in JIT compiler mode.

## 5.3 Application Benchmark

In this section, we report the performance of three multithreaded Java applications on JESSICA2. These include CPI, nBody and SOR programs. We use explicit *synchronized* method to synchronize the computation steps of the tested programs. The program CPI calculates the approximation of $\pi$ (PI) by evaluating the integral. Each thread calculates the area under different intervals and sum up all the results together to get the approximate value of $\pi$. The nBody follows the algorithm of Barnes & Hut to simulate the motion of particles in a 2D dimension due to gravitational forces over a fixed amount of time steps. SOR does red-black successive over-relaxation on a 2-D matrix for a number of iterations.

Figure 5 compares the performance between JESSICA and JESSICA2 based on the execution time of CPI program. We run CPI program with 50,000,000 iterations. In the tests, the number of Java threads created is the same as the number of cluster nodes used during the test. All the threads are originally created and running on the master JVM. Later, only one thread is left in the master JVM and the rest are migrated to the worker JVMs, one thread per worker JVM. The thread migration takes place when iteration number is approximate to 2,000,000. From the figure, we can see that running in the interpretation mode (JESSICA) is far slower than running in the JIT compiler mode (JESSICA2).

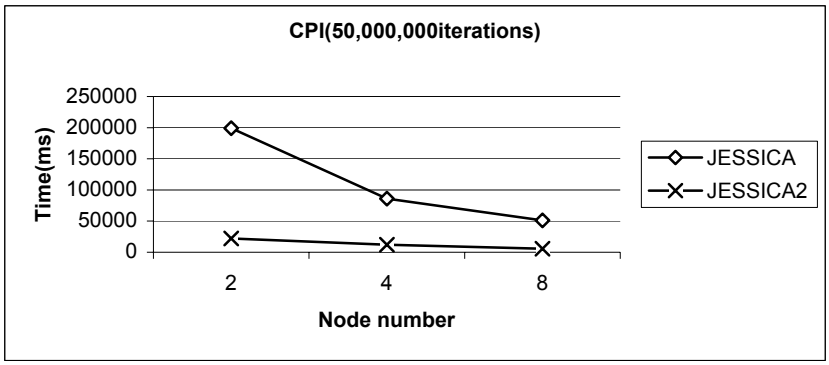


**Figure 5. Performance Comparison of JESSICA and JESSICA2 based on a CPI Program**

Figure 6 shows the raw execution time of running nBody and SOR respectively. We run nBody with 640 particles in 10 iterations and SOR with 1024x1024 matrix in 20 iterations. We didn't include the data of JESSICA because the performance of JESSICA is far slower than JESSICA2 and with large problem size, JESSICA is not able to run using the Treadmark DSM.

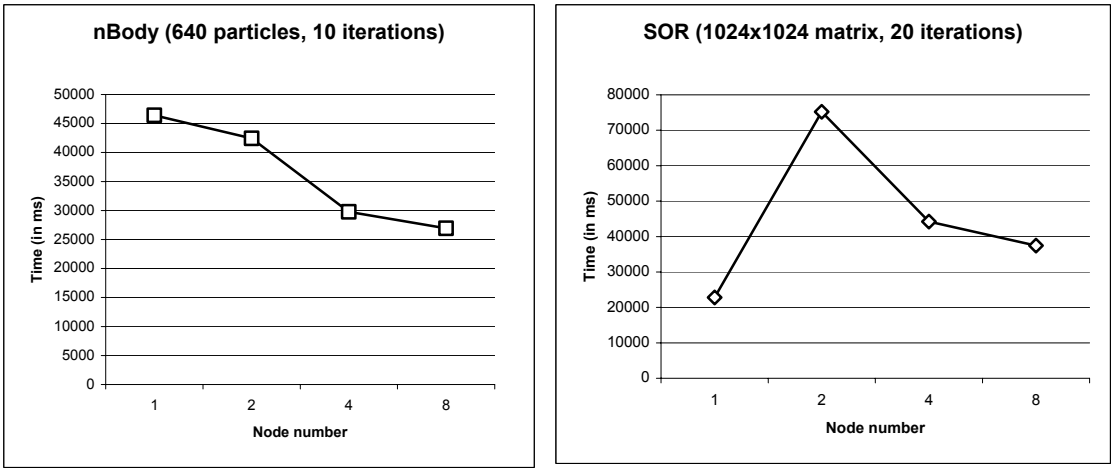


**Figure 6. Performance of nBody and SOR under JESSICA2**

Figure 7 shows the speedup of CPI, nBody and SOR by comparing the execution time between JESSICA2 and Kaffe 1.0.6 under JIT compiler mode. From the figure, we can see nearly ideal speedup in JESSICA2 considering the fact that all the threads run in the master JVM for 4% of the time at the very beginning. For multithreaded programs with little communication between threads like CPI, the performance improvement is due to the lower cost of migration and enhanced computation power of a JITEE. For nBody program, the speedup is only 1.5 for 8 nodes as the program suffers from many communication overheads between the worker threads and the master thread that is responsible for calculating the Barnes-Hut Tree. The SOR program achieves the worst speedup among the three programs. The resulting speedup is less than 1 for even 8 nodes. It is because the large array data has to be transferred between the master thread and the worker threads in every synchronization step. In such programs like nBody and SOR, communication part dominates the total execution time in JESSICA2 because the computation in JIT compiler mode in relatively fast. Further optimization on the GOS layer based on Java semantics can be exploited to achieve better speedup. Several optimization techniques, such as object pre-caching, object escape analysis, have been considered to be included in JESSICA2 in the future.

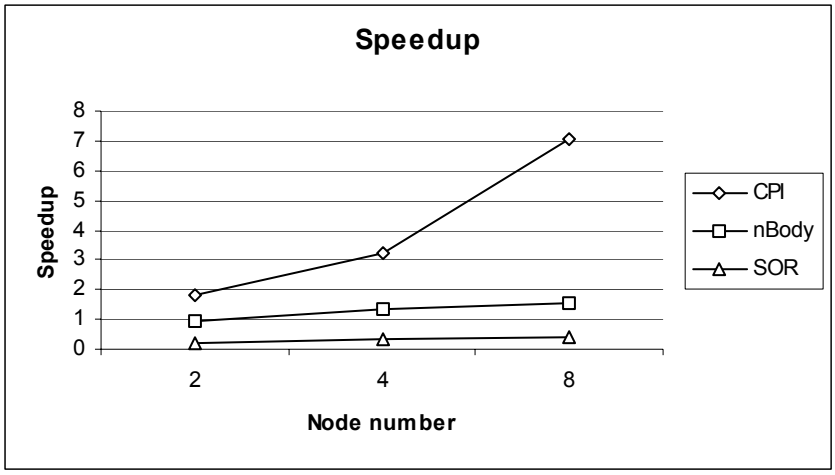


**Figure 7. Speedup Measurement of Three Java Applications**

## 6. Related Works

The research on DJVM has been a promising area in recent years. Many prototypes are been developed to support running multithreaded Java applications on clusters to achieve the Single System Image illusion at the JVM level.

cJVM[6] is a cluster-aware JVM that provides SSI of a traditional JVM running on cluster environments. cJVM was implemented by modifying the interpreter loop based on Sun JDK1.2 interpreter on NT clusters. cJVM distributes the Java threads on clusters at the time of thread creation and support the remote object access by a smart proxy model. For a remote Java object reference, a proxy of the object will be created. Later access to the object will be directed to the proxy and the proxy will use a technique called *method shipping* to ship the current method to the node where the master copy of the object resides based on the analysis on different object access.

Java/DSM[3] is a DJVM that runs on a cluster of heterogeneous computers based on an underlying Treadmark page-based DSM. The design was based on the JDK 1.0.2 JVM. The DSM is used to realize the global heap and class repository for Java program. Java/DSM relies on the underlying DSM to maintain the consistency of shared data. However Java/DSM requires the threads in the Java program be modified to specify the location to run. This violates the transparency or SSI requirement of DJVM.

Hyperion [5] provides a running support for multithreaded Java applications upon an object-based DSM. Hyperion takes a static compiling approach by statically compiling the multithreaded Java programs to C programs. The C programs are then compiled by the C compiler and are linked with the PM2 run-time library into a parallel native application. In a strict sense the approach of Hyperion didn't realize a real DJVM, for it requires all classes be available before execution thus loses the flexible of a DJVM to dynamically load a class on demand.

There are other prototypes to practice the DJVM that follow the approach of using a DSM like our previous project JESSICA [2] and Kaffemik [14]. And there are other DJVMs that use the static compiling approach like Jackal [15].

## 7. Conclusions

JESSICA2 is our new DJVM that use transparent Java thread migration mechanism to run multithreaded Java applications on clusters. We have successfully implemented the efficient transparent Java thread migration mechanism in JESSICA2 which can be performed in the JIT compiler mode. The mechanism provides us a flexible way for distributing threads among cluster nodes to balance the workload of clusters. The performance improvement over the old JESSICA is significant due to the introduction of JITEE in JESSICA2.

To support shared object access in JESSICA2, we implemented a *global object space* (GOS) layer without using a page-based DSM. The consistency protocol adopted in our GOS is close to the definition of Java memory model. Meanwhile, it is embedded within the JVM. Such design makes GOS able to work efficiently with the JESSICA2 thread system. It is found that the non-blocking I/O support of GOS makes JESSICA2 possible to avoid the whole JVM to be blocked while a thread is issuing a remote object access. From all various benchmark tests, we conclude that JESSICA2 is a promising DJVM design which has a good potential to achieve a high-performance parallel execution environment for multithreaded Java applications on a parallel or distributed environment.

At the current stage, JESSICA2 still suffers from excessive communication overheads for some applications with high degree of object sharing. Our future work will focus on the optimization of the GOS. Runtime shared object detection together with a good object pre-fetching technique could be a possible solution. Besides, there are good opportunities for us to develop more intelligent load balancing strategies as JESSICA2's JITEE is able to detect various runtime thread information.

## References


[1] K. Hwang, H. Jin, E. Chow, C.L. Wang, and Z. Xu; "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space," *IEEE Concurrency Magazine*, Vol. 7, No. 1, pp. 60-69, Jan-Mar., 1999.

[2] M.J.M. Ma, C.L. Wang, and F.C.M. Lau, "JESSICA: Java-Enabled Single-System-Image Computing Architecture," *Journal of Parallel and Distributed Computing*, Vol. 60, No. 10, October 2000, 1194-1222.

[3] W. Yu and A.L. Cox, "Java/DSM: A platform for heterogeneous computing," *Proc. of ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[4] M.J.M. Ma, C.L. Wang, and F.C.M. Lau; "Delta Execution: A Preemptive Java Thread Migration Mechanism", *Cluster Computing: The Journal of Networks, Software Tools and Application*, Vol. 3, No. 2, 2000, pp. 83-94.

[5] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst, "The Hyperion System: Compiling Multithreaded Java Bytecode for distributed execution," *Parallel Computing*, 2001.

[6] Yariv Aridor, Michael Factor, and Avi Teperman, "CJVM: a Single System Image of a JVM on a Cluster," *Proc. ICPP'99*.

[7] C. Amza, A.L.Cox, S.Dwarkadas, P. Keleher, H.Lu, R. Rajamony, W. Yu and W. Zwaenepoel, "Treadmarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer* 29(2), February, 1996.

[8] Transvirtual Technologies In., Kaffe Open VM, http://www.kaffe.org.

[9] T. Lindholm and F. Yellin. "The Java(tm) Virtual Machine Specification," *Addison Wesley*, second edition, 1999.

[10] Michael G. Burke, *et. al.,* "The Jalapeno dynamic optimizing compiler for Java," *In ACM 1999 Java Grande Conference*, pages 129--141, June 1999.

[11] A. Krall and R. Grafl. "CACAO -- a 64 Bit JavaVM Just-in-Time Compiler," In G. C. Fox and W. Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997. ACM.

[12] S. Bouchenak, D. Hagimont, "Approaches to Capturing Java Threads State," *Middleware 2000*, New York - USA, 3rd-7th April 2000
[13] William Pugh. The Java memory model is fatally flawed. Concurrency: Practice and Experience, 12(1):1--11, 2000.
[14] Andersson Weber Cecchet, "Kaffemik - A distributed JVM on a single address space architecture," http://wwwbode.cs.tum.edu/archiv/smile/scieur2001/scieuro2001-s1-p3.pdf
[15] R. S. Cost, T. Finin, Y. Labrou, X. Luan, Yun Peng, I. Soboroff, J. Mayfield, and A. Boughannam, "Jackal: a java-based tool for agent development," 1998.
[16] Alfred V.Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools," *Addison-Wesley*, 1986.
[17] The Java Grande Forum Multi-threaded Benchmarks, http://www.epcc.ed.ac.uk/javagrande/threads/contents.html
[18] Intel Cor., " Intel Architecture Optimizations Manual," 1999.
[19] W.L. Cheung, C.L. Wang, and F.C.M. Lau, "Building a Global Object Space for Supporting Single System Image on a Cluster," *Annual Review of Scalable Computing*, Volume 4. World Scientific, 2002, to appear.
[20] The Java Memory Model, http://www.cs.umd.edu/~pugh/java/memoryModel/.