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

A distributed Java Virtual Machine (DJVM) spanning multiple cluster nodes can provide a true parallel execution environment for multi-threaded Java applications. Most existing DJVMs suffer from the slow Java execution in interpretive mode and thus may not be efficient enough for solving computation-intensive problems. We present JESSICA2, a new DJVM running in JIT compilation mode that can execute multi-threaded Java applications transparently on clusters. JESSICA2 provides a single system image *(SSI) illusion to Java applications via an embedded* global object space *(GOS) layer. It implements a* cluster-aware Java execution engine *that supports transparent Java thread migration for achieving dynamic load balancing. We discuss the issues of supporting transparent Java thread migration in a JIT compilation environment and propose several lightweight solutions. An adaptive migrating-home protocol used in the implementation of the GOS is introduced. The system has been implemented on x86-based Linux clusters, and significant performance improvements over the previous JESSICA system have been observed.*

1. Introduction

A distributed Java Virtual Machine (DJVM) spanning multiple cluster nodes and offering a *single system image* (SSI) [12] illusion to Java threads can provide a true parallel execution environment for multi-threaded Java applications.

Some existing DJVMs follow the monolithic approach. They modify the JVM so that the it can support distributed class loading, shared object placement and access, distributed thread management and synchronization, etc. A notable example is cJVM [5] which makes use of Java's semantics to optimize the execution of Java threads in cluster environments. However, due to the complexity in modifying a JVM to support the above functions, DJVMs of this type were usually based on modifications of the Java interpreter, and hence can only support execution of Java programs in interpreter mode. To achieve higher performance especially for computation-intensive applications, a DJVM that supports JIT compilation is needed.

The other type of DJVM relies on a *Distributed Share Memory* (DSM) system to support the parallel execution of Java threads. All distributed Java threads can transparently access objects in the common object space created by the underlying DSM. In this environment, thread synchronization and object consistency are managed through the DSM's locking/unlocking mechanisms and data consistency protocols. This approach allows any JVM to be turned into a DJVM with only minor changes. Java/DSM [20], Hyperion [4], Jackal [19] and our previous project, JESSICA [17], are examples of this type. This in fact is a layered approach, and as such runtime information at the JVM level cannot be easily channeled to the underlying DSM, which could lead to poor running performance. Also, due to the mismatch between the memory models of Java and the underlying DSM [8], nontrivial optimizing techniques need to be employed to enable efficient object sharing among distributed Java threads.

To fully exploit the power of clusters, a thread migration mechanism can be built into the DJVM to enable dynamic load balancing through migrating Java threads between cluster nodes at runtime without programmers' involvement. Transparent thread migration has long been used as a load balancing mechanism to optimize the resource usage in distributed environments [10, 13]. "Transparent" means that the migration operation is automatically triggered without explicit instructions from the source program. Java's uniform thread model and its machine-independent bytecode format make the migration mechanism more portable even in heterogeneous environments.

*This research is supported by Hong Kong RGC grant HKU-7030/01E.

For systems that support thread migration using C/C++ on DSM [13], the native pointer problem remains a difficult issue. These systems usually rely on reserving some virtual address space in all the nodes and require the system to be totally homogeneous. The resulting system imposes many constraints on the program and is unfriendly to the end users. The system we present here solves the problem by allowing the reallocation of the thread stack in different virtual address spaces. Thus, multi-threaded Java programs need only be written in the usual fashion.

Our DJVM is a totally new design. We exploit the power of Just-in-Time (JIT) compiler to enhance the performance. Incorporating JIT compilation in DJVMs is rarely discussed in the literature; yet we prove that it is a promising feature for DJVMs through our implementation. Our prototype uses the Kaffe JVM V1.0.6 [1], 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 compiler mode. The results we obtained show a major improvement in performance over the old interpreter-based implementation.

The other new feature not shared by our previous work is a *global object space* (GOS) layer. This layer supports the access of shared objects by multiple distributed Java threads. The GOS per se is not a full-fledged object-based DSM supporting general-purpose distributed computations, but is built following the memory consistency model defined in the current Java Memory Model. Our design tries to exploit the Java semantics to minimize the remote object access overheads in a distributed environment. From our experiences we found that a naive implementation of a GOS without exploiting object access patterns in multi-threaded Java programs can result in poor overall performance. In this paper, we introduce our optimization technique which is based on an adaptive home-migration strategy for objects.

The rest of the paper is organized as follows. Section 2 presents the overall architecture of JESSICA2. In Section 3, we discuss the details of the Java thread migration mechanism in a JIT environment. In Section 4, we explain the design of the GOS. Section 5 shows the experimental results of our prototype system. Related work is discussed in Section 6, and a short conclusion is given in Section 7.

2. Overview of 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 in different cluster nodes. We call a node that starts the Java program as the *master node* and the JVM running on it as the *master 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 in any node of the cluster. The load monitor is responsible for monitoring the system load of the cluster and activating Java thread migration. The Java threads in the application can migrate from one node to another node upon receiving requests from the load monitor.

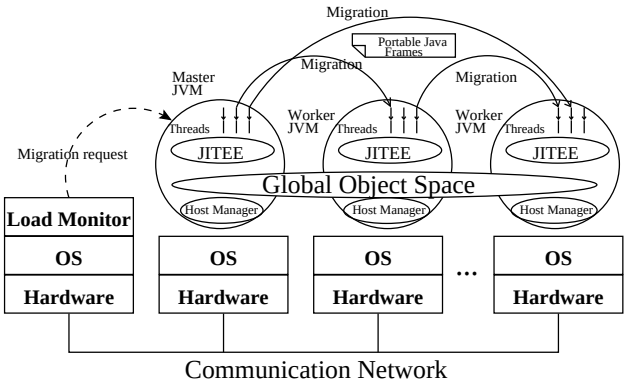

Figure 1. The JESSICA2 system architecture

Each modified JVM on the cluster node uses the modified Java execution engine, JITEE, to execute the bytecodes in JIT mode. The Java thread migration mechanism is built inside the JITEE to enable the mobility of Java threads at bytecode boundaries. The details of this mechanism will be discussed in Section 3.

The GOS layer is embedded inside the DJVM to provide an SSI view for distributed Java threads. Two main kinds of operations are included: the data access and thread synchronization operations. The data access operations are used to hide the data location for the distributed Java threads. They correspond to the bytecode instructions that access the class static data, object fields, and the array components. The thread synchronization operations implement the Java thread synchronization primitives transparently in a distributed environment, which include the operations such as *lock()*, *unlock()*, *wait()*, *notify()* and *notifall()*.

A daemon thread called the *host manager* runs inside the JVM to provide basic communication supports for data movements in GOS and control movements in JITEE. The host mangers in different JVMs communicate with each other through TCP connections.

3. Transparent Java thread migration

The JVM specification [15] defines various runtime data structures, including the heap, the method area, and the JVM stack. They together form the execution context of a Java thread.

In our design, we have abandoned the use of a page-based DSM because of performance and portability reasons. Under this environment that is without a DSM, we need to

apply different strategies to handle different types of Java thread context during thread migration.

The global heap of the DJVM will be realized by the embedded GOS, which will be discussed in detail in the next section. Therefore during thread migration, we can migrate all the objects used by the migrated thread by shipping only the global Java object references without actually moving the object data to the remote node. The GOS will handle these object data movements once the migrated thread restarts its execution on the remote node and needs to access the objects.

For the method area, we partially load each class independently on each JVM. The correctness of rebinding the class structures on the remote machine is guaranteed by preserving a single copy of static data for non-system classes. Similar approaches can be found in common DJVM prototypes such as cJVM and Hyperion.

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 the JITEE to generate efficient native codes to manage the thread execution context at runtime and extend the JVM thread system to support stack capturing and restoring. To be able to rebuild Java thread stack at a different virtual memory space in a remote node, the thread stack context is captured at the bytecode boundary. The captured stack context is then translated into a machine-independent text format and is able to be restored by the target JVM. The machine-independent description of Java stack can help realize thread migration among different types of JVM.

3.1. Stack capturing

The JIT compiler makes the execution context of Java thread more complicated to capture compared to methods used in the interpreter mode. Several issues need to be solved when migrating threads within JIT compilers:

(1) *Migration point.* In order to migrate thread stack to a different virtual address space, we need the migration point to be set at the bytecode boundary. In JIT mode, the thread runs its native codes generated by the JIT compiler. 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 at the bytecode boundary. How to "slide" to the bytecode boundary is an issue for thread migration in JIT mode.

(2) *Register context.* As JVM is a stack-based machine, one of the important tasks of a JIT compiler is to allocate registers for Java variables in the register-based machines [7, 14]. 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 variables may take place in the allocated registers. That the variables don't have the most recent values in the memory will prevent the stack capturing operation from getting the correct values from the stack. The problem has the same characteristics as the optimized code debugger. But it is impractical for us to store heavy data structures like the code debugger to support thread migration in the execution engine. We should be able to get the variable information out of appropriate registers in JIT mode.

(3) *Variable types.* As the stack variables are dynamically pushed into or popped from the thread stack, the variable types of specific stack slots cannot be determined in advance. To tell if a variable is a reference or not and to encode the variable in a machine-independent format, it is required that the type of the variable be known at the time of thread migration. In [6], it is proposed using a separated *type stack* operated synchronously for interpreter during thread execution to the trace the variable types. 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) *Frame boundary.* The JVM stack in an interpreter is defined explicitly in the internal data structures known by JVM. But for a JIT compiler, the stack is implicitly managed by the native codes generated. In this case, the frame boundary is known to the running codes but not known to JVM. Moreover, one single running stack for a Java thread is often shared among the Java methods, and the internal JVM functions (including the JIT compiler). All these make the frame boundary in the stack become blurred to JVM, which in turn makes the stack capturing difficult. One approach can allocate a dynamic data structure to store the stack frames upon method invocations during runtime. However, it will consume more execution time and memory space, especially for Java programs in which method invocations are frequent. Therefore, an efficient mechanism to reveal the stack frame boundary to JVM 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 additional checking codes at these points. The codes are used to check the migration request and to spill the machine registers to the memory slots of the variables. The candidate threads to migrate, when running to such points

can detect the migration request, and will call the appropriate functions to do the stack capturing and migration. The resulting effect is like that the thread slides to a safe point before migration.

To tackle the third problem, we choose to do the *type spilling* at the migration points discussed above. The type information of stack variables at such points will be gathered at the time of bytecode verification before compiling the Java methods. We use one single type to encode the reference type of stack variable as we can identify the exact type of a Java object from the object reference. Therefore, we can compress one type into 4-bit data. Eight compressed types will be bound in a 32-bit machine word, and an instruction to store this 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 efficiency than the type stack method mentioned before.

The fourth problem is solved by generating native codes that link the Java thread stack dynamically upon method invocations. The codes only need a few instructions to spill the previous Java frame stack pointer and previous machine stack pointer. Such arrangement makes it possible to tell a Java frame from the internal JVM functions frames (we call it C frames [16]). 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.

3.2. Stack restoring

The stack restoring needs to recover the machine registers in the migration target node. Most previous approaches supporting thread stack restoring often build the stack by simulating the method invocation and use additional status variables to distinguish the restoring execution flow and the normal execution flow inside the methods [6]. This will results in large overheads because it needs to add such branching codes in all the methods. Rather we directly build the thread stack and use de-compiling techniques to get the mapping between the thread variables and the machine registers at all restoration points. Each mapping is then used by a generated code stub that will be executed before the restoration point to recover the machine registers. In our implementation, we allocate the code stubs inside the restored thread stack so that they will be freed automatically after execution.

4. Global object space

As we mentioned before, our previous JESSICA project uses a page-based DSM to realize 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 does not match well with the access granularity of the object-based Java 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 distributed Java threads.

4.1. Memory model and object access

The *Java Memory Model* (JMM) [15] specifies the semantics of memory operations issued by Java threads. Based on the common understanding of the JMM, multi-threaded 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 copies of objects from 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. In addition, Java threads use monitors 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 it 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 the GOS layer. Object caching is adopted in JVM for improving the performance. Each JVM in the DJVM will contribute a portion of its heap, the *cache heap area*, for storing cached remote objects. The other portion of heap, the *global heap area*, is used to store the master copies of objects. The node that holds the master copy of an object is called the *home* of the object. For all the threads inside one JVM, the access to objects in global heap area is just the same as the single-node JVM implementation, i.e., all the objects in the area are the master copies and can be

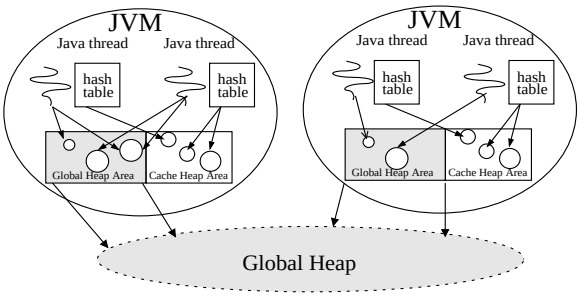

Figure 2. GOS architecture

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 a per-thread data structure. It can be viewed as an extension to the thread's working memory. Figure 2 shows the architecture of GOS.

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. The approach can make better use of the computing resource on clusters and is more appealing than the approach of running JVM on top of an existing DSM on clusters, in which case a page-fault will make the whole JVM stop from running until underlying DSM has brought in the memory pages needed from the network. It is also superior to the approach of simply adopting an existing Object-based DSM without multi-threading support.

4.2. Adaptive object home migration

With the introduction of Java thread migration among cluster nodes, the home of an object may be different from the location of threads that access this object. This will cause the object to be cached on the remote node after thread migration. According to the semantics of JMM, at synchronization points when entering or exiting a monitor, this cached object has to be flushed and be re-fetched again from its home. The communication cost of such operations will slow down the overall system performance.

To address such problem, we introduce the *adaptive object home migration concept* in the GOS design. Object home migration means that we change the home of an object to another node. The master object at the old home becomes the cache copy and the remote cache copy at the new home becomes the master copy. By doing so, if an object is frequently accessed by the threads on the new home in a period of time, the communication cost to flush and to re-fetch this object will be eliminated. This could result in great performance improvement for the Java applications.

However, it is hard to determine whether the home of an object should be migrated or not. With complicated compiler support [9], it is possible to detect some of the objects that are local to one thread. However such support will be too heavy to be exploited at runtime. In our system, we choose simple heuristic method to adaptively migrate the home of an object. We follow the rules below to migrate the home of an object: if the number of the accesses from a thread dominates the total number of accesses to an object, the home of the object will be migrated to the node where the thread is running on. We perform this object home migration decision during the flushing operation from the remote node so that both the new home and the old home will have the same copy of object data at the time of home migration. Later the threads in other nodes except the new home node will be informed an access redirection when they are trying to access this object the first time after the home migration operation.

5. Performance results

A JESSICA2 prototype has been implemented based on Kaffe JVM V1.0.6 on 540MHz Pentium-II cluster nodes running the Linux 2.2.1 kernel and connected by Fast Ethernet. Various performance results are reported.

5.1. Microbenchmarks on thread migration

We perform microbenchmarks to measure the cost of the transparent Java thread migration. Table 1 shows the measurements of different operations during thread migration for different sizes of Java frames.

Table 1. Timing breakdowns of thread migration (in μs)

Frame number	1 frame (475Bytes)	2 frames (482Bytes)	11 frames (3,049Bytes)
Stack capturing	232	437	12,993
Frame parsing	166	328	1,383
Resolution	3,431	13,747	227,587
Frame setup	9	13	49
Overall time	3,838	14,525	242,012

From the above measurements we can see that using our techniques in JIT mode discussed in Section 3, the costs of stack capturing and frame setup account for an insignificant part in the overall cost. Because the proposed type spilling operation can quickly determine the type of variables, the stack capturing operation costs only a few milliseconds for large stack frames. The frame setup operation is extremely fast as we directly manipulate the thread stacks instead of

building the stacks by simulating the function calls as used in other similar thread migration systems [18]. The most costly part of thread migration lies in the class and method resolution for the Java stacks. The cost to resolve class and method is caused by the disk I/O to perform class loading when the remote node does not have the needed classes in the JVM's class pool. This cost will be amortized with more threads migrating to this node which needs the same classes.

There are other operations involved in the thread migration. For example, the native thread creation cost on the remote node has a constant overhead about 423 μs on Pentium-II machine. Also the migration mechanism will insert checkpointing instructions to do the type spilling and migration checking in the generated native code. This kind of native code instrumentation occurs only to methods of the application classes. According to our measurements, the increase in code size due to the inserted checkpoints is less than 1%.

5.2. Performance of GOS

The embedded GOS support introduces both the space overheads and time overheads. The object checking increases the total native code size by nearly 50% for typical applications because the object checking operations are performed on every object access even if the object is local. This is a common problem for object-based DSM compared to using page-fault hardware mechanism to check the object status.

The adaptive object home migration plays an important role in optimizing the GOS performance. We use SOR to study the effect of home migration. SOR does red-black successive over-relaxation on a 1024×1024 matrix in 20 iterations.

Figure 3 shows the execution time of SOR application while enabling and disabling the adaptive object home migration. The improvement resulted from enabling adaptive object home migration mechanism is significant from the figure. If we disable this mechanism, we will have hundreds of messages transferring between master node and worker nodes in every iteration in order to fetch and flush the needed array elements. After adapting the home migration, the communication volume during each iteration steps decreased dramatically. Each iteration requires only 20 messages that are necessary for getting the real shared array and for performing locks.

5.3. Kaffe versus JESSICA2

Table 2 shows the performance comparison between the original Kaffe JVM and JESSICA2 using Java Grande Forum's Benchmark Suite Thread Version 1.0 [11]. The measurements shows that the worst time cost is below 22% for all the benchmarking applications.

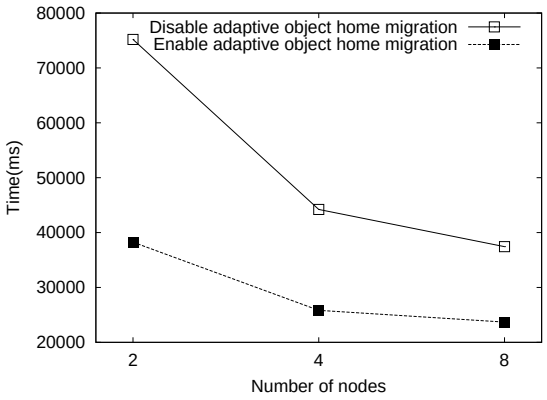

Figure 3. Effect of adaptive object home migration

Table 2. Single node performance benchmarking using Java Grande Forum Benchmark suit (in seconds)

Time (in seconds)	Kaffe JIT	JESSICA2	Slowdown
Barrier	22.58	26.79	18.64%
ForkJoin	6.91	7.2	4.20%
Sync	71.65	52.18	-27.17%
Crypt	9.85	11.98	21.62%
LUFact	9.29	10.97	18.08%
SOR	46.01	36.52	-20.63%
Series	36.91	36.38	-1.44%
SparseMatmult	26.17	31.92	21.97%

From the above table, the Crypt program has 21.62% slowdown which is due to the large number of iterations on checking array objects. In some cases, such as SOR and Sync, JESSICA2 may even outperform Kaffe. This is contributed by replacing the original Kaffe's *thin locks* with our new locking native codes.

5.4. Application benchmark

In this section, we report the performance of four multi-threaded Java applications on JESSICA2, and show how thread migration, JIT compilation and communication overheads in the GOS, could affect Java programs' performance. The applications are CPI, TSP, Raytracer, and nBody. The program CPI calculates an approximation of π by evaluating the integral. The Travel Salesman Problem (TSP) finds the shortest route among a number of cities using parallel branch-and-bound algorithm. The Raytracer program mea-

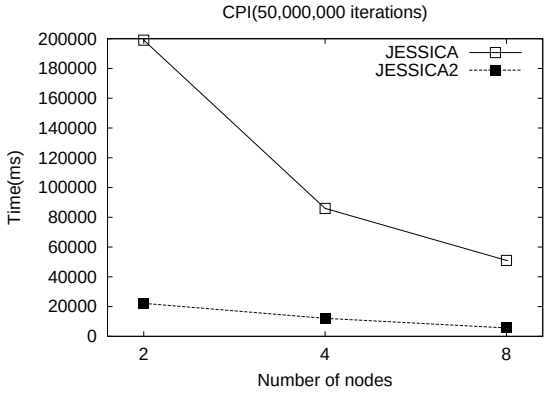

Figure 4. Performance comparison of JESSICA and JESSICA2 based on a CPI program

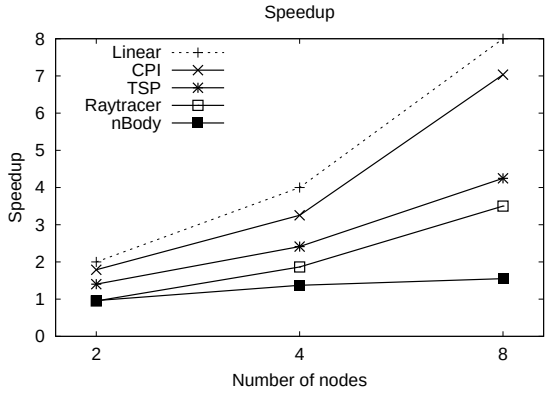

Figure 5. Speedup Measurement of Java applications

sures the performance of a 3D Raytracer. The nBody follows the algorithm of Barnes & Hut to simulate the motion of particles in a 2D space due to gravitational forces over a fixed amount of time steps. We use explicit synchronized methods to synchronize the computation steps of these tested programs.

Figure 4 compares the performance between JESSICA and JESSICA2 based on the execution time of the CPI program. We run the 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 around 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). The result clearly demonstrates the importance of incorporating the JIT compiler in DJVM.

Next we run TSP with 14 cities, Raytracer with in a 150x150 scene containing 64 spheres, nBody with 640 particles in 10 iterations. We show the speedups of CPI, TSP, Raytracer and nBody in Figure 5 by comparing the execution time between JESSICA2 and Kaffe 1.0.6 (in a single-node) under JIT compiler mode. From the figure, we can see nearly linear speedup in JESSICA2 for CPI, despite the fact that all the threads needed to run in the master JVM for 4% of the overall time at the very beginning. For the TSP and Raytracer, the speedup curves show about 50% to 60% of efficiency. Compared to the CPI program, the number of messages exchanged between nodes in TSP has been increased because the migrated threads have to access the shared job queue and to update the best route during the parallel execution, which will result in flushing of working memory in the worker threads. In Raytracer the number of messages is small, as it only needs to transfer the scene data to the worker thread in the initial phase. The slowdown comes from the object checking in the modified JVM as the application accesses the object fields extensively in the inner loop to render the scene. But for the nBody program, the speedup is only 1.5 for 8 nodes. The poor speedup is expected, which is due to the frequent communications between the worker threads and the master thread in computing the Barnes-Hut Tree.

6. Related work

The research on DJVM has been a promising area in recent years. Many prototypes have been developed to support running multi-threaded Java applications on clusters to achieve the SSI illusion at the JVM level.

cJVM [5] is a cluster-aware JVM that provides SSI of a traditional JVM running on cluster environments. cJVM distributes the Java threads on clusters at the time of thread creation and support the remote object access by a smart proxy model. The cJVM prototype was implemented by modifying Sun JDK1.2 interpreter. Hence it has performance weaknesses stemming from the slow execution of the Java interpreter, and may not be efficient enough for solving computation-intensive problems in practice.

Java/DSM [20] is a DJVM that runs on a cluster of heterogeneous computers. The design was based on the JDK 1.0.2 JVM and relies heavily on the underlying Treadmark page-based DSM to maintain the consistency of shared data. Java/DSM requires the threads in the Java program be modified to specify the location to run. This violates the SSI requirement of DJVM. Also using a page-based DSM may suffer from false sharing problems for Java applications.

There are other DSM-based DJVM prototypes, for ex-

ample, JESSICA [17] and Kaffemik [3]. Our previous work, JESSICA, implemented a preemptive Java thread migration mechanism, called Delta Execution [16] by modifying the Kaffe JVM V0.9.1 interpreter. JESSICA provides an SSI illusion to Java programs on top of an existing paged-based DSM, e.g., Treadmark [2] and JUMP [8]. Kaffemik is developed on clusters interconnected with SCI. The current prototype is at its early stage, and lacks support for object replication and caching.

There are systems developed to support thread migration. Arachne [10] is one of such systems. It provides a portable user-level programming library that supports thread migration over a heterogeneous cluster. However the thread migration is not transparent to the user as it required that programs be written using special thread library or APIs.

7. Conclusion

JESSICA2 is a new DJVM that applies transparent Java thread migration to multi-threaded Java applications running in a cluster. We have successfully implemented a prototype that operates in the JIT compiler mode. The migration mechanism provides us with a flexible way to distribute threads among cluster nodes at runtime. The performance improvement over the previous JESSICA system is significant due to the introduction of a JIT execution engine (JITEE) in JESSICA2.

To support shared object access, we implemented a global object space (GOS) layer without using a page-based DSM. The GOS is embedded in the JVM, and by design works efficiently with the JESSICA2 thread system.

From the various benchmark tests, we conclude that JESSICA2 is a promising DJVM system that can serve as a high-performance parallel execution environment for multi-threaded Java applications on parallel or distributed hardware.

The current JESSICA2 implementation suffers excessive communications for certain applications requiring a high degree of object sharing. Our ongoing work will focus on optimization of the GOS in response to this. Runtime shared object detection together with good object pre-fetching techniques is a possible solution under study. There are also opportunities to develop load balancing strategies that are more intelligent, as JESSICA2's JITEE is able to obtain various runtime thread information.

References

[1] Kaffe open vm. Technical report, Transvirtual Technologies Inc., http://kaffe.org, 2002.

[2] 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):18–28, February 1996.

[3] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill. Kaffemik - a distributed jvm on a single address space architecture. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[4] G. Antoniu et al. The hyperion system: Compiling multithreaded java bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

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

[6] S. Bouchenak and D. Hagimont. Approaches to capturing java threads state. In *Middleware 2000*, New York, USA, Arpil 2000.

[7] M. G. Burke et al. The jalapeno dynamic optimizing compiler for java. In *ACM 1999 Java Grande Conference*, pages 129–141, 1999.

[8] W. L. Cheung, C. L. Wang, and F. C. M. Lau. Building a global object space for supporting single system image on a cluster. In *Annual Review of Scalable Computing*, volume 4. World Scientific, 2002.

[9] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for java. In *Conference on Object-Oriented*, pages 1–19, 1999.

[10] B. Dimitrov and V. Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5), 1998.

[11] EPCC. The java grande forum multi-threaded benchmarks. http://www.epcc.ed.ac.uk/javagrande/threads/contents.html.

[12] 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*, 7(1):60–69, January 1999.

[13] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *Systems and Software*, 42(1):71–87, 1998.

[14] A. Krall and R. Grafl. Cacao – a 64 bit javavm just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

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

[16] 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*, 3(2):83–94, 2000.

[17] M. J. M. Ma, C. L. Wang, and F. C. M. Lau. Jessica: Java-enabled single-system-image computing architecture. *Parallel and Distributed Computing*, 60(10):1194–1222, October 2000.

[18] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in java. In *ASM*, pages 29–43, 2000.

[19] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal. Jackal, a compiler based implementation of java for clusters of workstations. PPoPP 2001, 2001.

[20] W. Yu and A. L. Cox. Java/dsm: A platform for heterogeneous computing. *Concurrency - Practice and Experience*, 9(11):1213–1224, 1997.