JIT-Compiler-Assisted Distributed Java Virtual Machine [*]

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

Abstract

There is now a strong interest in high-performance execution of multithreaded Java programs in a cluster. Previous efforts to provide for such executions have either used static compilation tools that can transform multithreaded Java programs into parallel versions, or interpreter-based cluster-aware JVMs that offer the needed support. They failed however to be fully compliant with the Java language specification or to achieve high performance due to their weaknesses in supporting thread distribution and global object sharing. We present our research experience in the design and implementation a JIT-compiler-assisted distributed Java Virtual Machine. In our system, we make use of a JIT compiler to realize and optimize dynamic thread migration and global object sharing in a distributed environment.

1. Introduction

The Java programming language [11] supports concurrent programming with multithreading. It realizes a shared memory programming model where Java threads access the created objects in a heap without explicit data partitioning and message passing. Java's concurrent features make it a potential language for parallel computing. Although Java has for a long time been criticized for its slow execution, recent advances in better Java class libraries and Just-in-Time (JIT) compilation techniques have greatly boosted the performance of Java. Some results indicate that Java can deliver a performance in the 65–90% range of the best Fortran performance for a variety of benchmarks [4] and can compete with the performance of C++ [17].

In recent years, there is increased interest in using Java for high-performance computing. An attractive direction is to extend the Java Virtual Machine (JVM) to be "cluster-aware" so that a group of JVMs running on distributed cluster nodes can work together as a single, more powerful JVM to support true parallel execution of a multithreaded Java application. With the cluster-aware JVM, the Java threads created within one program can run on different cluster nodes to achieve a higher degree of execution parallelism. Similar to a desktop JVM, the distributed JVM system provides all the virtual machine services to Java programs, and should be fully compliant with the Java language specification. We refer to such a distributed system is a Distributed JVM (DJVM).

The outstanding merit of the DJVM approach is that the existing multithreaded Java programs running on a DJVM can fully utilize the available resources in all participating nodes. The DJVM automatically schedules Java threads on different nodes to execute in parallel. Unlike the stand-alone JVM which is limited by all the available resources in a single server, the DJVM running on a cluster potentially has unlimited resources. In the latest (Nov. 2003) TOP 500 supercomputer list [23], cluster-based systems occupy 7 out of the top 10 places. People can now easily build a high-performance cluster having hundreds of machines within a few days [22]. A DJVM would be a great software addition to these clusters.

The design of a DJVM needs to consider the following key issues: how to manage the Java threads, how to store the data (i.e., the objects), and how to process the bytecode in a distributed environment.

The thread scheduler of the JVM picks a thread to execute and performs the switch of thread context accordingly. In a DJVM, the task is much more complex, which involves the decision to place threads in different nodes. Most existing DJVMs [3] and related static compiler systems [25, 2] adopt the simple approach of doing the placement only initially when a thread is created. A newly created thread can be placed in a remote node. The result could be a good one for cases where all threads embody balanced workload. However it is not so good for many other applications where

[*] This research is supported by Hong Kong RGC Grant HKU-7030/01E and HKU Large Equipment Grant 01021001.

there is imbalanced workload among the threads. In these applications, a thread's workload is not known until some stage during runtime and could be quite varied from stage to stage. In this situation, it would be ideal to provide support for threads to migrate from node to node during runtime in order to achieve better load balancing.

The heap is the shared memory space for Java threads to store created objects. In a distributed environment, threads are running on different nodes, and they need to access objects created by themselves or by other threads during execution. A distributed shared heap is therefore necessary in the design of a DJVM. A natural question to ask is whether an off-the-shelf software Distributed Shared Memory (DSM) system would satisfy the requirement or do we need a new distributed heap design.

There exist a number of DJVMs [18, 30] that are directly built on top of an unmodified software DSM. For these systems, there is the issue of for achieving good performance because there tends to be a mismatch between the memory model of Java and that of the underlying DSM. The Java Memory Model (JMM) cannot be easily fit into one of consistency models that are supported by existing DSM systems. The JMM is probably closest to the Home-based Lazy Release Consistency (HLRC) model but with several differences, and it is currently evolving [20]. The mismatch in memory consistency between the chosen DSM and the JMM will cause the DJVM built this way to be inefficient or incomplete in supporting the required semantics. Moreover, the runtime information at the JVM level such as object type information cannot be easily channelled to the DSM. Indeed, it is difficult to extend an off-the-shelf DSM to support other desirable services such a single-system image (SSI) view of I/O objects. In our design, we opt for a customized built-in distributed global heap that realizes the JMM. Because it is tightly coupled with the DJVM kernel, we can make use of the runtime information inside the DJVM to reduce object access overheads.

To achieve high performance and to support dynamic thread migration, we set out to incorporate the Just-in-Time (JIT) compiler option in the design of the DJVM. With JIT compilation, a bytecode method is translated into the equivalent native code that can be executed directly on the underlying machine. In order to implement the thread migration feature in a DJVM that supports JIT compilation, the JIT compiler must be made cluster-aware so local machine-level entities can be converted to global data structures accessible by distributed VM components. For example, thread states in native machine form must be captured and represented in an intermediate form that can be interpreted by the destination JVM before the migration. These inevitably will add extra overhead to the JIT compiler. The challenge is to design a lightweight thread state capturing and restoration mechanism so that the overhead would not offset the benefits of thread migration.

In the paper, we describe our DJVM design, called JESSICA2. Our design is novel in the sense that it supports execution of runtime-migratable threads in JIT mode, and efficient global object sharing in the distributed environment.

The rest of the paper is organized as follows. In Section 2, we describe the general architecture of JESSICA2. Section 3 describes the implementation of thread migration based on JIT compilation. Section 4 discusses the implementation of distributed shared object support for the DJVM. Section 5 gives the experimental results with our prototype system. Section 6 discusses the related work. Section 7 concludes the paper and alludes to some possible future work.

2. JESSICA2 system overview

Figure 1 shows the overall architecture of our JESSICA2 DJVM system. The system runs in a cluster environment and comprises a collection of modified JVMs that run in different cluster nodes and communicate with each other using TCP connections.

We call a node that starts the Java program the *master node* and the JVM running on it the *master JVM*. All the other nodes in the cluster are *worker nodes*, each running a worker JVM to participate in the execution of a Java application. The worker JVMs can dynamically join the execution group. The Java threads in the application can migrate from node to node.

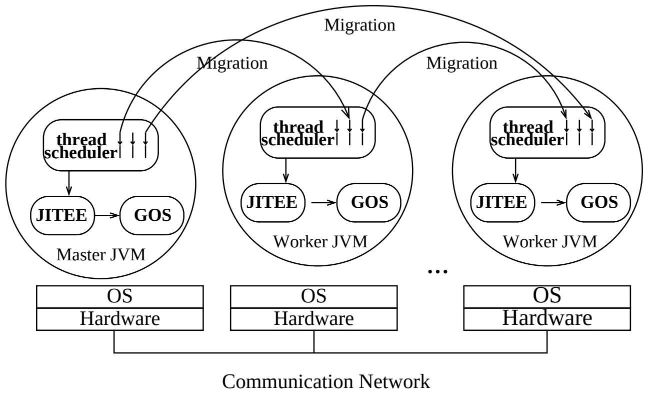

Figure 1. The system architecture.

To create a single system image, the system provides a global space spanning all the nodes for the objects, a method area that is available to all nodes, and support for the transfer of thread stacks needed for thread mobility. In JESSICA2, there is a *global object space* (GOS) that "virtualizes" a single Java object heap across multiple cluster nodes to facilitate transparent shared object access by the master JVM and other worker JVMs. The GOS takes care of all the memory consistency issues, such as object faulting, addressing, replication policy, and those related to object transmission [27, 26]. Therefore during thread migration, we can migrate the objects in the heap by shipping only the Java object references without actually moving the object data to the remote node.

To globalize the method area, we load each class into every JVM. The correctness of rebinding the class structure in the remote node is guaranteed through preserving a single copy of the static data of the class concerned.

Our system does not rely on a shared distributed file system such as NFS; nor does it need to be restricted to a single IP for all the nodes in the running cluster. The system has built-in I/O redirection functionalities to create the single system image view for file and networking I/O operations.

A Java thread can be dynamically scheduled by the thread scheduler to migrate to another node during runtime in order to achieve a balanced system load throughout. Being transparent, the migration operation is completed without explicit migration instructions to be inserted in the source program.

JITEE stands for JIT compiler based execution engine. It is the core of the system. It extends the JIT compiler to support cluster-wide computations, such as thread migration and global object access.

3. JIT-compiler-assisted dynamic thread migration

Dynamic thread migration has long been used as a load balancing tool for optimizing the resource usage in distributed environments [10]. Such systems usually use the *raw thread context* (RTC) as the communication interface between the source node and target node. The RTC usually includes the virtual memory space, the thread execution stack and hardware machine registers.

Existing solutions for Java thread migration mainly use the *bytecode-oriented thread context* (BTC) as the interface. The BTC consists of the identification of the Java thread, followed by a sequence of frames. Each frame contains the class name, the method signature and the activation record of the method. The activation record consists of the bytecode program counter (PC), the JVM operand stack pointer, operand stack variables, and the local variables encoded in a JVM-independent format. There are three main approaches in existing systems: extending a JVM interpreter [18], static bytecode instrumentation [21], and using the JVM Debugger Interface (JVMDI) [13].

To extend a JVM interpreter seems to be the most obvious approach since the interpreter has the complete picture and control of the BTC. However, modifying a JVM interpreter to deal with the BTC adds to the already rather slow execution by the interpreter.

Static bytecode instrumentation can be used to extract limited thread stack information, but the price to pay for is a significant amount of additional high-level bytecodes in *all* the Java class files. This additional amount could result in large space overhead. For example, in JavaGoX [21] and Brakes [24] which use static bytecode instrumentation, about 50% additional space overhead can be observed in running the simple recursive Fibonacci method.

In a JIT-enabled JVM, the JVM stack of a Java thread becomes a native stack and no longer remains bytecode-oriented. As such, JVMDI appears to be a convenient solution. The earlier JVMDI implementations did not support JIT compilers and only the latest JDK from Sun [13] is able to support full-speed debugging using deoptimization techniques that were introduced in the Self compiler [12]. However, JVMDI needs huge data structures and incurs large time overhead in supporting the general debugging functions. Moreover, the JVMDI-based approach needs to have the Java applications compiled with debugging information using specific Java compilers such as *javac* of the Sun JDK, which will not work for Java applications distributed in bytecode format but without debugging information. Furthermore, not all existing JVMs have realized the JVMDI defined in the Sun JDK.

In contrast to the aforementioned approaches, we solve the transformation of the RTC into the BTC directly inside the JIT compiler. Our solution is based on two main functions, *stack capturing* and *stack restoration* (see Figure 2). Stack capturing is to take a snapshot of the RTC of a running Java thread and transforms the snapshot into an equivalent BTC. Stack restoration is to re-establish the RTC using the BTC. Such a process via an intermediate BTC takes advantage of the portability of the BTC.

The general idea of our approach is to use the JIT compiler to instrument the native code so that they become helpful in the transformation of the RTC into BTC. The native code will “spill” the most recent information of variables in the stack at various points, i.e., the latest values will be written to memory from registers. When the migration request arrives, the thread scheduler can perform on-stack scanning to derive the BTC from the RTC instead of using a standalone process to collect the context like what the JVMDI does. For this part of the work, we emphasize simple and efficient solutions that solve the Java thread migration problem without introducing large auxiliary data structures and costly or unnecessary transform functions.

We choose the high-level bytecode-oriented thread context as the execution trace to maintain. The context includes a sequence of Java thread frames. Each frame contains the method id, the bytecode Program Counter (PC), the stack

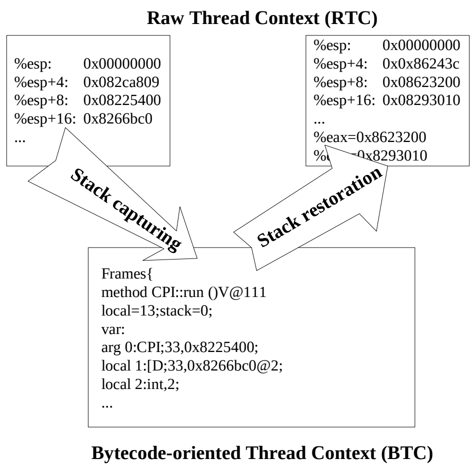

Figure 2. The thread stack transformation.

pointer for the JVM operand stack, the local variables and the JVM stack operands. The above items will be synchronized periodically during thread executions at certain program locations called migration checkpoints. Using the JIT compiler, we are able to perform optimizations to reduce the number of such checkpoints [28].

Figure 3 shows the work flow of the JIT compiler in our system for generating migration checkpointing code to support the thread capturing operation. During the compilation phase, the JIT compiler will try to choose the appropriate migration points upon bytecode verification. A migration point will be marked before at the head of a basic block pointed to by a back edge in the control graph and at the boundary of the following bytecode instructions: INVOKESTATIC, INVOKESPECIAL, INVOKEVIRTUAL and INVOKEINTERFACE. During the bytecode translation phase, the migration points will be translated into a number of native instructions for spilling variables and their types in the thread stack.

To support thread context restoration, we introduce a technique called "dynamic register patching" to rebuild register context just before the control returns to the restored points. Figure 4 illustrates the dynamic register patching on i386 architecture. Shaded areas represent the native codes. "Ret Addr" is the return address of the current function call and "%ebp" is the i386 frame pointer. The dynamic register patching module will generate a small code stub using the register-variable mapping information at the restored point of each method invocation. The thread execution will switch to the code stub entry point for each method invocation. The last instruction to be executed in the code stub will be a branching instruction to jump to the restored point of the method. To make our solution efficient, we allocate the code stub inside the thread stack so that when the stub

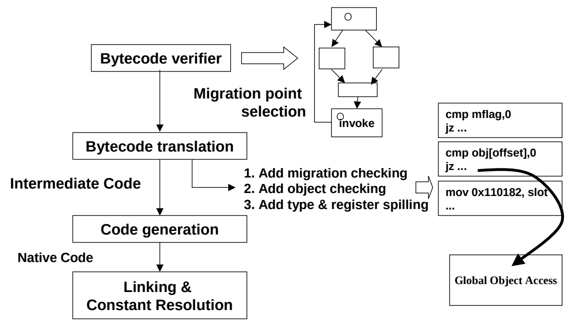

Figure 3. The work flow of the JIT compiler.

jumps to the restored point, the code stub will be automatically freed to avoid memory fragmentation caused by the small-size code stub.

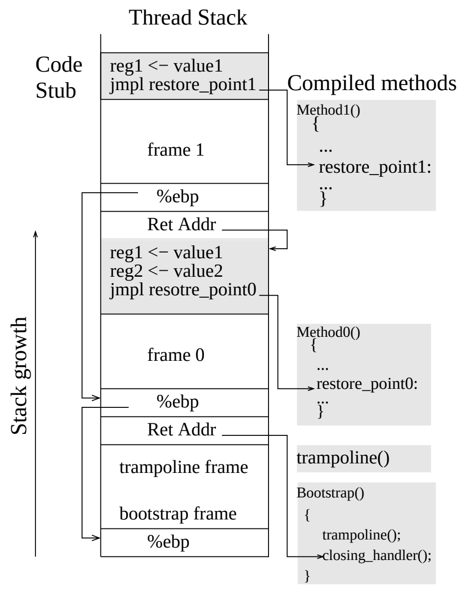

Figure 4. One example of dynamic register patching on i386 architecture.

4. Global object sharing

The JIT compiler also compiles bytecode instructions for object access such as GETFIELD, PUTFIELD, AALOAD, AASTORE, etc., into appropriate interface functions for managing the shared objects. The functions for managing the consistency of shared objects are grouped in a module in

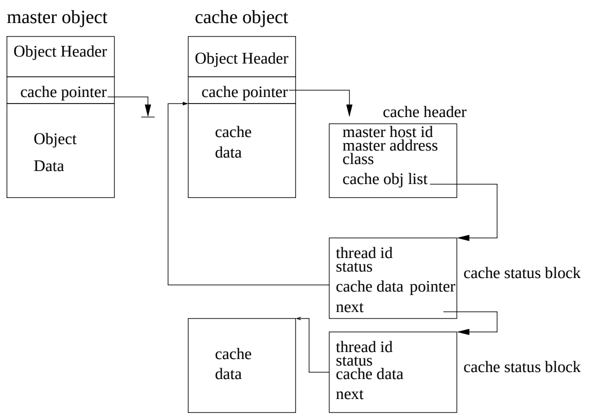

Figure 5. GOS runtime data structure

the DJVM kernel. We call it the *global object space* (GOS) support.

Figure 5 shows the main data structure of the GOS. We added a cache pointer in the original object header to distinguish between a cached object and a master object. For a cached object, the cache pointer will point to a shared cache header for all local threads that need to cache the data from the home of the object. Unlike some implementations such as [2] that use a single cache copy for all the Java threads, we use different cache copies for different local threads. First, it is closer to the JMM. Second, it will prevent different threads from interfering each other's cache copies. For example, in the case of sharing a cache copy by all local threads, if a thread enters a lock and tries to flush all the cache data, data that are still valid for other threads may get flushed. Third, such decision can lead to finer-grain caching, as is done for huge array object. Each thread can cache a portion of one huge array used, and the first cache object will be used as the representative of the object in the local host. The address of the first cache copy can be stored inside other objects in the local host like a master object.

Many state-of-the-art optimizing caching protocols are adopted in our GOS design [26]. We employ an adaptive object "home" migration protocol to address the problem of frequent write accesses to an object remotely, and a timestamp-based fetching protocol to prevent redundant fetching of remote objects. For an object, O, the node that holds its master copy is called the *home* of the object, denoted by $HOME(O)$. Adaptive object home migration means that we dynamically change $HOME(O)$ to another node to avoid the communication cost incurred in having to flush the object and to re-fetch the object as required by the JMM [16]. This can result in substantial message reduction during the execution of Java threads in the DJVM.

Besides, we also embedded into the design several optimizations such as object pushing and fast software state checking by exploiting the JVM runtime information and JIT compiler techniques in the implementation of the GOS. Object pushing is a kind of pre-fetching technique that makes use of the connectivity of Java objects. Based on the internal field definition of an object, we can aggregate the communication messages to transfer several objects in one single message. The pre-fetching level is limited by the the message size in the implementation. Fast software checking is to use the JIT compiler to generate native code for object state checking instead of simply directing it to an interface function.

5. Performance result

JESSICA2 is implemented using the Kaffe open JVM (version 1.0.6) [29] as a base. We use a Linux cluster connected by Fast Ethernet as the testbed. The cluster consists of 2GHz Pentium IV processor running kernel 2.4.18-3. Each machine has 512MB memory.

5.1. Thread migration overheads

We first tested the space and time overheads charged to the execution of Java threads by the JIT compiler after enabling the migration mechanism. Then we measured the latency of a single migration operation.

The time overheads are mainly due to checking at the migration points; and the space overheads are mainly due to the instrumented native code. We did not require the benchmarks to be multithreaded in the test since the dynamic native code instrumentation will function also in single-threaded Java applications.

We used SPECjvm98 [9] benchmark in the test. Table 1 shows the test results. The space overheads are in terms of the average size of native code per bytecode instruction, i.e., the blowup of the native code compiled from the Java bytecode.

From the table we can see that the average time overhead charged to the execution of Java thread with thread migration is about 2.21% and the space overhead due to the generated native code is 15.68%. Both the time and space overheads are much smaller than the reported results from other static bytecode instrumentation approaches. For example, JavaGoX [21] reported that for four benchmark programs (Fibo, qsort, nqueen and compress in SPECjvm98), the additional time overhead ranges from 14% to 56%, while the additional space cost ranges from 30% to 220%.

We also measured the overall latency of a migration operation using different multithreaded Java applications. These applications include a latency test (LT) program, π calculation (CPI), All-pair Shortest Path (ASP), N-Body simulation and Successive Over-Relaxation (SOR). The latency measured includes the time from the point of stack capturing to the time when the thread has finished its stack

Benchmarks	Time (seconds)		Space (native code / bytecode)	
	No migration	Migration	No Migration	Migration
compress	11.31	11.39(+0.71%)	6.89	7.58(+10.01%)
jess	30.48	30.96(+1.57%)	6.82	8.34(+22.29%)
raytrace	24.47	24.68(+0.86%)	7.47	8.49(+13.65%)
db	35.49	36.69(+3.38%)	7.01	7.63(+8.84%)
javac	38.66	40.96(+5.95%)	6.74	8.72(+29.38%)
mpegaudio	28.07	29.28(+4.31%)	7.97	8.53(+7.03%)
mtrt	24.91	25.05(+0.56%)	7.47	8.49(+13.65%)
jack	37.78	37.90(+0.32%)	6.95	8.38(+20.58%)
Average		(+2.21%)		(+15.68%)

Table 1. The execution overheads using SPECjvm98 benchmarks.

restoration in the remote node and has sent back the acknowledgement. Table 2 shows the overall migration latency of LT, CPI, ASP, N-Body and SOR. CPI only needs 2.68 ms to migrate and restore thread execution because it only needs to load one single frame and one Java class during the restoration. LT and ASP need about 5 ms to migrate a thread context consisting of one single frame and restore the context. Although they only have one single frame to restore, they both need to load two classes inside their frame contexts. For SOR which migrates two frames, the time is about 8.5 ms. For N-Body, which needs to load four classes in 8 frames, it takes about 10.8 ms.

In addition, the breakdown of the latency test program LT is shown. LT accepts a parameter for the number of nested function calls, so that we can migrate different numbers of Java frames in different tests using the same program. Using LT, we give a fine-grain view of the various steps inside the migration mechanism. These steps include stack capturing, frame parsing, cloning a thread, partial compilation to retrieve the register mapping, and the restoration of the frames.

Table 3 shows the migration time breakdown of LT. The first three rows show the information about the bytecode context migrated, including the frame number, the number of variables of all the frames, and the size of the frame context in JVM-independent format. The last five rows show the breakdown of each major step in the migration mechanism with different frame numbers between 1 and 10. The capturing time, frame parsing time, compilation time and stack building time are linear functions of the size of the frame.

5.2. GOS optimization evaluation

We also evaluated the effect of GOS optimization techniques in reducing the communication messages. We used the multithreaded Java programs including the Travel Salesman Problem (TSP) program, the N-Body simulation program, and the ASP program.

We used 8 nodes in the test. The problem size for TSP is 14 cities. In N-Body we used 512 objects. And for ASP we used a graph of 512 vertices. Figure 6 shows normalized result of message reduction after applying the adaptive object home migration protocol (H), object pushing (P) and time stamp based object fetching (T), respectively.

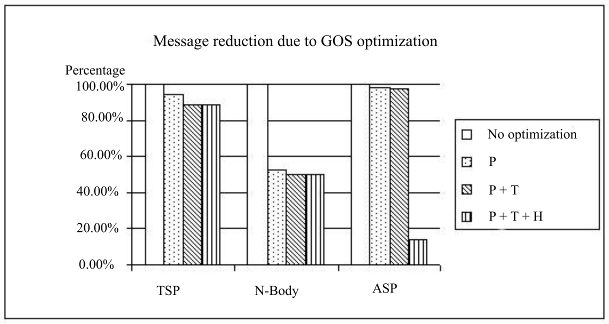

Figure 6. GOS optimization results.

For TSP, the object pushing and time stamp based technique reduced the number of messages by about 10% and the adaptive object home migration did not reduce the number. The reason is that in TSP, only limited objects such as the shortest path array are updated during the execution. In N-Body, object pushing introduces a significant reduction in number of messages because it needs to create new objects in rebuilding the Barnes & Hut tree. For ASP, the adaptive object home migration protocol reduced the number of messages to 13.50% because it needed to update a large number of objects during each iteration in calculating the shortest path.

Program (frame #)	LT (1)	CPI (1)	ASP(1)	N-Body(8)	SOR(2)
Latency (ms)	4.997	2.680	4.678	10.803	8.467

Table 2. Overall migration latency of different Java applications. The number in parentheses is the number of frames.

Frame#	1	2	4	6	8	10
Variable#	4	15	37	59	81	103
Size (bytes)	201	417	849	1281	1713	2145
Capture (us)	202	266	410	495	605	730
Parse (us)	235	253	447	526	611	724
Create (us)	360	360	360	360	360	360
Compile (us)	478	575	847	1,169	1,451	1,720
Build (us)	7	11	14	16	21	28
Total (us)	1,282	1,465	2,078	2,566	3,048	3,562

Table 3. Migration breakdown of latency test (LT) program for different frame sizes.

5.3. Application speedup

In this section, we present the speedup of four mulithreaded applications.

We ran TSP with 14 cities, Raytracer with a 150x150 scene containing 64 spheres, and N-Body with 640 particles in 10 iterations. We give the speedups of CPI, TSP, Raytracer and N-Body in Figure 7 based on comparing the execution time of JESSICA2 and that of 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 N-Body 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.

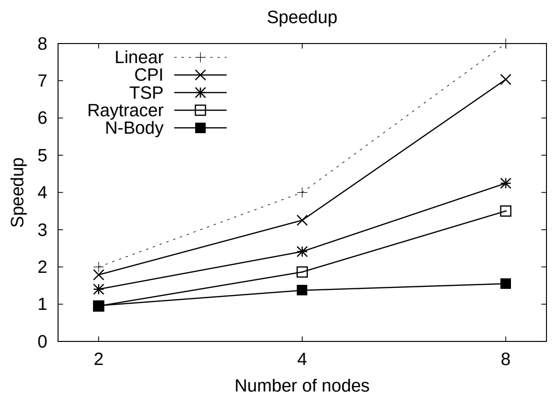

Figure 7. Speedup Measurement of Java applications

6. Related work

Parallel programming on a parallel computer such as a cluster is far from an easy task. Compared with sequential programming, parallel programming raises new issues such as data sharing, synchronization, and load balancing. Several programming paradigms exist for parallel computing, namely, *data parallel*, *message passing*, and *shared memory*. A substantial amount of effort, either using a pure runtime system or a combination of compiler and runtime system, has been spent in supporting these programming paradigms during the past two decades [15, 7, 14, 6, 5, 1]. Our effort was motivated by recent work on distributed JVM, particularly cJVM [3] and JESSICA [18]. Several design issues encountered in our GOS and thread migration are also related to software DSM systems and classical thread/process migration research. These techniques are exploited in our JESSICA2 system in the context of JIT compilation.

6.1. Software Distributed Shared Memory

Software-based Distributed Shared Memory (DSM) has been a subject for intensive study for a decade or more. Orca [5] is an object-based DSM that uses a combination of compile-time and runtime techniques to determine the placement of objects. Our GOS differs from Orca in that we provide the shared object abstraction supports entirely at runtime through the JVM JIT compiler.

TreadMarks is a page-based DSM [1] that adopts lazy release consistency protocols and allows multiple concurrent writers writing to the same page. Treadmarks uses hardware page-faulting support, and therefore it can eliminate the overheads of software checking on object status. One of the drawbacks, however, is that page-based DSM tend to have problems of false sharing if directly applied to an object-based language such as Java.

6.2. Computation migration

Computation migration has also been studied for many years. Process migration can be regarded the ancestor of thread migration. The paper [19] reviews the field of process migration up to 1999. It provides a detailed analysis on the benefits and drawbacks of process migration. The systems that are mentioned in the paper range from user-level migration systems to kernel-level migration ones. In contrast to existing computation migration techniques, we try to solve the computation migration problem from a new perspective by bringing in JIT compilation into the solution.

There have been systems developed to support thread migration. Arachne [10] is one of them. It provides a portable user-level programming library that supports thread migration over a heterogeneous cluster. However, the thread migration there is not transparent to the user as it requires that programs be written using a special thread library or API. Cilk [6] is a system that is very similar to our system in supporting C-based thread migration for C. It allows the programmer to explicitly and aggressively create new threads to maximize the concurrency. Cilk's runtime system then takes the responsibility to schedule the threads to achieve load balance and improved performance.

There are related systems in the mobile computing area that support the mobility of Java threads. For example, JavaGoX [21] and and Brakes [24] both use a static preprocessor to instrument Java bytecodes to support the migration of Java threads. These systems do not address the distributed shared object issues.

6.3. Distributed JVMs

Nearly all DJVMs are based on a set of cooperative JVMs running on different cluster nodes. Among them, the cJVM [3] prototype was implemented by modifying the Sun JDK1.2 interpreter. cJVM does not support thread migration. It distributes the Java threads at the time of thread creation.

There are other DSM-based DJVM prototypes, including JESSICA [18] and Java/DSM [30]. Both systems are based on the Java interpreter. JESSICA supports thread migration by modifying the Java interpreter. Java/DSM lacks support for the location transparency of Java threads. It needs programmers' manual coding to place the threads in different cluster nodes.

Jackal [25] and Hyperion [2] adopt static compilation approaches to compile the Java source code directly into native parallel code. The parallel code is linked to some object-based DSM library package. Such a static compilation approach, however, will usually disable some useful features of Java such as dynamic class loading. As a result, the output parallel code is not fully compliant with the Java language specification.

7. Conclusion and future work

In the JESSICA2 project, we combine thread migration and distributed shared object to build a high-performance single-system image cluster middleware. The implementation is at the JVM level, and the system can transparently execute multithreaded Java applications in parallel without any modification. The novelty is in the application of a JIT compiler to combine and optimize the above two features. The following components of the original single-node JVM have been modified or extended: the JIT compiler, the class loader, the heap and the I/O class libraries. Our experimental results confirm that it is feasible to deliver high performance for multithreaded execution using Java through the support of a Distributed JVM in a cluster environment.

Possible future work will try to deploy other more advanced JIT compilers to further optimize the thread migration mechanism and distributed object sharing. It is possible that with some advanced compilation analysis technique such as escape analysis [8], the object checking overhead can be much reduced. For the realization of the complete SSI ideal for all Java applications including GUI applications, more supports will be needed in areas including Java I/O libraries (such as the AWT or the Swing library). Also, in a large cluster environment, the master node could easily turn into a bottleneck for I/O requests; new techniques for duplicating the master node can be explored.

References

[1] 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.

[2] G. Antoniu et al. The Hyperion System: Compiling Multithreaded Java Bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

[3] 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.

[4] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. High Performance Numerical Computing in Java: Language and Compiler Issues. In *Languages and Compilers for Parallel Computing*, pages 1–17, 1999.

[5] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance Evaluation of the Orca Shared-Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998.

[6] R. D. Blumofe, C. F. Joerg, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[7] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++ — A C++ Dialect for High Performance Parallel Computing. *Lecture Notes in Computer Science*, 1049, 1996.

[8] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape Analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[9] T. S. P. E. Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/org/jvm98, 1998.

[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] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[12] U. Hlzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, USA, June 1992.

[13] S. M. Inc. Java Platform Debugger Architecture, 2002.

[14] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[15] C. Koelbel, D. Loveman, et al. *The High Performance Fortran Handbook.* . The MIT Press, Cambridge, MA, 1994.

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

[17] C. Mangione. Performance test show Java as fast as C++. Technical report, JavaWorld, 1998.

[18] Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Parallel and Distributed Computing*, 60(10):1194–1222, October 2000.

[19] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[20] W. Pugh. Fixing the Java Memory Model. In *Java Grande*, pages 89–98, 1999.

[21] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Joint Symposium on Agent Systems and Applications / Mobile Agents*, pages 16–28, 2000.

[22] T. U. o. H. K. The System Research Group, The Department of Computer Science. HKU Gideon Cluster. http://www.srg.csis.hku.hk/gideon/, 2004.

[23] TOP500.ORG. Top500 supercomputer sites. http://www.top500.org.

[24] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, pages 29–43, 2000.

[25] R. Veldema, R. F. H. Hofman, R. Bhoedjang, and H. E. Bal. Runtime optimizations for a Java DSM implementation. In *Java Grande*, pages 153–162, 2001.

[26] Weijian Fang, Cho-Li Wang, and Francis C.M. Lau. Efficient Global Object Space Support for Distributed JVM on Cluster. In *The 2002 International Conference on Parallel Processing (ICPP-2002)*, pages 371–378, British Columbia, Canada, August 2002.

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

[28] Wenzhang Zhu, Cho-Li Wang, and F. C. M. Lau. Lightweight Transparent Java Thread Migration for Distributed JVM. In *International Conference on Parallel Processing*, pages 465–472, Kaohsiung, Taiwan, October 2003.

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

[30] W. Yu and A. L. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency - Practice and Experience*, 9(11):1213–1224, 1997.