A NEW TRANSPARENT JAVA THREAD MIGRATION SYSTEM USING JUST-IN-TIME RECOMPILATION

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

ABSTRACT

Thread migration is to support the movement of threads across machine boundaries in a distributed computing environment. It can improve load balancing and the execution efficiency of multithreaded programs. In this paper, we introduce a new approach that employs the technique of Just-in-Time (JIT) recompilation to support transparent thread migration. With JIT recompilation, the native thread execution mode is preserved, and much of the space and time overheads of previous solutions based on code instrumentation can be eliminated. The new thread migration system is integrated into the JESSICA2 distributed JVM. The measured results show that our approach is beneficial to the overall system in supporting the transparent execution of Java applications on clusters.

KEY WORDS
Just-in-Time compiler, Java, Thread migration, JVM

1 Introduction

Java is increasingly popular in high-performance computing circles due to its portability and built-in support for multithreaded programming. In most situations, however, a multithreaded Java program runs with virtual parallelism in a single machine. An attractive direction is to extend the Java execution environment—i.e., the Java Virtual Machine (JVM)—to take advantage of the multiplicity of processing nodes in a cluster or the like. The ideal is to have a group of JVMs running on distributed cluster nodes to work together as a single, virtual JVM to support true parallel execution of a multithreaded Java application. With true parallelism, threads in an application can be executed in different nodes in the cluster simultaneously to achieve better overall efficiency. We call such an execution environment a distributed JVM (DJVM) [1, 4, 8]. Other than providing parallel execution support to threads, the DJVM maintains a virtual shared memory space to contain the objects that need to be accessed by the executing threads. This paper focuses on the parallel execution support.

The best form of support for parallel execution of Java threads should include low-cost, efficient movements of threads during runtime. Therefore, one of the desired features of a DJVM is to support transparent thread migration. A thread is a fine-grained computation unit with a lightweight context. Thread migration is the movement of thread contexts across machines. Being transparent, the migration is done without the involvement of the application programmer. Thread migration enables mobility of executing threads, which is needed to achieve fault tolerance and load balancing.

In this paper, we propose a new transparent thread migration system that makes use of a Just-in-Time (JIT) compiler. Unlike previous solutions based on bytecode or JVM-level instrumentation, we use a JIT compiler to recompile the Java methods in the stack context to aid the extraction and restoration of the thread context at migration time. Such actions are called for only when a migration is requested by some policy module. In this way, we achieve both high-speed native thread execution without redundant code and thread mobility.

The rest of the paper is organized as follows. Section 2 discusses the related work. In Section 3, we give an overview of the design of our approach to thread migration based on JIT recompilation. Section 4 presents the detailed steps of the migration process, from extraction of context to resuming the execution of the thread in the other node. Section 5 gives the experimental results of our prototype system. Section 6 concludes the paper.

2 Related work

JavaGoX [6] and Brakes [7] use static bytecode instrumentation to realize transparent Java thread migration. The benefit of such an approach is that they can be more portable than JVM-level approaches. However, this approach usually introduces much more overheads than the JVM-level approaches because the use of higher level bytecode for the instrumentation.

Sumatra [5] extends the JVM interpreter to enable capturing and restoring of Java thread context during migration. The interpreter-based thread migration mechanism, however, has poorer execution performance when compared to an approach based on the JIT compiler, such as our system.

JESSICA [4] provides a single-system image view for multithreaded Java applications on clusters. It implements thread migration based on the interpreter of the Kaffe Open JVM (Version 0.9.1). The improved version, JESSICA2, is able to support JIT compilation using Version 1.0.6 of the Kaffe Open JVM. Using JESSICA2 as the base, our previous research [9] modified the JIT compiler to support lightweight thread migration. The approach was to apply native code instrumentation when a method is first translated into native code. This approach is superior to the bytecode instrumentation as it can make use of the JVM's kernel functions to handle the capturing and restoration of thread contexts. Nevertheless, the instrumented native code will run longer and take up more space than the original code, even when migration is never triggered.

3 The approach

A Just-in-Time (JIT) compiler is a dynamic compiler in the JVM that translates Java methods into native code when the methods are first called. As the JIT-enabled JVM runs the native code generated by the JIT compiler, the thread context is native, i.e., in a machine-dependent form. The Program Counter (PC) will be a real instruction pointer of the underlying hardware. The thread stack will be a native thread stack. Also, hardware machine registers will be used to hold the values of the variables. We call such a context a *Raw Thread Context* (RTC).

To implement thread migration in the JIT compiler-based JVM, a brute-force approach would move everything in the hardware machine context of the migrated thread from the source JVM to the destination JVM. Unless the two JVMs are running on homogeneous machines having an identical virtual address space and classes are loaded at same locations, the brute-force approach will not work. These constraints would make such an approach not practical [2].

We aim at an design with which a thread context can be portable. A portable context can be used to resume a thread's execution in any kind of node. As Java bytecode is portable, we therefore set out to construct a context that is in bytecode, which we call *bytecode-oriented thread context* (BTC). The BTC therefore has the same portability as any bytecode. The BTC consists of the identification of the Java thread, and a sequence of frames. Each frame contains the class name, the method signature, bytecode PC, operand stack pointer, operand stack variables, and the local variables encoded in a JVM-independent format.

Our design eliminates the machine-dependent hardware context during the migration operation. A context at the bytecode level is usually several times smaller than the equivalent at the hardware level, thus saving some communication cost during thread migration. Java objects and methods are represented by their symbolic names in the BTC. This makes easy the relocation of Java objects and methods when the execution of a thread is restored at the destination JVM.

At the point when a migration is to be carried out, our approach is to transform the RTC of the thread in question into an equivalent BTC. The BTC will then be transferred to the destination node where it will converted back to an RTC. In both steps, we need to make sure that the two types of thread context remain equivalent in semantics. For this to be possible, certain difficulties have to be overcome.

From RTC to BTC, which is when a thread is suspended for migration, the native PC in the RTC may not be pointing at the first instruction of the native code block that was compiled from a bytecode instruction. Also, the values of some local variables in the RTC may be kept in machine registers; as the BTC does not have the equivalent register set, the transformation needs to copy the latest values of the local variables in the registers to some equivalent variable definitions in the BTC.

Another difficulty is that to identify and then copy a value needs the knowledge of the variable type. The type information of the local variables can be known at compilation time. But the types of the stack variables can only be known at runtime. During the execution of a Java thread, the stack operands are dynamically pushed onto or popped from the thread stack. For the same stack slot, the type of its variable varies from time to time. For example, the bytecode instruction "*f2d*"(convert *float* to *double*) would pop off a *float* from the operand stack and push a *double* onto the stack.

From BTC to RTC, which is when the thread is to be resumed at the destination JVM, the transformation needs to restore the calling sequences of the frames in the BTC. The native PC in each frame needs to be set according to the bytecode PC in the BTC. Also, registers should be allocated and their values restored at the restoration point based on the architecture of the destination node.

To solve the above problems, we adopt a novel mechanism called *Just-in-Time recompilation*. When the JVM thread scheduler stops a Java thread, it analyzes the thread's native frames. The JIT compiler in the JVM is "re-run" to compile those methods that created these frames. During the recompilation, the information such as the corresponding bytecode PC, the variable types and the register allocation pattern, will be determined. Code stubs will be generated by the scheduler to be used to gather the BTC a moment later when the thread is resumed temporarily at this node. Later on when the Java thread context is restored in the destination JVM, another phase of compilation will be performed using the captured frames. A dynamic register patching technique will be used to re-establish the register allocation for each frame in the thread context. Dynamic register patching is to use the JIT compiler to generate the code stub that moves the values defined in the input BTC context into the machine registers based on the information available at the restoration point. In essence, the capturing and restoration of thread context use the JIT compiler during a thread migration operation to reconstruct the "symbol table".

4 Transparent Java thread migration using JIT recompilation

This section elaborates on the detailed steps involved in a thread migration act using our approach. Note that the normal execution of Java threads will not run any redundant code. It is only when there is a migration request would JIT recompilation be invoked by the thread scheduler. The recompilation will analyze the RTC in the thread stack and generate the migration supporting code. The thread will trap into this migration supporting code when it is executed again, which will extract the BTC and send it to the destination JVM.

In the JESSICA2 system, no matter where a thread has moved to, the Java objects needed by the thread are accessible through a distributed shared heap, a built-in object-based DSM-like service available in each JVM [8].

Figure 1 gives an overview of the steps of the JIT recompilation. A source JVM initiates a migration, and a destination JVM accepts the migrated thread. The shadeless boxes are operations in the source JVM; the shaded boxes are operations in the destination JVM.

4.1 RTC-to-BTC transformation

The JIT recompilation consists of seven steps: the stack walk, frame segmentation, bytecode PC positioning, breakpoint selection, type derivation, translation, and native code patching.

4.1.1 Stack walk

This is to traverse the native stack of the thread to be migrated and gather the information of each frame into a linked list of frames. The information includes the *frame pointer* (FP), the *stack pointer* (SP), the saved native PC, and the address of the PC. During the walk, all the native frames including those used by the JVM internal functions or the signal handlers will be gathered.

4.1.2 Frame segmentation

This step identifies the frames created by pure Java methods, which we call *Java-frames*. The other frames created by the JVM internal functions or Java native methods are *C-frames*. The topmost consecutive Java-frames will be selected for migration since they have the corresponding BTC.

The identification of Java-frames is done by matching the native PC against the code ranges stored in the Java method cache created by the JVM. When the native PC is found in a Java method's native code range, the Java method is identified. After the consecutive Java-frames are identified, a filter function will be applied to mark those to be migrated. The filter allows only a subset of the frames to be migrated. For example, only the frames created by the methods that contain computation loops will be chosen, so that only the heavy computations will be migrated to other nodes for execution.

4.1.3 Bytecode PC positioning

The third step is to position the bytecode PC in the selected Java-frames according to the saved native PC. Only when the bytecode PC is known can we get the other information such as the operand stack size and the variable types in the operand stack.

When the Java bytecode is compiled into native code, there is no simple one-to-one mapping of a bytecode instruction to a native code instruction. One bytecode instruction may correspond to a few native code instructions that form a native code block. When the JIT compiler performs code optimization, the mapping becomes many-to-many. This makes it difficult to directly extract the bytecode PC.

One approach to solve the problem is to save the mapping when the method is first compiled by the JIT compiler. This could consume much memory because all Java methods compiled would need the storage even if they will not be involved in a migration act. When a thread is stopped at a certain point, it is unlikely that it hits the entrance of the native code block to which the current PC belongs. Our approach is to re-run the code generation of the JIT compiler on the method to arrive at the same offset as the saved native PC.

4.1.4 Breakpoint selection

Because the native PC may not hit the boundary of a native code block compiled from a bytecode instruction or bytecode block, the RTC may not have corresponding BTC at the stopped native PC. In this case we need to delay the RTC-to-BTC transformation to the first native code instruction following the current native PC that has the matching of RTC to BTC. The transformation will be carried out by setting a breakpoint at the next eligible position. In normal cases, the native PC that maps to the immediately following bytecode PC will be the breakpoint target. But there may be more than one target position in some cases. For the compound branch bytecode instructions, *TABLESWITCH* and *LOOKUPSWITCH*, which behave like the *switch statement* in C, there may be many jump targets. All these target bytecode PCs will be collected. Afterwards, the type derivation and migration supporting native code will be generated at these breakpoints just identified.

4.1.5 Type derivation

We derive the type information at the breakpoints by simulating the simplified bytecode verification on the Java method. The bytecode verification is used in JVM to ensure the code integrity before executing the code [3]. In

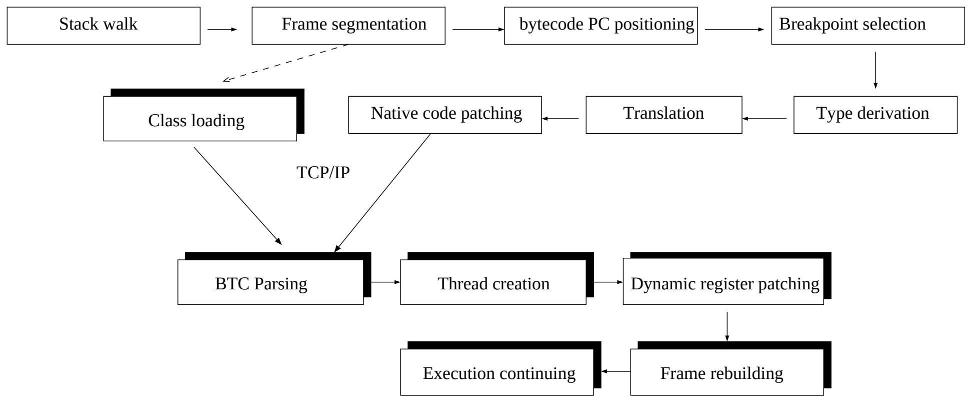

Figure 1. Transparent Java thread migration using JIT recompilation.

this step, no real verification on the type consistency will be carried out since the method has been verified to be correct before. Instead only the type information and stack operation will be updated along with the verification. According to the JVM specification [3], at any given point in the program, the operand stack is always of the same size and contains the same types of values no matter what code path is taken to reach it. Therefore we can follow any path in this process and the type result is still valid for the current execution path in the stopped method. Once a breakpoint target is reached, the type information of the local and stack variables will be saved.

4.1.6 Translation

In this step, we run the JIT compilation code generator to generate the new native code for the current method. For all the locations that are not marked as breakpoints, the same native code will be generated as the original compiled result. Special migration supporting code will be inserted at the breakpoints. This special code contains native instructions that save the bytecode PC, the Java stack pointer, the operand types and values into the thread's private area created for migration's sake. Except for the bottom frame, native instructions will be generated to simulate the epilogue of the current method. The instructions will unwind the thread stack and return the control to the caller. The bottom frame will return control to a migration handler which will then pack up the thread context and send to the destination JVM.

4.1.7 Native code patching

After the new code with breakpoints have been generated, the thread's native stack will be patched so that when the thread is scheduled to run again, its execution will be based on the new native code for each frame.

Recall that we have collected the addresses of the return address in each frame during the stack walk. The native code patching step then replaces all the native return addresses in the stack frames with the corresponding new native PC. Hence, when the thread is re-scheduled to run again, the execution will go through the new generated native code. Eventually when the thread reaches one of the breakpoints, the migration handler will execute. The BTC will be collected by the handler and sent to the destination JVM.

4.2 BTC-to-RTC transformation

The restoration of the thread is done through the BTC-to-RTC transformation. We have constructed a sequence of stack frames with the return 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.

The recompilation of the frames in the destination JVM will be carried out. The bytecode PC in each frame of the BTC will be translated into the corresponding native PC, which will be the restoration point of the frame. During the recompilation, a dynamic register patching function will generate a small code stub using the register-variable mapping information at the restored point of each frame. When the thread is scheduled to run later, for each restored frame, the execution will switch to the code stub to recover the registers for this frame. The last instruction in the code stub will be a branching instruction to jump to the restored point of the method for the frame.

4.3 Migration latency hiding

When there is a migration to be carried out, the issue of overhead would come into the picture, which is due to the various steps or our recompilation technique. At the destination JVM, the Java classes needed by the migrated thread have to be loaded from the local disk or the network. From our previous experience [9], the class loading step could easily be the dominant overhead in the entire process.

We observe that at the source JVM, once the frames in the thread context have been selected, we know which

Java classes should be loaded in the destination JVM. As shown in Figure 1, the dashed arrow indicates a signal to start loading the class files in the destination JVM. With that signal, the destination JVM can preload all the class files into its method area while the recompilation is still progressing at the source JVM. As a result, a good part of the latency of migration is hidden through this parallel operation.

5 Results

5.1 Experiment setting

We use our JESSICA2 DVJM system, into which we install our new thread migration mechanism. JESSICA2 is developed based on the Kaffe open JVM 1.0.6 [10]. JESSICA2 runs on the HKU Gideon 300 Linux cluster, which consists of up to 300 2GHz Pentium-4 nodes, each running the Linux kernel 2.4.22; the nodes are connected by a Fast Ethernet switch.

We compare our current approach with our previous approach based on dynamic native code instrumentation which has proved to be more light weight and efficient than other static bytecode instrumentation approaches [9]. We are also interested in how much latency can be hidden through preloading of class files at the destination JVM.

We run four multithreaded Java benchmark applications: the π calculation (CPI), Successive Over-Relaxation (SOR), All-pair Shortest Path (ASP), and N-Body simulation (NBody). The program CPI calculates an approximation of π by evaluating the integral. SOR does red-black successive over-relaxation on a 2-D matrix. ASP calculates the shortest path between any pair of nodes in a graph using a parallel version of Floyd's algorithm. 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 first observe that, when there is no migration, our approach incurs zero time and space overhead on top of normal execution. The dynamic native code instrumentation approach has the largest time overhead of about 16% when running the SOR program. The space overhead due to the extra instrumentation code of the same approach in the NBody program reaches to about 13%.

5.2 Migration latency

We evaluate the time overhead (migration latency) of the migration operation using two nodes, one serving as the source and the other one the destination JVM. This latency is measured from the time of stack capturing in the source JVM to the time when the thread has finished its stack restoration on the destination JVM.

As shown in Figure 2, the latency hiding technique (JITR + Preload) proves to be effective for all the four benchmark programs. Comparing with the dynamic native code instrumentation approach (DNCI), our new approach has slightly larger overheads. It will, however, be a small price to pay if migration is not excessively frequent, since there will be zero overhead for normal execution using our approach.

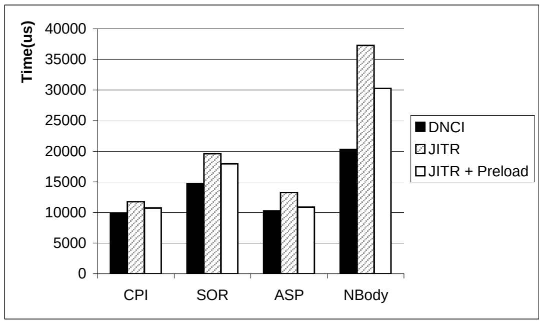

Figure 2. Java thread migration latency.

5.3 Initial placement vs. dynamic thread migration

The ultimate question is whether dynamic thread migration can in fact improve the overall execution time of an application. We run a Java application server simulation benchmark for this test, using different numbers of cluster nodes. We compare the throughput of initial placement versus that of dynamic thread migration. Initial placement is to place a thread in some chosen node when the thread is first created, and the thread will stay there until it terminates. Both initial placement and dynamic thread migration requires our thread migration mechanism.

The benchmark simulates a pure Java application server that accepts external requests to run different applications. Each request is handled by one Java thread. The Java thread then executes the requested application and returns the result to the server. The execution time of the application is randomly assigned.

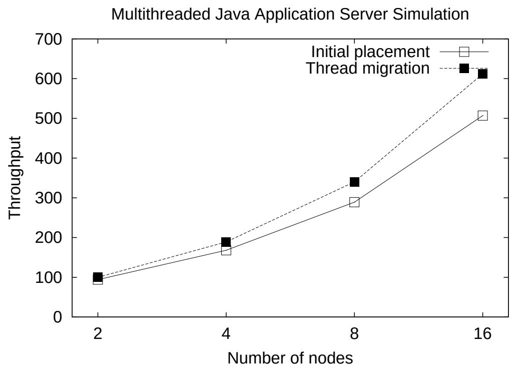

Figure 3. Initial placement versus thread migration in multithreaded Java application server test.

The throughput is defined as the number of requests performed per given time unit. As shown in Figure 3, comparison, spreading the threads over more nodes does result in improved execution performance. The speedups average to between 30–40%. Dynamic thread migration can help to further improve the throughput of initial placement by 10%–20% for all the configurations. The reason is that with initial thread placement only, the random workload of the requested application could still cause imbalances among the nodes; dynamic thread migration can correct those imbalances.

6 Conclusion

This paper presents an efficient transparent Java thread migration system using the technique of JIT recompilation. Dynamic thread migration can be used to improve load balancing; it can also be used to support fault tolerance if coupled with a mechanism to detect impending node failures. Our design introduces a portable interface, the bytecode-oriented thread context, for the movement of Java thread contexts, thus allowing threads to be migrated to heterogeneous nodes. Our solution preserves the high-performance JIT compilation and its benefits in the presence of thread migration.

The idea of using JIT recompilation demonstrates a new use of JIT compilers. It can be generalized to support many other useful applications such as debugging, profiling, etc. One key advantage of recompilation is that the system does not have to provide space to store the compilation information in advance. Further optimizations on the technique are possible such as to preload selected classes at much earlier times in order to completely eliminate the class loading overhead.

References

[1] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre 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.

[8] 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.

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

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