September 10, 2009

RT0877
Computer Science pages

Research Report

Profiling user-defined events

Takeshi Ogasawara

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

PROFILING USER-DEFINED EVENTS

Takeshi OGASAWARA

1. Background

In recent years, with the advent of software development technologies, the sizes of programs have been increased, and the programs have been highly integrated. Therefore, it is difficult to inspect whether programs operate as designed. One of existing technologies describes a method for analyzing the operation of a program using a performance counter. A performance counter is a special counter provided in a CPU. The value in the performance counter is incremented by one each time an event of the CPU occurs (refer to, for example, "The POWER2 performance monitor" (http://www.research.ibm.com/journal/rd/385/welbon.pdf) and "Software Optimization Guide for AMD Family 10h Processors" (http://www.amd.com/us-en/assets/content_type/hite_papers_and_tech_docs/40546.pdf)). In general, a performance counter is provided in a CPU so as to count the occurrence of a CPU event, such as a data cache miss. The CPU having such a configuration executes a program. The CPU detects and counts up the occurrence of a data cache miss concurrently with executing the program. Each time the number of the events reaches a predetermined value, a CPU exception is started. A handler of the CPU exception detects the location in the program that caused the data cache miss and stores the detected location. The distribution of the stored locations in the program that caused data cache miss is used for the analysis of the program performance. This technique has an advantage in that the number of the occurrences of low-level events, such as data cache misss, per unit of time can be obtained without changing the program. A disadvantage of this technique, however, is that, in order to detect the location in the program that causes a CPU event, the CPU exception that has high overhead needs to be used. To address this issue, in general, the overhead is reduced by reducing the number of CPU exceptions using a sampling method. Another disadvantage is that CPU events that can be detected are limited to predefined ones.

Another technique has been proposed in which a program issues an event and the number of the events is counted. In a typical case, the number of method calls and the execution time of each of the methods are measured. When calling a method, the program increments the number of method calls by one and acquires the current time (a method entry time). The program stores the current time. When the method exits, the program acquires the current time again. Subsequently, the program computes the elapsed time of the method using the stored method entry time. A typical example is Dtrace (refer to, for example, http://en.wikipedia.org/wiki/DTrace). This technique is useful for detecting a software-level event. A disadvantage of this technique, however, is that the overhead is significant, since profile code is executed in series with the program execution context. If the overhead is significant, the operation of the original program may change. In particular, if the execution timing points of parallel programs are changed, the events to be detected may not be detected at all or may not be detected properly.

2. User-Defined Event Profiling

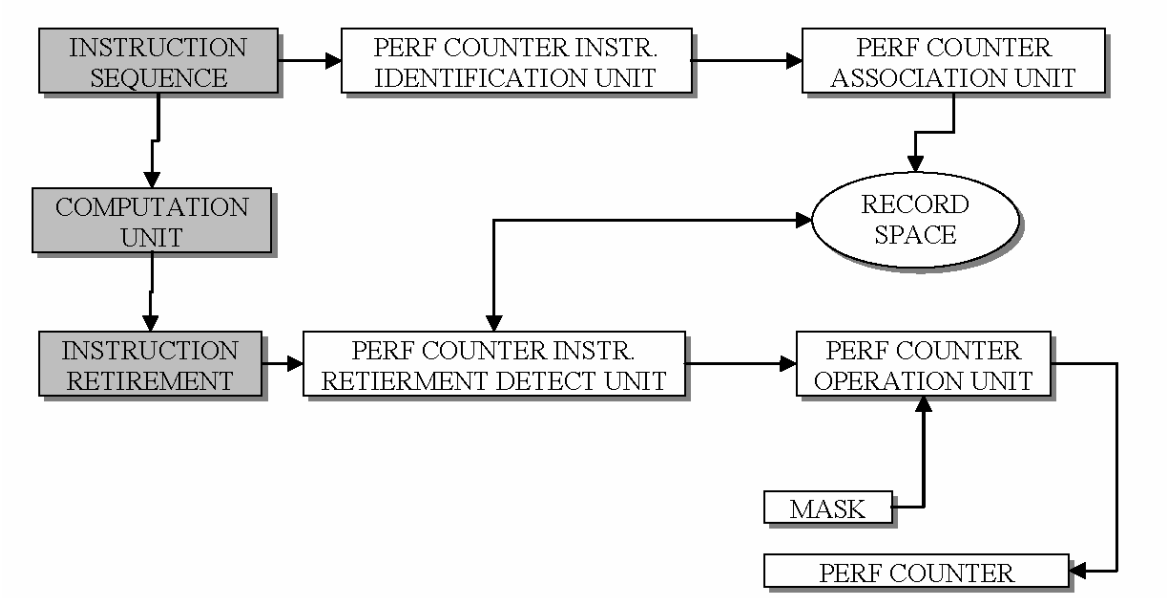

Fig. 1 OVERVIEW OF OUR APPROACH

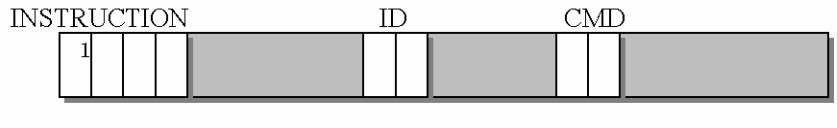

Fig. 2 INSTRUCTION FORMAT

We provide a method for informing the execution of code at a specified location of a program with minimized overhead. Accordingly, we provide a user-defined performance counter instruction to be inserted in a program by, for example, a compiler. In this way, software developers or users can acquire the profile for a program section that is frequently executed without changing the behavior of the program. Fig. 1 is a schematic illustration of our technique.

First, the identification unit (PERF COUNTER INSTR. IDENTIFICATION UNIT) identifies a performance counter instruction using an address at which a CPU attempts to execute an instruction and an instruction sequence. For example, the instruction sequence can be obtained from a wait queue of instructions decoded by the CPU. The identification unit extracts an instruction having a format that matches that of the performance counter instruction from the instruction sequence. The identification unit then sends the extracted instruction to the association unit (PERF COUNTER ASSOCIATION UNIT). For example, when the instruction format is defined as a fixed-length bit string shown in Fig. 2, the identification unit determines whether the first four bits of the instruction format are 1001, which are specific to the performance counter instruction.

The bit string may have a fixed length or a variable length. When the bit string has a fixed length, the instruction format of the performance counter instruction is pre-defined by the CPU. The performance counter instruction may be implemented as a new instruction, or an existing instruction may serve as a performance counter instruction. When the bit string has a variable length, a special counter used for defining the instruction format that indicates a performance counter instruction is provided. Thus, any performance counter instruction can be defined. Note that, by using a plurality of performance counter instruction registers, continuous instructions may be defined to serve as a performance counter instruction.

Fig. 3 PERFORMANCE COUNTER INFORMATION

The association unit associates the format of the sent instruction with a performance counter number and a performance counter operation command (hereinafter simply referred to as a "command"). Association can be performed in a variety of ways. One of the association methods is a method for encoding a counter number and a command into the instruction format. For example, as shown in Fig. 2, the bit strings other than the bit string that indicates the type of instruction are used for representing the counter number and the command. In another association method, by using a performance counter instruction register, a counter number and a command are specified, as in the normal instruction.

A set of the counter number (a counter ID), the command, and the address of the instruction associated in this manner (hereinafter, this set is referred to as "performance counter information") is stored in a performance counter information record space (hereinafter simply referred to as a "record space") in a FIFO manner. For example, an instruction buffer of the CPU used for managing an instruction sequence and the retirement of the instructions can be expanded and used for the record space. The above-described method is summarized in the flow chart shown in Fig. 4.

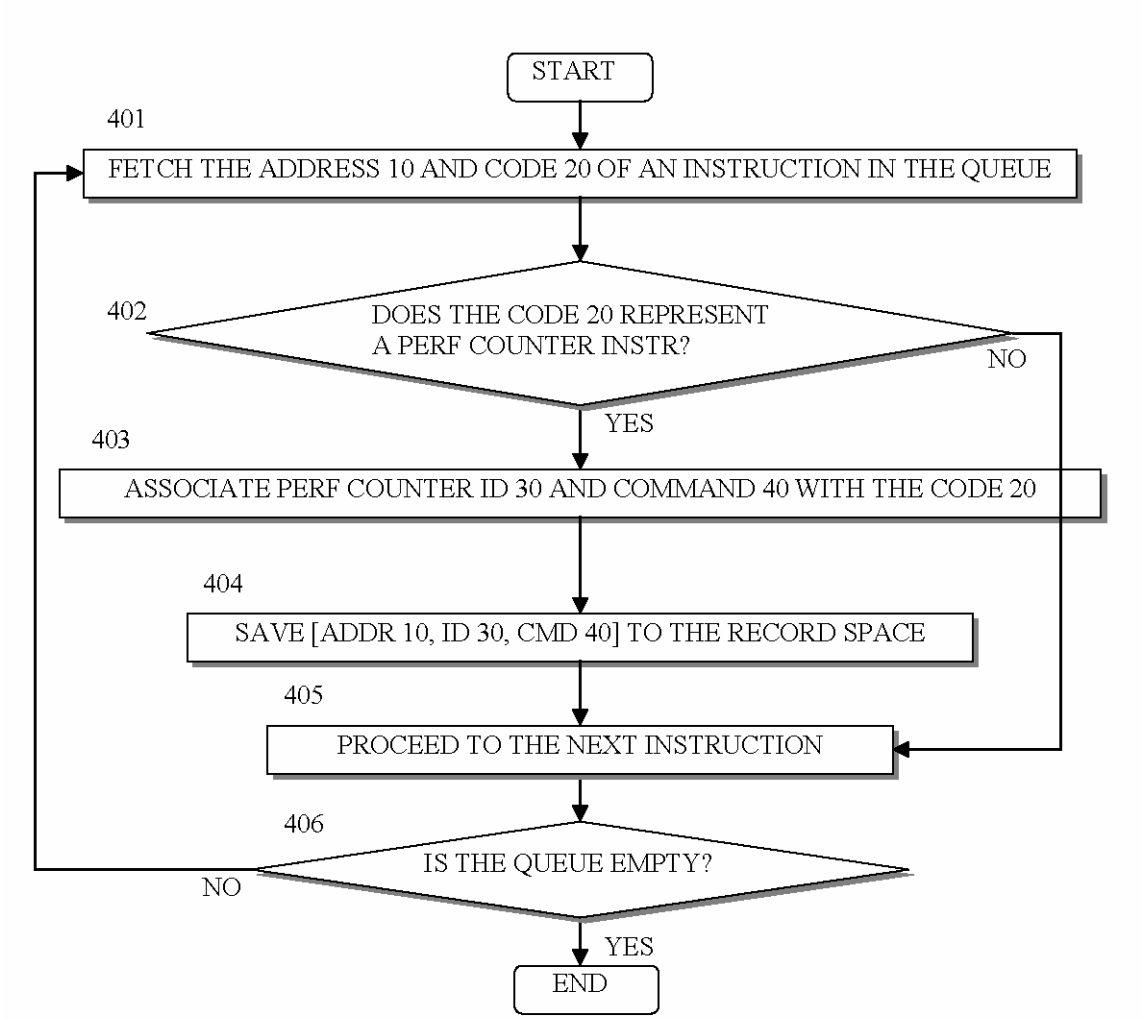

Fig. 4 DETECTION OF PERFORMANCE COUNTER INSTRUCTION

In general, retirement processing is performed after an instruction is executed by a CPU. The term "retirement processing of an instruction" refers to processing for considering the instruction as a completed instruction. The retirement detection unit identifies a performance counter instruction from among a series of instructions subjected to retirement processing. The retirement detection unit compares the instruction address of the first performance counter information that was stored in the record space with the address of a retirement instruction. If the addresses match, the retirement detection unit sends the performance counter information to the operation unit. Note that, if a record space is available in the instruction buffer, the performance counter information can be simply acquired in the form of the attribute of the retirement instruction.

The operation unit executes the command for the performance counter identified by the performance counter number. A typical command is a command for incrementing the value of the performance counter by one. Other commands can be supported as long as the number of commands is less than or equal to the number represented by the length of the command. For example, if the length of the command is 1 bit, two types of commands can be supported. Thus, an increment process of the

value or a decrement process of the value can be specified. The execution of the command can be controlled using a mask. The mask is a bit string stored in a predetermined register. The mask indicates performance counter numbers (a bit string), commands (a bit string), or the both. Thus, the matched operation can be disabled.

The above-described operation is summarized in the flow chart shown in Fig. 5.

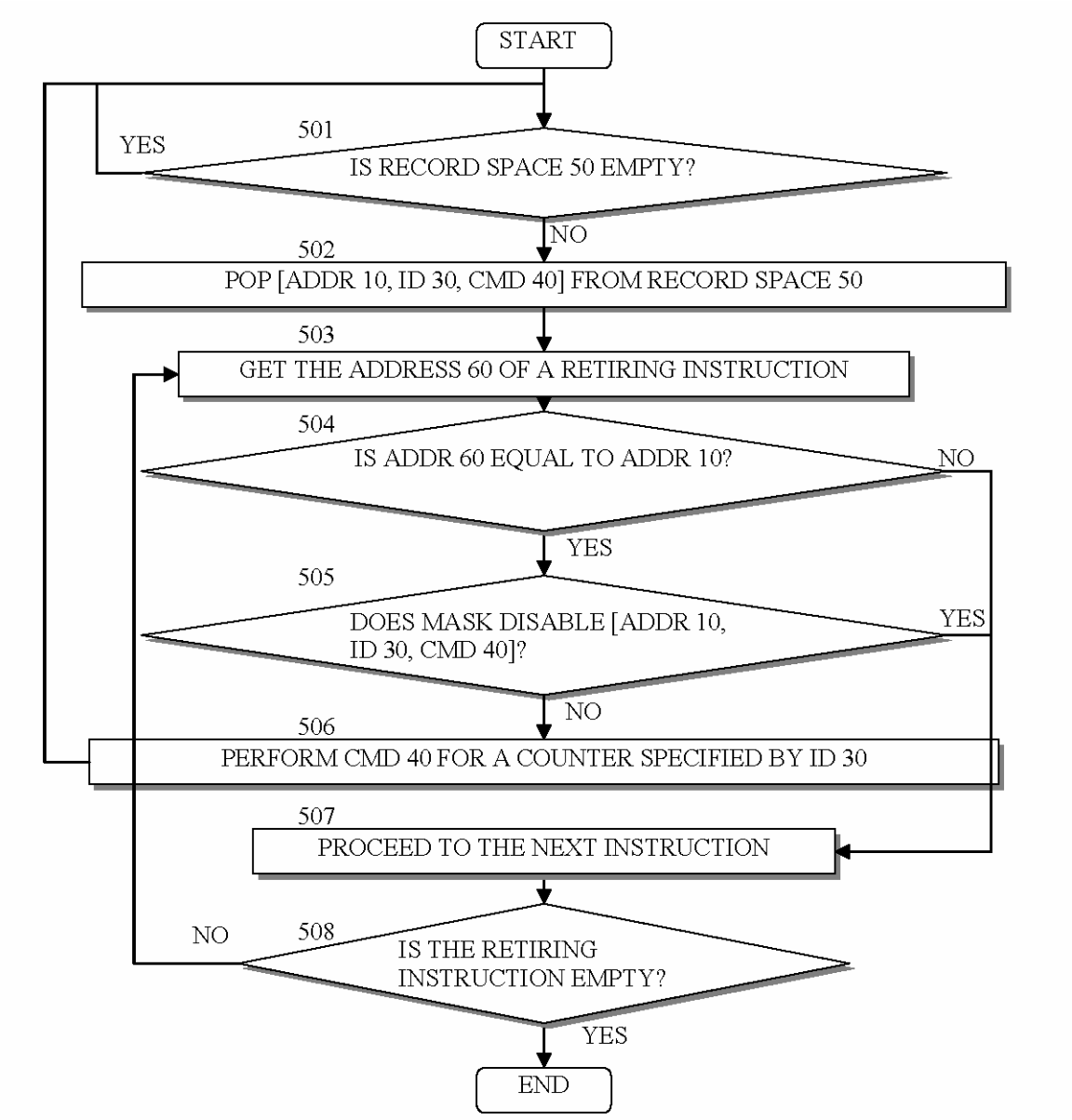

Fig. 5 OPERATION OF PERFORMANCE COUNTER

An example of application of the performance counter instruction to a program is described below. The following two methods for specifying a location at which a performance counter instruction is inserted are provided.

One of the methods is a method in which a user defines how the performance counter

instruction is inserted. The user can define the location at which the performance counter instruction is to be inserted or define the condition of inserting the performance counter instruction. For example, when defining the location, the user can use the line number in the file or the offset in a machine language. In contrast, when defining the insertion condition, the user specifies a method signature, and the start point and all of the return points of a method that matches the specified method signature are determined to be the insertion locations. More specifically, when the location is specified and if the line of the source program is specified, the instruction can be inserted in the machine language at the corresponding location by a compiler when the compiler generates the machine language. In contrast, if the offset in the machine language is specified, binary conversion software inserts the instruction into the machine language at the corresponding location. When the insertion conditions are specified, the compiler or the binary conversion software returns the location if the compiler or the binary conversion software detects the location that satisfies the conditions. Subsequently, the instruction is inserted when the machine language of the specified method is generated, for example.

The other method is a method for inserting the performance counter instruction at a predefined location. A software library pre-inserts some code into a predetermined location of the program code. When the library is developed, a mark (e.g., a macro) is attached to the corresponding location of the source code. When the source code is converted to a machine language, the mark is changed to the performance counter instruction.

By providing the CPU with the following functions, the usability of the CPU can be improved. The CPU has, in the instruction set thereof, dedicated instructions for reading and writing data from and to the performance counter and resetting one or all of the performance counters (e.g., setting the value of the performance counter to zero). In addition, when the performance counter overflows, an overflow interruption occurs. Subsequently, an interruption handler processes the overflow. In this way, the performance counter can continue the measurement of the performance.

Furthermore, the following function can be provided by software when the software is executed. For supporting context switching, the current value of the performance counter is read out for the context to be preempted in the context switching and is stored. Subsequently, the value of the performance counter stored for the dispatched context is restored. An operating system (OS) reads a mask and stores only the values of the performance counter that is enabled. In this way, the overhead can be reduced. Furthermore, the OS may accumulate the value of the performance counter when storing the value, and may reset a storage area S at the next dispatch time.

The following application may be employed. The application inserts the performance counter instruction into a particular spin loop or designates a particular instruction in the spin loop as a performance counter instruction. In this way, the number of spins can be measured with a minimum overhead. In addition, in order to provide a diagnostic function, the value of the performance counter stored in the OS during execution of the program is periodically backed up in an external storage unit (e.g., HDD). By examining the log of the values of the performance counter using an appropriate tool, the user can detect an abnormal behavior of the program, such as a significant variation in the value.

Also, a user can acquire a program profile SL generated by a compiler. The program profile SL indicates the frequency of executions of a spin loop. The compiler has a function of acquiring the profile. If the user enables this function using an execution option, the compiler automatically

generates code for acquiring the profile in the generated code. At that time, the compiler determines whether a target CPU supports the udpc instruction. If the target CPU supports the udpc instruction, the compiler can generate efficient code. However, if the target CPU does not support the udpc instruction, the compiler generates a library call for acquiring the profile (see Fig. 6).

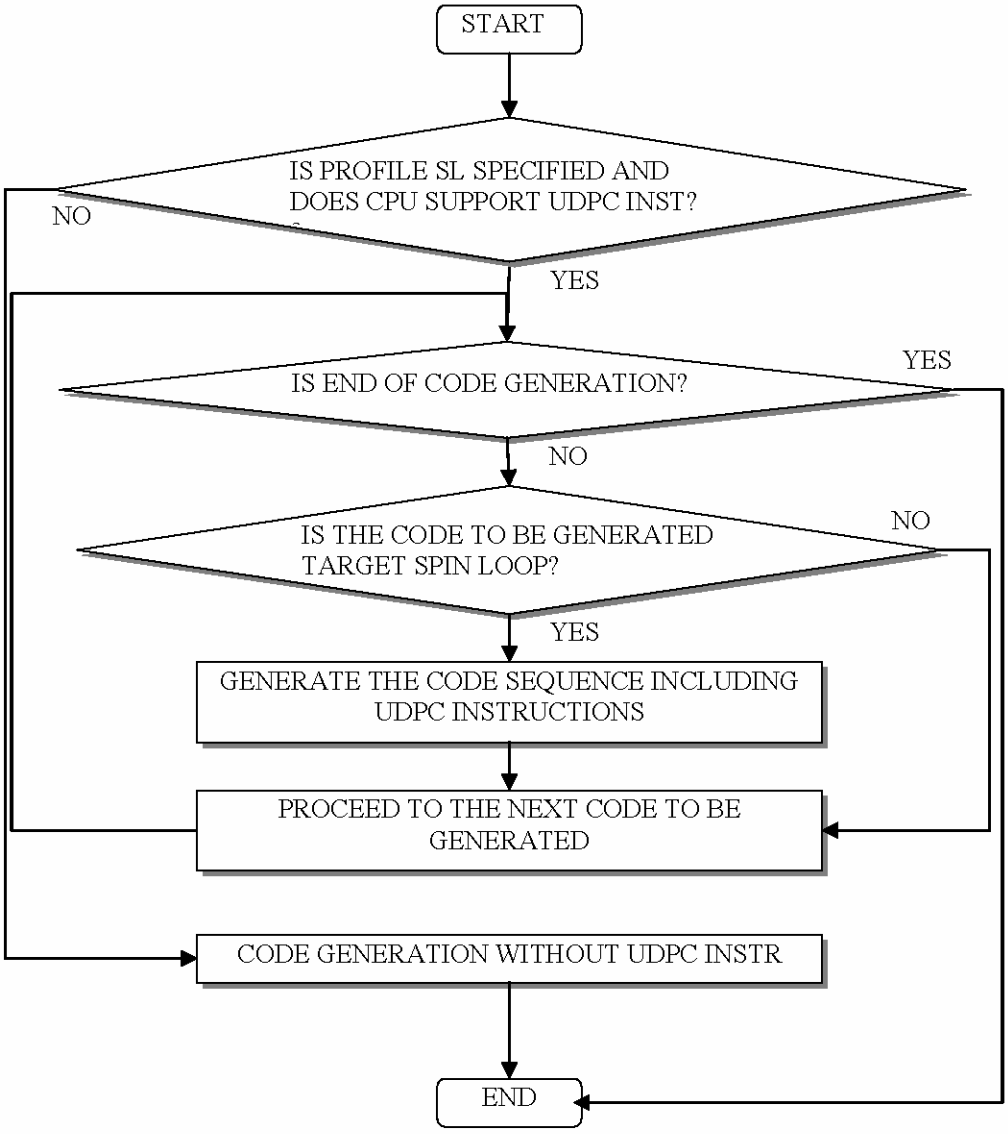

Fig. 6 OPERATION PERFORMED BY COMPILER FOR ACQUIRING PROFILE ON CPU

Here, suppose that the compiler generates the following spin loop code shown in Fig. 7.

```
100     r1 = 0
101     r3 = load mem1
102     r1 = cas mem2, r1, r3
103     cmp r1, 0
104     jne 100
```

Fig. 7 EXAMPLE OF SPIN LOOP CODE GENERATED BY COMPILER

Also, suppose that the profile indicates the frequency of the spin loop code and the number of spins, and the CPU has a dedicated udpc instruction. Then, when the profile function is enabled, the compiler generates the spin loop code including a udpc instruction inserted therein, as shown in Fig. 8.

```
100     r1 = 0
101     r3 = load mem1
102     r1 = cas mem2, r1, r3
103     cmp r1, 0
104     udpc2inc
105     jne 100
106     udpc1inc
```

Fig. 8 EXAMPLE OF COMPILER-GENERATED SPIN LOOP CODE HAVING FUNCTION FOR ACQUIRING PROFILE

A mnemonic "udpc1inc" represents a performance counter instruction including a performance counter operation command "inc" (for incrementing the counter by one) for a performance counter identified by number 1. More specifically, the instruction bit string having a length of one word includes bit substrings corresponding to an instruction code (e.g., 0xdcff), a performance counter number (0x01), and a performance counter operation command (0x00). The instruction sequence shown in Fig. 8 is processed by the CPU in accordance with the flow charts shown in Figs. 4 and 5. After the CPU has processed the instructions at addresses 100 to 103 in accordance with the flow chart shown in Fig. 4 (iteration of steps 402, 405, and 406), the CPU determines the type of the instruction at an address 104. Thus, the CPU can determine that the instruction is a performance counter instruction (steps 402 and 403 shown in Fig. 4). The CPU then stores the performance counter information (104, 2, inc) therein (step 404 shown in Fig. 4). The same processing is applied to the instructions at addresses 105 and 106. Thus, performance counter information (106, 1, inc) is stored. In this example, it is assumed that the conditional branch instruction at the address 105 does not cause a branch. Then, in the CPU, the instructions at addresses from 100 to 106 are sequentially retired. At the same time, in accordance with Fig. 5, when the performance counter information is stored (step 501 shown in Fig. 5), the operation unit pops the information (104, 2, inc) first (step 502 shown in Fig. 5). Subsequently, the operation unit skips the retired instructions at addresses 100 to 103 until the address of the retired instruction matches the popped address 104 (iteration of steps 503, 504, 507, and 508). When the instruction at address 104 is retired and if the profile is not masked, the value in the performance

counter identified by number 1 is incremented by one. The performance counter information (104, 2, inc) is then retired (steps 505 and 506 shown in Fig. 5). Subsequently, the processing proceeds to the processing of the next performance counter information (steps 506 and 501 shown in Fig. 5). In a similar manner, the performance counter information (106, 1, inc) is processed. In this way, the number of executions of the code section is stored in the performance counter identified by number 1, and the number of spins is stored in the performance counter identified by number 2. The profile code uses no registers, no stacks, and no heaps. In addition, the profile code does not alter the instruction sequence of the original program code. Furthermore, the profile code can be executed concurrently with the original program code. Accordingly, the processing performance of the original program code is not affected by the profile code. Still furthermore, if more detailed information is needed, for example, if a plurality of spin loops appear in the program and the profile for each of the spin loops is needed, different performance counter numbers can be assigned to the representative code locations of the code sections by sampling the code sections using the above-described overflow processing of a performance counter. Thus, each of the profiles can be identified (note that this processing can be autonomously performed by a dynamic compiler).

A manner how the CPU executes code that includes code for acquiring a profile and that is generated by a compiler is described next. Here, a manner how the code shown in Fig. 8 is processed using the method shown in Fig. 1 is described with reference to Figs. 9 to 12. Fig. 9 illustrates a state in which processing is just started from address 100.

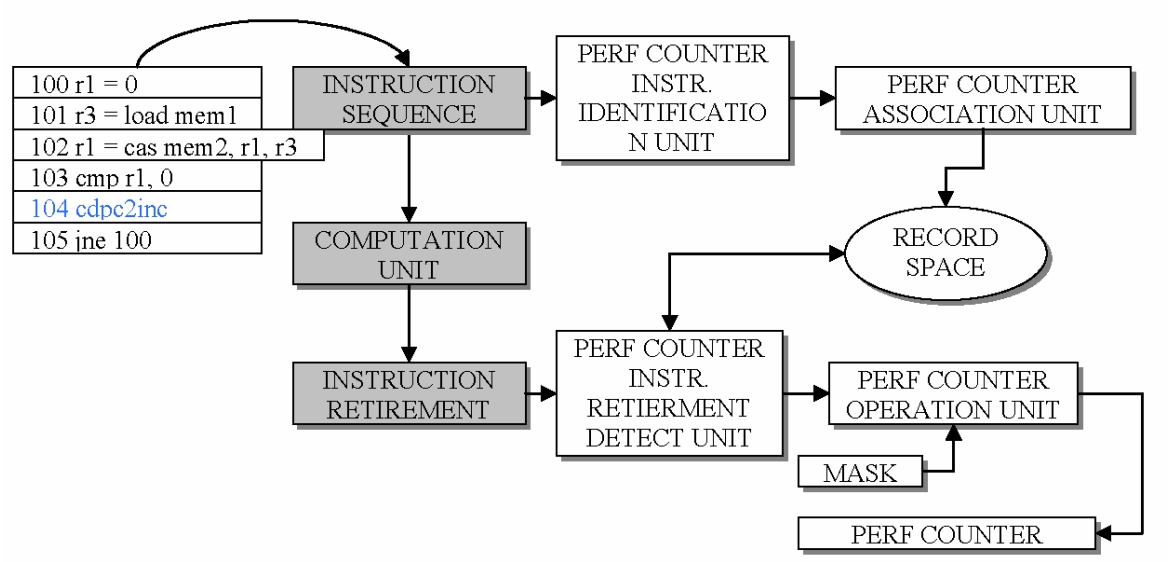

Fig. 9 PROCESSING OF SPIN LOOP CODE SEQUENCE INCLUDING CODE FOR ACQUIRING PROFILE (AT INITIAL TIME)

The processing continues. In Fig. 10, execution of the instructions at addresses 100 and 101 (shown by hatchings) has already been completed, and the retirement processing for the instruction at address 102 is being performed. Concurrently, the instruction at address 103 is executed in a computation unit. Similarly, the instruction at an address 104, that is, the instruction is concurrently

processed by the mechanism shown in Fig. 4, and the information is stored in the record space.

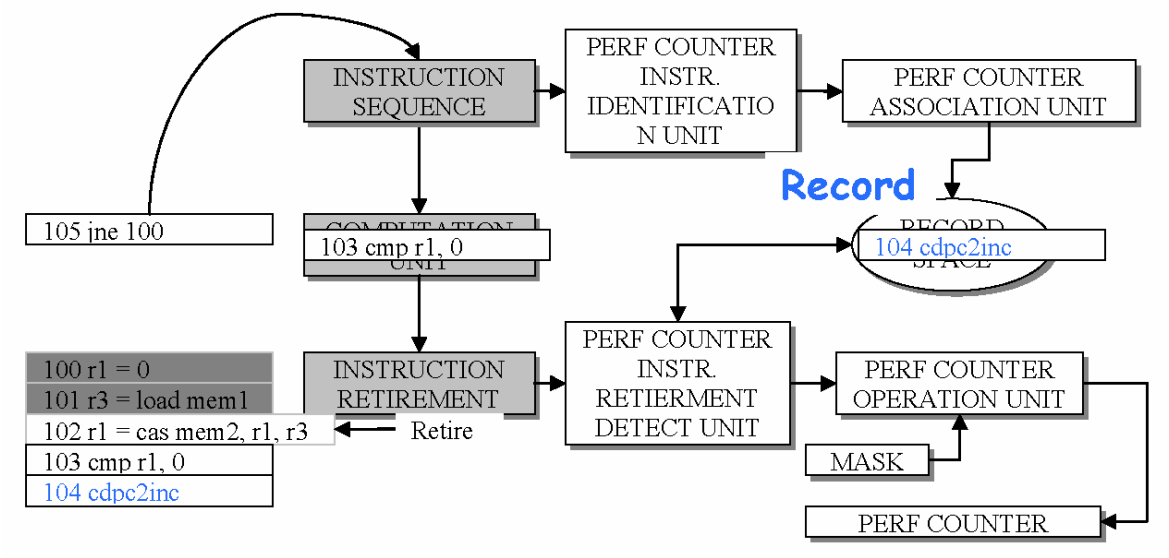

Fig. 10 PROCESSING OF SPIN LOOP CODE SEQUENCE INCLUDING CODE FOR ACQUIRING PROFILE (AT TIME WHEN DECODING OF INSTRUCTION AT ADDRESS 104 IS COMPLETED)

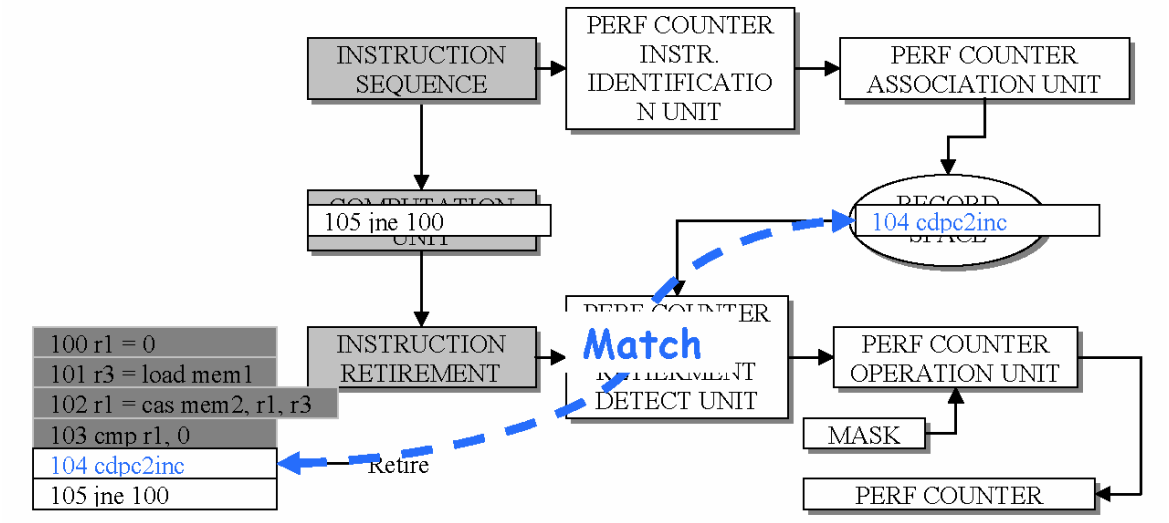

Fig. 11 PROCESSING OF SPIN LOOP CODE SEQUENCE INCLUDING CODE FOR ACQUIRING PROFILE (RETIREMENT OF INSTRUCTION AT ADDRESS 104)

The processing still continues. In Fig. 11, execution of the instructions at addresses from 100 to 103 has already been completed, and the retirement processing for the instruction at address 104 (the instruction) is being performed. Using the mechanism shown in Fig. 5, it can be determined that this address matches the address 104, which was stored before. The retirement processing for the instruction at address 104 is performed concurrently with the operation.

The processing still further continues. In Fig. 12, execution of the instructions at addresses from 100 to 104 has already been completed, and the retirement processing for the instruction at address 105 is being performed. At the same time, a counter operation is performed for the instruction at address 104 by the mechanism shown in Fig. 5.

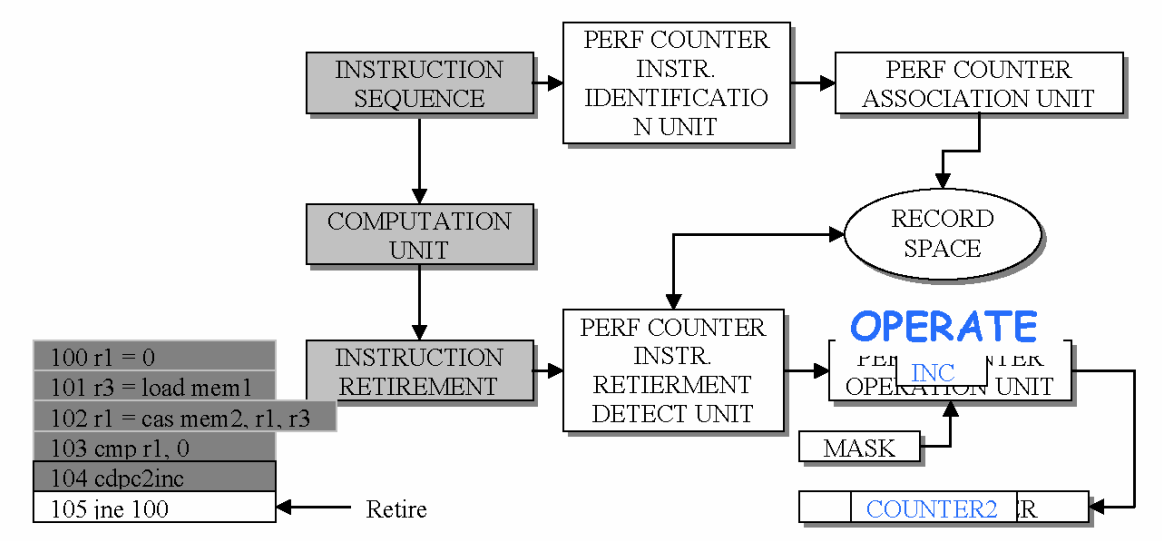

Fig. 12 PROCESSING OF SPIN LOOP CODE SEQUENCE INCLUDING CODE FOR ACQUIRING PROFILE (COUNTER OPERATION)