Delta Execution: A Preemptive Java Thread Migration Mechanism*

Matchy J. M. Ma   Cho-Li Wang   Francis C. M. Lau

Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam, Hong Kong

Abstract

Delta Execution is a preemptive and transparent thread migration mechanism for supporting load distribution and balancing in a cluster of workstations. The design of Delta Execution allows the execution system to migrate threads of a Java application to different nodes of a cluster so as to achieve parallel execution. The approach is to break down and group the execution context of a migrating thread into sets of consecutive machine-dependent and machine-independent execution sub-contexts. Each set of machine-independent sub-contexts, also known as a *delta set*, is then migrated to a remote node in a regulated manner for continuing the execution. Since Delta Execution is implemented at the virtual machine level, all the migration-related activities are conducted transparently with respect to the applications. No new migration-related instructions need to be added to the programs and existing applications can immediately benefit from the parallel execution capability of Delta Execution without any code modification. Futhermore, because the Delta Execution approach identifies and migrates only the machine-independent part of a thread's execution context, the implementation is therefore reasonably manageable and the resulting software is portable.

Keywords: Thread migration, load balancing, cluster computing, single-system-image, Java.

1 Introduction

Traditional UNIX-like systems such as Sprite [7] and MOSIX [2] support dynamic load balancing in a cluster by implementing process migration at the kernel level. Because the kernel has the best knowledge of all the activities and resources that are distributed within the cluster, it can perform resource management effectively. However, as UNIX was not designed with the the notion of process migration from ground up, extending it to support migration usually

*This research was supported by the Hong Kong RGC Grant HKU 7032/98E.

requires a substantial amount of software effort and the portability of the system could be compromised as a result [1].

Delta Execution aims at providing a high-level and portable implementation of migratable Java threads that is not entangled with any low-level and system-dependent issues. The implementation is carried out at the middle-ware level by extending the Java Virtual Machine. Because of this high-level approach, the Delta Execution mechanism is also applicable to other language environments that are based on virtual machines for supporting preemptive thread migration.

The Delta Execution mechanism identifies and separates the machine-independent sub-contexts from the machine-dependent sub-contexts in the execution context of a running thread. Machine-independent sub-context is the state information that can be expressed in terms of execution state of the virtual machine, such as data stored in the virtual machine's registers. Mahine-dependent sub-context is the state information that are part of the internal state of the executing program that implements the virtual machine, such as the hardware program counter that points to the current machine instruction when the virtual machine is executing a native method. When a thread migrates to a *remote node*, only the machine-independent sub-contexts are migrated in a regulated manner. The manipulation of any machine-dependent sub-contexts is avoided by keeping them intact at the *home node* where execution involving these sub-contexts will continue to be performed. Active execution of the migrated thread is observed as a sequence of exections on the machine-dependent and machine-independent sub-contexts that switches back and forth between the home and the remote node. Because the migrated thread only *incrementally advances* its execution by a *delta* amount every time the execution control switches between the two nodes, thus we give this mechanism the name *Delta Execution*.

The Delta Execution mechanism is implemented as part of the JESSICA (Java-Enabled Single-System-Image Computing Architecture) middle-ware [8, 9] for cluster computing. JESSICA is a distributed Java Virtual Machine that abstracts away the distributed nature of the cluster hardware and provides multi-threaded Java applications with an illusion of a single multi-processor machine.

In summary, the main features of Delta Execution include:

- Transparent migration—The Delta Execution mechanism is implemented entirely at the virtual machine level which is transparent to a migrating thread; together with a redirection mechanism for forwarding location-dependent operations and a distributed shared-memory subsystem that supports location-independent memory access, migration transparency can be guaranteed.

- Bytecode-level migration granularity—A thread can be stopped at any time and be migrated once the execution of the current bytecode instruction is finished. The system can swiftly respond to any change in system load across the cluster for load balancing.

- Portable and manageable implementation—Unlike traditional systems that implement migration at the kernel level, Delta Execution is implemented on top of the kernel without having to deal with any low-level details of the operating system or the hardware. This approach provides better portability and the software effort required is more manageable.

- Compatibility with existing Java applications—Because the migration mechanism does not require adding any new migration-related instructions to a Java program, the implementation offers maximum compatibility to the vast number of existing Java applications.

- A language-neutral solution—The design of Delta Execution and its accompanying redirection mechanism for handling migration transparency is not only applicable to the Java programming language alone. It is a generic solution that can also be applied to other programming languages which execute by means of virtual machines for supporting transparent thread migration.

The rest of the paper is organized as follows. Section 2 presents the system architecture of JESSICA. Section 3 discusses the Delta Execution mechanism in detail. Section 4 evaluates the performance of an experimental prototype. Section 5 discusses related work and finally Section 6 concludes the paper.

[please insert figure: journal-sysArch.eps here by uncommenting the following line]

Figure 1: Global Thread Space creates an SSI illusion over a cluster of computers

2 JESSICA System Architecture

JESSICA comprises a group of daemons running in all the nodes in a cluster of computers. They execute as user-level processes on top of the UNIX operating system, and it is the collaboration and coordination among these JESSICA daemons that an single-system-illusion illusion is offered to Java applications, as depicted in Fig. 1.

The single-system-image illusion is established through the provision of a *global thread space*. The global thread space is a logical thread space that spans all the nodes in the cluster, where the machine boundaries are hidden and threads can freely migrate from one node to another by following the Delta Execution mechanism. The distributed-shared memory (DSM) subsystem enables memory objects to be globally-accessible by threads independent of their current physical locations. Together with a master-slave redirection mechanism that forwards I/O and other location-dependent operations back to the home node for execution, thread migration within the global thread space is entirely transparent to applications. As a result, this free movement of threads provides an opportunity for optimizing the utilization of shared resources within the cluster.

In JESSICA, we classify cluster nodes as either *console* or *worker* as follows:

- Console Node—Java applications can be started in any node in the JESSICA cluster. The node in which a Java application is initiated will become the home of that application and is known as the console node.

- Worker Node—When an application is instantiated in the console node, other nodes in the JESSICA cluster will play the role of supporting the console to execute the application. The worker nodes are there in stand-by mode and would serve any requests forwarded from the console.

2.1 JESSICA Daemon

Each JESSICA daemon is composed of the following four components that provide bytecode execution, memory management, thread creation, scheduling and synchronization to Java applications the same way as a standard Java Virtual Machine (JVM).

- Bytecode Execution Engine (BEE)—It is responsible for binding an active thread and executing its method code. Parallel execution of a multi-threaded application is realized by having multiple BEEs running on multiple machines to execute multiple threads simultaneously.
- Distributed Object Manager (DOM)—It is responsible for managing the memory resources in its local node and to cooperate with DOMs running on other nodes to create a glob al object space. The physical locations of objects are transparent to the threads within the global object space.
- Thread Manager (TM)—It is responsible for creating, scheduling and destroying threads running on the local node. During the course of migration, it coordinates with TMs running on other nodes to marshal, ship, and demarshal the execution contexts of migrating threads.
- Migration Manager (MM)—It is responsible for collecting load information of the local node and exchanging that information with MMs running on other nodes in order to enforce a migration policy.

2.2 A Master-Slave Model for Migrating Threads

When a thread running in the console node migrates, it does not actually pack its entire execution context and move to the destination worker node. Instead, it is split into two cooperating entities, one running in the console node called the *master*, and the other running in the worker node called the *slave*. The slave thread is in fact created anew at the worker node and acts as the migrated thread image to continue the execution of the original thread. The execution context is divided into sets of machine-dependent and machine-independent sub-contexts as identified by the Delta Execution mechanism. All the machine-dependent sub-contexts are

[please insert figure: cc-tea-fig3.eps here by uncommenting the following line]

Figure 2: Interactions between the master and the slave threads that hide migration from the rest of the system

processed locally by the master thread while the remaining machine-indepdent sub-contexts are migrated to the slave thread for execution. As as result, this master-slave design provides an opportunity for the implementation to isolate and process machine-dependent contexts from machine-independent contexts in a manageable way.

The master thread remaining at the console represents the original thread and is now responsible for performing any location-dependent operations such as I/O for the slave. With this design we are able to create the global thread space that maintains the same semantics and relationships between all the objects in the execution environment as the case when there is no migration, as shown in Fig. 2. All the interactions between the slave and other threads have to go through the master. The redirections make the master appear to other threads as the only thread they are interacting with. On the slave side, all the location-dependent operations are redirected transparently back to the console to be performed by the master. As a result, the execution environment observed by a running thread in JESSICA is the same as that in a standard JVM, regardless of whether the thread has been migrated or not.

3 Delta Execution

The primary task for performing thread migration is to capture the execution context of a migrating thread completely at the console node so that the context can be correctly reproduced at the destination worker node. Execution context represents the current state of a running thread. The execution state reconstructed at the worker node will be used to resume execution of the migrated thread at exactly the same place where the thread was frozen at the console.

3.1 Method Invocation Sequence and Java Method Frame

Observe that the life of an active Java thread begins with the calling of `java.lang.Thread .start()` method, or the equivalent if this thread is a sub-class of `java.lang.Thread`. The bytecode instructions of this method will be executed by the bytecode execution engine, which may further call other methods, and so on. Finally, the thread terminates when this `start()` (or the equivalent) method returns. In other words, the execution context of a running thread can be represented by a sequence of method calls and their respective execution context local to each method.

Here a structure called *Java method frame* (JMF) is defined to help represent the method invocation sequence of an active thread. The purpose of JMF is to capture the execution context of a Java method under execution. A JMF contains:

- a program counter (PC) pointing to the bytecode that is currently under execution;
- NPC, the location of next bytecode instruction to be executed;
- a stack pointer (SP) pointing to the top of the Java method stack;
- the set of local variables of this method;
- the Java method stack;
- other miscellaneous information such as that for Java exception handling.

Notice that information containing in a JMF is platform-independent.

With the JMF defined, we will be able to describe the execution context of a running thread. When the bytecode exeuction engine first activates a thread by invoking its `java.lang.Thread .start()` method (or the equivalent), it pushes the corresponding JMF, labeled as f_0, onto the thread runtime stack before executing code that implements the method. The inserted f_0 is responsible to store state information that comprise the execution context of the `start()` method during execution. Consider at some point the `start()` method tries to invoke another method, m_1, by using one of the `invokeMethod`[1] instructions, the thread will instantiate a new JMF, f_1, for storing the execution context of the invoking method m_1 and pushes it onto the thread runtime stack. When m_1 returns, its method execution context will also be discarded. This is done by popping f_1 from the runtime stack. The popping action will also cause the execution context of the `start()` method, i.e. f_0, to be restored automatically. Since at this moment the NPC of f_0 is pointing to the instruction immediately following the previous m_1

[1]The bytecode execution engine can invoke a Java method by executing one of the `invokevirtual`, `invokespecial`, `invokeinterface`, or `invokestatic` bytecode instructions.

[please insert figure pdpta-fig1.eps here by uncommenting the following line]

Figure 3: `Factorial.java`

[please insert figure pdpta-fig2.eps here by uncommenting the following line]

Figure 4: Disassembled bytecode of the class `Factorial`

call, the thread can then continue its execution from the point of return of the m_1 call. Notice that because of this arrangemnet, the JMF sitting on the top of the runtime stack will always correspond to the execution context of the current method.

3.2 A Simple Thread Migration Example

The `Factorial.java` example shown in Fig. 3 illustrates how thread migration is carried out and how execution states are captured and transferred based on the JMF structure. The compiled bytecode instructions are shown in Fig. 4. The numbers following the line numbers in Fig. 4 represent the corresponding locations of the bytecode instructions to be stored in the program counter (PC).

Consider the console node triggering a migration when the `Factorial` program has already recursed three times and the running thread is frozen at PC = 8 of method `f()` (Fig. 4, line 18). After the current instruction at PC = 8, i.e. `iload_0`, has been completed, the execution context of the *main* thread can be represented pictorially as in Fig. 5.

Suppose at this moment this *main* thread migrates, the above five JMFs which constitute the entire execution context of *main*, are packed and shipped to a worker node. After the execution context arrives at the destination worker node, they are unpacked and reconstructed, and eventually bound to the execution context of a newly created slave thread. When the slave thread resumes its execution, it continues from the tail-JMF, and updates the value of PC to 9, i.e. `iconst_1` (Fig. 4, line 19), according to the content of the NPC. The result is an integer constant 1 being pushed onto the tail-JMF's method stack. From this point onwards, the slave thread continues its execution correctly from the point where the master was previously stopped. When the recursive method `f()` at this level (`n` = 7) returns, the JMF is popped and

[please insert figure pdpta-fig3.eps here by uncommenting the following line]

Figure 5: Execution context of thread *main* after a number of iterations

[please insert figure pdpta-fig4.eps here by uncommenting the following line]

Figure 6: `Bar.java` and `Foo.java`

[please insert figure pdpta-fig5.eps here by uncommenting the following line]

Figure 7: Disassembled bytecode of the class `Bar`

the context of the previous JMF ($\mathtt{n} = 8$) is restored. The method stack now should contain two integer elements: 8, and 5040, i.e. the value of `n` and the evaluated value of `f(7)` respectively. From this point onwards, the execution context of the slave thread further unwinds itself and the JMFs of the method `f()` are popped from the thread runtime stack one after another as the recursive calls return. Eventually when all the JMFs are exhausted the slave thread will have completed its execution, and the result is the return value of `f(10)` stored at the top of the method stack.

3.3 How Machine-dependent Sub-context Comes About

Although the JMF structure appeared to be comprehensive enough to capture the execution context of a running thread, it turns out that such solution is only good for some cases. If a thread invokes methods directly through the bytecode execution engine using one of the `invokeMethod` instructions, then the scheme of using a sequence of JMFs as described previously can sufficiently capture and represent the execution context of this running thread. On the other hand, if this thread invokes a native method[2], or if the bytecode execution engine executes an instruction such that the instruction will invoke another method as a side-effect, this will generate machine-dependent sub-context that a sequence of JMFs alone cannot capture. An example of machine-dependent state information so generated is the hardware return address of the native method. This can be illustrated by the following `Bar` and `Foo` class as shown in Fig. 6. Their corresponding disassembled bytecode are shown in Fig. 7 and Fig. 8 respectively.

[2] A native method is a method implemented by the native machine code of the underlying hardware.

[please insert figure pdpta-fig6.eps here by uncommenting the following line]

Figure 8: Disassembled bytecode of the class `Foo`

[please insert figure pdpta-fig7.eps here by uncommenting the following line]

Figure 9: Simplified implementation for the `new` bytecode instruction in C/C++

[please insert figure pdpta-fig8.eps here by uncommenting the following line]

Figure 10: Execution context of thread *main* after entering the class initialization method ''`<clinit>`''

An Illustrating Example

This example demonstrates how machine-dependent state information is introduced into a thread's execution context when the bytecode execution engine executes a bytecode instruction that in turn invokes another method. The `Bar` class contains a class initialization method which will be executed once when it is loaded by the bytecode execution engine the first time. When class `Foo` tries to create a new instance of object `Bar` using the `new` bytecode instruction, the bytecode execution engine will load the `Bar` class and invoke its initialization method just before allocating a new `Bar` object. Machine-dependent state information is introduced into the execution context at this point. If the thread migrates before the initialization method returns, such machine-dependent sub-context cannot be captured by a JMF sequence. In this example, the `static` block in the class `Bar` defines the class initialization method (Fig. 6, line 03) and is internally denoted as method ''`<clinit>`'' by the virtual machine implementation (Fig. 7, line 12). According to this example, the initialization method will initialize the class variable `count` to 100 when class `Bar` is first loaded (Fig. 6, line 03).

When the execution of class `Foo` begins, the *main* thread of the bytecode execution engine enters the `main()` method of class `Foo` (Fig. 6, line 08) and executes its first bytecode instruction, i.e. `new`, at PC = 0 (Fig. 8, line 06). Within the bytecode execution engine, the implementation looks up and loads class `Bar` into the system (Fig. 8, line 06) and invokes the class initialization method. The bytecode execution engine then allocates a free memory block and returns it as a new `Bar` object. The C/C++ code segment in Fig. 9 shows a possible way to implement the `new` bytecode instruction for the bytecode execution engine, which summarizes the above actions.

When the *main* thread invokes the class initialization method ''`<clinit>`'' by calling the `vmExecuteJavaMethod()`[3] function from the `instruction_new()` implementation at line 10 of

[3] `vmExecuteJavaMethod()` is an internal function of the JESSICA implementation which allows invocation of

Fig. 9, it pushes a new JMF onto the thread runtime stack so as to represent the execution context of `‘‘clinit’’`. If at this moment the console decides to migrate the thread when the bytecode execution engine is executing the first instruction of `‘‘clinit’’`, i.e. `bipush` (Fig. 7, line 13), the *main* thread will be frozen once the `bipush` instruction has been completed. At this stage the execution context of the thread can be represented by the JMF sequence as illustrated in Fig. 10.

After the JMF sequence of the migrating *main* thread is transmitted to a destination worker node, its slave thread will resume its execution according to the reconstructed method exection context of `‘‘<clinit>’’`, where the value of PC at this point equals to 2. This will cause the integer value 100, as obtained from the top of the method stack, to be stored to the class variable `count` before returning from the initialization method (Fig. 7, line 14). When the `‘‘clinit’’` method returns the corresponding JMF will be popped from the thread runtime stack and the execution context of `Foo.main()` will also be restored. Without special arrangement the bytecode execution engine will continue the execution from the execution context of `Foo.main()` and execute the next instruction at PC = 3, i.e. `inovkestatic` (Fig. 8, line 07). But this will lead to incorrect result as the slave thread will have “forgotten” to allocate memory for the new `Bar` object, since the instructions from line 13 to line 19 of Fig. 8 are skipped.

Note that if there is no migration, the control flow of the *main* thread after returning from `‘‘<clinit>’’` should continue from line 13 of the `instruction_new()` code, as shown in Fig. 9. In this case, the thread will obtain a memory block from the memory manager and push the memory handle onto the method stack (Fig. 9, line 13 to line 19) before returning to the method execution context of `Foo.main()` at PC = 3. In other words, there are certain machine-dependent state information that failed to be captured between the two JMFs when the execution context is migrated, as shown in Fig. 11.

The missing information, including the machine-dependent return address at line 13 of the `instruction_new()` code in Fig. 9, are in fact stored in the *UNIX process stack* of the daemon process running at the console. They are the machine-dependent sub-context of a

Java methods from within the implementation itself, the same function may takes other names and forms in other Java Virtual Machine implementations.

[please insert figure pdpta-fig9.eps here by uncommenting the following line]

Figure 11: Revised execution context of thread *main* after entering the class initialization method ``<clinit>''

[please insert figure pdpta-fig10.eps here by uncommenting the following line]

Figure 12: Revised thread execution context representation, shaded blocks are the machine-dependent sub-contexts not captured by JMFs

thread execution context which by all means are implementation- and hardware-dependent.

3.4 Partitioning Execution Context into Delta Sets and Machine-dependent Blocks

Concluding from the two examples discussed above, we can categorize JMFs into two types:

- *B Frame*—is the JMF that is pushed onto the thread runtime stack as a result when a method is invoked using either `invokevirtual`, `invokespecial`, `invokeinterface`, or `invokestatic` *bytecode instruction.* For example, the JMF being pushed as method `f()` recurrsively invokes itself at line 21 of Fig. 4 is a B frame.

- *C Frame*—is the JMF that is pushed onto the thread runtime stack when the implementation code of a bytecode instruction, or a native method, invokes a method using the `vmExecuteJavaMethod()` *C/C++ function.* For example, the JMF being pushed at line 10 of the `instruction_new()` code in Fig. 9 is a C frame.

As illustrated by the shaded block in Fig. 11, it is the machine-dependent sub-context introduced by the implementation code before pushing a C frame onto the thread runtime stack that cannot be captured directly. We will label such machine-dependent sub-context as a *D block.* Consequently, for any thread execution context, the correct representation should be expressed as a sequence of interleaving B frames and C frames, with a D block inserted before each C frame. Note that every sequence must begin with a C frame, since any thread execution must be initiated by the virtual machine implementation itself (Fig. 12).

Our approach is to partition a thread execution context into chunks of JMFs that are separated by D blocks, each chunk is known as a *delta set*, ΔE. A *delta set* begins with a C frame and is followed by zero or more B frames. As shown in Fig. 12, ΔE_1 is the first *delta set* inserted into the execution context when a thread first enters its `start()` method. ΔE_2 and ΔE_3 are the subsequent *delta sets* appended to the tail of the execution context as the thread invokes methods from native code by means of the `vmExecuteJavaMethod()` function. Now, instead of shipping the whole execution context to the destination all at once as done in the traditional way of thread migration, we ship them in an incremental manner, one *delta set* at a time.

When the console node performs a migration operation, the thread manager marshals and ships the tail *delta set* in the execution context of the migrating thread. The slave thread at the destination worker node, once it is instantiated, will bind to this *delta set* and continue execution until the JMFs in this set is exhausted. The execution control will then switch back to the master thread, to let it finish the machine-dependent D block there. Immediately after the machine-dependent execution is completed, control flow will switch back to the slave with the next *delta set* migrated to continue execution at the worker node.

With this arrangement of *Delta Execution*, we are able to isolate those machine-dependent parts of a thread's execution context that are not migratable and to have them executed locally at the console, while those machine-independent parts are migrated across machines in the form of delta sets and are executed remotely at the worker nodes. The cooperating console and worker nodes need only incrementally advance the execution of a migrating thread by a *delta* amount each time.

4 Performance Evaluation

To study the effectiveness of the Delta Execution mechanism, we have implemented the mechanism in a JESSICA prototype that runs on a cluster of 12 SUN Ultra-1 workstations interconnected by a Fore ASX1000 ATM switch. The implementation is based on the KAFFE virtual machine [11] and the Treadmarks package [5] for Distributed Shared-Memory (DSM) support.

4.1 Applications Performance

We have implemented the following three multi-threaded Java applications to measure the impact on performance due to thread migration.

[please insert figure cpi.eps here by uncommenting the following line]

Figure 13: Performance results of the approximation of the value π with 100M intervals

- π Calculation—This application approximates π by evaluating an integral. The area under the corresponding graph is divided into multiple regions and multiple threads are deployed to find the sub-areas. The value π is obtained by summing up all the sub-areas once all the threads have finished. This application shows the raw parallel performance of Delta Execution since there is no interaction between the worker threads until all the computations are done; the extra overhead due to migration is minimal.

- Recursive Ray-Tracing—We have implemented a multi-threaded recursive ray-tracer in Java where the worker threads render the pixels of a projected 2D image by shooting rays into a given 3D scene. The threads obtain the next line of pixels to compute from a globally shared job queue, which provides a load-balancing effect at the application level. All worker threads are tightly synchronized between themselves when they access the job queue in order to maintain its consistency. This application demonstrates how distributed thread synchronization affects the performance for the worker threads that are distributed across the cluster.

- Red-Black Successive Over-Relaxation (R/B-SOR) on a Grid—The R/B-SOR program creates multiple threads to compute matrix elements in parallel. A huge $1024x1024$ matrix is divided into two sub-matrices, the Red and the Black matrix, which in turn are divided into roughly equal-size bands of rows, with each band assigned to a different thread. The threads repeatedly retrieve values from one matrix, compute the average, and write the result back to the other matrix. Since the huge matrix is allocated from the globally shared DSM space, the execution imposes a significant amount of loading on the DSM subsystem. Hence, this is a good candidate for studying how the DSM overhead contributes to the overall execution time as a result of migration.

Each of the applications was tested on the 12-node cluster using 1, 2, 4, 8 and 12 processors with 1, 2, 4, 8 and 12 worker threads respectively running. The results are presented in Fig. 13, Fig. 14, and Fig. 15.

[please insert figure raytracer.eps here by uncommenting the following line]

Figure 14: Performance results of the recursive ray-tracer to produce a 480x640 image

[please insert figure rbsor.eps here by uncommenting the following line]

Figure 15: Performance results of the red/black successive-over-relaxation application on a 1024x1024 matrix

According to Fig. 13, it can be seen that almost ideal speedup and efficiency are achieved in the π approximation application, since there is no communication or coordination between worker threads until all the computations are completed. The recursive ray-tracing experiment shows that the efficiency is less than optimal and drops moderately as more processors are used. The efficiency decreases from 69% when using two processors to 47% when using twelve processors. This is because, as indicated in Fig. 14, the distributed synchronization overhead contributes a significant amount to the total execution time. As shown in Fig. 15, the R/B-SOR application can gain moderate speedup when running with four or more processors. When running with two processors, the speed gained by overlapping the computation is offset by the extra overhead incurred due to remote memory access. The efficiency stays at around 53% and improves slightly when the number of processors is progressively increased from two to twelve. This could be due to the fact that the amount of data shared, i.e., the sizes of the Red and the Black matrices, remain the same when executed by any number of processors, and therefore the DSM overheads constitute roughly the same percentage of the execution time.

4.2 Analysis of Migration Latency

Fig. 16 illustrates in detail the interactions between the console and the worker node when a thread is migrated from the former to the latter. The migration latency is the time between the moment the migrating thread is frozen by the console and the moment it is restarted later as a slave thread at the worker node. Let $T0$ be the time taken to notify the destination node and to have the destination node prepare itself for the migration. The value of $T0$ is relatively constant. Let $T1$ be the time taken to marshal a delta set at the console node, to send the marshaled data across the network, and eventually to de-marshal received data at the destination node. The value of $T1$ is therefore proportional to the size of the transferring delta set. The migration latencies, i.e. $T0 + T1$, for different sizes of the delta sets are measured.

[please insert figure pdpta-fig12.eps here by uncommenting the following line]

Figure 16: Interactions between the console and the worker during migration

According to the data collected, when the size of delta set is zero, the migration latency is about 24.55 milliseconds ($T0$). $T0$ is the time taken to execute the `java.lang.Object.clone()` method in the slave as well as that for sending the four handshake messages between the master and the slave, as shown in Fig. 16. The purpose of the `clone()` method is to create an image of the migrating thread at the destination node which will then become the *slave* thread. A further breakdown of this $T0$ value reveals that the time required to invoke the `clone()` method is about 17 milliseconds and that for a handshake message to be sent between the master and the slave is about 2 milliseconds. Now consider the case when a thread is migrated just before it starts executing the first instruction; the delta set contains only one JMF and the corresponding method stack is empty. If there is no local variable defined in the method, the size of this minimal JMF is 208 bytes. Consequently, the minimum migration latency of the Delta Execution mechanism is measured to be about 28.12 milliseconds.

It can be seen that the minimum migration latency is dominated by the time for invoking the `clone()` method, which accounts for over 60% of the migration time. It is relatively expensive for the virtual machine in JESSICA to invoke a method like `clone()`, when compared to traditional migrating mechanisms which execute in native machine code. It is the intrinsic characteristic of JESSICA that a thread object cannot be duplicated by simply copying its memory content but a particular `clone()` method has to be invoked to clone the object. Although this cloning method is expensive, it is only to be invoked once for migrating a thread, and there is no addition overhead in the later part of the Delta Execution process. Besides, despite this overhead, we are still able to achieve good speedup by migrating threads to different nodes in the cluster for parallel execution, as demonstrated in the experimental applications discussed.

5 Related Work

MOSIX [2] supports transparent process migration and follows the monolithic kernel approach. NOW MOSIX [1] uses the home model whereby user processes are created to run seemingly at the user's home workstation. Location transparency is maintained for migrated processes by redirecting any system calls that are location-dependent. However, considerable modifications have been made to the kernel in order to support the transparency.

On the other hand, since Delta Execution is implemented at the middle-ware level, it enables the migration mechanism to focus only on the machine-independent part of a thread execution context, and to leave the remaining machine-dependent part untouched. As a result, the implementation of Delta Execution did not have to delve into any low-level layers of the underlying operating system or the hardware. Consequently, the implementation requires much less effort than that of MOSIX and the resulting software is more manageable.

Bouchenak's MobileThread package [10] supports Java thread mobility by extending the Java Virtual Machine. MobileThread inherits and extends the `java.lang.Thread` class in order to provide an interface for extracting and restoring the execution context of an active Java thread. This allows thread migration and checkpointing for the purpose of load balancing and fault-tolerance.

Bouchenak's work is different from the Delta Execution in that the Bouchenak's migration is not transparent. The MobileThread package requires the application programmer to decide what to do with the extracted state, whether to save the state to some non-volatile storage or to send the state to another machine to resume execution. Existing applications cannot benefit from the MobileThread package until they are modified and adapted to the MobileThread API. The Delta Execution mechanism introduces no new primitives nor requires any modification in existing applications in order to support thread migration.

Java/DSM [12] is a distributed Java Virtual Machine that runs on a cluster of heterogeneous computers. It provides an illusion to Java applications as if they are running on a single JVM with multiple processors attached. Parallel execution of a multi-threaded application is possible by having multiple threads running on multiple nodes in the cluster.

Both Java/DSM and JESSICA follow a similar approach by implementing a distributed virtual machine at the middle-ware level. They utilize DSM systems to simplify their implementations. However, in Java/DSM, load distribution is achieved by remote invocations of Java threads alone, while JESSICA supports also transparent thread migration. Besides, The

[Please insert Table 1 here by uncommenting the following line]

Table 1: Comparison of characteristics between JESSICA and the related work discussed

current Java/DSM prototype focuses mainly on supporting DSM in a heterogeneous environment, other issues such as location transparency are not addressed.

Arachne [4] is a portable user-level programming library that supports thread migration over a heterogeneous cluster. However, migration is not transparent to the application programmers—they have to include the Arachne migration-related primitives into the code. It is the programmer's responsibility to decide when to migrate, which thread to migrate, and where to migrate. In addition, it introduces new keywords to the C++ programming language in order to facilitate the implementation of thread migration. Even though the primary objective of Arachne is to support efficient thread migration on a heterogeneous network, it lacks certain features such as thread synchronization that is fundamental to a thread package. Besides, migrated threads cannot share data.

In contrast, the Delta Execution mechanism enables JESSICA to provide a flexible, portable, efficient and useable thread package to the application programmers. Migration is entirely transparent to programmers in JESSICA; there is no migration primitive that programmers need to insert into their code. Migrated threads can share object of any data types. Furthermore, the Delta Execution mechanism is portable as it does not introduce any new keyword to the Java programming language.

Table 1 provides a comparison between JESSICA and the related work discussed in this section.

6 Conclusion

Delta Execution takes a novel approach in dealing with thread migration. Instead of moving the whole execution context to the destination, the context is separated into machine dependent and machine-independent parts, and only the machine-independent parts are transported in a controlled manner. This design imposes no limitation on the type of threads that can migrate, such as whether they are using location-dependent resources or not. The master-slave model makes the whole system easy to control and monitor. In addition, the design also prepares the ground for further development of thread migration in a heterogeneous environment as all the state information migrated are hardware independent. Hence, the Delta Execution approach provides maximum flexibility and portability for thread migration. This advantage is a direct consequence of using the Java programming language, whereby execution via a virtual machine provides an extra layer of abstraction. The virtual machine approach allows the implementation of Delta Execution to be carried out entirely at the user level, without having to deal with any operating system specific or hardware specific issues. Moreover, Delta Execution needs no insertion of new migration instructions into a Java program, thus enabling JESSICA to provide maximum compatibility to the vast number of existing Java applications.

The Delta Execution mechanism and JESSICA together create a parallel execution platform with good speedup for all the experimental applications discussed. Efficiency we measured ranges from about 50% to 95% when executing on two to twelve nodes. The major overhead comes from remote object accesses made by migrated threads as well as distributed thread synchronization. Although the current prototype is not all that optimized because we traded simplicity for efficiency during the implementation process, we are optimistic about the future development of JESSICA. We are in the process of porting the JESSICA prototype to a Linux-based cluster which is equipped with the Direct-Point fast communication subsystem [6] and the JUMP DSM subsystem [3]. Our future work will study the effect of different DSM consistency models on the remote object access overhead; we will also try to fine-tune the implementation and to reduce the distributed synchronization overhead.

References

[1] A. Barak, O. Laden and Y. Yarom, "The NOW MOSIX and its Preemptive Process Migration Scheme", *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments*, Vol. 7, No. 2, pp. 5-11, 1995.

[2] A. Barak and O. Laden, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing", *Journal of Future Generation Computer Systems*, Vol. 13, No. 4-5, pp. 361-372, Mar 1998.

[3] B. Cheung, C.L. Wang and K. Hwang, "A Migration-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations", *Proc. of 1999 International*

Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Jun 1999.

[4] B. Dimitrov and V. Rego, "Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 5, pp. 459-469, May 1998.

[5] 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*, Vol. 29, No. 2, pp. 18-28, Feb 1996.

[6] C.M. Lee, A. Tam and C.L. Wang, "Direct-Point: An Efficient Communication Subsystem for Clusterr Computing" Proc. of 1998 International Conference on Parallel and Distributed Computing Systems (IASTED), Oct 1998.

[7] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *Software Practice and Experience*, Vol. 21(8), Aug 1991.

[8] M.J.M. Ma, C.L. Wang, F.C.M. Lau, and Z. Xu, "JESSICA: Java-Enabled Single-System-Image Computing Architecture", *Proc. of 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Jun 1999.

[9] M.J.M. Ma, "JESSICA: Java-Enabled Single-System-Image Computing Architecture", M.Phil. Thesis, Department of Computer Science and Information System, The University of Hong Kong, Feb 1999.

[10] S. Bouchenak, "Pickling Threads State in the Java System", *Proc. of Third European Research Seminar on Advances in Distributed Systems (ERSADS)*, Apr 1999.

[11] Transvirtual Technologies Inc., Kaffe Open VM, *http://www.transvirtual.com.*

[12] W. Yu and A.L. Cox, "Java/DSM: A Platform for Heterogeneous Computing", *Proc. of ACM 1997 Workshop on Java for Science and Engineering Computation*, Jun 1997.

About the Authors

Matchy J.M. Ma received his B.Sc. and M.Phil. degrees in Computer Science from the University of Hong Kong in 1994 and 1999 respectively. He is currently working as a software engineer at the Advanced Academic Java Campus, the Unversity of Hong Kong. His research interests include Internet and Web computing, object-oriented distributed systems, and cluster computing.

Cho-Li Wang received his B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1985. He obtained his M.S. and Ph.D. degrees in Computer Engineering from University of Southern California in 1990 and 1995 respectively. Currently, he is an Assistant Professor of Department of Computer Science and Information Systems at the University of Hong Kong. His research interests include high-speed networking, cluster computing, and software environment for Web-based applications. Wang can be reached by Email: clwang@csis.hku.hk.

Francis C.M. Lau is an associate professor in computer science and information systems at the University of Hong Kong. His major research interests include parallel and distributed systems, operating systems, and Internet and Web computing. He is an active volunteer of the IEEE Computer Society, serving as a vice president in 1999.