METHODS AND APPARATUS TO PROVIDE A MODULAR NATIVE METHOD INVOCATION SYSTEM

Methods and apparatus to provide a modular native method invocation (NMI) system are described herein. In an example method, NMI information associated with an NMI call from one or more virtual machines is received. The NMI information is translated via an NMI adapter. To generate a native stub, the translated NMI information is provided to a modular NMI component.
METHODS AND APPARATUS TO PROVIDE A MODULAR NATIVE METHOD
INVOCATION SYSTEM

TECHNICAL FIELD

[0001] The present disclosure relates generally to managed runtime environments, and
more particularly, to methods and apparatus to provide a modular native method
invocation system.

BACKGROUND

[0002] As applications migrate toward managed runtime environments (MRTEs) such
as Java® Virtual Machine (JVM) and Common Language Runtime (CLR) provided by
Microsoft® .NET, unmanaged application components may need to be integrated into or
coexist with new MRTE-based systems. In particular, managed code is code executing
under the control of an MRTE (e.g., any code written in C# (“C-sharp”) from Microsoft®
or Visual Basic .NET) whereas unmanaged code is code executing outside of the MRTE
(e.g., COM components and WIN32 application program interface (API) functions).
However, low-level platform components and legacy application components may not be
compatible with the new MRTE-based systems. That is, some legacy application
components are kept outside of MRTEs (i.e., unmanaged code) because current MRTEs
cannot comprehend certain platform specific features of the legacy application
components. Another reason for keeping some legacy application components outside of
MRTEs is the high cost involved in converting the legacy application components to
managed code.

[0003] Native method invocation (NMI) such as platform invoke is a service that
enables managed code to call unmanaged functions implemented in dynamic link libraries
(DLLs) such as Microsoft® WIN32 API. Typically, an NMI component is used during
common language infrastructure (CLI) runtimes to improve performance of a processor system (i.e., a platform). In particular, the NMI component interacts with many different data structures of a virtual machine to process managed code and unmanaged code. Currently, however, there is no standard for implementation of the NMI component so that information associated with the NMI component may be customized for optimization from one virtual machine to another.

**BRIEF DESCRIPTION OF THE DRAWINGS**

[0004] FIG. 1 is a block diagram representation of an example native method invocation (NMI) system configured in an existing system.

[0005] FIG. 2 is a block diagram representation of an example modular NMI system configured in accordance with an embodiment of the teachings of the invention as disclosed herein.

[0006] FIG. 3 is a block diagram representation showing additional detail of an example modular NMI system as in FIG. 2.

[0007] FIG. 4 is a code representation of example code associated with NMI that may be used to implement an example modular NMI system as in FIG. 3.

[0008] FIG. 5 is a flow diagram representation of example machine accessible instructions that may be executed to implement an example modular NMI system as in FIG. 3.

[0009] FIG. 6 is a block diagram representation of an example processor system that may be used to implement an example modular NMI system as in FIG. 3.
DETAILED DESCRIPTION

[0010] Although the following discloses example systems including, among other components, software or firmware executed on hardware, it should be noted that such systems are merely illustrative and should not be considered as limiting. For example, it is contemplated that any or all of the disclosed hardware, software, and/or firmware components could be embodied exclusively in hardware, exclusively in software, exclusively in firmware or in some combination of hardware, software, and/or firmware.

[0011] Referring to FIG. 1, an NMI system 100 configured in a known manner typically includes a virtual machine (VM) 110 and an NMI component 120. The VM 110 is an execution environment (i.e., runtime) that operates as an interface between a compiler and a processor in a processor system (e.g., the processor system 1000 of FIG. 6) such as Java® Virtual Machine (JVM) and/or Common Language Runtime (CLR) provided by Microsoft®.NET. In particular, the VM 110 includes a marshaling language (ML) stub 112, metadata 114, a garbage collector (GC) 116, and a prestub 118. As used herein “stub” refers to a portion of dynamically-generated code to perform various tasks during execution of a method. As used herein the term “method” refers to one or more applications, programs, functions, routines, or subroutines for manipulating data. The metadata 114 (i.e., NMI information) includes a native method signature, a method return type, and call site information such as a native library path and/or a native library name to generate the ML stub 112. Persons of ordinary skill in the art will readily recognize that the GC 116 recovers memory storage that is no longer used by a method so that the memory storage is available for other processes of that method and/or other methods. The prestub 118 includes code that indicates a NMI call of a method to an entry point during runtime. In general, the performance of a processor may be optimized if the processor operates in its native code because native code is code compiled to run with a particular
processor. Accordingly, the prestub 118 calls into the NMI component 120 when a native method is invoked by the VM 110 to generate a native stub of the native method.

[0012] The NMI component 120 typically includes a plurality of NMI stubs 150 and an NMI cache 160. The plurality of NMI stubs 150 includes a compiled stub 154, an interpreted stub 156, and a platform specific stub 158 as described in detail below. The NMI cache 160 includes an ML stub cache 162 and a native stub cache 164. The ML stub cache 162 stores the ML stub 112 while the native stub cache 164 stores the native stub (not shown) generated based on the ML stub 112 to execute the native method.

[0013] Upon being called by the VM 110 via the prestub 118, the NMI component 120 generates the ML stub 112 to invoke the native method during runtime based on the metadata 114 of the VM 110. In particular, the NMI component 120 translates the ML stub 112 into native instructions as parts of the native stub (i.e., data marshaling). If the data marshaling is simple (e.g., only isomorphic type parameters exist), the ML stub 112 may be compiled to native instructions (i.e., the compiled stub 154). Alternatively, if the data marshaling is complex, the ML stub 112 may be interpreted at runtime (i.e., the interpreted stub 156). The GC 116 is disabled during execution of the native stub and is enabled after the execution of the native stub to ensure that parameters of the NMI method signature passed to the native method are not moved in memory (e.g., the main memory 1030 of FIG. 6). After the native stub is generated based on the ML stub 112, the native stub is stored in the native stub cache 164. As a result, the NMI component 120 may retrieve the native stub from the native stub cache 164 during subsequent calls of the method.

[0014] In the example of FIG. 2, the illustrated modular NMI system 200 includes one or more VMs 210 generally shown as VM #1 212, VM #2 214, and VM #n 216, a modular NMI component 220, and an NMI adapter 230. In general, the NMI adapter 230 operates
as an interface between the VMs 210 and the modular NMI component 220 to solve any mismatch/compatibility problems so that the modular NMI component 220 may be plugged into any of the different VMs 210. In contrast to the VM 110 and the modular NMI component 120 interacting directly with each other as described in connection with FIG. 1, the VMs 210 and the modular NMI component 220 interact with the NMI adapter 230. The NMI adapter 230 acts as a wrapper between the VMs 210 and the modular NMI component 220.

[0015] In an example operation, the VM #1 212 may call the NMI adapter 230 to invoke the modular NMI component 220. The NMI adapter 230 translates NMI information received from VM #1 212 into the API of the modular NMI component 220. Further, the modular NMI component 220 may request the NMI adapter 230 to retrieve additional information from the VMs 210. The NMI adapter 230 separates the modular NMI component 220 from different VM implementations so that the modular NMI component 220 is independent of any particular VM (i.e., an independent NMI module).

The data structure used in the different VMs 210 is irrelevant to the modular NMI component 220 because the modular NMI component 220 retrieves information from any of the different VMs 210 via the NMI adapter 230, which formats retrieved data in a manner known by the modular NMI component 220. That is, the NMI adapter 230 permits the modular NMI component 220 to be plugged into any of the different VMs 210.

As a result, the modular NMI component 220 may operate in connection with VM #1 212, VM #2 214, VM #n 216 and/or any other suitable execution environments.

[0016] In the example of FIG. 3, an illustrated VM 312 includes metadata 314, a GC 316, and a prestub 318. The modular NMI component 320 includes an NMI stub manager 340, a plurality of NMI stubs 350, and an NMI cache 360. The NMI stub manager 340 is the core component in the modular NMI component 320 because the NMI stub manager
340 processes NMI calls from the VM 312 via an NMI adapter 330. In particular, the NMI stub manager 340 receives a VM request to perform an NMI call from the NMI adapter 330, retrieves running background information from an application and a VM, and analyzes the VM request and the running background information to dynamically select a proper stub from the plurality of NMI stubs 350 (i.e., a compiled stub 354, an interpreted stub 356, and/or a platform specific stub 358) to process the NMI call. In contrast to the NMI component 120 described in connection with FIG. 1, the modular NMI component 320, rather than the VM 312, generates an ML stub 352 because the NMI adapter 330 permits the modular NMI component 320 to customize optimization with the VM 312.

The modular NMI component 320 generates a native stub based on the ML stub 352 during runtime to execute a native method. The NMI cache 360 includes an ML stub cache 362 and a native stub cache 364. The modular NMI component 320 stores the ML stub 352 in the ML stub cache 362 and a native stub generated based on the ML stub 352 to execute native method in the native stub cache 364.

In the example of FIG. 4, the illustrated code 400 includes a “DIIImport” function 410 and an “extern” method declaration 420 to generate the ML stub 352 to execute the NMI call from the VM 312 by the modular NMI component 320. In particular, the modular NMI component 320 passes control to the “DIIImport” function 410 to locate a native code library (e.g., a DLL) in “mylib.dll” and load the DLL into memory (e.g., the memory 1030 of FIG. 6). Further, the modular NMI component 320 locates the address of the native method in memory, and pushes arguments (e.g., integers, strings, arrays, structures, etc.) into a stack. The “extern” method declaration 420 indicates that the native method associated with NMI call (e.g., method “foo”) is outside of the VM 310. As a result, the modular NMI component 320 may generate the ML stub 352 to generate the native stub prior to transferring controls to the native method.
Referring back to FIG. 3, the modular NMI component 320 implements the plurality of NMI stubs 350 to perform different optimization strategies. That is, each of the plurality of NMI stubs 350 corresponds to a particular optimization strategy. In particular, the compiled stub 354 parses and compiles the ML stub 352 to native instructions so that no runtime support functions in the ML stub 352. The compiled stub 354 may run faster than the interpreted stub 356 but may include restrictions for parameters being marshaled. The interpreted stub 356 parses and interprets the ML stub 352 when an NMI call is executing. The interpreted stub 356 includes many runtime support functions to perform stack manipulation and data marshaling. The platform specific stub 358 applies optimizations related to the specific platform. For example, Streaming SIMD Extension (SSE) and/or SSE2 instructions may be used to improve performance of a processor system implemented using one or more of the Intel® Pentium® technology, the Intel® Itanium® technology, and/or Intel® Personal Internet Client Architecture (PCA) technology.

A flow diagram 500 representing machine accessible instructions that may be executed by a processor to provide a modular NMI system is illustrated in FIG. 5. Persons of ordinary skill in the art will appreciate that the instructions may be implemented in any of many different ways utilizing any of many different programming codes stored on any of many computer-accessible mediums such as a volatile or nonvolatile memory or other mass storage device (e.g., a floppy disk, a CD, and a DVD). For example, the machine accessible instructions may be embodied in a machine-accessible medium such as an erasable programmable read only memory (EPROM), a read only memory (ROM), a random access memory (RAM), a magnetic media, an optical media, and/or any other suitable type of medium. Alternatively, the machine accessible instructions may be embodied in a programmable gate array and/or an application specific integrated circuit.
(ASIC). Further, although a particular order of actions is illustrated in FIG. 5, persons of ordinary skill in the art will appreciate that these actions can be performed in other temporal sequences. Again, the flow diagram 500 is merely provided and described in conjunction with the example modular NMI system of FIG. 3 as an example of one way to provide a modular NMI system.

[0020] In the example of FIG. 5, the flow diagram 500 begins with the NMI adapter 330 receiving NMI information associated with the NMI call initiated by the VM 312 (block 510). For example, the NMI information may include a native method signature, a method return type, a native library path, and a native library name. Accordingly, the NMI adapter 330 translates the NMI information for the modular NMI component 320 (block 520). The NMI adapter 330 forwards the translated NMI information to the NMI component 320 to generate the ML stub 352 (i.e., data marshaling) (block 530). Based on the ML stub 352, the modular NMI component 320 generates the native stub associated with the NMI call (block 540). During execution of the native stub, the NMI adapter 330 disables the GC 316 (block 550). As noted above, the NMI adapter 330 disables to the GC 316 so that parameters of the NMI method signature passed to the native method are not moved in memory. After execution of the native stub, the NMI adapter 330 enables the GC 316 again so that the GC 316 may offers its services (block 560). Further, the modular NMI component 320 stores the native stub in the native stub cache 364 (block 570). As a result, the modular NMI component 320 and the NMI adapter 330 provide customized optimization in different execution environments.

[0021] The methods and apparatus disclosed herein are well suited for source code implemented using the European Computer Manufacturers Association (ECMA) Common Language Infrastructure (CLI) (second edition, December 2002) and the ECMA C# language specification (second edition, December 2002). However, persons of ordinary
skill in the art will appreciate that the teachings of the disclosure may be applied to source code in other standards or specifications.

[0022] FIG. 6 is a block diagram of an example processor system 1000 adapted to implement the methods and apparatus disclosed herein. The processor system 1000 may be a desktop computer, a laptop computer, a notebook computer, a personal digital assistant (PDA), a server, an Internet appliance or any other type of computing device.

[0023] The processor system 1000 illustrated in FIG. 6 includes a chipset 1010, which includes a memory controller 1012 and an input/output (I/O) controller 1014. As is well known, a chipset typically provides memory and I/O management functions, as well as a plurality of general purpose and/or special purpose registers, timers, etc. that are accessible or used by a processor 1020. The processor 1020 is implemented using one or more processors. For example, the processor 1020 may be implemented using one or more of the Intel® Pentium® technology, the Intel® Itanium® technology, Intel® Centrino™ technology, and/or the Intel® XScale® technology. In the alternative, other processing technology may be used to implement the processor 1020. The processor 1020 may include a cache 1022, which may be implemented using a first-level unified cache (L1), a second-level unified cache (L2), a third-level unified cache (L3), and/or any other suitable structures to store data as persons of ordinary skill in the art will readily recognize.

[0024] As is conventional, the memory controller 1012 performs functions that enable the processor 1020 to access and communicate with a main memory 1030 including a volatile memory 1032 and a non-volatile memory 1034 via a bus 1040. The volatile memory 1032 may be implemented by Synchronous Dynamic Random Access Memory (SDRAM), Dynamic Random Access Memory (DRAM), Rambus Dynamic Random Access Memory (RDRAM), and/or any other type of random access memory device. The non-volatile memory 1034 may be implemented using flash memory, Read Only Memory
(ROM), Electrically Erasable Programmable Read Only Memory (EEPROM), and/or any
other desired type of memory device.

[0025] The processor system 1000 may also include an interface circuit 1050 that is
coupled to the bus 1040. The interface circuit 1050 may be implemented using any type
of well known interface standard such as an Ethernet interface, a universal serial bus
(USB), a third generation input/output interface (3GIO) interface, and/or any other suitable
type of interface.

[0026] One or more input devices 1060 may be connected to the interface circuit 1050.
The input device(s) 1060 permit a user to enter data and commands into the processor
10 1020. For example, the input device(s) 1060 may be implemented by a keyboard, a
mouse, a touch-sensitive display, a track pad, a track ball, an isopoint, and/or a voice
recognition system.

[0027] One or more output devices 1070 may also be connected to the interface circuit
1050. For example, the output device(s) 1070 may be implemented by display devices
15 (e.g., a light emitting display (LED), a liquid crystal display (LCD), a cathode ray tube
(CRT) display, a printer and/or speakers). The interface circuit 1050, thus, typically
includes, among other things, a graphics driver card.

[0028] Further, the processor system 1000 may also include one or more mass storage
devices 1080 to store software and data. Examples of such mass storage device(s) 1080
include floppy disks and drives, hard disk drives, compact disks and drives, and digital
versatile disks (DVD) and drives.

[0029] The interface circuit 1050 may also include a communication device such as a
modem or a network interface card to facilitate exchange of data with external computers
via a network. The communication link between the processor system 1000 and the

10
network may be any type of network connection such as an Ethernet connection, a digital subscriber line (DSL), a telephone line, a cellular telephone system, a coaxial cable, etc.

[0030] Access to the input device(s) 1060, the output device(s) 1070, the mass storage device(s) 1080 and/or the network is typically controlled by the I/O controller 1014 in a conventional manner. In particular, the I/O controller 1014 performs functions that enable the processor 1020 to communicate with the input device(s) 1060, the output device(s) 1070, the mass storage device(s) 1080 and/or the network via the bus 1040 and the interface circuit 1050.

[0031] While the components shown in FIG. 6 are depicted as separate blocks within the processor system 1000, the functions performed by some of these blocks may be integrated within a single semiconductor circuit or may be implemented using two or more separate integrated circuits. For example, although the memory controller 1012 and the I/O controller 1014 are depicted as separate blocks within the chipset 1010, persons of ordinary skill in the art will readily appreciate that the memory controller 1012 and the I/O controller 1014 may be integrated within a single semiconductor circuit.

[0032] Although certain example methods, apparatus, and articles of manufacture have been described herein, the scope of coverage of this patent is not limited thereto. On the contrary, this patent covers all methods, apparatus, and articles of manufacture fairly falling within the scope of the appended claims either literally or under the doctrine of equivalents.
What is claimed is:

1. A method comprising:
   receiving native method invocation (NMI) information associated with an NMI call from one or more virtual machines;
   translating the NMI information via an NMI adapter; and
   providing the translated NMI information to a modular NMI component to generate a native stub.

2. A method as defined in claim 1, wherein receiving NMI information associated with the NMI call from the one or more virtual machines comprises receiving information associated with a platform invoke function call from the one or more virtual machines.

3. A method as defined in claim 1, wherein receiving NMI information associated with the NMI call from the one or more virtual machines comprises receiving one of a native method signature, a method return type, a native library path, and a native library name.

4. A method as defined in claim 1, wherein receiving NMI information associated with the NMI call from the one or more virtual machines comprises receiving NMI information associated with the NMI call from one of a Java Virtual Machine and a Common Language Runtime.
5. A method as defined in claim 1, wherein providing the translated NMI information to an NMI module comprises providing the translated NMI information to a platform invoke module.

6. A method as defined in claim 1 further comprising controlling a garbage collector associated with the one or more virtual machines.

7. A method as defined in claim 1 further comprising generating a marshaling language stub based on the translated NMI information via the NMI module and storing the marshaling language stub within the NMI module.

8. A machine accessible medium having instructions, which when executed, cause a machine to:

   receive native method invocation (NMI) information associated with an NMI call from one or more virtual machines;

   translate the NMI information via an NMI adapter; and

   provide the translated NMI information to an NMI module to generate a native stub.

9. A machine accessible medium as defined in claim 8, wherein the instructions cause the machine to receive NMI information associated with the NMI call from the one or more virtual machines by receiving information associated with a platform invoke function call from the one or more virtual machines.
10. A machine accessible medium as defined in claim 8, wherein the instructions cause the machine to receive NMI information associated with the NMI call from the one or more virtual machines by receiving one of a native method signature, a method return type, a native library path, and a native library name.

11. A machine accessible medium as defined in claim 8, wherein the instructions cause the machine to receive NMI information associated with the NMI call from the one or more virtual machines by receiving NMI information associated with the NMI call from one of a Java Virtual Machine and a Common Language Runtime.

12. A machine accessible medium as defined in claim 8, wherein the instructions cause the machine to provide the translated NMI information to an NMI module by providing the translated NMI information to a platform invoke module.

13. A machine accessible medium as defined in claim 8, wherein the instructions cause the machine to control a garbage collector associated with the one or more virtual machines.

14. A machine accessible medium as defined in claim 8, wherein the instructions cause the machine to generate a marshaling language stub based on the translated NMI information via the NMI module and to store the marshaling language stub within the NMI module.

15. A machine accessible medium as defined in claim 8, wherein the machine accessible medium comprises one of a programmable gate array, application specific
integrated circuit, erasable programmable read only memory, read only memory, random access memory, magnetic media, and optical media.

16. An apparatus comprising:

one or more virtual machines to provide native method invocation (NMI) information associated with an NMI call;

an NMI module to generate a native stub; and

an NMI adapter coupled to the virtual machine and the NMI module to receive the NMI information, to translate the NMI information, and to provide the translated NMI information to the NMI module.

17. An apparatus as defined in claim 16, wherein the NMI adapter controls a garbage collector associated with the virtual machine.

18. An apparatus as defined in claim 16, wherein the virtual machine comprises one of a Java Virtual Machine and a Common Language Runtime.

19. An apparatus as defined in claim 16, wherein the NMI module comprises a platform invoke module.

20. An apparatus as defined in claim 16, wherein the NMI module generates a marshaling language stub based on the translated NMI information via the NMI module and stores the marshaling language stub within the NMI module.
21. An apparatus as defined in claim 16, wherein the NMI information comprises a native method signature, a method return type, a native library path, and a native library name.

22. A processor system comprising:

a dynamic random access memory (DRAM); and

a processor coupled to the DRAM to identify native method invocation (NMI) information associated with an NMI call from one or more virtual machines, to translate the NMI information via an NMI adapter, and to provide the translated NMI information to a modular NMI component to generate a native stub.

23. A processor system as defined in claim 22, wherein the NMI adapter controls a garbage collector associated with the one or more virtual machines.

24. A processor system as defined in claim 22, wherein the virtual machine comprises one of a Java Virtual Machine and a Common Language Runtime.

25. A processor system as defined in claim 22, wherein the modular NMI component comprises a platform invoke module.

26. A processor system as defined in claim 22, wherein the processor generates a marshaling language stub based on the translated NMI information via the NMI module and stores the marshaling language stub within the modular NMI component.
27. A processor system as defined in claim 22, wherein the NMI information comprises a native method signature, a method return type, a native library path, and a native library name.
public class LibWrap {

    [DllImport("mylib.dll", CharSet=CharSet.Auto)]
    public static extern void foo (string s);

}
Start

Receive NMI information from VM

Translate NMI information

Provide translated NMI information to NMI module

Generate native stub based on ML stub

Disable GC during execution of native stub

Enable GC after execution of native stub

Store native stub in native stub cache

End