Lightweight Transparent Java Thread Migration for Distributed JVM*

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

Abstract

A distributed JVM on a cluster can provide a high-performance platform for running multi-threaded Java applications transparently. Efficient scheduling of Java threads among cluster nodes in a distributed JVM is desired for maintaining a balanced system workload so that the application can achieve maximum speedup. We present a transparent thread migration system that is able to support high-performance native execution of multi-threaded Java programs. To achieve migration transparency, we perform dynamic native code instrumentation inside the JIT compiler. The mechanism has been successfully implemented and integrated in JESSICA2, a JIT-enabled distributed JVM, to enable automatic thread distribution and dynamic load balancing in a cluster environment.

We discuss issues related to supporting transparent Java thread migration in a JIT-enabled distributed JVM, and compare our solution with previous approaches that use static bytecode instrumentation and JVMDI. We also propose optimizations including dynamic register patching and pseudo-inlining that can reduce the runtime overhead incurred in a migration act. We use measured experimental results to show that our system is efficient and lightweight.

Keywords: distributed JVM, multi-threading, Java thread migration, mobility, JIT compiler,

1 Introduction

The Java programming language supports threads and provides concurrency constructs at the language level for thread-based parallel computing. It is a more portable parallel programming tool than many other existing parallel languages or libraries for parallel computing. In contrast to message passing systems such as MPI, multi-threaded Java applications favor the alternative shared memory programming paradigm.

Cluster is becoming important for high-performance computing. Therefore, it is worth studying the possibility of extending the JVM to run on clusters in such a way that the execution of a single multi-threaded Java program can span multiple machines. Such an extended JVM is a "distributed JVM". An ideal distributed JVM can provide a *single system image* (SSI) to multi-threaded Java applications, which is a much desired feature in a cluster.

One of the essential features of a distributed JVM realizing SSI is the transparent migration of Java threads. This should happen not just once at the start of execution, but dynamically during runtime in order to achieve a balanced system load throughout. It is only through maintaining a balanced load will the system be able to achieve maximum speedup for its applications. Being transparent, the migration operation is done without explicit migration instructions to be inserted in the source program by the programmer. The runtime system would provide all the support necessary to schedule a migration when such a need arises. The migration will still be carried out efficiently and the application is unaware of the migration operation.

Among the many challenges in realizing a migration mechanism for Java threads, the transferring of thread contexts between cluster nodes requires the most careful design. One could use the raw thread context for the purpose, as is done in C/C++ thread migration systems [3]. Such systems however exhibit poor portability simply because the C/C++ thread context by design is not portable. For example, all nodes may need to reserve the same virtual address in order to properly access a stack variable [3]. On the contrary, Java threads operate with a bytecode-oriented context which is highly portable. This bytecode-oriented

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

thread context is understood by bytecode instructions, and no machine-dependent information will ever appear inside the bytecode context.

For parallel computing to achieve high performance, the JIT compilation mode is very much a necessity. The practical goal of our work to extend a JIT-enabled JVM is to provide an efficient transparent thread migration mechanism. We address the following issues.

- **Lightweight**. The JIT mode offers much higher performance than the interpreter mode. Hence, the migration overheads are more sensitive to the overall performance. Runtime overheads in terms of time and space to support thread migration should be minimized.

- **Dynamic**. Any preprocessing of Java code or bytecode of applications should be avoided so that a large variety of multi-threaded Java applications distributed in bytecode format can be downloaded and executed during runtime on our system.

- **Transparent**. The system should not introduce any special API for Java threads to make explicit calls for migration. The entire migration operation should be transparent to Java threads.

Based on the proposed design, a transparent Java thread migration mechanism has been implemented and successfully integrated in our distributed JVM, JESSICA2 [13], which allows JESSICA2 to perform automatic thread distribution and dynamic load balancing on a Linux PC cluster. This paper differs from a previous paper for the same project [13] in that it provides in-depth discussion on the design principles, implementation techniques and performance evaluation of the thread migration mechanism.

The rest of the paper is organized as follows. Section 2 discusses the overview of the transparent Java thread migration system. Sections 3 and 4 discuss the two main components of the system for stack capturing and stack restoration, respectively. Section 5 presents the experimental results. Section 6 discusses related work. The paper ends with a conclusion in Section 7.

2 Overview

2.1 Distributed Java Virtual Machine

A Distributed Java Virtual Machine (DJVM) is a middleware that supports parallel execution of multithreaded Java applications in a distributed system. DJVM supports the scheduling of Java threads on cluster nodes and provides location transparency on object access and I/O operations for Java threads. The semantics of Java thread execution on a DJVM will be preserved just as if it were executed in a single node. From the viewpoint of a multi-threaded Java application, the DJVM offers an SSI illusion.

JESSICA2 [13] is a DJVM running on a PC cluster to provide a single system image to multi-threaded Java applications. During runtime, the Java threads can be automatically migrated from one node to another to achieve dynamic load balancing. The JIT compilation support of the migration mechanism significantly improves the performance of JESSICA2 over the previous JESSICA project [7] which works only in interpreter mode. Figure 1 shows the overall architecture of JESSICA2. JITEE stands for JIT compiler based execution engine. The *global object space* provides a single Java object heap across multiple cluster nodes to facilitate location transparent object access in a distributed environment.

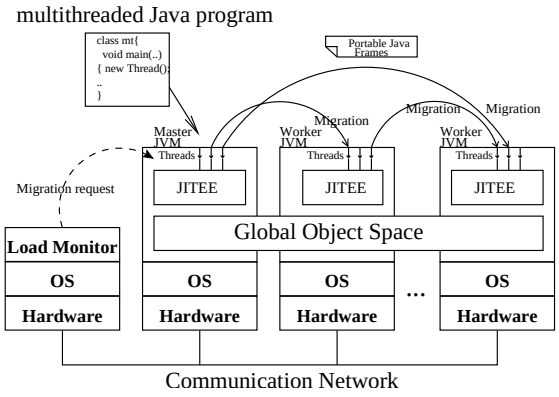

Figure 1. Overall architecture of JESSICA2.

2.2 Transparent Java thread migration

Transparent thread migration has long been used as a load balancing mechanism to optimize the resource usage in distributed environments [3]. Such systems usually use the *raw thread context* (RTC) as the communication interface between the migration source node and target node. RTC usually includes the virtual memory space, thread execution stack and hardware machine registers.

Existing solutions for Java thread migration mainly use *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 bytecode program counter(PC),

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 [7], static bytecode instrumentation [10], and using the JVM Debugger Interface (JVMDI) [5, 8].

To extend a JVM interpreter seems to be an 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 overheads. For example, in JavaGoX [10] and Brakes [11] 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 native stack and no longer remains bytecode-oriented. In the face of this, JVMDI is a convenient solution. The earlier JVMDI implementations did not support JIT compilers and only the latest JDK [1] from Sun is able to support full-speed debugging using deoptimization techniques that were introduced in the Self compiler [4]. 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 the *javac* in Sun JDK, which will deny many Java applications distributed in bytecode format but without debugging information. Furthermore, not all existing JVMs have realized the JVMDI defined in Sun JDK.

2.3 Our solution

In contrast to the aforementioned approaches, we solve the transformation of the RTC into the BTC directly inside the JIT compiler. Our solution is built 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 following two sections discuss in detail the operation of these two important functions, and optimizations that help to reduce the time overheads and the memory footprint.

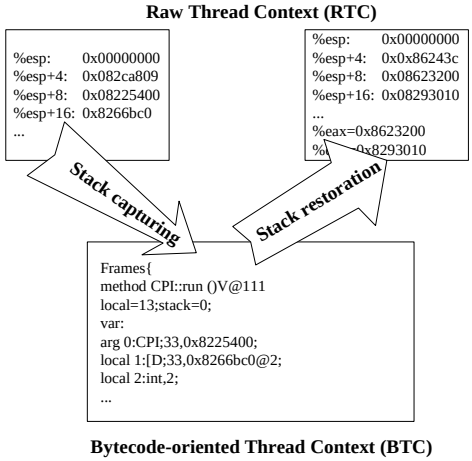

Figure 2. The thread stack transformation.

3 Stack capturing

To capture a thread stack in JIT compilation environments, we identified the following items that are needed to be transformed from RTC into BTC: method id, bytecode Program Counter(PC), stack pointer for JVM operand stack, the local variables and the JVM stack operands.

The general idea of our approach is to use the JIT compiler to instrument native codes which help the transformation of the RTC into BTC. These native codes will spill the most recent information of variables in the stack at some points, i.e., the latest values will be written back 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 stand-alone process to collect the context like JVMDI. During this process, we emphasize simple and efficient solutions that solve the Java thread migration problem without introducing large volume of auxiliary data structures and costly or unnecessary transform functions.

3.1 Migration points and pseudo-inlining

The BTC requires that the bytecode PC be well-defined so that a thread must be stopped at a point that has equivalent bytecode PC. In other words, the stopped point should be at the bytecode boundary. However, when a thread is stopped by the scheduler and is chosen to be the migration candidate, it is most likely running at some point of native codes that is not at the bytecode boundary. It may be very hard to "slide" the execution by simulating the execution

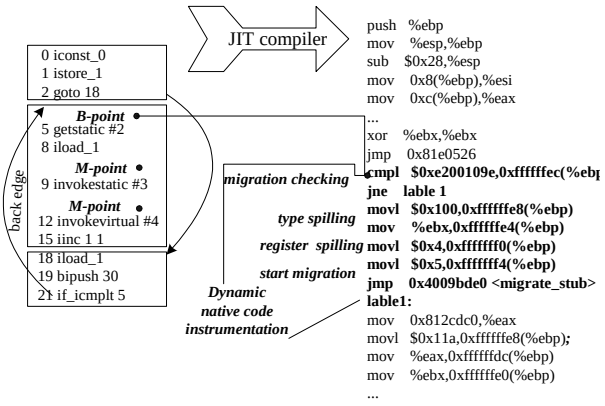

Figure 3. Dynamic native code instrumentation.

of native instructions from the stopped point to the next immediate bytecode boundary. In fact a transparent thread migration system does not need such fine-grained breakpoints. As long as the migration request can be acknowledged within a reasonable time, say a few microseconds, it still makes sense. In our system, instead of stopping and sliding, we use checking at some specific points in native codes. Such points are called *migration points*. The BTC will be consistent with the RTC at such points, i.e., the semantics of the stack context are identical to both BTC and RTC at the migration points. When the migration request is issued by JVM, the thread will delay the acknowledgement until it reaches the next migration point.

Generally, all points at the bytecode boundary can be chosen as the migration points. However, checking at all points will degrade the execution performance dramatically. We choose two types of points in our system. The first type (referred as M-point) is the site that invokes a Java method. The second type (referred as B-point) is the beginning of a bytecode basic block pointed by a back edge, which is usually the header of a loop. The concepts of migration points and dynamic code instrumentation are illustrated in Figure 3.

The M-point is necessary because we need to make sure that a frame should have consistent BTC before it is pushed in the stack so that later capturing can get the correct BTC from the pushed stack frame. At such points, we need to spill the values and types of variables, bytecode PC and stack pointer to the memory slots in the thread stack. We also have one test instruction to check if the migration request is issued.

The M-point will add overheads to the thread execution and too many migration points will lead to a performance degradation caused by the blowup in code size. The observations that short methods can skip the migration checking without delaying the migration responsive time too much lead to the following decisions made in our system: We treat Java library method invocations, which usually last for a relatively short time, as "straight" code sequences, i.e., no migration points will be inserted before such method invocations. Nevertheless, the advantage of such a decision is that the context will become more portable as the context contains only application methods. And such a decision can be generalized to inlined methods which are typically tiny. As migration will not happen inside an inlined methods, no additional efforts are needed to transform an inlined stack to a normal stack as used in the deoptimization technique (A debugger, however, needs this mechanism to support the user's request to breakpoint at inlined methods.).

The B-point is used to prevent a thread from being unable to respond to the migration request in a reasonable time when it is running inside a loop. If adopting the same spilling used for M-point, it will be much costly for JIT compilers to perform so many memory operations at each iteration in a loop. We observed that no spilling is needed if no migration request is issued. We check the migration request first. If no request happens at the migration point, no spilling will be performed. Therefore during normal execution, each iteration in a loop needs only one additional flag checking. Note that M-point can not have such optimization, because if a frame is not consistently pushed in the stack, the capturing caused by the later migration request will get the wrong data from the stack.

As Java applications typically have many small-size methods, if a JIT compiler has inlining optimization, the migration checking can be eliminated dramatically as many M-points will be eliminated. For a JIT compiler that does not introduce method inlining optimization, we propose a *pseudo-inlining* technique to eliminate the checking overheads with the same effect as inlining optimization. "Pseudo" means that the method is not actually inlined by the compiler. Rather our M-point checking tries to treat it as if it was inlined (see Figure 4). A small-size method is considered as an pseudo-inlined candidate if the method contains no further method invocations. The M-point will not place any checking and spilling if the callee is an inlined candidate. The B-point will not be called for inside such candidate methods either.

3.2 Type spilling

The thread context includes the values of stack operands together with their types. As stack operands

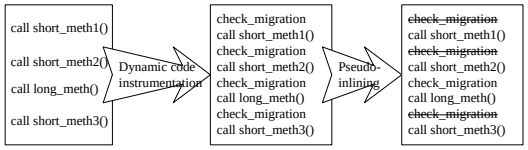

Figure 4. Example of Pseudo-inlining.

are dynamically pushed into or popped from the thread stack during thread execution, their types cannot be determined in advance. For example, the bytecode instruction "*f2d*" (convert *float* value to *double* value) will pop off a *float* variable from operand stack and push a *double* operand on the stack top. In Sumatra [9], it is proposed to use a separated type stack operating synchronously in the JVM interpreter during thread execution, so that at the time of migration the operand type can be known. Although such a method can be used in the case of JIT compilers, it doubles the operation time to access the stack operand.

To tackle this problem in JIT compilers, we choose to perform the type spilling at the migration points discussed above. The type information of stack operands at migration 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 the stack operand as we can deduce the real type of a Java object from the object header. We choose one encoding for each of primitive types. 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 at the migration points 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 operands in a method, which results in better performance improvement than the synchronous type stack method used in Sumatra [9].

4 Stack restoration

In this section we will discuss how to restore the execution of the migrated thread from the point it was stopped, given the BTC, the JVM-independent thread stack context as input. The approach of restoration a Java thread execution under JIT mode has rarely discussed in related projects. In interpreter-based JVM, a simple frame-by-frame interpretation mode can be used [7]. For static bytecode instrumentation approach, instrumented bytecodes will simulate the calling sequences [11].

Both approaches can not fit well in a JIT-enabled JVM, because the native codes compiled from the bytecodes of Java methods may assume certain usage of hardware registers at the restored points. We use a scheme called "dynamic register patching" in JIT compilers to rebuild register context.

4.1 Startup and closing

As the input is in the JVM-independent text format, the initial step to restore thread execution needs to quickly parse the input. The parser was written using YACC. Invalid inputs will be rejected by the JVM daemon thread which is responsible for accepting incoming thread migration. For valid inputs, a data structure containing the stack context will be created for later processing. As the daemon thread needs to handle all the requests from other JVMs, we can not use it to restore the execution of the migrated Java thread. Instead a new native thread will be created by the daemon thread and the input context will be assigned to it. Then the newly created thread becomes the clone of the migrated thread in current JVM. Given the stack context data structure as the input, the clone thread will start the bootstrapping.

The thread will load the necessary classes in the thread stack context and resolve the reference variables. After that, it will bring back the calling sequence as described by the input context, which is the most difficult task in the bootstrapping. In our system, we build a sequence of stack frames with the returning addresses and the frame pointers properly linked together to simulate the method invocation. The local variable inside the frames will be initialized to the values according to the input thread context. Next the dynamic register patching module will generate small code stubs to handle the restoration of machine registers. The detail of dynamic register patching will be discussed in next subsection.

A trampoline function will then be used to swap the current stack frame with the newly created stack frames. It also makes sure that upon completion the thread will return the control to the closing handling function. The closing handling function will collect the return data and notify the migration source. Then the thread can terminate its migration journey.

4.2 Dynamic register patching

In our system, we introduce a scheme called "dynamic register patching" to rebuild register context just

before the control returns to the restored points. The register-variable mapping information comes from the JIT compilers. 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 one 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 control jumps to the restored point, the code stub will be automatically freed to avoid memory fragmentation caused by the small-size code stub. Figure 5 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.

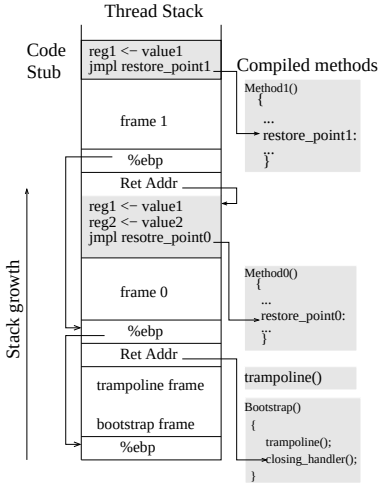

Figure 5. Dynamic register patching on i386 architecture.

5 Experimental results

To evaluate the proposed transparent Java thread migration mechanism, we use our distributed JVM, JESSICA2, which is developed based on Kaffe open JVM 1.0.6 [6] and runs on the HKU Gideon 300 Linux cluster. The cluster consists of 300 2GHz Pentium 4 PCs running Linux kernel 2.4.14 that are connected by a 312-port Foundry Fastiron 1500 Fast Ethernet switch.

5.1 Space and time overhead

We first measure the execution overheads caused by the migration mechanism. The time overhead is mainly due to the checking at the migration points; and the space overheads are mainly due to the instrumented native code at the migration points.

We use SPECjvm98 [2] benchmark in the tests. The initial heap size is set to 48 MB. We compare the differences in time and space costs between enabling and disabling the migration checking at migration points. We also measure the impact of using pseudo-inlining.

Table 1 shows the time and space overheads caused by thread migration, when migration checking is disabled (M-), enabled but without pseudo-inlining (M+I-), or enabled with pseudo-inlining turned on (M+I+). The space overhead is in terms of the average length of native code per bytecode instruction.

From the table we can see that if pseudo-inlining is disabled, on average, the time overhead reaches 36.33%, and the space overhead reaches 42.74%. However, if we enable pseudo-inlining, the average time overhead charged to the execution of Java thread with thread migration drops to 3.66% and the generated native code overhead becomes 15.73%. The additional costs in the case of pseudo-inlining being disabled are caused by the checking at migration points for the short methods.

Both the time and space overheads after applying the pseudo-inlining technique are much smaller than the reported results from other static bytecode instrumentation approaches. For example, JavaGoX [10] 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%.

For JVMDI-based approaches, we use the fibo and the nqueen program to measure the space cost caused by the embedded debugging information using the *javac* in Sun JDK 1.4. Both programs need about an additional 25% space for the debugging information in the class files.

5.2 Migration latency and breakdown

We also measured the overall latency of a migration operation using different multi-threaded Java applications. These applications include a latency test (LT) program, π calculation (CPI), All-pair Shortest Path (ASP), NBody 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 restoration on the re-

Table 1. SPECjvm98 benchmarks.

Benchmarks	Time overhead		Space overhead	
	M+I-/M-	M+I+/M-	M+I-/M-	M+I+/M-
compress	178.16%	106.78%	121.45%	110.18%
jess	116.34%	101.38%	160.78%	122.27%
raytrace	183.04%	106.38%	149.70%	113.65%
db	100.73%	101.51%	119.51%	108.86%
javac	113.33%	102.91%	180.30%	129.52%
mpegaudio	125.20%	104.96%	118.09%	107.13%
mtrt	171.96%	104.30%	149.73%	113.66%
jack	101.91%	101.04%	142.38%	120.58%
Average	136.33%	103.66%	142.74%	115.73%

Table 2. Migration breakdown.

Frame#	1	2	4	8
Variable#	4	15	37	81
Size(bytes)	201	417	849	1713
Capture(μs)	202	266	410	605
Parse(μs)	235	253	447	611
Create(μs)	360	360	360	360
Compile(μs)	478	575	847	1,451
Build(μs)	7	11	14	21
Total(μs)	1,282	1,465	2,078	3,048

mote node and has sent back the acknowledgement. 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 5.0 ms and 4.7 ms respectively to migrate a thread context consisting of one single frame and to 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 NBody, which needs to load four classes in 8 frames, the time is about 10.8 ms.

In additional, the breakdown of the latency test program LT is shown. LT accepts a parameter to specify the nested level 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 different steps inside the migration mechanism. These steps include stack capturing, frame parsing, cloning a thread, partial compilation to retrieve the register mapping, and to build the new frame layout.

Table 2 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 inside all 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 ranging from 1 to 8. In this breakdown, the time to create the clone thread is constant, and its average time is about 360 microseconds for different frame sizes. The capturing time, frame parsing time, compilation time and stack building time are a linear function of the size of the frame. The total cost of all these operations listed in last row shows that the major steps of the migration only charge about 1.282 milliseconds to the overall migration latency for migrating one frame, and less than 3.05 milliseconds for migration up to 8 frames.

5.3 Discussion

Compared with other proposed systems realizing limited thread migration, our system provides a higher performance platform with dynamic load balancing achieved through Java thread migration in running multi-threaded Java applications. The lower cost in execution with the migration mechanism enabled (M+I+) in the first part of the evaluation speaks for the high-performance execution of Java threads in the migration system. In the tests using the SPECjvm98 benchmark, the time overhead is about 3.66% on average and the space overhead is about 15.73%, which points to the fact that using dynamic native code instrumentation in JIT compilers inside the JVM is a promising solution to achieving high performance in applications with thread migration.

In the latency test, we observe that communication costs for transferring thread contexts and class loading from local disk dominate the overall migration latency (about 74% in the LT program). The major steps in the migration mechanism excluding the communication costs contribute only a small part of the overhead. This attests to the lightweight characteristics of our design in terms of time overhead of the migration operation inside the JVM.

6 Related work

cJVM [12] is a cluster-aware JVM that provides SSI of a traditional JVM running on cluster environments. The cJVM 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.

JavaGoX [10] and Brakes [11] use static bytecode instrumentation to realize transparent Java thread migration. Unlike their approach, our solution instruments native code during runtime, and only instruments executed methods.

Sumatra [9] extends the JVM interpreter to enable

capturing and restoring of Java thread context in order to support resource aware mobile threads. The interpreter-based thread migration mechanism, however, will result in poor execution performance compared to the approach based on JIT compilers as used in our system.

M-JavaMPI [8] uses JVMDI to support transparent migration of single-threaded Java process to achieve dynamic load balancing. Migrated processes can continue their MPI communication with other processes. Our system supports the migration of threads, which suits best multi-threaded Java applications.

7 Conclusion

We have presented a lightweight solution to transparent Java thread migration in a JIT-enabled JVM based on dynamic native code instrumentation and dynamic register patching. Our dynamic native code instrumentation is different from existing static bytecode instrumentation approaches in that it instruments finegrain native code on demand at runtime so that it is able to preserve the important features of Java such as dynamic class loading. The approach puts little constraints on the Java bytecode distribution, which is in contrast to the JVMDI-based thread migration approach which requires embedding debugging information in Java class files. The dynamic register patching scheme represents a new way to solve the restoration of the native Java thread stack in a JIT-enabled JVM.

Our design represents a balance between traditional native thread migration at the system level and static bytecode instrumentation at the user level. It uses the portable Java thread context as an interface to glue together independent JVMs running in different nodes. Our solution preserves high-performance JIT compilation execution in the presence of thread migration.

References

[1] Java Platform Debugger Architecture. http://java.sun.com/j2se/1.4.1/docs/guide/jpda/.

[2] The Standard Performance Evaluation Cooporation. SPEC JVM98 benchmarks. http://www.spec.org/org/jvm98, 1998.

[3] 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.

[4] U. Hlzle, C. Chambers and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *the SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[5] Torsten Illmann, Tilman Krueger, Frank Kargl and Michael Weber. Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture. In *Proceedings of the MA'01*, Atlanta, USA, December 2001.

[6] Transvirtual Technologies Inc. Kaffe Open VM. http://www.kaffe.org.

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

[8] Ricky Ma, Cho-Li Wang and Francis C.M. Lau. M-Javampi: A Java-mpi Binding with Process Migration Support. In *The Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, 2002.

[9] Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, USA, 1997.

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

[11] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen and Pierre Verbaeten. Portable Support for Transparent Thread Migration in Java. In *ASM*, pages 29–43, 2000.

[12] M. F. Yariv Aridor and A. Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[13] 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.