

(21) Application No: 0507943.9

(22) Date of Filing: 20.04.2005

(71) Applicant(s):
Transitive Limited
(Incorporated in the United Kingdom)
5th Floor Alder Castle, 10 Noble Street,
LONDON, EC2V 7QJ, United Kingdom

(72) Inventor(s):
Gavin Barraclough
Kit Man Wan
Abdul Rahman Hummaida

(74) Agent and/or Address for Service:
Appleyard Lees
15 Clare Road, HALIFAX, West Yorkshire,
HX1 2HY, United Kingdom

(51) INT CL:
G06F 9/318 (2006.01)

(52) UK CL (Edition X):
G4A AFI AFP APX

(56) Documents Cited:
US 20050005265 A1 **US 20040015627 A1**
US 20020092002 A1

(58) Field of Search:
UK CL (Edition X) **G4A**
INT CL⁷ **G06F**
Other: **Online:WPI,EPODOC**

(54) Abstract Title: **Precise handling of exceptions during program code conversion**

(57) Precise exception handling relies on a precise subject state including an accurate program counter and register values of a subject processor. Subject code (17) is translated into target code (21) executable by a target processor (13). The generated target code (17) includes counterpart target instructions (214) associated with fault-vulnerable subject code instructions (174). Further, each of the counterpart target code instruction (214) is associated with recovery information (195). When an exception (e.g. a fault) occurs, the recovery information (195) is retrieved and used to recover a precise subject state, in particular by taking account of optimizations to generate the common-case target code (21). The precise subject state is then used to precisely handle the exception. Interrupts to the target code execution may be handled by continuing execution until a target instruction, which corresponds to a potentially faulting subject instruction, and using the associated recovery information to handle the interrupt (figures 10 and 11).

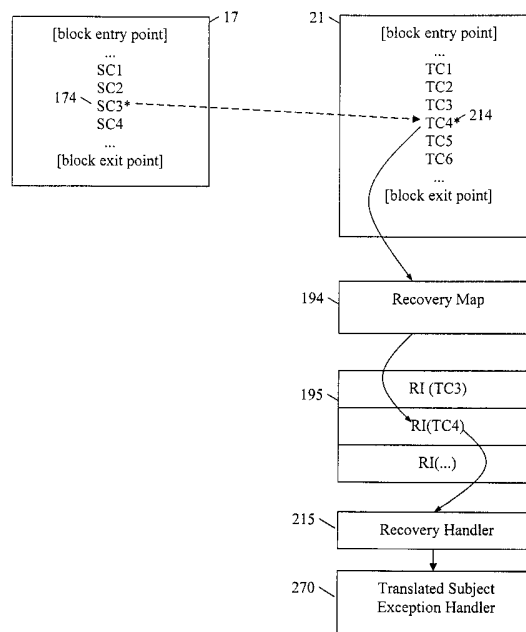
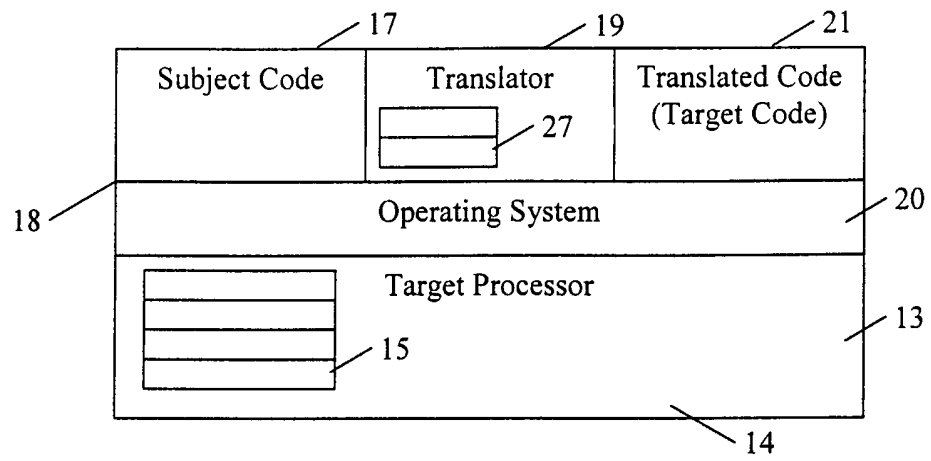
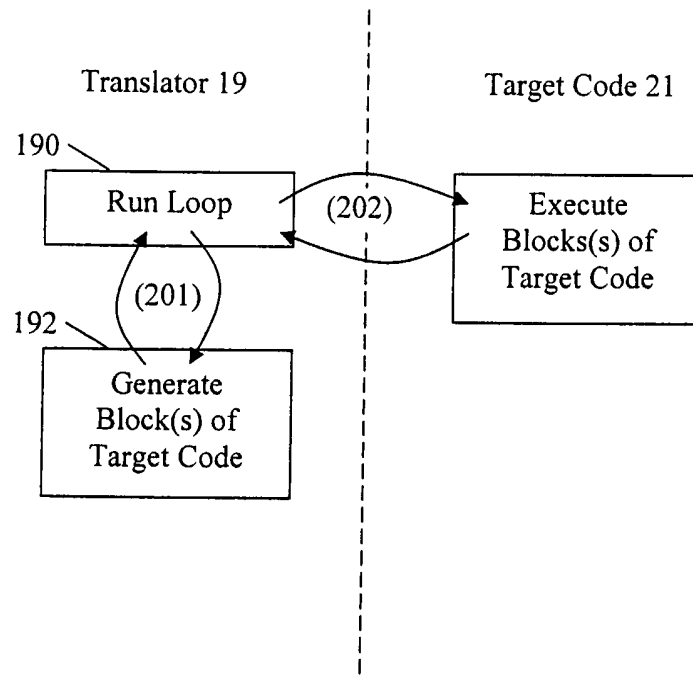
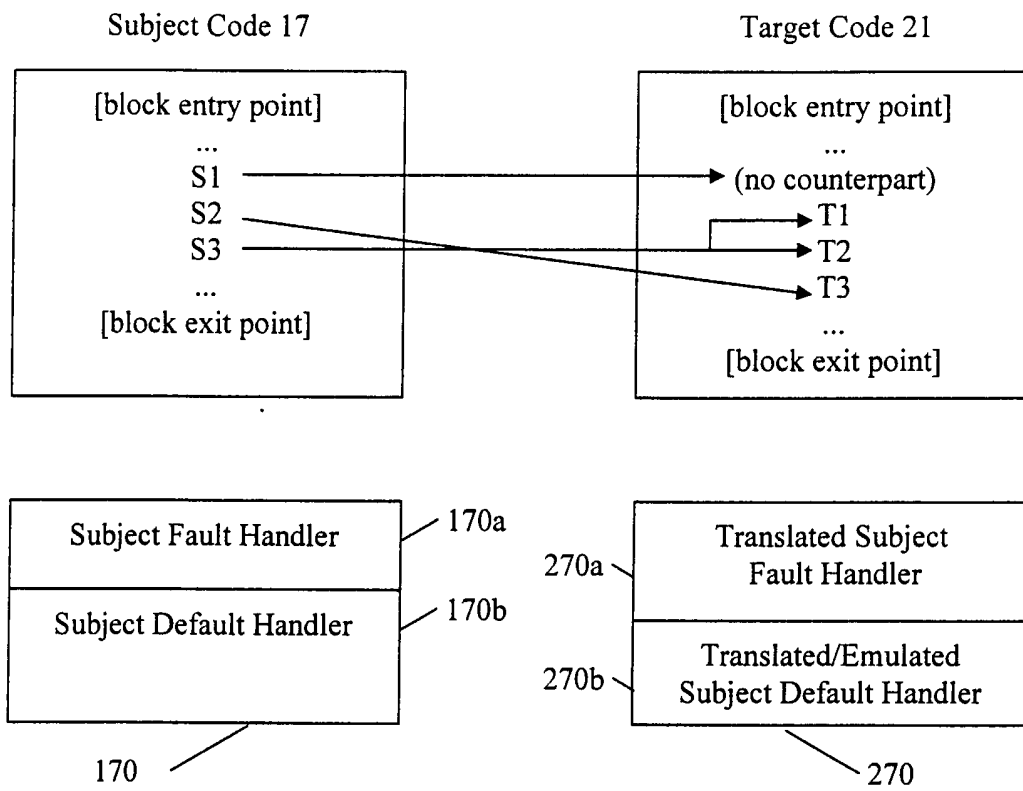
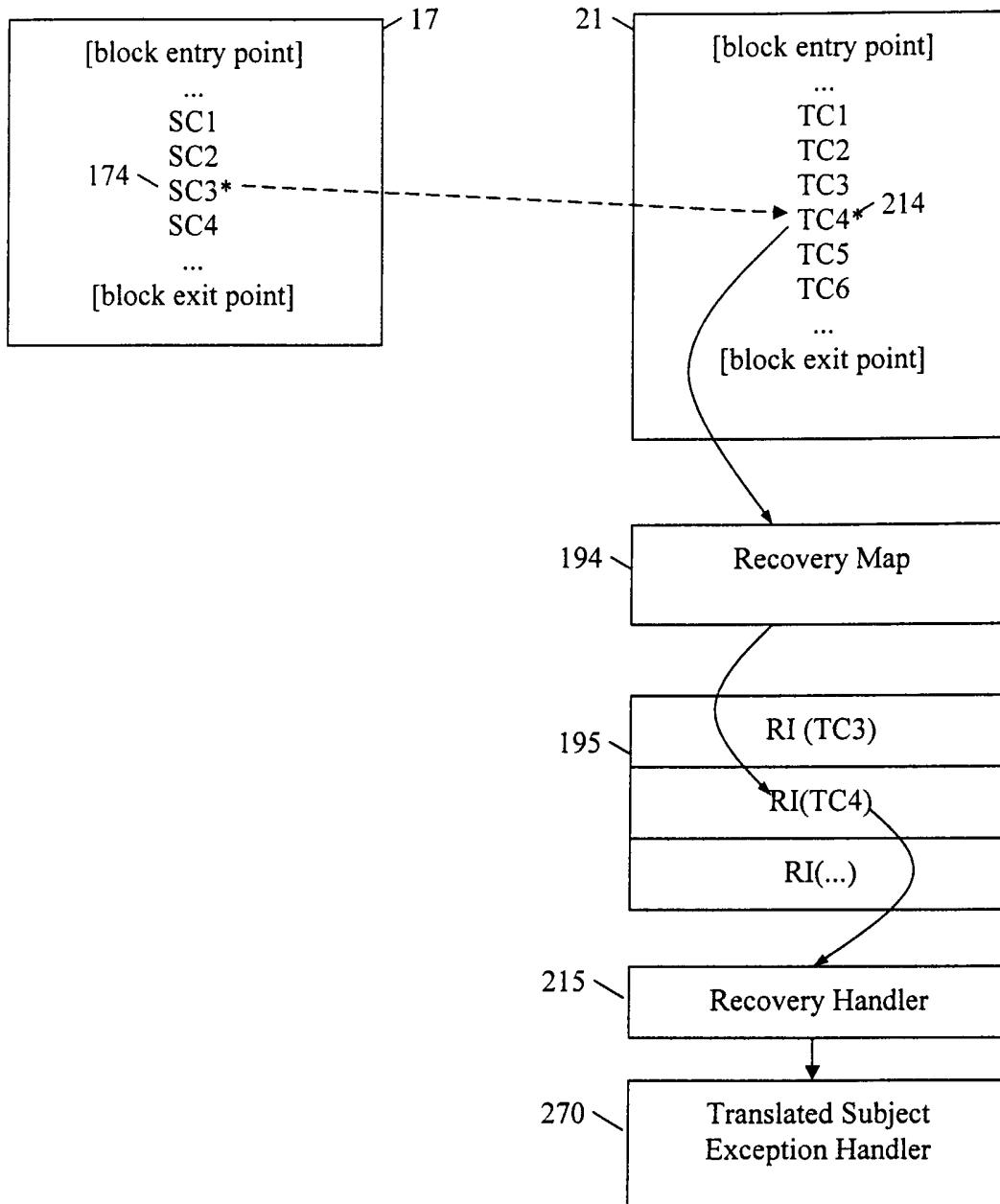


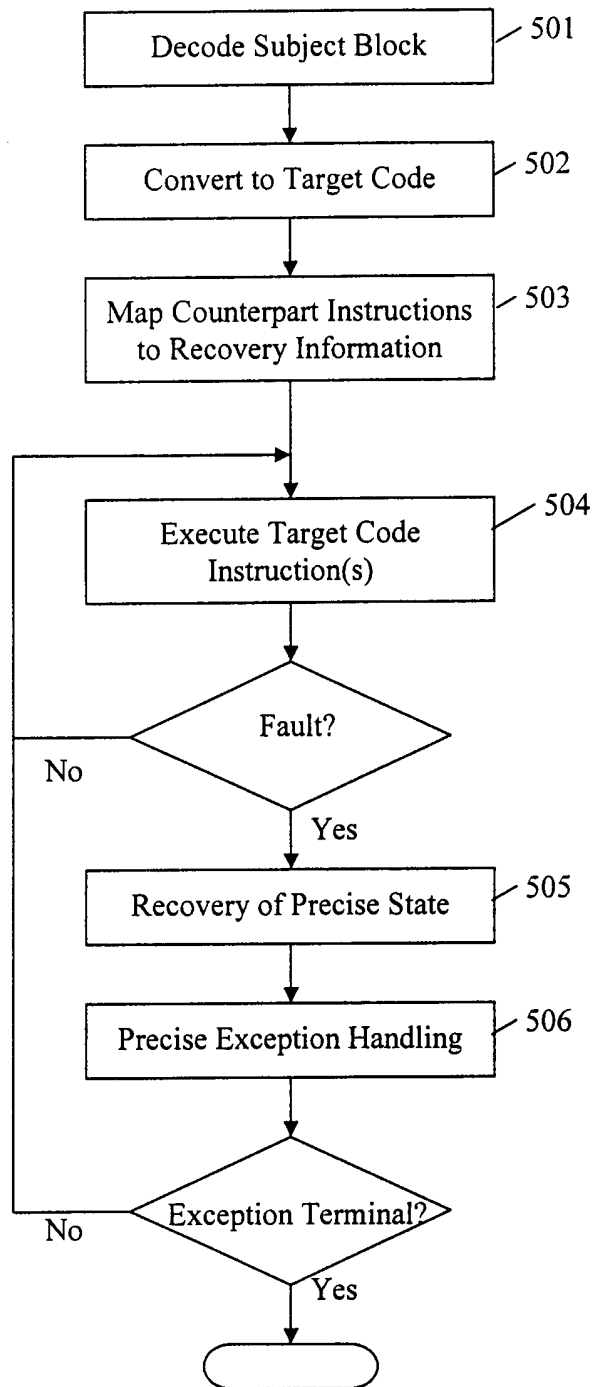
Fig. 4

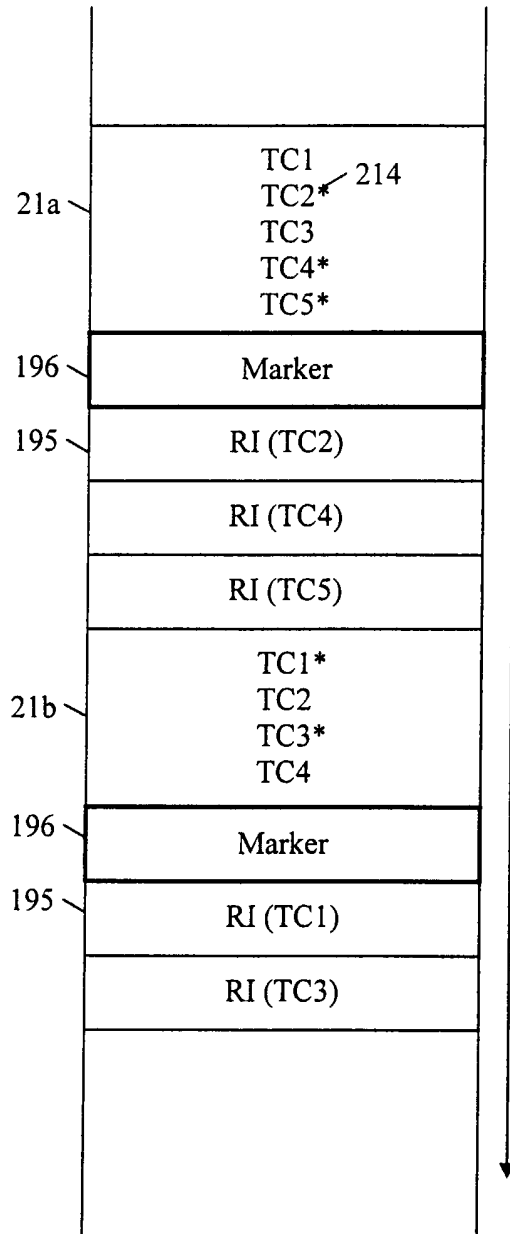
**Fig. 1**

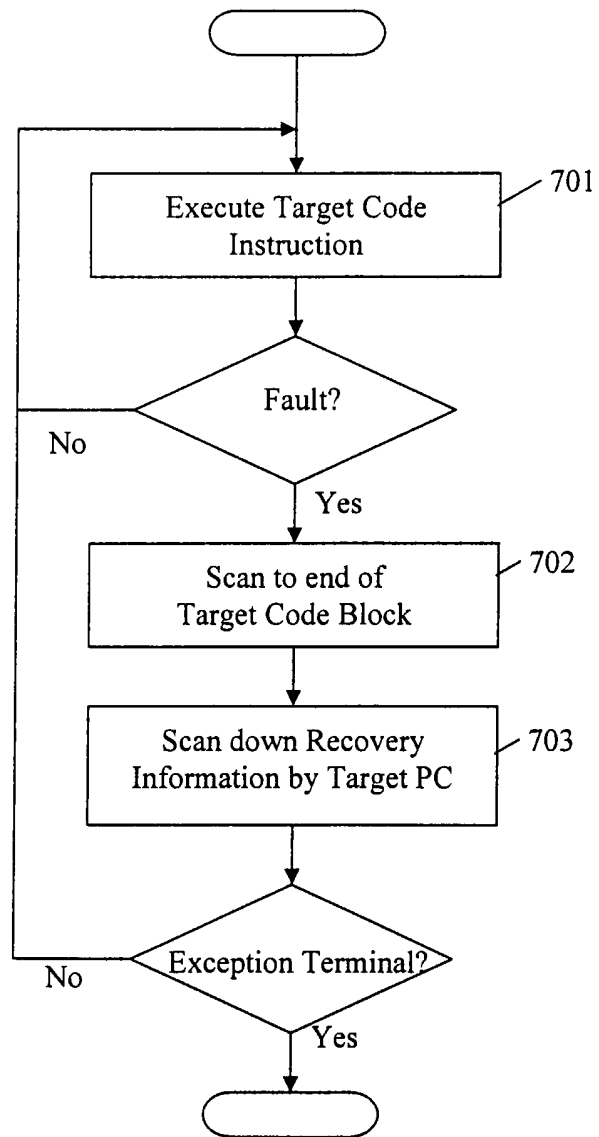
**Fig. 2**

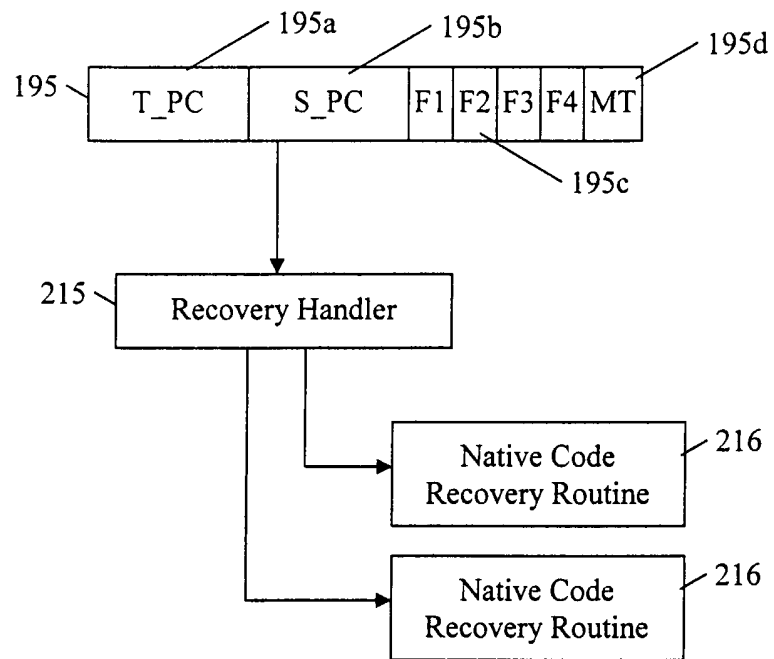
**Fig. 3**

**Fig. 4**

**Fig. 5**

**Fig. 6**

**Fig. 7**

**Fig. 8**

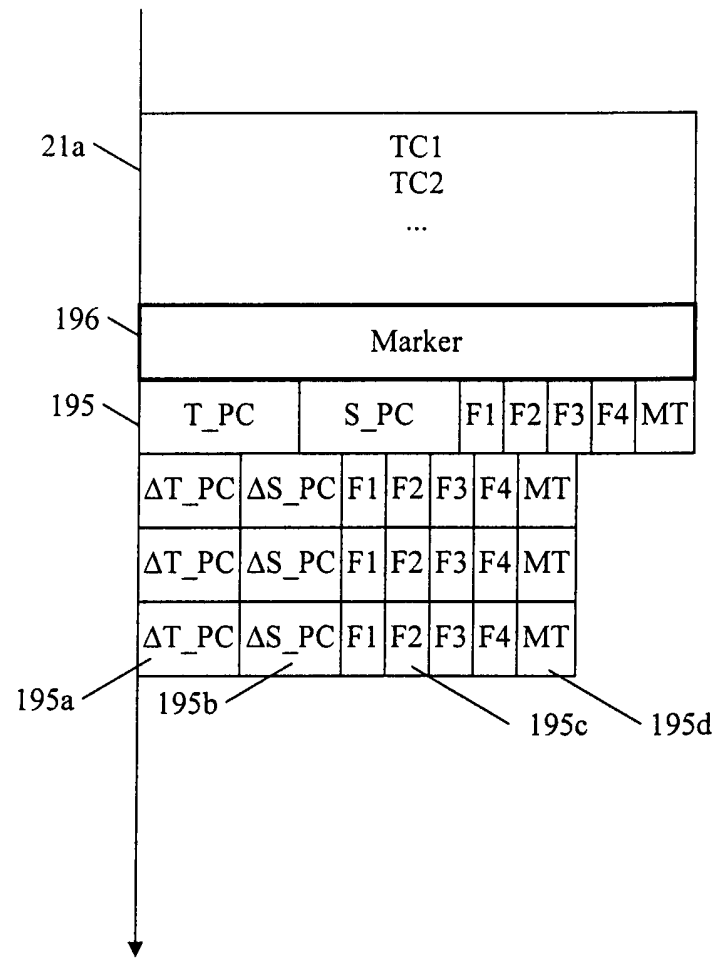


Fig. 9

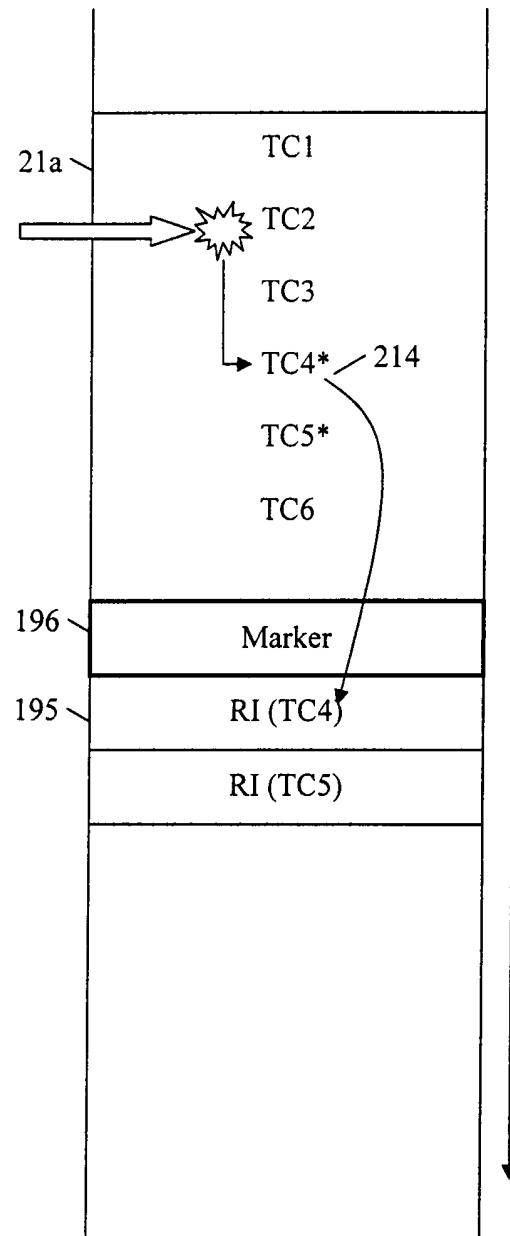
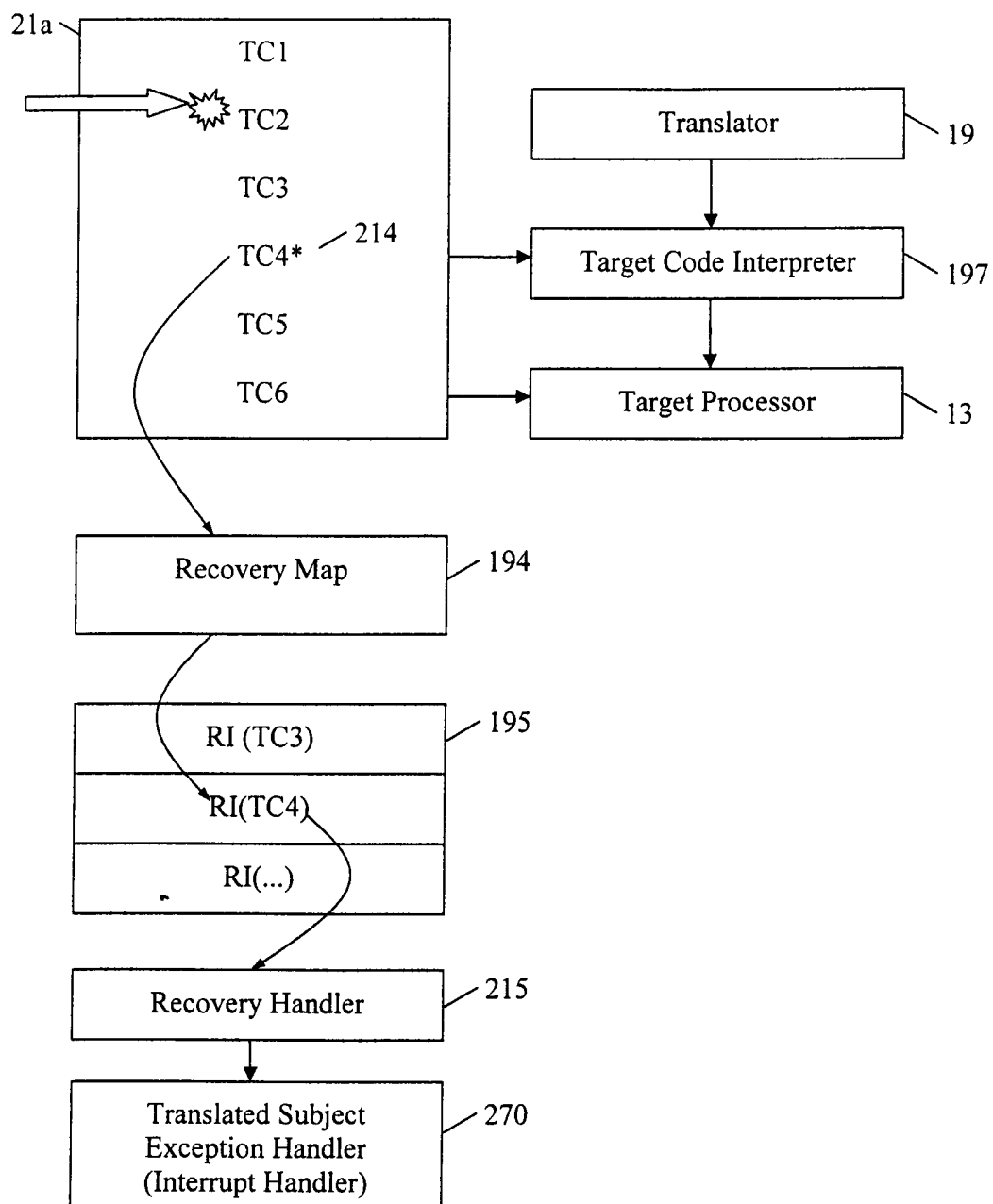


Fig. 10

**Fig. 11**

**METHOD AND APPARATUS FOR
PRECISE HANDLING OF EXCEPTIONS
DURING PROGRAM CODE CONVERSION**

5 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.

10

 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
15 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
20 would benefit from a software-oriented processor co-architecture.

 Program code conversion methods and apparatus to facilitate such acceleration, translation and co-
25 architecture capabilities are disclosed, for example, in published PCT application WO00/22521, and others.

 A problem arises in that performing program code conversion inevitably brings overheads in the conversion
30 process, compared with native execution of the subject program on a subject processor. A particular problem arises in relation to the handling of exceptions.

An exception is a condition that changes the normal flow of control in a program. An exception indicates that a condition has occurred somewhere within the system that requires the attention of the processor and usually needs
5 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
10 particular architectures.

An exception may be generated ("raised") by hardware or by software. Hardware exceptions include such signals as resets, interrupts, or signals from a memory management
15 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.

20

Software exceptions are varied respectively 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
25 the subject code causes an exception to be reported if the value of one register is greater than the value of a second register.

An exception handler is special code which is called
30 upon when an exception occurs during the execution of a program. If the subject program does not provide a handler for a given exception, a default system exception handler will be called, usually resulting in abortion of

the program being run and an error indication being returned.

Exception signals are a common mechanism for raising
5 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
10 events that trigger exceptions are when a process
implemented by a program tries to (i) access an unmapped
memory region or (ii) manipulate a memory region for which
it does not have the correct permissions. Other common
events that trigger exception signals are (iii) the
15 receipt of a signal sent from another process, (iv) the
execution by a process of an instruction that the process
does not have the privilege level to execute, or (v) an
I/O event in the hardware.

20 The present invention is particularly concerned with
the precise handling of exceptions, also often referred to
as precise exceptions. A precise exception is reported on
a boundary between subject code instructions. When a
precise exception is reported, the effects of all previous
25 subject instructions are complete, the exception points to
the first instruction of the subject code which has not
been executed, and no effects from that subject
instruction or any subsequent instruction have yet taken
place.

30

In the context of binary translation, it is apparent
that when a target instruction is performed on a target
processor and causes an exception to be reported, the

target instruction generally will not fulfil the conditions for reporting a precise exception to an exception handler written in subject code. Instructions are almost always performed on the target processor in a different order to the order of instructions in the corresponding block of subject code, firstly due to the differences between the instruction set of the subject processor for which the subject code was written and the target processor on which the target code is run, and secondly because of the optimisations that typically occur during program code conversion.

One known approach is the use of "recovery points", which return a virtual subject machine to the conditions that applied at entry into a section of code being translated or executed, i.e. by returning the virtual subject machine to the condition prevailing at the point of entry into the current block of subject code instructions being translated or executed.

20

US 5832205 (Kelly et al) discloses an emulator which uses a set of "working" registers which are then copied to a set of "official" virtual subject registers at the end of each section of subject code, using a gated store buffer. If an exception occurs, this will affect only the working registers and the condition of the virtual subject machine can be recovered from the "official" registers at the point of entry into that section of subject code. However, the use of "working" and "official" registers adds significantly to the overhead of the emulation process in the target processor.

30

As another example, WO00/65440 discloses alternating between A and B register sets to provide historical recovery points during program code conversion.

5 Another approach is discussed in detail in co-pending published application GB2403563 and co-pending application US 10/726,857. Here, a variable precision exception handling is applied, by recording different levels of detail for each of a plurality of types of instructions in
10 the subject program. That is, the quantity and type of information to be recorded is varied according to the nature of the subject instruction. Suitably, each subject instruction is determined to be one of four types, and a corresponding level of detail is recorded. A first level
15 records no state at all. A second level provides imprecise (inaccurate) state information using a last known stack frame. A third level records a precise subject program counter in a temporary abstract register during translation, which is then retrieved to handle an
20 exception. A fourth level generates and plants additional target code to rectify a precise state (i.e. record the subject program counter and rectify any lazy or pending subject register values) immediately preceding an exception-raising target instruction (particularly a trap-
25 type instruction which triggers an exception).

These prior art approaches rely on recording information which allows a subject processor state to be accurately determined at the point when an exception
30 occurs, by rolling back to a previously defined recovery point or by rectifying subject state in advance of that point. However, in the prior art, obtaining a precise state for precise exception handling is still restrictive

on the program code conversion process and involves considerable overhead and/or specific hardware support.

An aim of the present invention is to improve
5 performance when undertaking program code conversion.

A preferred aim of the present invention is to reduce overhead associated with handling of exceptions during program code conversion, and most particularly during
10 dynamic binary translation.

Another preferred aim of the embodiments of the present invention is to reduce overhead associated with providing a precise subject state for precise handling of
15 exceptions.

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
20 apparent from the dependent claims, and the description which follows.

The following is a summary of various aspects and advantages realizable according to embodiments of the
25 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.

30

In particular, the inventors have developed methods directed at expediting program code conversion, particularly useful in connection with a run-time

translator which provides dynamic binary translation of subject program code into target code.

In one aspect of the present invention there is provided a method of precisely handling exceptions in the context of program code conversion from subject code to target code. The method includes generating target code including one or more counterpart target instructions from a fault-vulnerable subject code instruction. Each of the counterpart target code instructions is associated with recovery information. After an exception (e.g. a fault) occurs during execution of the target code instructions, the recovery information is retrieved and used to rectify a precise subject state.

15

Preferably, the recovery information takes account of optimisations performed during target code generation. Suitably, the recovery information is used to perform recovery routines which rectify a precise subject state representing a subject processor at the point of a fault-vulnerable subject code instruction.

Preferably, the method includes storing one or more recovery information items immediately following the generated target code. Retrieving the recovery information includes scanning down the target code to locate the recovery information subsequent thereto, and retrieving a recovery information item associated with a respective faulting counterpart target instruction.

30

Preferably, the target code and recovery information are interleaved in blocks.

In a particularly preferred embodiment, a target program counter is used to index the recovery information.

In a second aspect of the present invention there is
5 provided a method of precisely handling exceptions with
program code conversion from subject code to target code,
including the steps of generating target code including at
least one counterpart target instruction which corresponds
to a potentially faulting subject instruction in subject
10 code, and storing recovery information associated with the
at least one counterpart target instruction. Upon
receiving an interrupt signal, execution of the target
code instructions continues until one of the counterpart
target instructions is encountered. Upon reaching the
15 counterpart target instruction, the associated recovery
information is retrieved and used to recover a precise
subject state.

In a preferred embodiment, the precise exception
20 handling method rolls forward from the point where an
exception occurs, until a precise subject state is
recoverable using the stored recovery information.
Suitably, execution of the target code instructions
continues one by one, preferably by invoking a target code
25 interpreter or invoking a stepwise target processor mode.

The present invention also extends to a translator
apparatus arranged to perform any of the methods defined
herein. Also, the present invention extends to computer-
30 readable storage medium having recorded thereon
instructions implementable by a computer to perform any of
the methods defined herein.

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

5

Figure 1 is a block diagram illustrative of apparatus wherein embodiments of the invention find application;

Figure 2 is a schematic flow diagram illustrating a preferred method of execution control during program code conversion;

Figure 3 is a schematic diagram to show a relationship between subject instruction and target instructions following program code conversion;

Figure 4 shows an example program code structure in a target machine during program code conversion according to a preferred embodiment of the present invention;

20

Figure 5 is a schematic overview of a method to create target code for precise exceptions, according to a preferred embodiment of the present invention;

Figure 6 is a schematic diagram showing a preferred mechanism to store recovery information and generated target code;

Figure 7 is a schematic diagram of a preferred method of obtaining recovery information;

30

Figure 8 is a schematic representation of recovery information used in a preferred embodiment of the present invention;

5 Figure 9 is a schematic representation of recovery information used in another preferred embodiment of the present invention;

Figure 10 shows an example target code sequence to
10 illustrate handling of interrupts; and

Figure 11 is a schematic view showing a preferred implementation to handle interrupts within a target computing platform.

15

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
20 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.

25

In the terminology below, a subject program is intended to execute on a subject computing platform including a subject processor. A target computing platform including a target processor is used to execute the
30 subject program, through a translator which performs dynamic program code conversion. The translator performs code conversion from subject code to target code, such

that the target code is executable on the target computing platform.

Figure 1 illustrates an example target computing platform comprising a target processor 13 having a plurality of target registers 15, and a memory 18 to store a plurality of software components 17, 19, 20, 21, and 27. The software components include an operating system 20, subject code 17, translator code 19, and translated target code 21.

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.

20

The translator 19, i.e., the compiled version of source code implementing the translator, and the translated code 21, i.e., the translation of the subject code 17 produced by the translator 19, run in conjunction with the operating system 20 running on the target processor 13, which is typically a microprocessor or other suitable computer.

It will be appreciated that the structure illustrated in Figure 1 is exemplary only and that, for example, software, methods and processes according to the invention may be implemented in code residing within or beneath an operating system. 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.

5 In the apparatus according to Figure 1, program code conversion is preferably performed dynamically, at run-time, while the target code 21 is running. The translator 19 runs inline with the translated program 21. The translator 19 is preferably employed as an application
10 compiled for the target architecture. The subject program 17 is translated by the translator 19 at run-time to execute on the target architecture 14.

 Running the subject program 17 through the translator
15 19 involves two different types of code that execute in an interleaved manner: the translator code 19; and the target code 21. The translator code 19 is generated such as by a compiler, prior to run-time, based on a high-level source code implementation of the translator 19. By
20 contrast, 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.

 The subject program 17 is intended to run on a subject
25 processor (not shown). In one embodiment, the translator 19 functions as an emulator. That is, the translator 19 emulates the subject processor, whilst actually executing the subject program 17 as target code 21 on the target processor 13. In the preferred embodiment, at least one
30 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 processor state is provided by components of the translator 19 and the target code 21. That is, the translator 19 stores the subject processor state in a variety of explicit programming language devices such as variables and/or objects. The compiler used to compile the translator 19 determines how the state and operations are implemented in target code. The 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, the 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.

Figure 2 is a schematic flow diagram illustrating a preferred method of execution control during program code conversion.

As shown in Figure 2, control initially resides with a translator control loop 190. In step 201, the control loop 190 calls a code generation function 192 of the translator code 19, which translates a block of the subject code 17 into a corresponding block of translated code 21. Then, in step 202, that block of translated code 21 is executed on the target processor 13. Conveniently, the end of each block of translated code 21 contains instructions to return control back to the control loop 201. In other words, the steps of translating and executing the subject

code are interlaced, such that portions of the subject program 17 are translated and then executed in turn.

Here, 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 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.

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 (DAGs), but are referred to colloquially as "trees".

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
5 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.

10

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
15 to run. For example, there is a unique abstract register definition for each physical register on the subject architecture ("subject register"). As such, abstract register definitions in the translator may be implemented as a C++ object which contains a reference to an IR node
20 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
25 suitably form part of the translator code generation function 192.

Exception Handling

30 Figure 3 is a schematic diagram to show a relationship between instructions in a subject program and instructions in a target program, following program code conversion in preferred embodiments of the present invention.

In this example, subject instructions S1-S3 result in functionally equivalent target instructions T1-T3. The subject instruction S1 has been removed such as by a dead code elimination optimisation and has no counterpart in the generated target code. Subject instruction S2 results in one equivalent target instruction T3. By contrast, subject instruction S3 results in two target instructions T1 & T2. There may be a one to none, one to one, one to many or many to one relationship between the target and subject code instructions.

As also shown in Figure 3, another commonly used optimisation is to perform code rescheduling, whereby an instruction sequence in the target code is not equivalent to the original sequence in the subject code. Here, the second subject instruction S2 has been rescheduled as the third target instruction T3.

Figure 3 also shows a subject exception handler 170. Typically, the subject environment would provide one or more specific subject fault handlers 170a (i.e. specific to a particular type of exception) and/or one or more default exception handlers 170b (to be employed where no specific exception handler is registered).

In this example, any of the subject instructions S1, S2 or S3 potentially results in an exception, which needs to be handled by subject exception handler 170.

A translated subject exception handler 270 provides an exception handler in target code to execute on the target processor, which emulates the subject exception handler

170. In practice, one or more translated subject specific
exception handlers 270a or default exception handlers 270b
are provided. Once the exception has been handled, the
exception handler 270 will usually return control to the
5 target code 21.

The translated subject exception handler 270 expects
to receive a precise subject state in order to precisely
handle the exception. As noted above, recreating the
10 precise subject state is both difficult and expensive.
First, there is an actual cost associated with calculating
and collecting the subject state. For example, translator
optimizations such as lazy evaluation may postpone the
calculation of subject register values by storing the
15 underlying data necessary to calculate those values.
Recreating the subject state in response to an exception
requires those values to be rectified (i.e., calculated)
immediately. Even if the subject registers have been
calculated previously, they must be retrieved from memory,
20 such as from the subject register bank 27.

Second, there is an opportunity cost associated with
the capability of calculating a precise subject state at
an arbitrary point in the subject program. Many key
25 optimizations in a dynamic binary translator, such as code
rescheduling, involve departures from a strict model of
binary compatibility. Binary compatibility means that the
translator can recreate the exact state of the subject
architecture. A strict model is one in which the subject
30 state can be recreated at any point in the translated
program (i.e., at any subject instruction). In order to
preserve the information necessary to recreate the subject
state at any point in execution, the translator must

typically forego significant optimizations. When those optimizations are in use, the translator has no way to recreate the subject context accurately. Thus, the real cost of precise exception handling is not just the work of
5 generating the precise subject state. There are restrictions to the code conversion process in order to be able to generate the subject state at all.

Faults

10

It is helpful to illustrate the present invention by concentrating firstly on a particular category of exceptions commonly known as faults.

15 Faults are typically used for error handling. Many operations may, in exceptional circumstances, be unable to produce a correct result. For example, an integer add instruction may produce a result which is too large to be represented (assuming that the size of the result matches
20 that of the inputs). Processor architecture designers decide how such a fault will be flagged, if at all. Three typical design options are: to silently ignore the overflow, to flag the error by modifying internal processor state (e.g. status flags) but continue execution
25 with the following instruction, or to raise an overflow exception. A particular architecture may offer a mix of these options, by offering faulting and non-faulting versions of a particular instruction.

30 As other examples, some common classes of faults are: arithmetic exceptions, various forms of memory fault (invalid addresses, protection violations, and alignment

checks), and malformed or illegal (at current privilege level) instruction faults.

Faults are typically synchronous, coerced exceptions, since they occur due to the execution of a given instruction, in the context of the operand data, and processor state (including memory allocation, etc). Faults occur within instructions, which is to say that the faulting instruction cannot be completed. Typically a subject processor will resolve the register state to that at the start of the faulting instruction, allowing a fault handler to attempt to resolve the error, and for execution to be resumed. In certain circumstances it may not be possible to resolve such a state, which in turn can cause the fault to be terminal.

An example of a recoverable fault (typically at OS or sub-OS level) is a page fault - when a memory access attempts to access a page in virtual memory that is not present, an exception is raised. If the page is mapped, the virtual memory system will allocate a physical page, prepare it according to the mapping (e.g. load data if a file has been mapped at the address), and then update the page table structures accordingly. The processor may then resume execution at the same instruction, which now should be able to complete correctly.

Some faults may be maskable from user-mode applications - for example, alignment-checks are commonly available as a maskable fault on architectures that support misaligned accesses; on those that do not the alignment fault is non-maskable.

Some instructions in a subject instruction set architecture are known to be vulnerable to producing faults, whilst other instructions are not. That is, most instruction sets include some fault-vulnerable instructions (also termed faulting instructions or potentially faulting instructions), together with some fault invulnerable instructions which do not fault.

Faults are problematic since they are synchronous and are usually recoverable. Further, fault-vulnerable instructions tend to be executed relatively frequently in any real-life subject program. Hence, an improvement in the handling of faults has an immediate benefit in program code conversion and particularly in dynamic binary translation.

Fault Handling

Figure 4 shows an example program code structure in a target machine during program code conversion according to a preferred embodiment of the present invention.

In Figure 4, an example block of subject code 17 comprises subject instructions SC1 to SC4. One of these instructions (SC3*) is a fault-vulnerable instruction 174, whilst the remainder are fault-invulnerable.

Also, a corresponding block of target code 21 is shown with equivalent target code instructions TC1 to TC6. The fault-vulnerable subject instruction 174 (SC3) relates to one or more counterpart target instructions 214 in the generated target code 21 (in this case TC4*).

As shown in Figure 4, a recovery map 194 is provided which links each of the counterpart target instructions 214 to a respective recovery information item 195. That is, each fault-vulnerable subject instruction 174 has one
5 or more counterpart target code instructions 214, each of which is mapped to a respective target specific recovery information item 195.

The recovery information 195 allows precise handling
10 of exceptions (especially faults) which occur during execution of the target code 21. In particular, the recovery information 195 allows additional native code instructions to be performed in order to create the precise subject state for precise handling of exceptions
15 in the translated subject exception handler 270.

In one preferred embodiment, a recovery handler 215 is provided which uses the recovery information 195 to rectify the precise subject state, ready for the exception
20 to be handled by the exception handler 270.

Figure 5 is a schematic overview of a method to create the target code 21 capable of precise handling of fault-type exceptions, according to a preferred embodiment of
25 the present invention.

In step 501, a section of subject code 17 is decoded, suitably a basic block. During decoding, one or more fault-vulnerable instructions 174 are identified according
30 to the definitions of the subject ISA.

In step 502, program code conversion is performed to convert the subject code 17 into a corresponding block of

target code 21. Optionally, the program code conversion includes optimisations such as dead code removal and/or code rescheduling, amongst others. The generated block of target code 21 includes one or more counterpart target instructions 214 for each of the fault-vulnerable subject instructions 174.

In step 503, recovery information 195 is provided for each of the counterpart target instructions 214. Suitably, entries are created in a recovery mapping table 194 linking each counterpart target instruction 214 to the respective recovery information 195.

In step 504, instructions in the generated block of target code 21 are executed. If no exceptions (faults) occur, then execution continues until the end of the block and then control flows to call the next block, such as through the translator run loop of Figure 2.

In step 505, when an exception (a fault) occurs in relation to one of the counterpart target instructions 214, the respective recovery information 195 is obtained and used to rectify a precise subject state. That is, the recovery information 195 is employed to execute a set of additional recovery code instructions (ideally in the recovery handler 215).

The recovery handler 215 is suitably written in native code which is natively executable by the target processor (as opposed to being translated target code dynamically generated by the translator 19).

At step 506, the exception is precisely handled by the exception handler 270. That is, the recovery at step 505 provides a precise subject state to meet the conditions of precise exception handling by the exception handler 270.

5 Usually, the exception is handled successfully and control returns to the executing program. Typically, control returns to complete execution of the current block. In other cases the fault is terminal and the program ends.

10 **Storing Recovery Information**

Figure 6 shows a preferred mechanism to store recovery information 195 and generated target code 21 on the target computing platform.

15

The recovery mechanism discussed above itself brings some cost to the translator 19. Firstly, memory space is required to store the recovery information 195. Further, space is required to store the recovery map 194 which
20 links each counterpart target code instruction 214 to the respective recovery information 195. When an exception (fault) occurs, it is desired to obtain the relevant recovery information 195 quickly and efficiently.

25 Figure 6 shows an area of memory in the target computing platform which stores a generated block of target code 21a. The target code block 21a includes one or more counterpart target instructions 214 which correspond to fault vulnerable instructions 174 in the
30 subject code.

The recovery information 195 is stored in memory immediately following the respective target code block

21a. That is, each of the counterpart target instructions 214 has a respective recovery information item 195, which is stored immediately after the relevant target code block 21a.

5

Preferably, a marker 196 is provided in order to readily distinguish between the end of the target code block 21a and the start of the recovery information 195.

10 Further, as shown in Figure 6, another translated target code block 21b is stored in memory, again followed by the respective recovery information 195 and a marker 196. That is, the target code 21 and the recovery information 195 are interleaved in memory.

15

When an exception (a fault) occurs in a particular counterpart target instruction 214, the following steps are performed in order to obtain the respective recovery information.

20

The first step is to scan down to the end of the target code block. Conveniently, this step comprises reading successive instructions until the marker 196 is identified. Secondly, scanning down the list of recovery
25 information 195 identifies the respective desired recovery information.

In a particularly preferred embodiment, each recovery information 195 is indexed according to the program
30 counter of a counterpart target instruction 214. That is, the program counter value of the counterpart target code instruction 214 forms an index in the list of recovery information 195. When an exception occurs, the target

program counter is readily available in the target processor. Scanning the memory firstly for the marker 196 and then secondly for the target program counter in the list of recovery information efficiently locates the
5 respective recovery information 195.

Figure 7 shows a preferred method of obtaining recovery information using the memory layout of Figure 6.

10 In practical embodiments, each basic block typically contains around twelve instructions. Hence, it is relatively fast to scan to the end of the current target code block since the distances involved are relatively short. In a preferred embodiment, each subject block is
15 limited to a maximum size of say 100 or 200 instructions. Even in these rare cases of relatively long target blocks, scanning is still an effective and cheap mechanism.

Figure 8 shows a particularly preferred embodiment of
20 the recovery information 195.

In the example of Figure 8, the recovery information 195 includes a target program counter 195a, a subject program counter 195b, a set of recovery indicators 195c
25 and a mapping template 195d.

The recovery information 195 allows the recovery handler 215 to rectify a precise subject state. In particular, the precise subject state comprises a subject
30 program counter value and subject register values.

Suitably, the recovery handler 215 uses the recovery information to rectify the subject program counter (if it

is not already available). In this example, the recovery information 195 stores an accurate subject PC 195b for each target PC 195a. The subject PC is rectified by reading this field 195b in the stored recovery
 5 information.

The mapping template 195d identifies an appropriate register mapping applicable at the point of the counterpart target instruction 214. The recovery handler
 10 215 rectifies the subject processor register values. In one embodiment, the register values are rectified to the abstract register bank 27 in memory. Alternatively, memory stores are left in the target registers. Here, the recovery handler 215 uses the stored mapping template 195d
 15 to set a register mapping whereby subject registers are mapped to physical target registers.

As shown in Figure 8, the stored recovery information 195 preferably includes a set of recovery indicators
 20 (optimisation flags) 195c which denote optimizations which are outstanding at the point of a particular target code instruction. In this example there are four recovery indicators set as flags F1, F2, F3 and F4. Suitably, the recovery handler 215 calls one or more native code
 25 recovery routines 216 to perform the outstanding optimizations, based upon the recovery indicators F1-F4.

In particular, the recovery handler performs lazy work which had been deferred at the point when the exception
 30 occurred. For example, many processors provide a set of condition code flags. A preferred optimization is to delay rectification of the condition code flags by instead storing underlying information which allows the correct

state of the condition code flag to be rectified when needed. The recovery routine 216 performs the lazy condition code flag evaluation from the stored underlying information.

5

Figure 9 shows another particularly preferred embodiment of the stored recovery information.

The recovery information 195 is ideally stored in a compressed form. Advantageously, a memory footprint of the recovery information is reduced. In some environments, the translator 19 operates on a computing platform along with many other processes, and it is desirable to reduce the amount of memory which is consumed by the overhead of the translator 19.

As shown in Figure 9, a first Target PC 195a and a first subject PC 195b are stored as full values. Subsequent PCs are each stored as offsets from these base values. Suitably, the first recovery information 195 in each block of target code 21a, 21b is stored with full PC values as a base value, and each subsequent recovery information 195 stores a program counter value as an offset from the base value. Alternatively, a base PC is defined for the block, such as the PCs for first target code instruction in the block, whether or not this is a counterpart target instruction 214.

Example 1: PPC to x86 Translation

30

An example sequence of subject code 17 is shown below, as an illustrative example of the presently described

techniques. The example subject code 17 is appropriate to a PowerPC (PPC) type processor:

Subject code (PPC):

5

```
0x0001003c: ...
0x00010040: add. r3, r3, 68
0x00010044: ld r3, 0(r3)
0x00010048: ...
```

10

The example PPC subject code adds a constant value 68 to register "r3", and then loads from this address into r3. The addition is performed with a PPC dotted add instruction, which sets flags in field 0 of the PPC condition field register. Should the load fault then, for precise handling of the exception, the program counter value of the load instruction (0x00010044) should be reported, r3 should contain the value after the add operation has been performed, and the condition field 0 should have been updated.

20

After program code conversion, the example target code 21 below is produced. In this example the target code is executable by an x86 type processor:

25

Target code (x86):

```
0x40083200: mov 12(%ebp), %eax
0x40083203: add $68, %eax
0x40083206: mov (%eax), %ecx
0x40083208: mov %ecx, 12(%ebp)
```

30

The target code loads the value for subject register r3 (found at offset 12 from the ebp register, which here points to the abstract register bank 27), into register eax. The target code now adds 68 to eax, and then loads
 5 from this address into ecx. Finally this result is stored back to the subject register r3 in the abstract register bank.

In this example, an exemplary instance of the recovery
 10 information 195 is:

```

0x40083206:
  subj_addr=0x00010044,
  flags=cr0_lazy1_cmp0,
15   register_map=[lazy1(%eax), r3(%eax)].

```

The target PC (0x40083206) is used to index the recovery information. In this case, the target mov instruction is the counterpart target instruction to the
 20 potentially faulting subject load instruction. The accurate subject PC is recorded (0x00010044). A flag is used to show that condition register field 0 needs to be rectified based on a comparison to zero of the value in a special subject register, called 'lazy1'. Also, recovery
 25 information indicates that the value for subject registers lazy1 and r3 are in the target register eax (the value produced by the add target instruction 0x40083203). In order to recover, the value from eax will be spilled to the appropriate subject registers, the condition field can
 30 now be rectified by performing the appropriate comparison using the value now spilled to lazy1, and the appropriate PC can be reported.

In summary, the recovery information allows work to be performed only when needed. That is, work is performed only in the exceptional case, when an exception occurs. Most of the time, in normal execution, a precise subject
5 state is not required and is not rectified. The recovery mechanism discussed herein allows work to be deferred, or not performed at all, during normal execution of the generated target code. As a result, normal execution of the target code becomes faster and more efficient.
10 However, when an exception occurs, the recovery information 195 still allows a precise subject state to be rectified, and for the exception to be handled precisely. The recovery work is performed after the exception has occurred, in order to rectify the precise subject state.

15

Interrupts

A further preferred aspect of the present information will now be described in more detail, relating to the
20 precise handling of interrupts.

Interrupts are typically used for hardware device support. When an event occurs in a hardware device external to the processor, a signal is sent to inform the
25 processor of the event. An example is the availability of data from an I/O device. The external device interrupts the current flow of instructions through the processor, in order to invoke a routine to service the interrupt. The use of interrupts alleviates the need for the system to
30 continually poll devices to see if they are in need of servicing. Another example is timer interrupts which allow user applications to be interrupted at predetermined intervals, with control typically transferred to the

operating system. Regular interrupts are a convenient mechanism to enforce the sharing of processor resource in a multi-tasking system.

5 Interrupts are asynchronous exceptions, in that they are instigated in reaction to an event in an external device, rather than the execution of a particular instruction within the processor. As such, interrupts may occur at an arbitrary point in an instruction stream. An
10 interrupt is unrelated to the current instruction stream being executed, and execution resumes once the exception has been serviced.

Figure 10 shows an example target code sequence to
15 illustrate handling of interrupts.

The block of target code 21a contains instructions TC1-TC6. In this example, an interrupt occurs while the processor is executing instruction TC2. Execution of the
20 current instruction (TC2) is completed. Also, zero or more additional instructions are executed (TC3) until a counterpart target instruction 214 (in this case instruction TC4) is encountered which is associated with recovery information 195. That is, execution of the
25 target code instructions rolls forward until the next point where a precise subject state is achievable.

When the processor reaches the next counterpart target instruction 214 with associated recovery information 195,
30 the recovery information is passed to the recovery handler 215 in order to rectify a precise subject state. The precise subject state is then passed to an appropriate

exception handler 270, which is suitably a subject interrupt handler.

Figure 11 shows three preferred implementations to
5 handle interrupts within the target computing platform.

In a first preferred embodiment, a target code interpreter 197 is invoked when an interrupt-type exception occurs. The target code interpreter 197
10 executes the target code instructions one by one, until a counterpart instruction 214 having recovery information 195 is reached.

In a second embodiment, a step processor mode is
15 invoked. That is, the target processor 13 is controlled to enter a stepped processor mode which executes the target code instructions one by one until a counterpart instruction 214 is reached.

20 In a third preferred embodiment, check criteria are invoked to execute the target code instruction by instruction until a counterpart instruction 214 with recovery information 195 is reached.

25 If no fault vulnerable counterpart target instructions 214 are encountered in the target code block 21a, execution suitably runs to the end of the block. In a preferred embodiment, a precise subject state is rectified at a block boundary.

Traps

Traps are typically explicitly and unconditionally user-requested exceptions. Where a subject instruction
5 raises a trap-type exception, the translator 19 generates target code for that instruction which directly performs the behaviour required to handle the trap event.

Some instruction set architectures define traps that
10 require further special handling, such as single-stepping traps and conditional traps (for example the INTO instruction in the IA-32 architecture). Conditional traps, where the common case is for the exception not to be raised, are preferably handled similar to faults as
15 discussed above. This way, work in the common case is minimised. In the exceptional case, the required additional work is then invoked through the recovery handler.

20 Aborts

Aborts are typically non-recoverable events, but may have a variety of characteristics. Aborts are typically used to manage critical system failure. Upon detecting a
25 hardware failure such as an inconsistency of an internal processor state, or a serious software failure such as a fault within critical regions of the operating system, then the processor may attempt to ignore the error and continue execution, halt or reboot for safety, or an
30 exception may be raised to signal the error to the operating system. Like faults, aborts occur within instructions. Aborts are often terminal. Aborts are usually not user maskable.

Should an abort, such as a machine check error, occur within execution of an instruction that is not expected to fault, it is difficult to fully reproduce a precise
5 subject state at that aborting instruction. Generally, it is very expensive to handle aborts precisely in the context of dynamic binary translation.

As will be appreciated from the above discussion, the
10 exception handling mechanisms discussed herein have many advantages. In particular, work is removed from the common case wherever possible, and is only performed in the exceptional case, which occurs infrequently during a typical execution of a program. However, a precise
15 subject state is achieved when required. Also, overhead to achieve the precise subject state is minimised.

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

Attention is directed to all papers and documents
25 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.

30

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.

5 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
10 feature disclosed is one example only of a generic series of equivalent or similar features.

The invention is not restricted to the details of the foregoing embodiment(s). The invention extends to any
15 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.

20

Claims

1. A method of handling precise exceptions with
program code conversion from subject code to target code,
5 comprising the steps of:

(a) decoding subject code executable by a subject
processor according to a subject instruction set
architecture, and identifying a fault-vulnerable
10 instruction which is potentially faulting in the subject
instruction set architecture;

(b) generating target code from the subject code,
including generating one or more counterpart target
15 instructions from the identified fault-vulnerable subject
instruction, and storing recovery information associated
with the or each of the counterpart target instructions;

(c) executing the target code on a target processor
20 according to a target instruction set architecture; and

(d) when executing the or one of the counterpart
target instructions indicates a fault, then performing the
steps of:

25

(i) retrieving the recovery information associated
with the counterpart target instruction;

(ii) recovering a precise subject state which
30 accurately represents the subject processor at the fault,
using the recovery information; and

(iii) precisely handling the fault using the precise subject state.

2. The method of claim 1, wherein generating the
5 target code includes performing one or more optimizations,
and wherein the recovery information denotes the one or
more optimizations.

3. The method of claim 1, wherein generating the
10 target codes includes performing one or more optimizations
which leave an inaccurate subject state at the one or more
counterpart target instructions, and the recovery
information renders obtainable a precise subject state
from the available inaccurate subject state.

15

4. The method of claim 1, wherein the precise subject
state includes at least an accurate subject program
counter.

20 5. The method of claim 1, wherein the precise subject
state includes at least accurate register values
representing subject registers of the subject processor.

6. The method of claim 1, wherein storing the
25 recovery information comprises storing the recovery
information immediately following the generated target
code.

7. The method of claim 6, wherein retrieving recovery
30 information comprises the steps of:

scanning down the target code to locate the recovery
information subsequent thereto; and

retrieving the recovery information associated with the respective counterpart target instruction.

5 8. The method of claim 1, comprising:

dividing the subject code into a plurality of blocks;

generating a plurality of blocks of target code; and

10

storing recovery information interleaved between the generated blocks of target code.

9. The method of claim 1, comprising storing a marker
15 separating the target code from the recovery information,
and retrieving the recovery information by scanning from
the counterpart target instruction indicating a fault to
locate the marker and then scanning from the marker to
locate the associated recovery information.

20

10. The method of claim 1, comprising storing the
recovery information indexed by a target program counter
denoting the associated counterpart target instruction.

25 11. The method of claim 1, wherein recovering the
precise subject state includes passing the retrieved
recovery information to a recovery handler, obtaining a
currently available subject state, and modifying the
currently available subject state using the recovery
30 information to provide the precise subject state.

12. The method of claim 1, comprising passing the
recovery information to a recovery handler, and calling

from the recovery handler to one or more native code recovery routines according to the recovery information.

13. The method of claim 1, wherein the recovery
5 information comprises a target program counter value and a subject program counter value.

14. The method of claim 13, wherein the recovery
information further comprises a set of recovery indicators
10 which denote optimizations which are outstanding at the point of the counterpart target code instruction.

15. The method of claim 13, wherein the recovery
information further comprises a mapping template which
15 identifies a mapping of target registers to abstract subject registers applicable at the point of the counterpart target instruction.

16. The method of claim 1, wherein the recovery
20 information is stored in a compressed form.

17. The method of claim 13, wherein the recovery
information comprises a first recovery information item
with respect to a block of target code stored with a full
25 program counter value as a base value, and one or more subsequent recovery information items each comprising a program counter value as an offset from the base value.

18. The method of claim 1, further comprising the
30 steps of:

upon receiving an interrupt signal at the target processor during execution of the target code, continuing

execution of the target code until encountering the or one of the counterpart target instructions;

retrieving the recovery information associated with
5 the respective counterpart instruction and recovering the precise subject state;

precisely handling the interrupt using the precise subject state.

10

19. A method of precisely handling exceptions with program code conversion from subject code to target code, the method comprising the steps of:

15 (a) translating the subject code executable by a subject processor into target code executable by a target processor, including:

(a1) identifying a fault-vulnerable instruction in
20 the subject code;

(a2) generating one or more counterpart target code instructions as counterparts to the fault-vulnerable subject instruction;

25

(a3) storing a recovery information associated with the or each counterpart target instruction; and

(b) executing the target code on the target processor,
30 including:

(b1) when an exception occurs in relation to the or one of the counterpart target instructions,

locating the recovery information associated with the
faulting counterpart target instruction;

5 (b2) executing a recovery routine to rectify a
precise subject state representing the subject
processor, according to the located recovery
information; and

10 (b3) invoking an exception handler to handle the
exception with respect to the rectified precise
subject state.

20. A method of precise handling of exceptions with
program code conversion from subject code to target code,
15 comprising the steps of:

generating target code including at least one
counterpart target instruction which corresponds to a
potentially faulting subject instruction in the subject
20 code, and storing recovery information associated with the
at least one counterpart target instruction;

upon receiving an interrupt, continuing execution of
target code instructions until encountering the at least
25 one counterpart target instruction;

upon reaching the counterpart target instruction,
recovering a precise subject state using the associated
recovery information; and

30 precisely handling the interrupt using the recovered
precise subject state.

21. The method of claim 20, comprising, upon receiving the interrupt, executing the target code instructions one by one.

5 22. The method of claim 21, comprising invoking a stepwise mode in the target processor to execute the target instructions one by one.

23. The method of claim 21, comprising invoking a
10 target code interpreter to interpret the target code instructions one by one.

24. A method of precisely handling exceptions with dynamic binary translation from subject code to target
15 code, comprising the steps of:

generating a sequence of target code instructions from a sequence of subject code instructions;

20 executing the sequence of target code instructions and, when an exception occurs, rolling forwards along the sequence of target code instructions until encountering a target code instruction where a precise subject state is achievable; and

25 deferring handling of the exception until the precise subject state is achievable.

25. A translator apparatus arranged to perform the
30 method of any preceding claim.

26. A computer-readable medium having recorded thereon instructions implementable by a computer to perform the method of any of claims 1 to 24.

5 27. A method of handling exceptions during translation of program code, substantially as hereinbefore described with reference to the accompanying drawings.



INVESTOR IN PEOPLE

Application No: GB0507943.9

Examiner: David Midgley

Claims searched: 1-27

Date of search: 30 June 2005

Patents Act 1977: Search Report under Section 17

Documents considered to be relevant:

Category	Relevant to claims	Identity of document and passage or figure of particular relevance
X	1,19 at least	US2005/0005265 A1 KNOWLES See, for example, paragraph, [0016].
A	1,19	US2002/0092002 A1 BABAIAH See, for example, paragraph [0009].
A	20,24	US2004/0015627 A1 DESOLI See SUMMARY.

Categories:

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

Field of Search:

Search of GB, EP, WO & US patent documents classified in the following areas of the UKC^X :

G4A

Worldwide search of patent documents classified in the following areas of the IPC⁰⁷

G06F

The following online and other databases have been used in the preparation of this search report

Online: WPI, EPODOC