US 20070294675A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2007/0294675 A1**

Barraclough et al. (43) **Pub. Date: Dec. 20, 2007**

(54) **METHOD AND APPARATUS FOR HANDLING EXCEPTIONS DURING BINDING TO NATIVE CODE**

(75) Inventors: **Gavin Barraclough**, Manchester (GB); **Kit M. Wan**, Oldham (GB); **Abdul R. Hummaida**, Manchester (GB)

Correspondence Address:
WILMERHALE/BOSTON
60 STATE STREET
BOSTON, MA 02109

(73) Assignee: **Transitive Limited**, Manchester (GB)

(21) Appl. No.: **11/546,012**

(22) Filed: **Oct. 10, 2006**

(30) **Foreign Application Priority Data**

Jun. 20, 2006 (GB) GB0612149.5

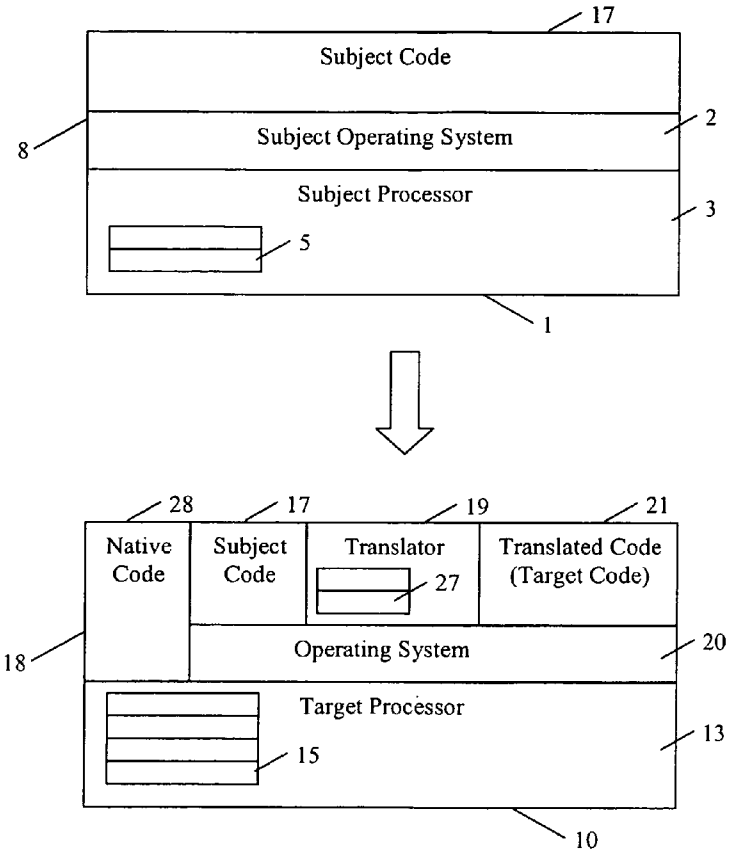
Publication Classification

(51) **Int. Cl.**
G06F 9/45 (2006.01)

(52) **U.S. Cl.** **717/137**

(57) **ABSTRACT**

A method of handling exceptions during native binding under program code conversion from subject code (**17**) executable by a subject computing architecture to target code (**21**) executable by a target computing architecture. Performing native binding executes a portion of native code (**28**) in place of translating a portion of the subject code (**17**) into the target code (**21**). When an exception occurs during the portion of native code (**28**), the method comprises saving a target state (T') which represents a current point of execution in the target computing architecture for the portion of native code (**28**), and creating a subject state (S') which represents an emulated point of execution in the subject computing architecture. The exception is handled through a subject exception handler (**170**, **170'**) with reference to the subject state (S'), such that, upon resuming execution from the exception using the provided subject state (S'), the saved target state (T') is restored to resume execution in the section of portion of native code (**28**). In one embodiment, the subject state (S') links to the saved target state (T') through a recovery unit (**192**).



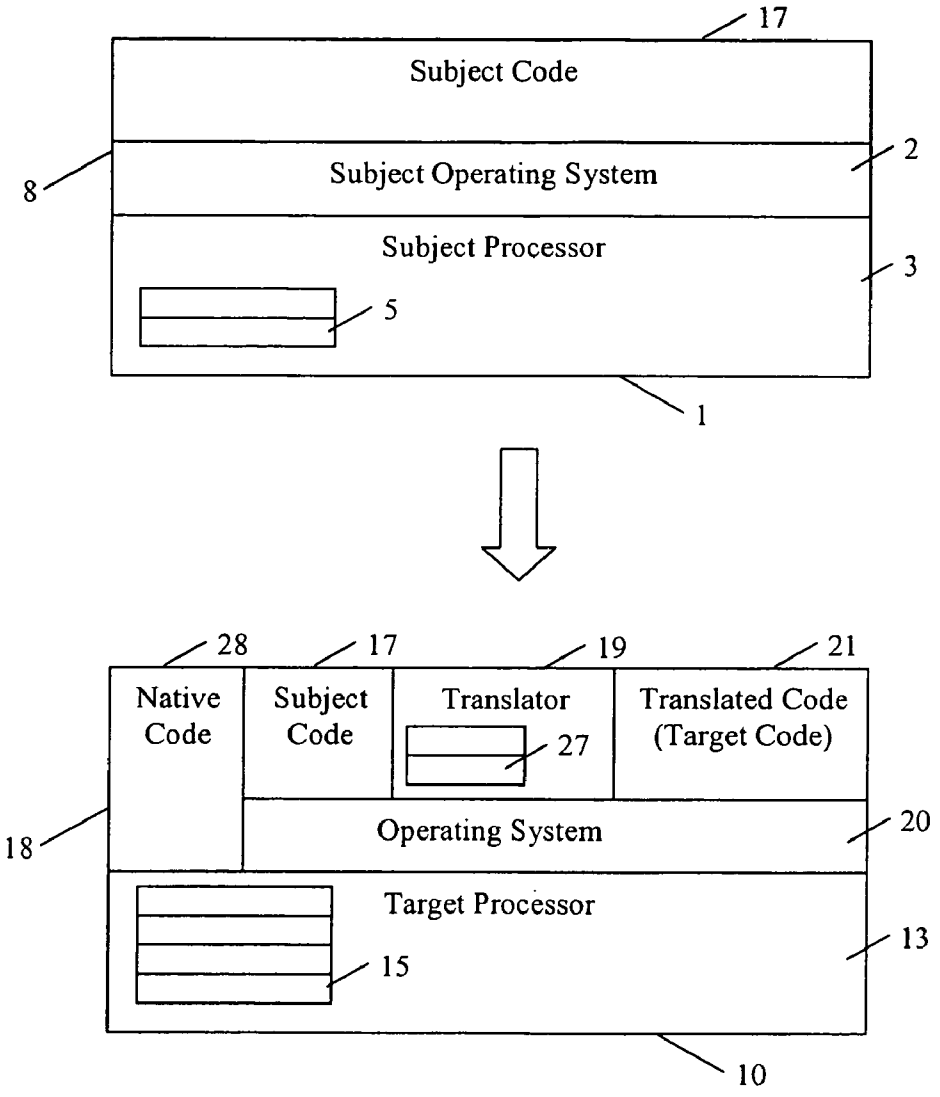

17
Subject Code
8
Subject Operating System
2
Subject Processor
3
5
1
28
17
19
21
Native Code
Subject Code
Translator
27
Translated Code (Target Code)
18
Operating System
20
Target Processor
13
15
10

Fig. 1

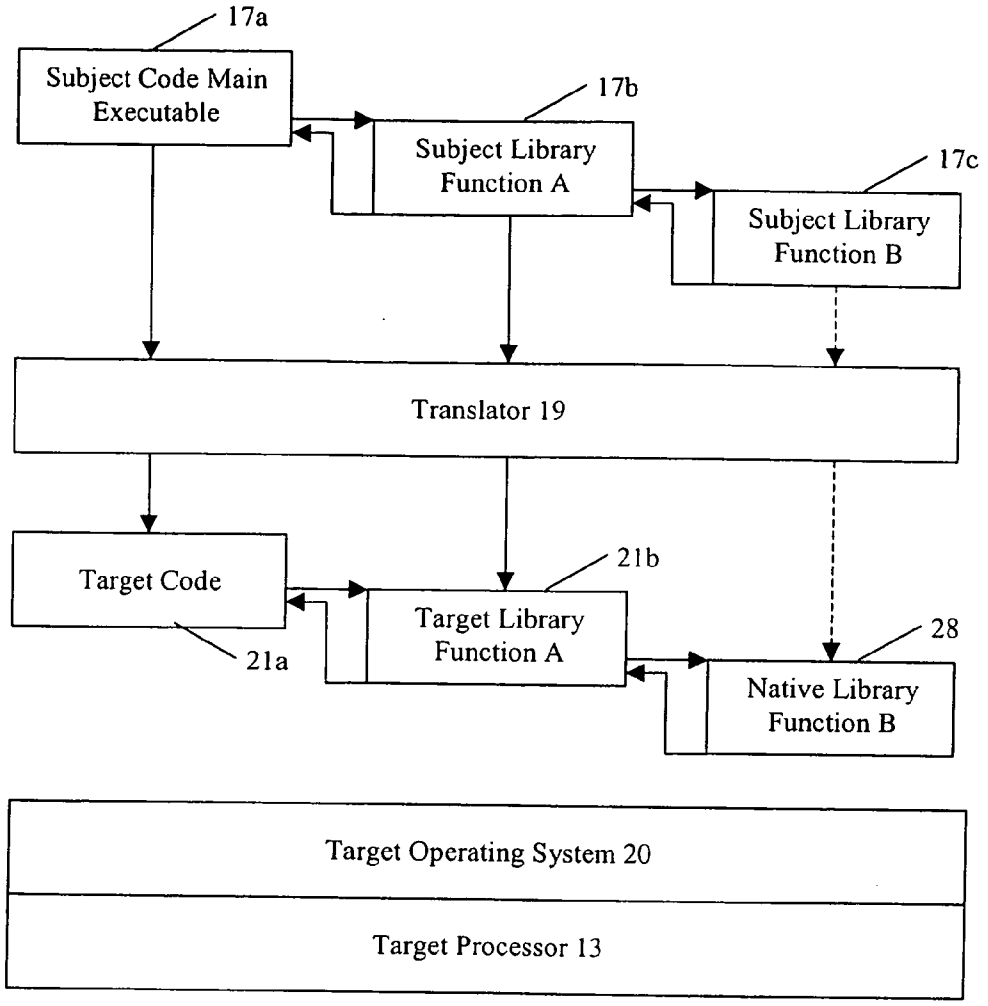

17a
Subject Code Main Executable
17b
Subject Library Function A
17c
Subject Library Function B
Translator 19
Target Code
21a
21b
Target Library Function A
28
Native Library Function B
Target Operating System 20
Target Processor 13

Fig. 2

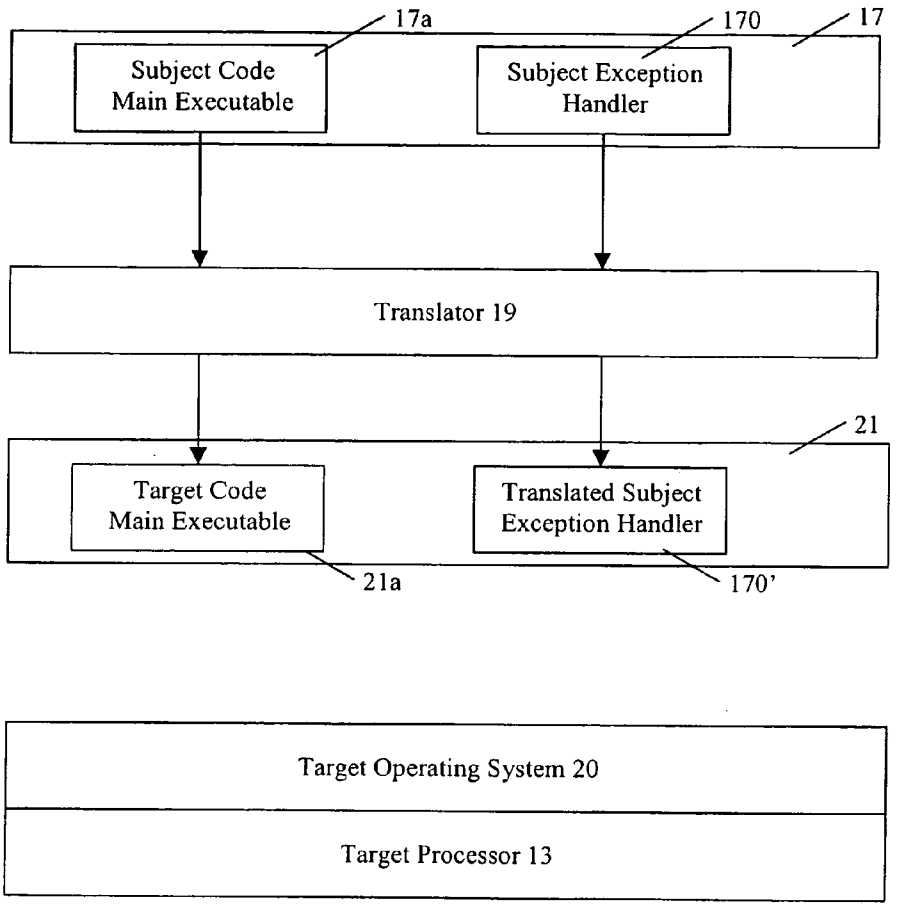

17a
170
17
Subject Code
Main Executable
Subject Exception
Handler
Translator 19
21
Target Code
Main Executable
Translated Subject
Exception Handler
21a
170'
Target Operating System 20
Target Processor 13

Fig. 3

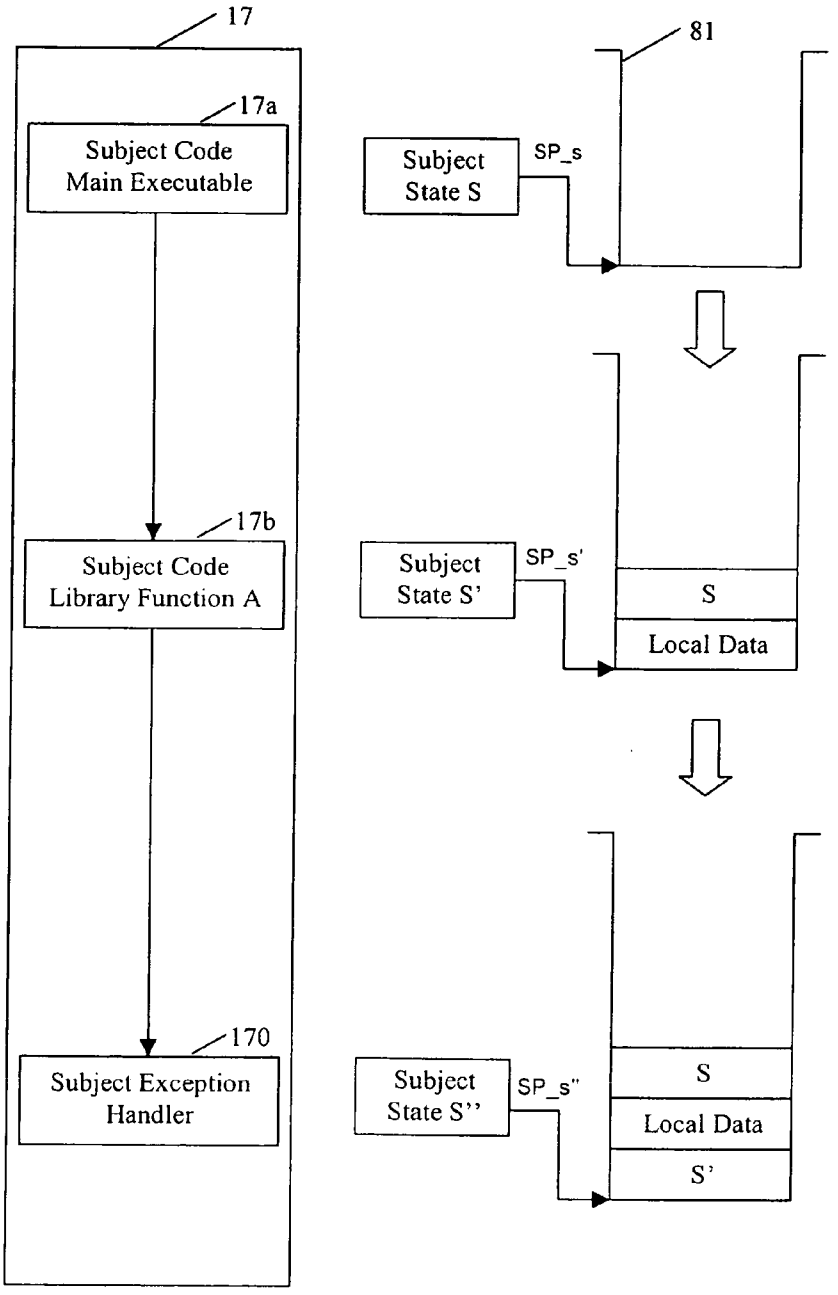

17
17a
Subject Code
Main Executable
17b
Subject Code
Library Function A
170
Subject Exception
Handler
81
Subject
State S
SP_s
Subject
State S'
SP_s'
S
Local Data
Subject
State S''
SP_s"
S
Local Data
S'

Fig. 4

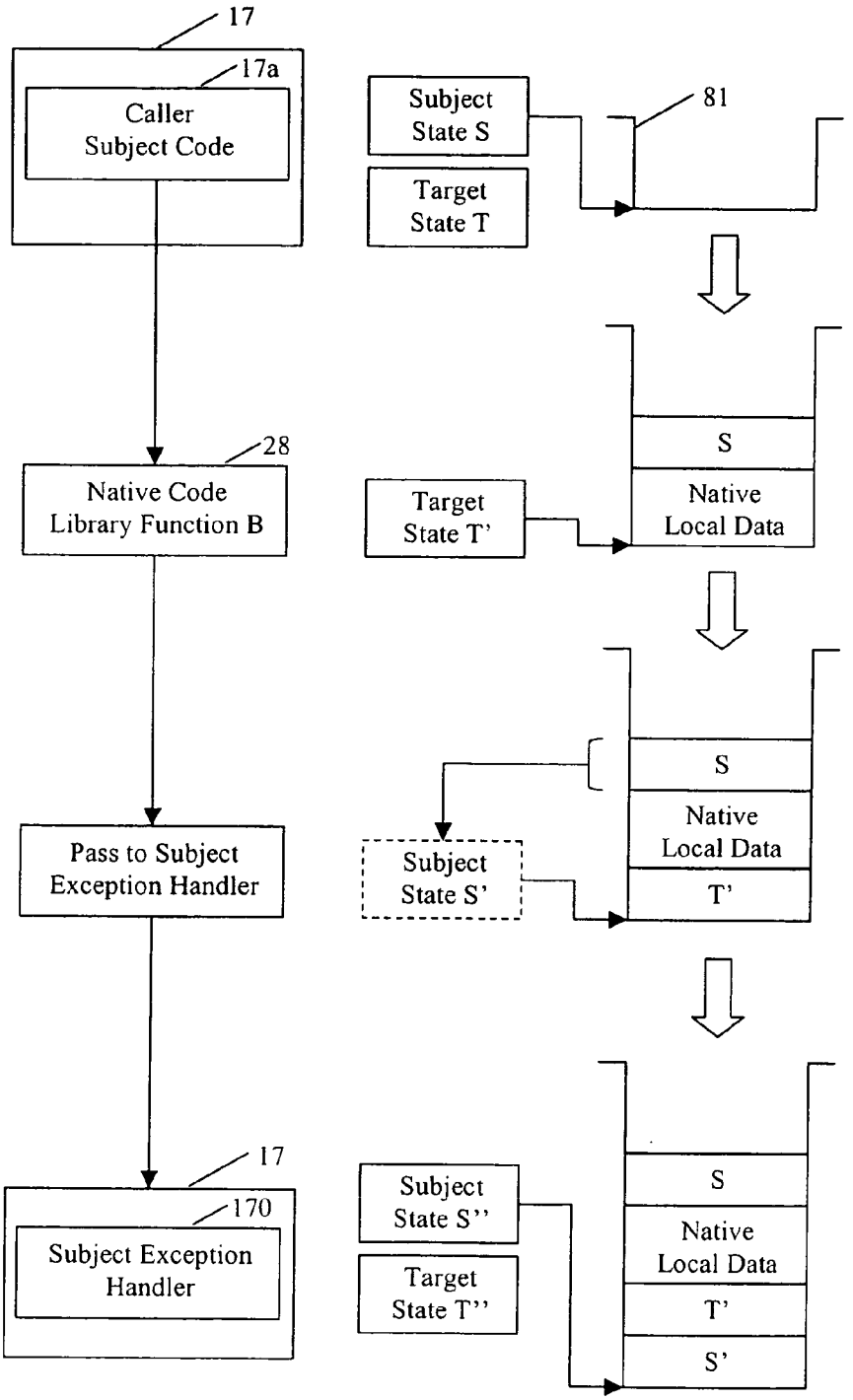

17
17a
Caller
Subject Code
28
Native Code
Library Function B
Pass to Subject
Exception Handler
17
170
Subject Exception
Handler
Subject
State S
Target
State T
81
S
Native
Local Data
Target
State T'
S
Native
Local Data
T'
Subject
State S'
Subject
State S''
Target
State T''
S
Native
Local Data
T'
S'

Fig. 5

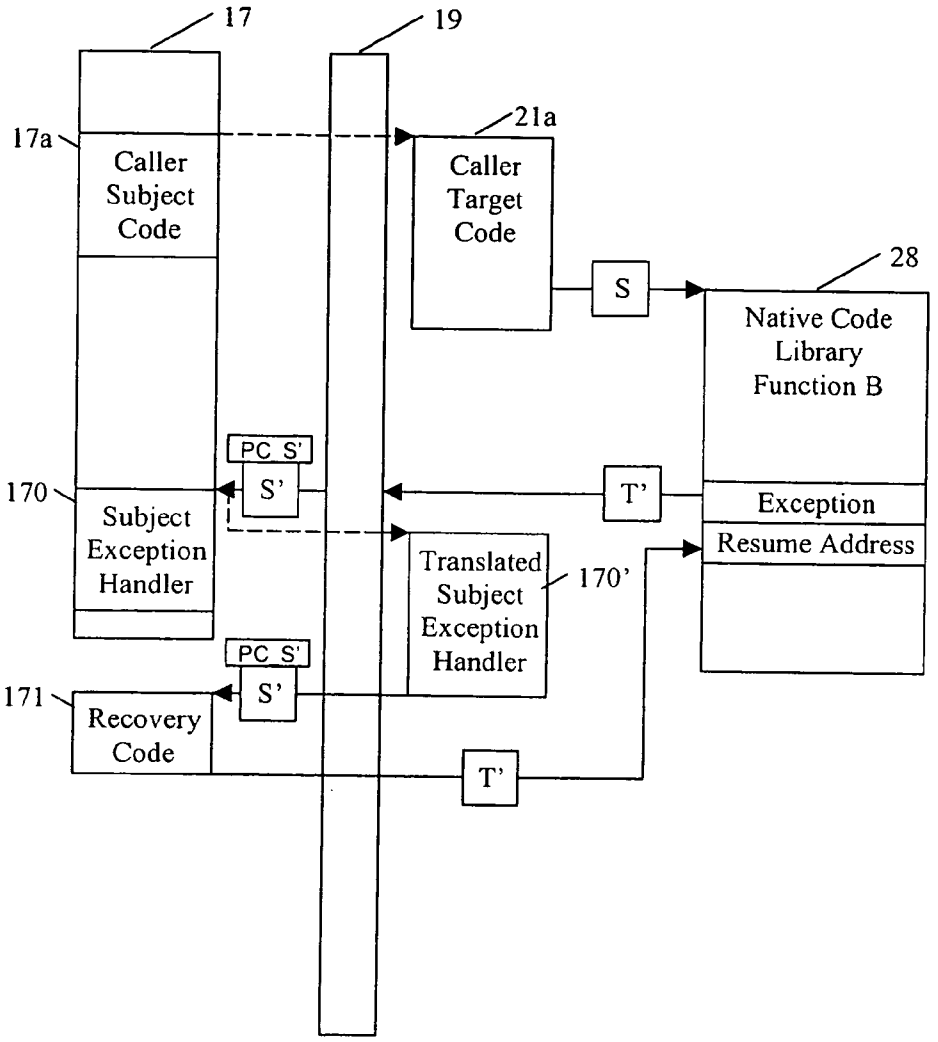

17
19
17a
Caller Subject Code
21a
Caller Target Code
S
28
Native Code Library Function B
PC S'
S'
170
Subject Exception Handler
T'
Exception
Resume Address
Translated Subject Exception Handler
170'
PC S'
S'
171
Recovery Code
T'

Fig. 6

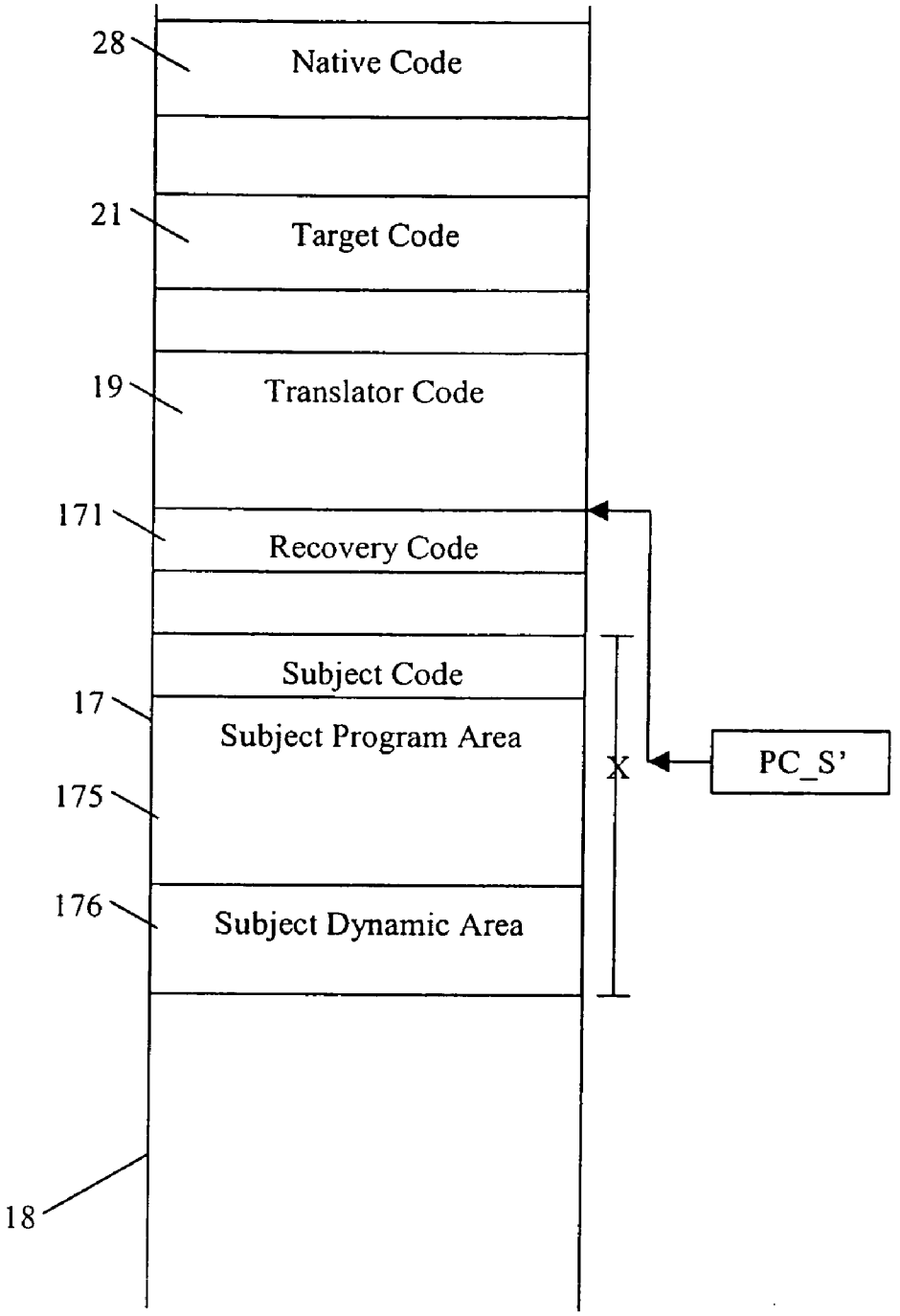

28
Native Code
21
Target Code
19
Translator Code
171
Recovery Code
Subject Code
17
Subject Program Area
175
176
Subject Dynamic Area
X
PC_S'
18

Fig. 7

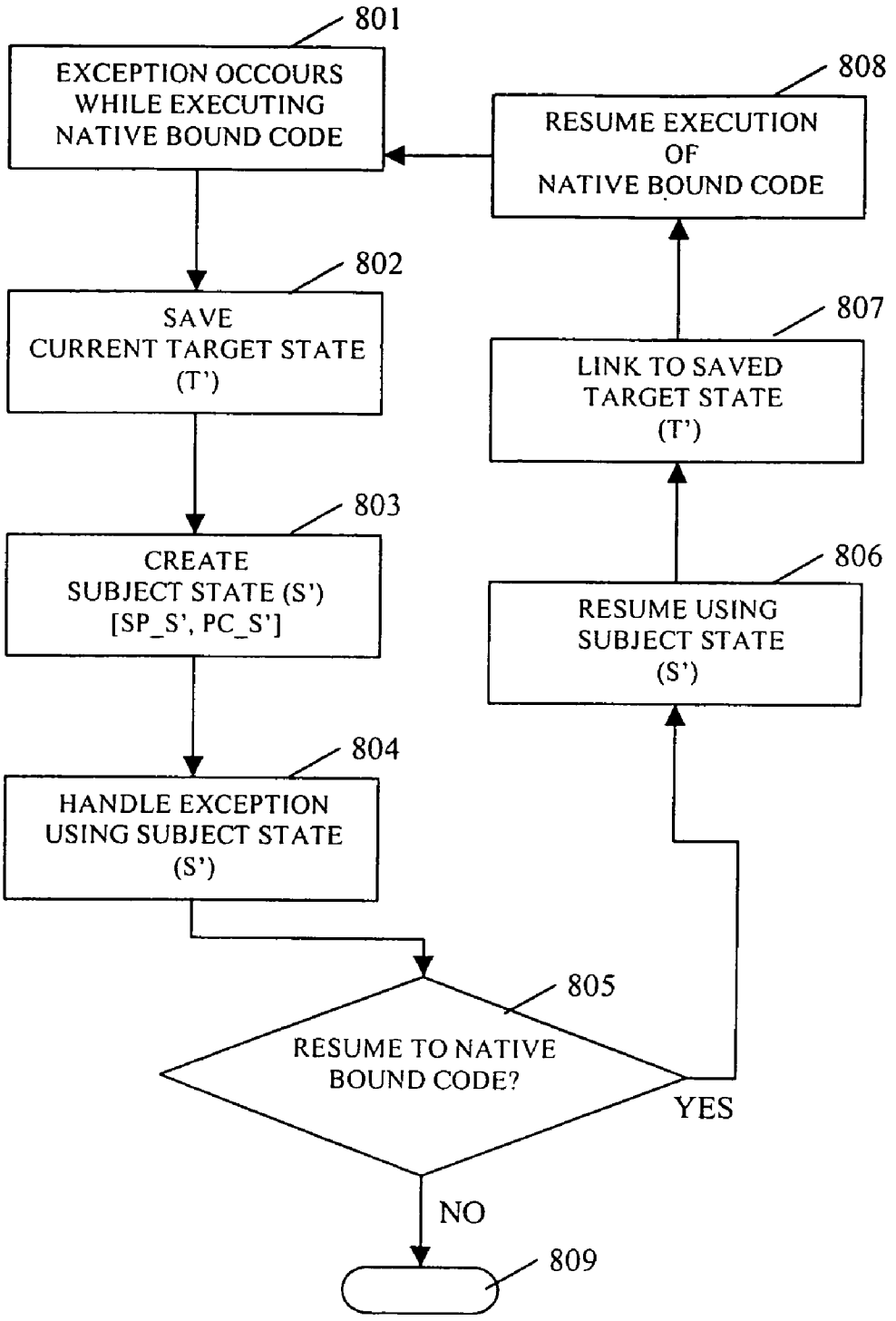

801
EXCEPTION OCCOURS WHILE EXECUTING NATIVE BOUND CODE
802
SAVE CURRENT TARGET STATE (T')
803
CREATE SUBJECT STATE (S') [SP_S', PC_S']
804
HANDLE EXCEPTION USING SUBJECT STATE (S')
805
RESUME TO NATIVE BOUND CODE?
YES
NO
806
RESUME USING SUBJECT STATE (S')
807
LINK TO SAVED TARGET STATE (T')
808
RESUME EXECUTION OF NATIVE BOUND CODE
809

Fig. 8

METHOD AND APPARATUS FOR HANDLING EXCEPTIONS DURING BINDING TO NATIVE CODE

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application claims benefit of the following Patent Application:
U.K. Patent Application No. GB0612149.5, filed Jun. 20, 2006, which is hereby incorporated by reference in its entirety.

FIELD OF THE INVENTION

[0002] The present invention relates generally to the field of computers and computer software and, more particularly, to program code conversion methods and apparatus useful, for example, in code translators, emulators and accelerators which convert program code.

BACKGROUND OF THE INVENTION

[0003] In both embedded and non-embedded CPUs, there are predominant Instruction Set Architectures (ISAs) for which large bodies of software exist that could be "accelerated" for performance, or "translated" to a myriad of capable processors that could present better cost/performance benefits, provided that they could transparently access the relevant software. One also finds dominant CPU architectures that are locked in time to their ISA, and cannot evolve in performance or market reach. Such CPUs would benefit from a software-oriented processor co architecture.
[0004] PCT application WO00/22521 (also published as US2002/0100030A), which is incorporated herein by reference, discloses program code conversion methods and apparatus to facilitate such acceleration, translation and co-architecture capabilities as may be employed in embodiments of the present invention.
[0005] Performing program code conversion brings overheads in the conversion process, compared with native execution of a subject program on a subject processor. A subject program usually consists of multiple units of subject code, including a subject executable and a number of subject libraries, some of which may be proprietary and some of which are provided as part of the subject operating system. As the subject program runs, control flow passes between these different units of subject code as function calls are made to the subject libraries. In some circumstances, native versions of certain subject libraries may be available on the target architecture.
[0006] PCT application WO2005/008478 (also published as US2005/0015781A), which is incorporated herein by reference, discloses a native binding technique for inserting calls to native functions during translation of subject code to target code, such that function calls in the subject program to subject code functions are replaced in target code with calls to native equivalents of the same functions. This avoids the overhead of translating the subject versions of those functions. In addition, the native version may be a much more efficient implementation of the same functionality, as the native version can better exploit architectural features of the target architecture which may not be available on the subject architecture.
[0007] Although the native binding technique is very useful, the inventors have now identified in that a difficulty arises in relation to exception handling. An exception is a condition that changes the normal control flow in a program. An exception indicates that a condition has occurred somewhere within the system that requires the attention of the processor, and the exception usually needs to be handled before processing can continue. Exceptions can be subdivided into various different types such as interrupts, faults, traps or aborts. The terminology varies between different architectures, and particular types or categories of exceptions may be unique to particular architectures.
[0008] Undertaking the native binding mechanism reduces overhead in the translation process, such as the work of maintaining an accurate subject state. However, the execution of such native code also brings difficulties such as in the reporting of a subject state to a subject exception handler. Further, difficulties have now been identified in relation to resuming execution of the subject program after the exception has been handled, and particularly where it is desired to resume execution of the native code.

SUMMARY OF THE INVENTION

[0009] According to the present invention there is provided an apparatus and method as set forth in the appended claims. Preferred features of the invention will be apparent from the dependent claims, and the description which follows.
[0010] The following is a summary of various aspects and advantages realizable according to embodiments of the invention. It is provided as an introduction to assist those skilled in the art to more rapidly assimilate the detailed design discussion that ensues and does not and is not intended in any way to limit the scope of the claims that are appended hereto.
[0011] In particular, the inventors have developed methods directed at program code conversion, which are particularly useful in connection with a run-time translator that provides dynamic binary translation of subject program code into target code.
[0012] In one aspect of the present invention there is provided a method of handling exceptions during native binding under program code conversion from subject code executable by a subject computing architecture to target code executable by a target computing architecture. Performing native binding executes a portion of native code, usually in place of translating a portion of the subject code into the target code. When an exception occurs during execution of the native bound code, the method comprises saving a target state which represents a current point of execution in the target computing architecture for the native bound code, providing a subject state, and handling the exception with reference to the subject state, such that, upon resuming execution from the exception using the subject state, the saved target state is restored to resume execution of the native bound code.
[0013] In the example embodiments, the subject state represents a point of execution in the subject computing architecture, which is emulated by the target computing platform. Particularly, the subject state may include a stack pointer to a subject stack data structure in the memory of the target computing platform used to emulate a procedure stack (or control stack) of the subject architecture. Conveniently, the target state is saved to the subject stack, such that the subject stack pointer links to the stored target state. Further, in the example embodiments, execution of the native bound

code uses the subject stack. Further still, in example embodiments the subject state includes a modified program counter which is used to redirect execution control when returning from handling the exception. In some embodiments, a recovery routine is performed which links from the subject state to the saved target state.

[0014] The present invention also extends to a translator apparatus arranged to perform the embodiments of the invention discussed herein. Also, the present invention extends to computer-readable storage medium having recorded thereon instructions which may be implemented by a computer to perform any of the methods defined herein.

[0015] At least some embodiments of the invention may be constructed, partially or wholly, using dedicated hardware. Optionally, the target computing platform is provided as a function-specific dedicated translator apparatus.

[0016] Terms such as 'module' or 'unit' used herein may include, but are not limited to, a hardware device, such as a Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC), which performs certain tasks. Alternatively, elements of the invention may be configured to reside on an addressable storage medium and be configured to execute on one or more processors. Thus, functional elements of the invention may in some embodiments include, by way of example, components, such as software components, object-oriented software components, class components and task components, processes, functions, attributes, procedures, subroutines, segments of program code, drivers, firmware, microcode, circuitry, data, databases, data structures, tables, arrays, and variables. Further, although the preferred embodiments have been described with reference to the components, modules and units discussed below, such functional elements may be combined into fewer elements or separated into additional elements.

BRIEF DESCRIPTION OF THE DRAWINGS

[0017] The accompanying drawings, which are incorporated in and constitute a part of the specification, illustrate presently preferred implementations and are described as follows:

[0018] FIG. **1** is a block diagram illustrative of apparatus wherein embodiments of the invention find application;

[0019] FIG. **2** is a schematic diagram illustrating a native binding mechanism as employed in embodiments of the invention;

[0020] FIG. **3** is a schematic diagram showing components which are employed for the handling of exceptions by example embodiments of the present invention;

[0021] FIG. **4** is a schematic diagram showing execution in a subject computing platform when calling a subject library function and when handling an exception during execution of the library function;

[0022] FIG. **5** is a schematic diagram showing execution in a target computing platform when performing native binding and when handling an exception in native bound code, as employed in embodiments of the present invention;

[0023] FIG. **6** is a schematic diagram showing a flow of execution control when handling an exception in native bound code, as employed in embodiments of the present invention;

[0024] FIG. **7** is a schematic diagram of a portion of the memory of the target computing platform; and

[0025] FIG. **8** is a schematic flow diagram illustrating a method of handling exceptions for native bound code as employed by example embodiments of the present invention.

DETAILED DESCRIPTION

[0026] The following description is provided to enable a person skilled in the art to make and use the invention and sets forth the best modes contemplated by the inventors of carrying out their invention. Various modifications, however, will remain readily apparent to those skilled in the art, since the general principles of the present invention have been defined herein specifically to provide an improved program code conversion method and apparatus.

[0027] FIG. **1** gives an overview of a system and environment where the example embodiments of the present invention may find application, in order to introduce the units, components and elements that will be discussed in more detail below. Referring to FIG. **1**, a subject program **17** is intended to execute on a subject computing platform **1** having at least one subject processor **3**. However, a target computing platform **10** is instead used to execute the subject program **17**, through a translator unit **19** which performs program code conversion. The translator unit **19** performs code conversion from the subject code **17** to target code **21**, such that the target code **21** is executable on the target computing platform **10**.

[0028] As will be familiar to those skilled in the art, the subject processor **3** has a set of subject registers **5**. A subject memory **8** holds, inter alia, the subject code **17** and a subject operating system **2**. Similarly, the example target computing platform **10** in FIG. **1** comprises a target processor **13** having a plurality of target registers **15**, and a memory **18** to store a plurality of operational components including a target operating system **20**, the subject code **17**, the translator code **19**, and the translated target code **21**. The target computing platform **10** is typically a microprocessor-based computer or other suitable computer.

[0029] In one embodiment, the translator code **19** is an emulator to translate subject code of a subject instruction set architecture (ISA) into translated target code of another ISA, with or without optimisations. In another embodiment, the translator **19** functions as an accelerator for translating subject code into target code, each of the same ISA, by performing program code optimisations.

[0030] The translator code **19** is suitably a compiled version of source code implementing the translator, and runs in conjunction with the operating system **20** on the target processor **13**. It will be appreciated that the structure illustrated in FIG. **1** is exemplary only and that, for example, embodiments of the invention may be implemented within or beneath the operating system **20** of the target platform. The subject code **17**, translator code **19**, operating system **20**, and storage mechanisms of the memory **18** may be any of a wide variety of types, as known to those skilled in the art.

[0031] In the example apparatus according to FIG. **1**, program code conversion is performed dynamically, at runtime, to execute on the target architecture **10** while the target code **21** is running. That is, the translator **19** runs inline with the translated target code **21**. Running the subject program **17** through the translator **19** involves two different types of code that execute in an interleaved manner: the translator code **19**; and the target code **21**. Hence, the target code **21**

is generated by the translator code **19**, throughout run-time, based on the stored subject code **17** of the program being translated.

[0032] In one embodiment, the translator unit **19** emulates relevant portions of the subject architecture **1** such as the subject processor **3** and particularly the subject registers **5**, whilst actually executing the subject program **17** as target code **21** on the target processor **13**. In the preferred embodiment, at least one global register store **27** is provided (also referred to as the subject register bank **27** or abstract register bank **27**). In a multiprocessor environment, optionally more than one abstract register bank **27** is provided according to the architecture of the subject processor. A representation of a subject state is provided by components of the translator **19** and the target code **21**. That is, the translator **19** stores the subject state in a variety of explicit programming language devices such as variables and/or objects. The translated target code **21**, by comparison, provides subject processor state implicitly in the target registers **15** and in memory locations **18**, which are manipulated by the target instructions of the target code **21**. For example, a low-level representation of the global register store **27** is simply a region of allocated memory. In the source code of the translator **19**, however, the global register store **27** is a data array or an object which can be accessed and manipulated at a higher level.

[0033] The term "basic block" will be familiar to those skilled in the art. A basic block is a section of code with exactly one entry point and exactly one exit point, which limits the block code to a single control path. For this reason, basic blocks are a useful fundamental unit of control flow. Suitably, the translator **19** divides the subject code **17** into a plurality of basic blocks, where each basic block is a sequential set of instructions between a first instruction at a single entry point and a last instruction at a single exit point (such as a jump, call or branch instruction). The translator **19** may select just one of these basic blocks (block mode) or select a group of the basic blocks (group block mode). A group block suitably comprises two or more basic blocks which are to be treated together as a single unit. Further, the translator may form iso blocks representing the same basic block of subject code but under different entry conditions.

[0034] In the preferred embodiments, trees of Intermediate Representation (IR) are generated based on a subject instruction sequence, as part of the process of generating the target code **21** from the original subject program **17**. IR trees are abstract representations of the expressions calculated and operations performed by the subject program. Later, the target code **21** is generated based on the IR trees. Collections of IR nodes are actually directed acyclic graphs (DAFs), but are referred to colloquially as "trees".

[0035] As those skilled in the art may appreciate, in one embodiment the translator **19** is implemented using an object-oriented programming language such as C++. For example, an IR node is implemented as a C++ object, and references to other nodes are implemented as C++references to the C++ objects corresponding to those other nodes. An IR tree is therefore implemented as a collection of IR node objects, containing various references to each other.

[0036] Further, in the embodiment under discussion, IR generation uses a set of abstract register definitions which correspond to specific features of the subject architecture upon which the subject program **17** is intended to run. For example, there is a unique abstract register definition for each physical register on the subject architecture (i.e., the subject registers **5** of FIG. **1**). As such, abstract register definitions in the translator may be implemented as a C++object which contains a reference to an IR node object (i.e., an IR tree). The aggregate of all IR trees referred to by the set of abstract register definitions is referred to as the working IR forest ("forest" because it contains multiple abstract register roots, each of which refers to an IR tree). These IR trees and other processes suitably form part of the translator **19**.

Native Binding

[0037] FIG. **1** further shows native code **28** in the memory **18** of the target architecture **10**. There is a distinction between the target code **21**, which results from the run-time translation of the subject code **17**, and the native code **28**, which is written or compiled directly for the target architecture. Native code **28** is generated external to the translator **19**, meaning that the translator **19** does not dynamically generate the native code **28** and the translator **19** does have an opportunity to modify or optimize the native code **28**.

[0038] FIG. **2** is a more detailed schematic diagram illustrating a native binding mechanism as employed in embodiments of the present invention.

[0039] Native binding is implemented by the translator **19** when it detects that the subject program's flow of control enters a section of subject code **17**, such as a subject library, for which a native version of the subject code exists. Rather than translating the subject code, the translator **19** instead causes the equivalent native code **28** to be executed on the target processor **13**. In example embodiments, the translator **19** binds generated target code **21** to the native code **28** using a defined interface, such as native code or target code call stubs, as discussed in more detail in WO2005/008478 (and US2005/0015781A) referenced above.

[0040] The subject program **17** usually includes one or more subject executable files **17***a* which are translated into target code **21***a*. The subject executable **17***a* may in turn refer to and make use of a number of subject libraries including proprietary libraries and/or system libraries. Two example library functions **17***b*, **17***c* are illustrated. The translator **19** uses native binding to replace calls to certain of the subject library functions with calls to equivalent functions in native libraries provided in the native code **28**. In this example, the translator **19** has translated a first library function A into target code **21***b*, whereas a second library function B is native bound to a native library function in native code **28**. These native libraries are typically part of the target operating system **20**, but may also be provided to the target system along with the translator **19**.

[0041] As an illustrative example, the translator **19** is arranged to perform a MIPS to x86 translation. Here, the x86 target system library "libc" defines an advanced native memcpy( ) (memory copy) routine that takes advantage of SSE2 vector operations to perform extremely fast byte copies. Using native binding, calls to a subject memcpy function in the MIPS subject code are bound to the native memcpy( ). This eliminates the cost of translating the subject (MIPS) version of the memcpy( ) function. In addition, the native (×86) version of the memcpy( ) is adapted to the intricacies of the native hardware, and can achieve the function's desired effect in the most efficient way for that hardware.

[0042] Native binding is primarily applicable to library functions, but may also be implemented for any well-defined section of subject code for which a native code equivalent is available in the target architecture. That is, in addition to target system library calls, native binding may be used for more arbitrary code substitution, such as substituting a natively compiled version of a non-library function. Furthermore, native binding may be used to implement subject system calls on a native architecture, by replacing all calls to subject system functions with substitute native functions that either implement the same functionality as the calls to subject system functions or act as call stubs around target system calls. Native binding may also be applied at arbitrary subject code locations, beyond function call sites, to allow arbitrary code sequences (in either target code or native code) and/or function calls to be inserted or substituted at any well-defined point in the subject program.

Exception Handling

[0043] FIG. **3** is a more detailed schematic diagram of the target computing platform **10** of FIG. **1**, showing components which are employed relevant to the handling of exceptions as performed by an example embodiment of the present invention.

[0044] An exception may be generated ("raised") by hardware or by software. Hardware exceptions include signals such as resets, interrupts, or signals from a memory management unit. As examples, exceptions may be generated by an arithmetic logic unit or floating-point unit for numerical errors such as divide-by-zero, for overflow or underflow, or for instruction decoding errors such as privileged, reserved, trap or undefined instructions. Software exceptions occur in many different forms across various software programs and could be applied to any kind of error checking which alters the normal behaviour of the program. As an illustrative example, an instruction in the subject code causes a software exception to be reported if the value of one register is greater than the value of a second register.

[0045] Typically, one or more subject exception handlers **170** are provided (registered) to handle exceptions which occur during execution of the subject program **17**. An exception handler is special code which is called upon when an exception occurs during the execution of a program. If the subject program does not provide a handler for a given exception, then a default system exception handler may be called. The exception handler will usually try to take corrective action and resume execution, or abort running of the subject program and return an error indication. In the context of program code conversion, it is desirable to accurately model, on the target system, the behaviour of the subject exception handler(s).

[0046] Exception signals are a common mechanism for raising exceptions on many operating systems. The POSIX standard, which is adhered to by many operating systems, particularly Unix-like systems, specifies how this mechanism should behave so that exception signals are broadly similar across many systems. The most common events that trigger exceptions are when a process implemented by a program tries to access an unmapped memory region or manipulate a memory region for which it does not have the correct permissions. Other common events that trigger exception signals are the receipt of a signal sent from another process, the execution by a process of an instruction that the process does not have the privilege level to execute, or an I/O event in the hardware.

[0047] FIG. **3** shows a set of subject exception handlers **170**, which may include specific subject exception handlers that are specific to a particular type of exception and one or more default system exception handlers to be employed where a specific exception handler is not registered. Conveniently, the subject exception handlers **170** are made available on the target platform as part of the subject code **17**.

[0048] The translator **19** also provides a corresponding set of translated subject exception handlers **170'** in target code **21** to execute on the target processor **13**, which emulate the subject exception handlers **170**. In particular embodiments, the subject exception handlers **170** are dynamically translated into executable target code versions when needed. It will be understood that reference to a subject exception handler **170** in the following description includes, where appropriate, a reference to the translated target code version of the subject exception handler.

[0049] FIG. **4** is a schematic diagram showing execution in the subject computing platform when calling a subject library function and when handling an exception during execution of the library function.

[0050] When an exception occurs, a current subject state is stored to a predetermined location (e.g. to a stack) and execution control passes to the appropriate subject exception handler **170**. The subject exception handler **170** will often use this stored subject state information in order to handle the exception. Also, if the exception handler so determines, the subject state is used to resume execution of the subject program, either at the same point as where the exception occurred, or at some other point in the subject program. The subject exception handler may, as part of handling the exception, alter the stored subject state, such as by altering a stored program counter. Hence, in the context of program code conversion, it is desirable to accurately follow the expected behaviour of the subject exception handler **170**.

[0051] As will be familiar to persons skilled in the art, in architectures which use a stack for procedure calls, a subject stack **81** stores information about the active subroutines or library functions which have been called by the subject program. Usually, the subject stack **81** is provided in the memory **8** of the subject platform **1**, and many processors provide special hardware to manipulate such stack data structures in memory. The main role of the stack **81** is to keep track of the point to which each active function should return when it finishes executing, although the stack may also be used for other purposes such as to pass function parameters and results, and to store local data. Typically, each function call puts linking information on the stack, including a return address. This kind of stack is also known as an execution stack, control stack, or function stack. Usually, one stack is associated with each running program or with each task of a process. The exact details of the stack depend upon many factors including, for example, the subject hardware, the subject operating system, and the instruction set architecture of the subject platform.

[0052] In FIG. **4**, the subject state S includes information such as the content of at least some of the subject registers. In particular, the subject state S may include information such as current values of a subject stack pointer (SP_S) and a subject program counter (PC_S), amongst others. When the caller subject program **17***a* calls a subject library func-

tion **17***b*, the subject state S is stored by pushing the contents of the subject registers to the subject stack **81**, and the subject stack pointer SP_S is updated to point to the top of the subject stack **81**. During execution of the called library function **17***b*, local data may be stored on the subject stack **81** and a new subject state S' is formed. When an exception occurs, the new subject state S' is stored on the stack **81**, and execution passes to the subject execution handler **170**. After handling the exception, the subject platform will be in a third state S". In this example, the second stored state S' is recovered from the stack after handling the exception, and execution resumes at the point in the subject library function **17***b* where the exception occurred. Later, the first stored state S is recovered when execution returns from the library function **17***b* to the caller program **17***a*.

[0053] FIG. **5** is a schematic diagram showing a state of execution in the target computing platform when performing native binding and when handling an exception in native bound code, as employed in embodiments of the present invention. In particular, FIG. **5** illustrates an example embodiment of the present invention which allows a useful subject state S' to be reported to the subject exception handler **170**, and also allows execution to resume in the native bound code **28**.

[0054] For program code conversion as discussed herein, the translator **19** provides elements on the target platform **10** which are, in general terms, functionally equivalent to those on the subject platform **1**. In this example, the translator **19** provides a representation of the subject stack **81** in the target memory **18**, and represents the subject registers **5** using the abstract register bank **27**. Hence, the translator **19** is able to emulate all of the structures shown in FIG. **4** when the subject code **17***a* calls the subject library function **17***b* and when a exception is handled in the subject exception handler **170**. However, when performing native binding, the translator **19** no longer has close control over the native bound code **28** and the translator **19** cannot maintain a subject state S' which remains precisely equivalent to the subject state during execution of the subject library function **17***c* on the subject platform. However, the embodiments discussed herein allow the translator **19** to provide an appropriate equivalent of the second subject state S' to the subject exception handler **170**. Further, the embodiments discussed herein allow execution control to return to the native bound code **28** after the exception has been handled.

[0055] In FIG. **5**, the target platform **10** has a target state T which represents information such as a target program counter and a target stack pointer, and the current state of the target registers **15** in the target processor **13**. Initially, the target state T reflects the execution of target code **21** produced by the translator **19** from the caller subject program **17***a*. Execution of the target code **21** causes a subject state S to be provided on the target platform, as discussed above.

[0056] The native binding technique is employed to execute a native code library function B, and the first target state T evolves to a second target state T' during execution of the native bound code **28**. When an exception occurs during execution of the native bound code **28**, an exception signal is raised (i.e. by the target OS **20**) and passed to the registered exception handler. As part of handling the exception, the target state T' is saved to an appropriate storage location in the target system, in this case to the subject stack **81**. This second target state T' represents a current point of execution in the target processor **13** for the native bound code **28**, at the point when the exception occurred. Also, an exception handler unit **191** of the translator **19** creates and stores a subject state S', before passing execution control to the subject exception handler **170**. The subject execution handler **170** is invoked to handle the exception with reference to the created subject state S'. Here, the second subject state S' comprises at least a subject stack pointer (SP_S) pointing to the subject stack **81** above the saved target state T'. Conveniently, the previously saved subject state S is used as a foundation for the second subject state S', with a modification to include the required new value of the subject stack pointer (SP_S).

[0057] Execution of the subject exception handler **170** results in a third subject state S" and a third target state T" (due to the work done on the target platform to handle the exception). However, the subject exception handler **170** is now able to refer to the saved subject state S' in order to resume execution in the native bound code **28** at the point where the exception occurred. That is, the saved subject state S' owns the saved target state T' and resuming execution of the subject code at the saved subject state S' resumes execution of the native bound code **28**.

[0058] It is useful to note that, in this illustrated embodiment, the first and third target states T and T" refer to execution of the target code **21** produced by the translator **19** from the relevant subject code **17**. In this embodiment, execution of the target code **21** uses a target stack (not shown) also provided in the memory of the target platform separately from the subject stack **81**. By contrast, the second target state T' refers to execution of the native code **28** using the subject stack **81**. Hence, the second target state T' is shown to include a stack pointer (here illustrated with an arrow) which points to the subject stack **81**, whilst the first and third target states do not.

[0059] As shown in FIG. **5**, the example embodiments of the present invention use the subject stack **81** for execution of the native bound code **28**. As discussed above, the subject stack **81** is a designated area of the memory **18**, which the translator **19** manages on behalf of the subject code **17** as part of the subject code to target code translation. Using the subject stack **81** for execution of the native code **28** prevents the native context T' being lost as a result of servicing an exception during execution of that native bound code. In particular, the subject stack **81** is preserved when executing the translated version of the subject exception handler **170**. As a further advantage, using the subject stack **81** for native binding execution allows environment switches in the subject code **17** to be dealt with transparently, such as by executing library calls similar to a "longjmp" function. A longjmp function restores a subject stack and subject state previously saved by calling a setjmp. This provides a way to execute a non-local "go to" type instruction and is typically used to pass execution to recovery code from the subject exception handler **170**. Therefore, by using the subject stack **81** for native binding, resources can be reclaimed transparently if the subject code **17** calls a longjmp. As part of the translation of the subject longjmp function, the translated code resets the subject stack pointer SP_S to reclaim stack space allocated by the native binding mechanism.

[0060] In some alternate embodiments of the present invention, execution of the native code **28** may employ a different stack elsewhere in the memory **18** of the target platform **10**, such as a stack of the translator **19** (translator

stack) or a separately allocated native stack (not shown), instead of the subject stack **81**.

[0061] FIG. **6** shows a flow of execution control when handling an exception in native bound code, as employed in embodiments of the present invention.

[0062] In FIG. **6**, the caller subject code **17***a* (executing as caller target code **21***a*) calls into the native bound code **28** and an appropriate subject state S is saved on the subject stack **81** to allow a return into the caller target code **21***a*. An exception occurs during the native bound code **28** which, as discussed above, provides a target state T'. The native binding mechanism returns control to the translator **19** and the target state T' is saved.

[0063] As discussed above, the translator **19** generates the second subject state S'. In this aspect of the invention, the second subject state includes, inter alia, at least a subject program counter PC_S' which is specially modified by the translator **19**. In particular, the specially modified subject program counter value PC_S' passed to the subject exception handler **170** does not correspond to a program address of the subject program **17**. However, the subject exception handler **170** (executing as translated subject exception handler **170'**) may use this subject program counter PC_S' as a return address when attempting to restart execution at the point where the exception occurred.

[0064] In a first example embodiment as shown in FIG. **6**, the translator **19** includes a recovery unit **192** which acts to recover the target state T' and resume execution in the native bound code. In one embodiment, the recovery unit is form by providing a reserved location **171** (see FIG. **7**), such as a memory page within the translator's address space, which the translator **19** will treat as a portion of subject code **17**. Thus, when the translated subject exception handler **170'** attempts to pass control to the program location identified by the specially modified subject program counter PC_S', the translator **19** is directed to the reserved location **171**. In this first embodiment, the reserved location **171** contains subject code instructions which, when translated and executed as target code, cause the target state T' to be recovered such that the native bound code **28** resumes execution. Here, restoring the target state T' suitably includes filling the target registers with the saved version of their previous contents, such as popping these values from the subject stack **81**.

[0065] In a second example embodiment as also illustrated by FIG. **6**, the reserved location **171** contains special case instructions which do not fall within the instruction set of the subject architecture. Instead, the translator **19** recognises these special case instructions and in response executes a recovery routine to recover the stored target state T' and allow the native code **28** to resume. Here, the recovery routine is suitably provided as part of the translator code **19**.

[0066] In another example embodiment, the program counter PC_S' passed in the subject state S' is a predetermined notional value (such as 0X000000) which does not correspond to a real location in the memory of the target architecture. The translator **19** is configured to recognise this special program counter and, instead of passing control to the identified location, redirects the flow of execution to execute the recovery routine **171** which loads the stored target state T' and allows the native code **28** to resume. This particular embodiment relies on the translator **19** to detect when the subject program counter is a given predetermined value and to take a different action.

[0067] Referring again to FIGS. **5** and **6**, in the example embodiments discussed herein the first subject state S is employed in creating the second subject state S'. Suitably, the first subject state S is saved when the native code **28** is called, and hence the first subject state S is available to populate the second subject state S' when an exception occurs. That is, the old subject state S is copied forward to become the new subject state S', except that the program counter PC_S is modified to the special value PC_S', and/or the stack pointer is updated to the new head of the subject stack **81**. This mechanism provides the subject exception handler **170** with a workable subject state S' sufficient to handle the exception. The subject exception handler **170** may, for example, examine the subject program counter PC_S to determine where within the subject program the exception occurred. The subject handler **170** may then determine how to deal with the exception. The subject exception handler **170** is usually written in a manner which makes some assumptions about the program address space of the subject code program **17** and any dynamically allocated memory. However, the subject exception handler **170** usually cannot assume the memory areas where other components of the system, such as library functions, will reside. Where the subject program counter PC_S' reported to the subject exception handler **170** lies outside the address range allocated to the subject code **17**, the subject exception handler **170** cannot make detailed decisions based on the supposed state of the subject platform and will therefore handle the exception similar to the manner in which it would have been handled on the subject platform.

[0068] FIG. **7** is a schematic diagram of a portion of the memory of the target platform. As shown in FIG. **7**, the subject program **17** is allocated a subject program area **175** in the target memory **18**, alongside other components such as the translator code **19** and the target code **21**. This subject program area **175** typically comprises a linear range of virtual addresses, which map to one or more sets of physical storage locations as will be familiar to persons skilled in the art. The modified program counter PC_S' passed to the subject exception handler **170** suitably lies outside this subject program area **175**. Further, the subject program may have one or more dynamically allocated memory areas **176** such as for working storage, and the modified program counter PC_S' also suitably falls outside the dynamically allocated subject areas **176**.

[0069] FIG. **8** is a schematic flow diagram illustrating a method of exception handling for native bound code as employed by example embodiments of the present invention.

[0070] As discussed above, an exception occurs during execution of native bound code (step **801**). A current execution state is saved (step **802**), reflecting execution of the native bound code on the target platform (target state T'). Also, a subject state (S') is created (step **803**) reflecting an emulated point of execution on the subject platform, as if the exception had occurred whilst executing subject code on the subject platform. In particular, the created subject state S' includes a stack pointer SP_S' to the subject stack, where the target state T' is conveniently stored. Also, the subject state S' includes a specially modified program counter PC_S' as discussed above. The exception is handled with reference to the created subject state S' (step **804**). The subject exception handler **170** will include instructions which determine (step **805**) whether or not to resume execution at the point where

the exception occurred (i.e. return to the previous point of execution, which in this case lies in the native bound code). In some circumstances, execution of the subject program is halted, or control passes to a different portion of the program (step **809**). However, where it is determined to resume execution at the point where the exception occurred, execution is resumed using the created subject state S' (step **806**) which links to the saved target state T' (step **807**) to resume execution of the native bound code (step **808**).

[0071] In summary, the mechanisms and embodiments described herein have many advantages, including that exceptions occurring during native bound code are handled reliably and efficiently. In the example embodiments, storing the target state T' linked by the subject state S' allows execution of bound native code **28** to be resumed after handling an exception. Also, by modifying the subject program counter in the manner described above, the subject exception handler **170** can return control to the native code **28** by directly or indirectly performing the recovery function **171** which loads the stored target state T'. Further, using the subject stack **81** for execution of the native code **28** maintains the subject stack in good order and allows resources to be released efficiently. These and other features and advantages will be apparent to the skilled person from the above description and/or by practicing the described embodiments of the present invention.

[0072] Although a few example embodiments have been shown and described, it will be appreciated by those skilled in the art that various changes and modifications might be made without departing from the scope of the invention, as defined in the appended claims.

[0073] Attention is directed to all papers and documents which are filed concurrently with or previous to this specification in connection with this application and which are open to public inspection with this specification, and the contents of all such papers and documents are incorporated herein by reference.

[0074] All of the features disclosed in this specification (including any accompanying claims, abstract and drawings), and/or all of the steps of any method or process so disclosed, may be combined in any combination, except combinations where at least some of such features and/or steps are mutually exclusive.

[0075] Each feature disclosed in this specification (including any accompanying claims, abstract and drawings) may be replaced by alternative features serving the same, equivalent or similar purpose, unless expressly stated otherwise. Thus, unless expressly stated otherwise, each feature disclosed is one example only of a generic series of equivalent or similar features.

[0076] The invention is not restricted to the details of the foregoing embodiment(s). The invention extends to any novel one, or any novel combination, of the features disclosed in this specification (including any accompanying claims, abstract and drawings), or to any novel one, or any novel combination, of the steps of any method or process so disclosed.

1. A method of handling an exception, comprising the steps of:

(a) performing program code conversion to convert subject code executable by a subject computing architecture into target code executable by a target computing architecture;

(b) executing a portion of native code which is native to the target computing architecture in substitution for translating a portion of the subject code into the target code;

(c) in response to an exception arising during execution of the portion of native code, performing the steps of:

(i) saving a target state which represents a current point of execution in the target computing architecture with respect to the native code portion;

(ii) generating a subject state which represents an emulated point of execution in the subject computing architecture; and

(iii) handling the exception with reference to the subject state; and

(d) resuming execution from the exception by using the subject state to restore the target state, and then using the target state to resume execution of the native code portion.

2. The method of claim **1**, wherein the subject state comprises a link to the target state.

3. The method of claim **1**, wherein the step (d) comprises executing a recovery routine to restore the target state.

4. The method of claim **1**, wherein the subject state includes a modified subject program counter and the method further comprises passing execution control to the modified subject program counter thereby executing a recovery routine to load the target state and return execution control to the portion of native code.

5. The method of claim **4**, wherein the modified subject program counter has a predetermined value and the method further comprises detecting the predetermined value and redirecting execution control to execute the recovery routine.

6. The method of claim **1**, wherein the subject state includes a modified subject program counter, and the method further comprises providing a reserved location to be treated as a portion of the subject code at an address identified by the modified subject program counter.

7. The method of claim **6**, wherein the reserved location contains subject code instructions and the method comprises translating and executing the subject code instructions as target code, thereby to recover the target state and resume execution of the portion of native code.

8. The method of claim **6**, wherein the reserved location contains special case instructions which do not fall within an instruction set architecture of the subject computing architecture, and the method comprises detecting the special case instructions and in response executing a recovery routine to recover the saved target state and resume execution of the portion of native code.

9. The method of claim **6**, comprising using the modified subject program counter as a return address after handling the exception, when attempting to restart execution of the subject code at a point where the exception occurred, such that execution control passes to the reserved location.

10. The method of claim **1**, further comprising the steps of:

converting a caller portion of the subject code into target code and forming a first subject state including at least a subject program counter;

calling the portion of native code from the caller portion of target code; and

copying and modifying the first subject state to form the subject state.

11. The method of claim 1, further comprising providing a subject stack data structure in a memory of in the target computing architecture for emulating execution of the subject code on the subject computing architecture, and executing the portion of native code using the subject stack.

12. The method of claim 1, further comprising providing a subject stack in the target computing architecture for emulating execution of the subject code on the subject computing architecture, and storing the subject state on the subject stack prior to handling the exception whereby the stored subject state is available from the subject stack after handling the exception.

13. The method of claim 1, further comprising providing a subject stack in the target computing architecture for emulating execution of the subject code on the subject computing architecture, and saving the target state on the subject stack prior to handling the exception whereby the target state is available from the subject stack after handling the exception.

14. The method of claim 13, wherein the step of saving the target state comprises saving values of a set of registers of a target processor of the target computing architecture onto the subject stack, and the method further comprises restoring the target state including loading the saved values from the subject stack into the set of registers.

15. The method of claim 1, further comprising handling the exception and determining to resume execution of the subject code at a point where the exception occurred using the subject state, or to resume execution of the subject program at another point in the subject code by altering a subject program counter in the subject state.

16. A method of handling an exception, comprising the steps of:

converting subject code executable by a subject computing platform into target code executable by a target computing platform, including converting a caller portion of subject code into a caller portion of target code;

executing the caller portion of target code thereby generating on the target computing platform a first target state relating to execution of the target code and emulating a first subject state representing execution of the caller portion of subject code on the subject computing platform;

calling from the caller portion of target code to a portion of native code which is native to the target computing platform;

executing the portion of native code on the target computing platform thereby generating a second target state and, where an exception occurs during execution of the portion of native code, saving the second target state and generating a second subject state based on the first subject state wherein the second subject state comprises a link to the second target state; and

handling the exception with reference to the second subject state and using the second subject state to link to the second target state to thereby resume execution in the portion of native 7 code.

17. The method of claim 16, further comprising the step of executing of a recovery routine to restore the second target state when resuming execution from the exception.

18. The method of claim 16, wherein the second subject state comprises a modified subject program counter and the method further comprises passing execution control to the modified subject program counter, loading a recovery routine to load the saved second target state, and thereby returning execution control to the portion of native code.

19. The method of claim 18, wherein the modified subject program counter has a predetermined value and the method further comprises detecting the predetermined value and redirecting execution control to the recovery routine.

20. The method of claim 16, wherein the second subject state comprises a modified subject program counter, and the method further comprises providing a reserved location treated as a portion of the subject code at an address identified by the modified subject program counter.

21. The method of claim 20, wherein the reserved location contains subject code instructions and the method comprises converting and executing the subject code instructions as target code thereby causing recovery of the saved target state whereby the portion of native code resumes execution.

22. The method of claim 20, wherein the reserved location contains special case instructions which do not fall within an instruction set architecture of the subject computing platform, and the method comprises detecting the special case instructions and in response executing a recovery routine to recover the saved second target state thereby enable execution of the portion of native code to resume.

23. The method of claim 20, comprising using the modified subject program counter as a return address after handling the exception, when attempting to restart execution at a point where the exception occurred, whereby execution control passes to the reserved location.

24. The method of claim 16, wherein the first subject state comprises at least a subject program counter, and wherein generating the second subject state comprises copying the first subject state to form the subject state and modifying the subject program counter to a modified value.

25. The method of claim 16, further comprising providing a subject stack data structure in the target computing platform for emulating execution of the subject code on the subject computing platform, and executing the portion of native code with reference to the subject stack.

26. The method of claim 16, further comprising providing a subject stack in the target computing platform for emulating execution of the subject code on the subject computing platform, and storing at least the second subject state on the subject stack prior to handling the exception, such that the stored second subject state is available from the subject stack after handling the exception.

27. The method of claim 16, further comprising providing a subject stack in the target computing platform for emulating execution of the subject code on the subject computing platform, and saving at least the second target state on the subject stack prior to handling the exception, such that the saved second target state is available from the subject stack after handling the exception.

28. The method of claim 27, wherein the step of saving the second target state comprises saving values of a set of target registers of a target processor of the target computing platform onto the subject stack, and the method further comprises restoring the second target state including loading the saved values from the subject stack into the set of target registers.

29. The method of claim 16, further comprising handling the exception and determining to resume execution at a point where the exception occurred using the subject state, or to

resume execution of the subject program at another point in the subject code by altering a subject program counter in the subject state.

30. A target computing platform arranged to handle exceptions during binding to native code, comprising:

a translator unit arranged to translate subject code executable by a subject computing platform into target code executable by the target computing platform, and arranged to substitute a portion of native code which is native to the target computing platform instead of translating a portion of the subject code into the target code;

an execution unit arranged to execute the target code and the native code, and to raise an exception signal when an exception occurs;

an exception handler unit arranged to detect the exception signal raised by the execution unit during execution of the native code, cause the saving of a target state which represents a current point of execution in the execution unit for the native code, provide a subject state which represents an emulated point of execution in the subject computing platform, and handle the exception with reference to the subject state; and

a recovery unit arranged to restore the saved target state by linking from the subject state and thereby resume execution of the native code in the execution unit using the saved target state.

31. A translator apparatus arranged to handle exceptions during binding to native code, comprising:

a translator unit arranged to convert subject code executable by a subject computing platform into target code executable by a target computing platform, and arranged to cause the target computing platform to execute a portion of native code which is native to the target computing platform in substitution for converting a portion of the subject code into the target code, and further wherein the translator unit is arranged to convert caller subject code into caller target code for execution by the target computing platform to provide a first target state and a first subject state and to cause at least the first subject state to be saved when calling into a portion of the native code from the caller target code;

an exception handler unit arranged to detect an exception signal raised during execution of the native code, cause the saving of a second target state which represents a current point of execution in the target computing platform for the native code, and provide a second subject state, wherein the second subject state comprises a link to the second target state;

a subject exception handler unit arranged to handle the exception with reference to the second subject state; and

a recovery unit arranged to cause the target computing platform to resume execution of the native code by linking to the second target state from the second subject state.

32. A computer-readable medium having recorded thereon instructions implementable by a computer to perform a method of handling an exception, comprising the steps of:

(a) performing program code conversion to convert subject code executable by a subject computing architecture into target code executable by a target computing architecture;

(b) executing a portion of native code which is native to the target computing architecture in substitution for translating a portion of the subject code into the target code;

(c) in response to an exception arising during execution of the portion of native code, performing the steps of:

(i) saving a target state which represents a current point of execution in the target computing architecture with respect to the native code portion;

(ii) generating a subject state which represents an emulated point of execution in the subject computing architecture; and

(iii) handling the exception with reference to the subject state; and

(d) resuming execution from the exception by using the subject state to restore the target state, and then using the target state to resume execution of the native code portion.

33. A computer-readable medium having recorded thereon instructions implementable by a computer to perform a method of handling an exception, comprising the steps of:

converting subject code executable by a subject computing platform into target code executable by a target computing platform, including converting a caller portion of subject code into a caller portion of target code;

executing the caller portion of target code thereby generating on the target computing platform a first target state relating to execution of the target code and emulating a first subject state representing execution of the caller portion of subject code on the subject computing platform;

calling from the caller portion of target code to a portion of native code which is native to the target computing platform;

executing the portion of native code on the target computing platform thereby generating a second target state and, where an exception occurs during execution of the portion of native code, saving the second target state and generating a second subject state based on the first subject state wherein the second subject state comprises a link to the second target state; and

handling the exception with reference to the second subject state and using the second subject state to link to the second target state to thereby resume execution in the portion of native code.

* * * * *