(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 101278265 B
(45)授权公告日 2012.06.06

(21)申请号 200680036157.3

(22)申请日 2006.10.02

(30)优先权数据
11/240,703 2005.09.30 US

(85)PCT申请进入国家阶段日
2008.03.28

(86)PCT申请的申请数据
PCT/US2006/038898 2006.10.02

(87)PCT申请的公布数据
WO2007/038800 EN 2007.04.05

(73)专利权人 英特尔公司
地址 美国加利福尼亚

(72)发明人 C·纽伯恩 H·王 X·邹 R·奈特 A·切尔诺夫 R·杰瓦

(74)专利代理机构 永新专利商标代理有限公司 72002
代理人 王英

(51)Int.Cl.
G06F 11/34(2006.01)
G06F 9/38(2006.01)
G06F 9/46(2006.01)

(56)对比文件
CN 1523500 A,2004.08.25,
US 20030066060 A1,2003.04.03,全文.
CN 1523500 A,2004.08.25,说明书第6页第2段,说明书第8页2－4段,第10页第5段,说明书第13页第3段,说明书11页第2段.
US 20030066060 A1,2003.04.03,图4.

审查员 董乐

权利要求书 2 页 说明书 16 页 附图 6 页

(54)发明名称

用于收集剖析信息的方法和用于优化代码段的系统

(57)摘要

在一个实施例中,本发明旨在提供一种系统,该系统包括:优化单元,用于对代码段进行优化;以及剖析器,其耦合到该优化单元。优化单元可以包含编译器和剖析控制器。此外,该剖析器可以用于请求用场景对通道进行编程,以便于在该代码段执行期间收集剖析数据。还描述和声明了其它实施例。

CN 101278265 B

1. 一种用于收集剖析信息的方法，包括以下步骤：

在可管理的运行时环境 (MRTE) 中执行非插装代码；

在所述非插装代码执行期间，在一特权级中使用处理器的资源监视至少一个硬件事件；

当触发条件发生时，在所述特权级中收集与所述至少一个硬件事件相对应的剖析信息，收集所述剖析信息包括：当所述触发条件发生时，异步地从所述非插装代码调用一服务例程，以及获得在导致所述触发条件发生的指令之前所述处理器的架构状态信息；并且

利用所述至少一个硬件事件和所述触发条件对所述资源进行编程，其中，所述资源包括通道。

2. 如权利要求 1 所述的方法，还包括：

在所述特权级中将控制转移到所述服务例程。

3. 如权利要求 1 所述的方法，还包括：

在与所述特权级相对应的用户级特权级中，执行所述非插装代码。

4. 如权利要求 1 所述的方法，还包括：

通过所述服务例程，对与一个不同的硬件事件相关联的至少一个其它触发条件进行处理。

5. 如权利要求 1 所述的方法，还包括：

在所述触发条件未发生的情况下，读取与所述至少一个硬件事件相关联的计数。

6. 如权利要求 1 所述的方法，还包括：

暂停收集所述剖析信息，同时继续监视所述至少一个硬件事件。

7. 如权利要求 1 所述的方法，还包括：

在所述非插装代码执行期间，修改所述触发条件。

8. 如权利要求 1 所述的方法，还包括：

在所述服务例程中，基于所述指令的一部分以及所述架构状态信息，确定与所述指令相关联的存储单元的有效地址。

9. 如权利要求 8 所述的方法，还包括：

实时地确定所述有效地址而不存储所述架构状态信息。

10. 如权利要求 1 所述的方法，还包括：

对所述服务例程进行剖析。

11. 一种用于在系统中转移控制的方法，包括：

在应用程序执行期间，监视至少一个硬件事件；

当与所述至少一个硬件事件相关联的条件被触发时，指示一让步事件；

依据所述指示，在没有操作系统 (OS) 干预的情况下，将控制从所述应用程序转移到一让步事件例程并且收集剖析信息，该剖析信息包括在导致所述至少一个硬件事件发生的指令之前处理器的架构状态信息；并且

其中，利用关于所述条件的信息对所述系统的处理器的存储装置进行编程，所述信息包括所述至少一个硬件事件、所述条件的触发、以及所述让步事件例程的地址。

12. 如权利要求 11 所述的方法，还包括：

通过所述让步事件例程来访问所述存储装置以收集存储在所述处理器中的剖析信息。

13. 如权利要求 12 所述的方法，还包括：

在剖析缓冲器中对所述剖析信息进行缓冲，以便由代码优化系统进行访问。

14. 一种用于对通道进行编程的方法，包括以下步骤：

接收应用程序使用处理器的处理器通道的请求，以用于在所述应用程序执行期间收集剖析数据；

针对所述使用，选择多个处理器通道中的一个；

用场景对所选择的通道进行编程；以及

当所述场景触发时，通过由所述处理器直接调用的服务例程，从所述处理器通道收集所述剖析数据，包括获得在导致所述场景触发的指令之前所述处理器的架构状态信息。

15. 如权利要求 14 所述的方法，还包括：

接收关于所述场景的控制信息，并将所述控制信息存储在所选择的通道中。

16. 如权利要求 14 所述的方法，其中，所述选择的步骤包括：

确定所述多个处理器通道中一个可用的通道。

17. 如权利要求 14 所述的方法，还包括：

识别要针对其来收集所述剖析数据的一个或多个硬件事件，并且设置与要触发所述场景的计数器值相对应的采样值。

18. 一种用于优化代码段的系统，包括：

优化单元，用于优化代码段，所述优化单元包括编译器和剖析控制器；以及

剖析器，其耦合到所述优化单元，用于请求用场景对通道进行编程以便在所述代码段执行期间收集剖析数据，其中，剖析信息的收集包括获得在导致所述场景的触发的发生的代码段的指令之前所述系统的处理器的架构状态信息。

19. 如权利要求 18 所述的系统，其中，所述剖析器在所述场景触发时将控制从所述代码段转移到服务例程。

20. 如权利要求 19 所述的系统，其中，所述剖析器在没有操作系统 (OS) 干预的情况下转移所述控制。

21. 如权利要求 18 所述的系统，其中，所述编译器包括即时 (JIT) 编译器，并且所述优化单元还包括耦合到所述 JIT 编译器的剖析缓冲器以存储所收集的剖析数据。

22. 如权利要求 18 所述的系统，其中，所述优化单元基于对在因所述代码段的一指令导致所述场景触发时所收集的剖析数据的分析，将预取例程插入到所述代码段中。

23. 如权利要求 22 所述的系统，其中，所述剖析器在不对所述指令进行解码的情况下确定与所述指令相关联的有效地址。

24. 如权利要求 22 所述系统，其中，在所述指令执行之前的所述系统的架构状态在所述触发之后可用。

用于收集剖析信息的方法和用于优化代码段的系统

背景技术

[0001] 本发明的实施例涉及计算机系统，更具体地，涉及对这种系统的资源的有效利用。

[0002] 计算机系统使用该系统的不同硬件资源执行各种软件程序，所述硬件资源包括：处理器、存储器和其它这样的组件。处理器本身包括各种资源，包括：一个或多个执行核心、高速缓冲存储器、硬件寄存器等等。某些处理器还包括硬件性能计数器，其用于对在程序执行期间发生的事件或动作进行计数。例如，某些处理器包括用于对存储器访问、高速缓存缺失、执行的指令等等进行计数的计数器。另外，在软件中还可以存在性能监视器，以监视一个或多个软件程序的性能。

[0003] 总之，能够根据不同使用模型来使用这样的计数器和监视器。例如，它们可以在编译或者其它优化活动期间使用，以便基于在程序执行期间所获得的剖析信息 (profile information) 来改善代码执行。近几年，随着大量新软件都在用可管理的语言 (managed language) 进行编写，收集剖析信息以用于反馈式 (feedback-directed) 动态优化，这种操作变得非常重要。传统的反馈式优化技术依赖于：提供用来收集剖析信息的程序，要求进行编译以便插入钩子来收集该数据，以很高的开销来运行该程序，然后利用剖析信息进行重新编译来获得产品二进制码。插装代码 (instrumentation code) 不能收集与其不能直接观察到的行为（例如，硬件存储器高速缓存行为）相关的信息。在另一种使用模型中，一旦在程序执行期间在计数器或者监视器中有事件发生，就可以调用一个或多个辅助线程 (helper thread)。这种辅助线程是软件例程，它们由调用程序进行调用来改善执行，例如，从存储器预取数据或者执行另一活动以改善程序执行。

[0004] 经常，这些资源的使用是很低效的，并且在不同使用模型中对这样的资源的使用可能发生冲突。因此，就需要有改进的方式来在这些不同的使用模型中获得并使用监视器和性能信息。

附图说明

[0005] 图 1 是根据本发明的一个实施例的处理器的框图。

[0006] 图 2 是根据本发明的一个实施例，多个通道 (channel) 的硬件实现的框图。

[0007] 图 3 是根据本发明的一个实施例，系统中的硬件 / 软件交互的框图。

[0008] 图 4 是根据本发明的一个实施例的方法的流程图。

[0009] 图 5 是根据本发明的一个实施例，用于使用编程的通道的方法的流程图。

[0010] 图 6 是根据本发明的一个实施例，用于执行服务例程的方法的流程图。

[0011] 图 7 是根据本发明的一个实施例的多处理器系统的框图。

具体实施方式

[0012] 现在参考图 1，示出了根据本发明的一个实施例的处理器的框图。在一些实施例中，处理器 10 可以是单芯片多处理器 (CMP) 或者另一种多处理器单元。如图 1 所示，第一核心 20 和第二核心 30 可以用于执行各种软件线程的指令。在图 1 中还示出，第一核心 20 包

括监视器 40，该监视器可以用于管理资源并控制该核心的多个通道 50a-50d。第一核心 20 还可以包括执行资源 22，所述执行资源例如可以包括由该核心和其它执行单元构成的流水线。第一核心 20 还可以包括耦合到执行资源 22 的多个性能计数器 45，所述性能计数器可以用于对这些资源内的各种动作或者事件进行计数。由此，性能计数器 45 可以检测特定的条件和 / 或计数值，并对各种架构和 / 或者微架构事件进行和监视，然后这些事件被发送到例如监视器 40。

[0013] 监视器 40 可以包括各种可编程逻辑、软件和 / 或者固件，用来跟踪在性能计数器 45 和通道 50a-50d 中的活动。在一个实施例中，通道 50a-50d 可以是基于寄存器的存储介质。一个通道是一个架构状态，其包括针对一个场景 (scenario) 的详细说明和发生信息，在下面将要进行讨论。在各种实施例中，一个核心可以包括一个或多个通道。每个软件线程可以对应一个或多个通道，并且对于每个软件线程，所述通道可以被虚拟化。监视器 40 可以对通道 50a-50d 进行编程以针对各种使用模型，包括性能导向优化 (PGO) 或者与通过使用辅助线程等而改善的程序性能有关。

[0014] 虽然在图 1 的实施例中示出了包括 4 个这样的通道，但是在其它实施例中可以存在更多或者更少这样的通道。此外，虽然为了易于图示而仅仅在第一核心 20 中示出了通道，但是在多个处理器核心中都可以存在通道。让步 (yield) 指示器 52 可以与通道 50a-50d 相关联。在各种实施例中，让步指示器 52 可以担当一个锁，以便在让步指示器 52 处于设定状态（例如）时防止发生一个或多个让步事件（以下要进行讨论）。

[0015] 仍然参考图 1，处理器 10 可以包括附加的组件，例如耦合在第一核心 20 与第二核心 30 之间的全局队列 35。全局队列 35 可以用于为处理器 10 提供各种控制功能。例如，全局队列 35 可以包括监听过滤器和其它逻辑，用来处理在处理器 10 内的多个核心之间的交互。进一步如图 1 所示，高速缓冲存储器 36 可以担当最后一级高速缓存 (last level cache，LLC)。此外，处理器 10 可以包括存储器控制器中枢 (MCH)38，用来控制在处理器 10 和与其耦合的存储器（例如，动态随机存取存储器 (DRAM)）（在图 1 中未示出）之间的交互。虽然在图 1 中示出了这些有限的组件，但是处理器可以包括很多其它组件和资源。此外，图 1 中所示的至少一些组件可以包括硬件或固件资源，或者硬件、软件和 / 或固件的任意组合。

[0016] 现在参考图 2，示出了根据本发明的一个实施例，多个通道的硬件实现的框图。如图 2 所示，如同由软件所看到的那样，通道 50a-50d 可以分别对应于通道 0-3。在图 2 的实施例中，通道标识符 (ID)0-3 可以标识被用具体场景编程的通道，并且可以对应于通道的相对优先级。在各种实施例中，当多个场景因同一指令而触发时，通道 ID 还可以标识服务例程执行的顺序（即，优先级），但本发明的范围并不局限于此。如图 2 所示，每个通道在被编程后包括场景段 55、服务例程段 60、让步事件请求 (YER) 段 65、动作段 70、以及有效段 75。虽然在图 2 的实施例中示出为该特定实现，但是要理解的是，在其它实施例中，在编程的通道中可以存储其它的或不同的信息。

[0017] 场景定义了组合条件 (composite condition)。换而言之，一个场景定义了在处理器中执行指令期间可能发生的一个或多个性能事件或条件。在各种实施例中，这些事件或者条件（可以是单个事件或者一组事件或条件）可 以是架构事件、微架构事件或者其组合。因此场景定义了：什么能够以硬件进行检测和存储、并呈现给软件。场景包括触发条件，例如在程序执行期间多个条件的发生。虽然这些条件可能改变，但是在一些实施例

中，例如，所述条件可以涉及在执行资源 22 中发生的动作的低进度指示器 (lowprogress indicator) 和 / 或者其它微架构或结构细节。场景还可以定义可供收集的处理器状态数据，其反应了在触发时处理器的状态。在各种实施例中，可以将场景硬编码到处理器中。在这些实施例中，可以通过标识指令 (例如，x86 指令集架构 (ISA)(以下称为“x86 ISA”) 中的 CPUID 指令) 而发现特定处理器所支持的场景。

[0018] 服务例程是在让步事件发生时执行的每场景一个的函数 (per scenariofunction)。如图 2 所示，每个通道可以包括服务例程段 60，包含有其相关服务例程的地址。让步事件是一种架构事件，其将当前运行的执行流的执行转移到场景的相关服务例程。在各种实施例中，在满足场景的触发条件时发生让步事件。在各种实施例中，监视器可以在让步事件发生时启动服务例程的执行。当该服务例程完成时，先前执行的指令流恢复执行。存储在 YER 段 65 中的让步事件请求 (YER) 是每通道一个的位，其指示该通道的相关场景已经被触发并且一个让步事件未决。存储在动作段 70 中的通道的动作位定义了在通道的相关场景触发时该通道的行为。最后，有效段 75 可以指示相关通道的编程状态 (即，该通道是否被编程)。

[0019] 仍然参考图 2，让步指示器 52 在此也称为让步块位 (YBB)，其与通道 50a-50d 相关联。让步指示器 52 可以是每软件线程一个的锁。当让步指示器 52 被设置时，则冻结所有与那个特权级相关联的通道。即，当让步指示器 52 被设置时，相关联的通道不能进行让步，也不能估计 (例如，计数) 其相关场景的触发条件。

[0020] 软件利用场景对硬件进行编程，使得该硬件能检测预定义的事件并收集预定义的信息。软件因此可以在最初配置硬件，然后开始、暂停、恢复、以及停止收集。在一些实施例中，单独的软件例程 (即，一个服务例程) 可以执行数据收集。采样收集机制可以包括 :初始化通道，收集剖析样本和 / 或读取事件计数，以及将先前编程的通道改为暂停、恢复、停止，或者修改场景的当前参数。

[0021] 现在转到图 3，示出了根据本发明的一个实施例，在系统中的硬件 / 软件交互的框图。如图 3 所示，硬件包括具有多个通道 50 的处理器 10。在一些实施例中，可能仅存在单个通道。例如，处理器 10 可以对应于图 1 的处理器 10。剖析软件 (profiling software)80 可以与处理器 10 通信，以便使用通道 50 实现数据收集。因此如图 3 所示，剖析软件 80 发送配置 / 控制信号到处理器 10。进而，处理器 10 执行剖析活动，例如根据编程的通道来进行计数。当剖析软件 80 发出请求时，处理器 10 可以发送剖析数据，其进而被提供给动态剖析导向优化 (DPGO) 系统 90。

[0022] 如图 3 所示，DPGO 系统 90 可以包括虚拟机 (VM)/ 即时 (just-in-time，JIT) 编译器 92，其可以从热点检测器 96 接收控制与配置信息。热点检测器 96 可以耦合到剖析控制器 94，该剖析控制器根据所收集的数据生成剖析信息，并将其发送至剖析缓冲器 98。剖析数据可以被从剖析缓冲器 98 传送到 VM/JIT 编译器 92，以用于驱动进行优化，例如可管理的运行时环境 (managed run time environment，MRTE) 代码优化。因此 DPGO 系统 90 使用剖析软件 80 所收集的数据来识别当前执行的代码中的优化机会。

[0023] 在各种实施例中，剖析软件 80 在处理器 10 中编写轻量化的用户级控制让步机制，以监视具体的硬件事件 (即，场景)。当一个场景触发 (即，让步) 时，处理器调用服务例程，该服务例程本身可以在剖析软件 80 内。服务例程可以收集关于硬件状态的信息，并对其进

行缓冲以便稍后将其传递到例如 DPGO 系统 90。服务例程还在返回到所计划的执行流之前直接对该信息起作用。所述轻量化的控制让步（即，异步转移）可以在没有操作系统 (OS) 参与的情况下，使得从软件线程中计划的执行流转移到由一通道定义的服务例程函数，并且返回至该计划的执行流。换而言之，该用户级中断完全绕过了 OS，实现了对于 OS 而言透明的更细粒度的通信与同步。因此，在场景触发（即，让步）时导致的中断由用户级软件进行内部处理。从而，没有来自用户级软件的对 OS 的外部中断，并且在单个特权级中执行该让步机制。例如，可以在第一特权级（例如，ring 0）中执行 OS 活动，而在第二特权级（例如，ring 3）中执行用户级活动。采用该轻量化让步机制的实施例，当发生让步事件时，控制可以从一个 ring 3 程序直接传递到在同一 ring3 程序中的另一函数，避免了需要驱动程序或其它机制来引起对 OS 可见 的中断。

[0024] 现在参考图 4，示出了根据本发明的一个实施例的方法的流程图。如图 4 所示，根据本发明的一个实施例，方法 100 可以由例如监视器用来对通道进行编程。如图 4 所示，方法 100 开始于 ：设置让步块位 (YBB)，以便在对通道进行编程时防止让步（框 110）。在一个实施例中，可以使用 EWYB 指令来设置 YBB。当 YBB 被设置时，让步机制被锁定，并且可以避免在一特定 ring 级的所有通道上发生让步。因此，可以在多通道硬件实现中设置 YBB，以确保一个通道在另一通道正被编程时不会让步。例如，设想在通道 1 让步时软件已经开始对通道 0 进行编程。执行与通道 1 相关的服务例程。如果通道 1 的服务例程修改了通道 0 的状态，则通道 1 的服务例程可能在不知道软件期望对通道 0 进行编程的情况下改变和 / 或者破坏了通道 0 的状态。在对通道 0 进行编程之前设置 YBB 位可以避免这种情况的发生。

[0025] 仍然参考图 4，接下来可以确定是否存在可用通道（框 120）。在一些实施例中，当一个通道的有效位清零时，认为该通道可用。在一些实现中，可以执行一个例程来读取每个通道的有效位。例如，通过 CPUID 指令可以发现在特定处理器中存在的通道数量。以下的表 1 示出了根据本发明的一个实施例的一个示例的代码序列，用来找到可用通道。

[0026] 表 1

[0027]

```
int available_channel = -1;
if (YBB is not already set)
{
 Set YBB
 for (int i=0; i<numChannels; i++)
 {
  setup ECX;  // 通道 ID = i，匹配位 = 0,
              // ring 级 = 当前 ring 级
  EREAD
  check ECX;
  if (valid bit == 0)
```

[0028]

```
  {
    available_channel = i;
    i = numChannels;  // 跳出循环
    break;
  }
 }
}
if (available_channel == -1)
{
  // 初始化失败
}
```

[0029] 如表 1 所示，首先设置 YBB，然后可以设定好寄存器（即 ECX），并且可以执行用于读取当前通道的指令（即 EREAD），以确定当前通道是否可用。具体而言，如果当前通道的有效位等于 0，则当前通道可用，从而退出表 1 的例程并返回该可用通道的值。注意，通过将匹配位设置为 0，在表 1 的例程中的 EREAD 指令期间不写处理器状态信息。

[0030] 返回参考图 4，如果在菱形 120 中确定没有可用通道，则控制可以传递到框 125。在那里，在某些实施例中，如果没有发现可用通道，则可以将一个消息（诸如错误消息）返回到试图使用该资源的实体（框 125）。反之，如果在菱形 120 中确定有可用通道，则接下来控制传递到框 130。在那里，如果需要，可以动态地迁移一个或多个通道（框 130）。在多通道环境中，可以根据通道优先级将一个或多个场景移动到一个不同的通道，在此称为动态通

道迁移 (DCM)。动态通道迁移允许在希望时将场景从一个通道移动到另一个通道。假设一个具体实现支持两个通道 :通道 0 和通道 1,其中通道 0 是最高优先级通道。此外,假设通道 0 当前正在使用 (即,其有效位被设置),并且通道 1 可用 (即,其有效位被清零)。如果监视器确定要将一个新场景编程到最高优先级通道中,并且确定如果将当前编程到该最高优先级通道中的场景移动到较低优先级通道中该新场景不会对其造成任何问题,则可以发生动态通道迁移。例如,可以读取当前编程到通道 0 中的场 景信息,然后可以将该场景信息重新编程到通道 1。

[0031] 仍然参考图 4,在动态通道迁移之后,可以对所选择的通道进行编程 (框 140)。对一个通道进行编程可以使各种信息存储在被选择来与进行请求的代理相关联的通道中。例如,软件代理可以请求用特定场景来对一通道进行编程。此外,代理可以请求在与该场景相对应的让步事件发生时执行位于特定地址 (存储在通道中) 处的给定服务例程。另外,在通道中可以存储一个或多个动作位。

[0032] 在一些实施例中,可以使用单个指令 (例如 EMONITOR 指令) 对通道进行编程。在对通道进行编程时包含 3 个选择,即 :选择场景,选择采样后值 (sample-after value),以及在剖析和计数之间进行选择。首先,可以选择一个场景,用来监视关注的硬件事件。在运行过程中,当该硬件事件发生时,如果该通道被配置为进行计数,则可以对该硬件事件进行计数。

[0033] 如果要使用通道进行剖析,则选择采样后值。所述采样后值描述了在下溢位被设置之前要发生的硬件事件 (由场景定义) 的数量。直到已经设置了下溢位并且另一触发条件发生,才进行让步。如果希望进行非采样剖析,则在每次发生触发条件时都要进行让步事件,将下溢位预先设置为 1,从而在触发条件第一次发生和随后的每次发生时都进行采样。反之,如果希望进行采样剖析,则可以将下溢位设置为 0,并且可以将计数器设置为采样后值。采样后值的选择确定了如果通道被配置为进行剖析则场景的计数器何时将会下溢以及该通道何时会让步。例如,如果采样后值 100 被编程,则在通道让步之前将会发生 100+2+X(在此,X 是取决于硬件实现的一个较小的数) 次硬件事件 (即,100 个事件使得计数器到达 0,另一个事件设置下溢位,再另一个事件使让步发生)。

[0034] 最后,编程可以在对事件进行计数和 / 或基于事件进行剖析之间进行选择。可以使用对事件进行计数来表征处理器的行为。可以使用基于硬件事件的剖析来确定当让步发生时处理器正在执行什么代码。在一些实施例中,计数可以是比剖析开销更低的操作。如果选择计数,则可以将动作位设置为 0(例如,使得让步将不会发生) 并且将采样后值设置为最大值 (例如,0x7FFFFFFF)。如果选择剖析,则可以将动作位设置为 1(例如,导致让步)。一旦对一个通道进行编程,就可以设置有效位以指示该通道已经被编程 (框 150)。在一些实现中,可以在编程期间设置有效位 (例如,通过用于对通道进行编程且设置有效位的单个指令)。最后,可以对在编程之前所设置的让步位清零 (框 160)。虽然采用图 4 的实施例中的该特定实现进行了描述,但是应该理解的是,在其它实施例中,对一个或多个通道的编程可以有不同的处理。

[0035] 以下伪代码序列描述了根据一个实施例,如何对一个通道进行编程。如表 2 所示,可以将期望的通道信息加载到第一组寄存器中。然后,单个指令,即 x86 ISA 中的 EMONITOR 指令,可以利用该信息对所选择的通道进行编程。如表 2 所示,可以在调用诸如 EMONITOR

指令这样的编程指令之前，首先设定好寄存器 EAX、EBX、ECX 和 EDX。

[0036] 表 2

[0037]

```
setup EAX;   // EAX 包含场景的采样后值
setup EBX;   // EBX 包含服务例程地址
setup ECX;   // ECX 包含场景 ID、动作位、ring 级、通道 ID 及有效位
setup EDX;   // EDX 包含对于 EMONITOR 指令的场景专用提示
EMONITOR     // EMONITOR 利用以上数据对该通道进行编程
```

[0038] 现在参考图 5，示出了根据本发明的一个实施例，用于使用编程的通道的方法的流程图。如图 5 所示，方法 200 可以开始于：执行一个应用程序，例如用户应用程序（框 210）。在该应用程序的执行期间，处理器进行各种动作。在处理器中发生的这些动作中的至少一些可以影响在该处理器中的一个或多个性能计数器或者其它这种监视器。从而，当出现这种影响这些计数器或者监视器的指令时，（多个）性能计数器可以根据这些程序事件而递减（框 220）。接下来，可以确定当前处理器状态是否与一个或多个场景匹配（菱形 230）。例如，与高速缓存缺失相对应的性能计数器可以将其值与在不同通道中的一个或多个场景中编程的选定值进行比较。如果处理器状态与任何场景都不匹配，则控制传递回框 210。

[0039] 反之，如果在菱形 230 中确定处理器状态与一个或多个场景匹配，则控制传递到框 240。在此，可以设置针对与所匹配的（多个）场景相对应的一个或多个通道的让步事件请求 (YER) 指示器（框 240）。YER 指示器可以由此 指示被编程到通道中的相关场景已经满足其组合条件。

[0040] 因此，处理器可以为其 YER 指示器被设置的最高优先级通道生成一让步事件（框 250）。当对一通道进行编程以进行剖析时，当其场景触发时，其将会进行让步。该让步事件将控制转移到一个已将其地址编程到选定通道中的服务例程。从而，接下来，可以执行该服务例程（框 260）。将在以下进一步讨论对执行服务例程的实现。注意，在调用该服务例程之前，即在让步期间，处理器可以将各种值压入用户栈中，在该栈中，这些值中的至少一些要被（多个）服务例程所访问。具体而言，在一些实施例中，处理器可以将当前指令指针 (EIP) 压入到栈中。此外，处理器可以将控制与状态信息压入栈中，诸如修改版本的条件代码或者条件标志寄存器（例如，在 x86 环境中的 EFLAGS 寄存器）。另外，处理器可以将正在进行让步的通道的通道 ID 压入栈中。

[0041] 一旦该服务例程完成，就可以确定是否设置了其它 YER 指示器（菱形 270）。如果没有，则方法 200 可以返回到如上所述的框 210。反之，如果设置了其它 YER 指示器，则可以将控制从菱形 270 传递回如上所述的框 250。

[0042] 在不同的实施例中，服务例程可以采用很多不同形式。一些服务例程可以用来收集剖析数据，而其它服务例程可以用来改善程序性能，例如通过预取数据。无论如何，服务例程都可以执行某些高级功能。现在参考图 6，示出了根据本发明的一个实施例，执行服务例程的方法的流程图。如图 6 所示，方法 300 可以开始于：发现正在进行让步的通道（框 310）。在各种实施例中，服务例程可以从栈中弹出最近的值（即，通道 ID）。该值将映射到

让步的通道，并且可以用作在服务例程期间对于各种动作或指令（诸如收集数据和 / 或对通道进行重新编程）的通道 ID 输入。

[0043] 仍然参考图 6，接下来可以由该服务例程处理由正在进行让步的通道所提供的机会（框 320）。处理所述机会可以根据使用模型而采用不同的形式。例如，服务例程可以执行代码以利用处理器的当前状态（如由场景定义所定义的）、收集一些数据、或者读取通道状态。

[0044] 当收集数据时，在仅收集通道状态数据与收集通道及处理器状态数据之中进行选择。以下在表 3 中所示的伪代码描述了收集数据的一个实施例。 当然，其它实现也是可行的。

[0045] 表 3

[0046]

```
setup EAX;  // EAX 包含缓冲器指针，(用于收集处理器状态数据)
setup ECX;  // ECX 包含场景 ID、匹配位、
            // ring 级、以及发现的通道 ID
            // (如果场景 ID 输入与当前编程到
            // 通道中的场景 ID 匹配并且设置了
            // 匹配位，则将收集处理器状态数据)
EREAD
suspend_flag = 0;
error_flag = 0;
read EAX;  // EAX 包含当前硬件事件计数
// EBX 包含原先通过 EMONITOR 编程到通道中的服务例程地址
read ECX;  // ECX 包含通道的当前场景 ID、动作、
           // ring 级、通道 ID 和有效位的值
if (ECX is not programmed as expected)
{
  // 通道已经被窃取；采取适当步骤来报告/解决问题
  // (例如关闭或者重新编程该通道)，并跳过对样本数据的记录
  error_flag = 1
}
if (collecting processor state data and error_flag==0)
{
 // [EAX]包含由场景 ID 所定义的处理器状态数据
 adjust buffer pointer to move past processor state date collected;
 if (buffer pointer + sample size >= buffer end)
 {
  set flag indicating data is ready;
  // 使用一不同缓冲器继续收集，或者中止并
  // 等待优化子系统处理当前缓冲器
```

[0047]

```
    // 继续收集
    buffer pointer = a different buffer pointer;
    OR
    // 中止收集
    suspend_flag=1;
  }
}
```

[0048] 仍然参考图6，接下来，可以对通道进行重新编程（框330）。虽然在图6的实施例中示出了包含该框，但是应该理解，在很多实施例中可以不需要进行重新编程。然而，在实现时，可以在数据收集之后进行重新编程。更具体而言，可以对一个通道重新编程以重置其采样后值。如果没有对该通道重新编程，则在该通道最初下溢时设置的下溢位可以保持被设置，并且每次满足场景定义的硬件事件发生时该通道都将进行让步。此外，注意，当对通道重新编程时可以不设置YER位。为了对通道重新编程，可以在设定好某些寄存器（诸如EAX、EBX、ECX和EDX寄存器）之后使用EMONITOR指令。注意，可以保存先前从EREAD返回的EBX、ECX和EDX寄存器的值，并且在EMONITOR指令期间再次使用。在转换到服务例程的过程中可以将YER位清零。表4中示出了根据一个实施例，用于对通道进行重新编程的示例伪代码。

[0049] 表4

[0050]

```
setup EAX;  // EAX 包含场景的采样后值
setup EBX;  // EBX 包含服务例程地址
setup ECX;  // ECX 包含场景 ID、动作、ring 级、
            // 在进入服务例程时发现的通道 ID
            // 以及有效位(该有效位应被设置)
            // 如果设置了中止标志，则应将
            // 动作位设置为 0，以中止让步
setup EDX;  // EDX 包含对于 EMONITOR 指令的场景专用提示
EMONITOR
```

[0051] 最后参考图6，一旦重新编程（如果发生），服务例程可以将控制返回至例如在该通道的场景触发时正执行的原先的软件线程（框340）。为了退出服务例程，可以发生各种

动作。在一个实施例中，单个指令（例如，x86 ISA 中的 ERET 指令）可以执行各种功能。在让步入口（entry）期间压入栈的修改后的 EFLAGS 映象可以被从栈中弹出返回到 EFLAGS 寄存器中。接下来，在让步入口期间压入栈的 EIP 映象可以被从栈中弹出返回到 EIP 寄存器中。以这种方式，原先执行的软件线程可以恢复执行。注意，在退出操作期间，在让步开始时压入栈的通道 ID 不需要从栈中弹出。取而代之的是，如上所述，该栈值在服务例程期间被弹出。

[0052] 在一些实现中，一旦已经发生了让步，则就可以确定其它让步是否被挂起。例如，当正在执行针对已让步的通道的服务例程时，可以读取其它通道的状态（例如，通过 EREAD 指令）。如果另一通道的 YER 位被设置，则该通道的场景已经触发并且对其服务例程的调用被挂起。能够收集数据，并且能够对该通道重新编程。如果该通道的 YER 位没有清零，则让步可以保持挂起。

[0053] 使用这种机制，可以通过避免到服务例程的一些转换，来减少服务例程的开销。但是由于 DCM 的原因，软件不能假定其拥有哪个通道。如果每个通道都被用一个不同服务例程进行编程，则可以使用该通道的服务例程地址作为唯一标识符。每个通道在特定的软件线程中都是唯一的（假设针对每个软件线程，通道都被虚拟化）。假设每个软件线程都存活在单个进程的上下文中，则保证了服务例程地址是唯一的。

[0054] 因此，为了在单个服务例程中处理多个让步，可以采用唯一的服务例程地址对每个通道进行编程。然后，在处理一个挂起的让步之前，可以将通道的服务例程地址与预先编程的多个服务例程之一进行匹配。如果通过使得在每个（或者除了一个之外的全部）服务例程目标中的第一个指令都是向公共服务例程的跳转或者对其的调用来使它们共享同一服务例程，则仍然可以支持服务例程地址的唯一性。

[0055] 如上所述，当一个通道被编程以便对硬件事件进行计数时，其将不会 进行让步（由于其动作位被清零）。取而代之的是，软件线程可以周期性地或者在适当时刻（例如，方法的入口 / 出口）读取通道状态，以获得其当前硬件事件计数。在软件线程读取硬件事件计数之前，其必须找到用适当场景编程的通道。由于 DCM 的原因，活动的场景可能迁移到其它通道。如果将唯一的服务例程地址编程到每个通道中，则（例如通过 EREAD 指令）返回的服务例程地址可以用于唯一地标识正确的通道。表 5 中所示的伪代码序列可以用于找到当前用特定场景编程的通道并且保存当前的硬件事件计数。

[0056] 表 5

[0057]

```
int my_channel = -1;
in my_service_routine_address = (int)service_routine;
int sr;  // 用于保存从 EREAD 返回的服务例程地址的变量
int count;
for (int i=0; i<numChannels; i++)
{
 setup ECX;  // 通道 ID = i，匹配位 = 0，
             // ring 级 = 当前 ring 级
 EREAD
 mov count <- eax  // 在为选定的通道的情况下，保存当前计数
 mov sr <- ebx
 // 在该通道需要重新编程的情况下
 // 保存 ebx、ecx 和 edx 的值
 if (sr == my_service_routine_address)
 {
  my_channel = i;
  i = numChannels;  // 跳出循环
  break;
 }
}
```

[0058] 如果事件计数为负，则计数器已经下溢，并且可以对该通道重新编程。表 6 的伪代码示出了硬件计数积累和通道重新编程（如果需要）的一个实施例。

[0059] 表 6

[0060]

```
// total_count: 保持累积的计数
// previous_count: 保持通道读取的前一个计数
total_count = previous_count – count;
previous_count = count;
if (count < 0)
{
 // 通道已经下溢，对其重新编程
 // EAX 包含采样后值
 mov eax <- 0x7FFFFFFF
 // 恢复所保存的 ebx、ecx 和 edx 的值
 EMONITOR
 previous_count = 0x7FFFFFFF;
}
```

[0061] 以上代码假设将在多个下溢发生之前对通道进行读取。如果多个下溢是一种可能，则可以将动作位设置为 1，并且可以使用服务例程来在发生下溢时对其进行处理。

[0062] 有时，可能希望暂停数据收集。可以采用两种不同的方式来实现暂停剖析收集。要完全暂停收集，可以在适当的通道中将动作位清零。当动作位被清零时，该通道继续计数但是不让步。要恢复收集，可以将该适当的通道的动作位设置为 1。为了不使采样间隔不正常，可以在暂停时保存计数值，并且当继续通道的使用时将其恢复。如果一个通道的 YER 位被设置而该通道被暂停，则将不会发生让步。另一种用于暂停剖析收集的机制是，在服务例程中跳过数据收集。换而言之，当收集暂停时，在服务例程期间不调用用于读取数据的指令。第一种机制，即对动作位清零，与第二种机制相比可以引起更少的开销，因为没有执行服务例程。要完全停止收集，在一些实施例中，用于对在一个通道中的有效位清零的单个指令可以停止 剖析和 / 或计数收集。一旦将一个通道的有效位清零，该通道就可以被任何其它软件使用。

[0063] 如果一个服务例程进行了大量的工作，则可以对该服务例程本身进行剖析。为了对服务例程进行剖析，可以在服务例程执行期间对 YBB 清零，以允许在该服务例程执行的同时，硬件在场景触发时进行计数和 / 或让步。可以使用两种机制来对 YBB 清零。首先，可以使用设计用来写 YBB 的一个指令，例如在 x86 ISA 中的 EWYB 指令，来直接将 YBB 清零。第二，另一指令，例如在 x86 ISA 中的 ERET 指令，在被调用时可以隐含地将 YBB 清零。表 7 的伪代码序列示出了根据一个实施例，如何在退出一个服务例程之前对 YBB 清零。

[0064] 表 7

[0065]

```
void ServiceRoutine(void)
{
 pop channel   // 将通道 ID 弹出栈
 setup registers for EREAD;
 EREAD   // 在释放 YBB 锁之前进行 EREAD，以有效地
         // 避免在通道让步时丢失处理器状态信息
 // 接下来对通道重新编程，因此能够重新使用
 // 从 EREAD 返回的寄存器值
 setup registers for EMONITOR;
 EMONITOR
 // ERET 将会从栈中弹出两个值
 // 标志和 EIP。将这些寄存器的值压入栈。
 push 0   // 将伪标志压入栈，这些将会
          // 由第一个 ERET 指令弹出栈
 mov eax <- eip   // 操作当前 EIP 寄存器的值以
                  // 在 ERET 指令之后指向该 EIP
 add eax <- XYZ   // XYZ 是该 add 指令加上下面的
                  // push 指令再加上下面的 ERET
                  // 指令的大小(以字节计)
```

[0066]

```
 push eax
 ERET   // 将 YBB 清零，弹出下一个 EIP 和先前
        // 压入的标志，因此服务例程继续(YBB 被清零)，
        // 以便继续进行监视
 do work that needs to be monitored here;
 ERET
}
```

[0067] 为了对一个服务例程进行剖析，可以对通道重新编程，以使用一个不同的场景和/或一个较小的采样后值，来在服务例程的被剖析部分的执行时确保通道让步。或者，只要第一个通道一让步，就可以用较小的采样后值来对第二个通道进行编程。只要在第一个通道中 YBB 被清零，两个通道就都是活动的。

[0068] 很多剖析收集使用模型允许场景被复用和/或者允许特定场景所使用的采样后值在运行时被修改。其它的对于通道状态的运行时修改也是可行的。要改变通道状态，在一个实施例中可以实现以下操作序列：(1) 设置 YBB(在多通道硬件实现中)；(2) 找到通道；(3) 对通道重新编程；以及 (4) 将 YBB 清零(如果设置了的话)。

[0069] 另外，可以对通道进行保存、重新编程、并随后恢复到其原始状态。因此，要重新编程的通道可以使用例如 EREAD 指令来保存其状态。在重新编程之后和在执行期间，可以在特定的代码块或者时间段期间监视软件线程。一旦完成了监视，就可以设置 YBB，找到重新编程的通道，并且例如通过 EMONITOR 指令用原先存储的值来对状态进行恢复。

[0070] 在很多实施例中，存在两种不同类型的场景：类似陷阱 (trap-like) 的场景和类似故障 (fault-like) 的场景。类似陷阱的场景在触发该场景的指令引退之后执行其服务例程。而类似故障的场景一旦场景触发就执行其服务例程，然后重新执行触发该场景的指令。因此，在类似故障的场景中，在场景触发之前的架构寄存器状态在服务例程运行期间可以被访问。

[0071] 例如，指令 mov eax < -[eax] 将在执行期间修改 EAX 的原始值。如果在该指令执行期间触发类似陷阱的场景，则该场景的服务例程将不能确定 在该场景触发时 EAX 的值。但是如果在该指令执行期间触发类似故障的场景，则其服务例程能够确定在该场景触发时 EAX 的值。

[0072] 例如，如果所述触发涉及高速缓存缺失，则通过有效地使用在该指令执行之前的架构寄存器状态，可以确定在高速缓存中缺失的数据的地址(即，有效地址)。一旦确定了，就可以插入一个预取例程，从而优化应用程序来预取数据，避免了高速缓存缺失。在一些实施例中，可以对用于在类似故障的场景的情况下计算有效地址的软件进行优化，这是因为服务例程仅需要存储器地址，因而不需要对整个指令进行解码。从而，并非使用完整指令解码器，而是地址解码器可以使用指令集中的规律性来构建存储器地址和数据大小。

[0073] 在一个实施例中，地址解码器中的快速初始路径查找一个表来确定指令的存储器访问模式。换而言之，一个指令集中的各种指令具有类似的存储器访问模式。例如，多组指令可以请求同一长度的信息，或者可以将数据压入栈或者从栈中弹出，等等。因此，根据指令类型，可以提供高效的线性地址解码。表格条目还可以包括与用于对地址进行解码的、要从指令中获得的数据相关的信息。然后，其分派到选定的代码片段以构建故障指令的地址。可以对该表进行组织，以确保公共的分派路径共享高速缓存行，改善了连续解码的效率。因此，在各种实施例中，可以高效地对一个指令进行解码，以获得线性地址信息，同时忽略该指令的操作数部分。此外，可以在一个服务例程的上下文中快速地执行解码，显著地降低了执行数据收集的开支。此外，该地址解码可以在服务例程本身的上下文中进行(即，动态地、实时地)，避免了保存所捕获的大量数据并且随后执行完整解码的开支，后者本身也是一个很昂贵的处理过程。在一些实施例中，所获得的地址信息可以用于将预取插入到代码中，或者用于将数据放置在存储器中的不同位置处，以便减少高速缓存缺失的数量。可替换地，可以将地址信息作为信息提供给应用程序。

[0074] 作为例子，可以在运行着可管理的运行时应用程序和服务器应用程序的架构中使用各种实现。现在参考图 7，示出了根据本发明的一个实施例的多处理器系统的框图。如图 7 所示，该多处理器系统是点对点互连系统，并且包括经由点对点互连 450 耦合的第一处理

器 470 和第二处理器 480。 如图 7 所示，处理器 470 和 480 中每一个都可以是多核处理器，包括第一和第二处理器核心（即，处理器核心 474a 和 474b 以及处理器核心 484a 和 484b）。虽然为了便于图示而没有示出，但第一处理器 470 和第二处理器 480（更具体而言是其中的核心）可以包括多个在此所述的通道。第一处理器 470 还包括存储器控制器中枢 (MCH) 472 和点对点 (P-P) 接口 476 和 478。类似地，第二处理器 480 包括 MCH 482 和 P-P 接口 486 和 488。如图 7 所示，MCH 472 和 482 将处理器耦合到各自的存储器，即存储器 432 和存储器 434，其可以是接在本地的主存储器的一部分。

[0075] 第一处理器 470 和第二处理器 480 可以分别经由 P-P 接口 452 和 454 耦合到芯片组 490。如图 7 所示，芯片组 490 包括 P-P 接口 494 和 498。此外，芯片组 490 包括接口 492，以便将芯片组 490 与高性能图形引擎 438 耦合。在一个实施例中，可以使用高级图形端口 (AGP) 总线 439 来将图形引擎 438 耦合到芯片组 490。AGP 总线 439 可以符合加利福尼亚州 SantaClara 的英特尔公司在 1998 年 5 月 4 日公布的 Accelerated Graphics PortInterface Specification，revision 2.0。可替换地，点对点互连 439 可以耦合这些组件。

[0076] 进而，芯片组 490 可以经由接口 496 耦合到第一总线 416。在一个实施例中，第一总线 416 可以是外围组件互连 (PCI) 总线（如 1995 年 6 月的 PCILocal Bus Specification，Production Version，Revision 2.1 所定义的），或者是诸如 PCI Express 总线或者其它第三代输入 / 输出 (I/O) 互连总线之类的总线，但是本发明的范围并非局限于此。如图 7 所示，各种 I/O 设备 414 可以耦合到第一总线 416，还有总线桥 418 将第一总线 416 耦合到第二总线 420。在一个实施例中，第二总线 420 可以是少针脚型 (LPC) 总线。可以耦合到第二总线 420 的各种设备包括例如键盘 / 鼠标 422、通信设备 426 以及数据存储单元 428，所述数据存储单元在一个实施例中可以包含代码 430。此外，音频 I/O 424 可以耦合到第二总线 420。

[0077] 采用上述机制来收集剖析信息顾及到低开销的在线剖析以及动态编译。因此，所述轻量化控制让步机制及其对于用户级中断的应用的实施例可以完全绕过 OS，以对 OS 透明的方式实现了更细粒度的通信和同步。因此在各种实施例中，不需要 OS 的支持来收集和使用剖析信息，避免了 OS 进行编程和采用中断。因此，所述让步机制不需要设备驱动程序，不需要新的 OS 应用编程接口 (API)，以及不需要上下文切换代码中的新指令。使用本发明的实施例获得的剖析数据可以用于动态优化，例如，重新安排代码和数据以及插入预取。

[0078] 所述实施例可以以代码来实现，并且可以存储在存储介质上，所述存储介质上存储有指令，所述指令能够用来对系统进行编程以执行所述指令。所述存储介质可以是任意一种媒体，例如，盘片、半导体设备（诸如只读存储器 (ROM)、随机存取存储器 (RAM)、可擦写可编程只读存储器 (EPROM)、闪速存储器、电可擦写可编程只读存储器 (EEPROM)）、磁卡或者光卡、或者适于存储电子指令的任何其它类型的媒体。

[0079] 虽然已经针对有限数量的实施例描述了本发明，但是本领域技术人员将会理解，从中可以有各种修改和变型。所附权利要求意在覆盖落入本发明的实质和范围内的所有这样的修改和变型。

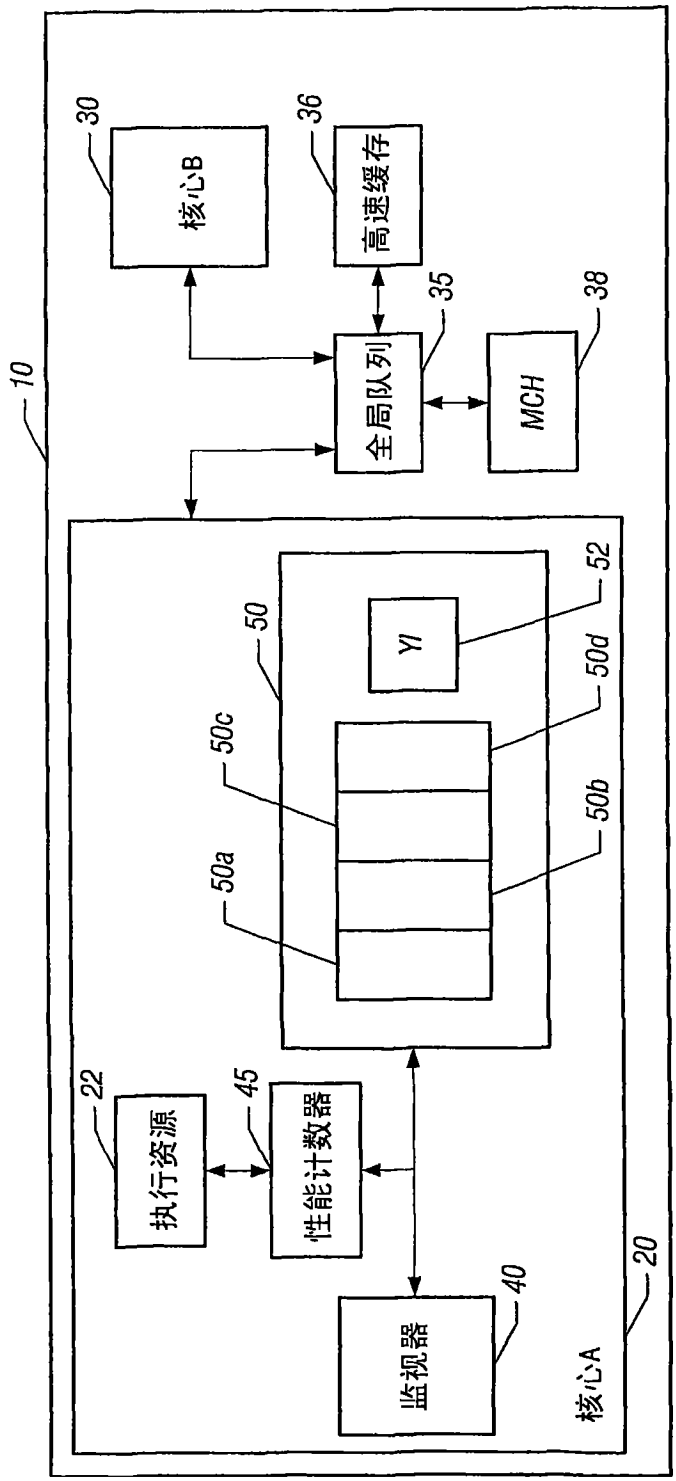
10
30
核心B
36
高速缓存
35
全局队列
38
MCH
50
52
YI
50d
50c
50b
50a
22
执行资源
45
性能计数器
40
监视器
20
核心A

图 1

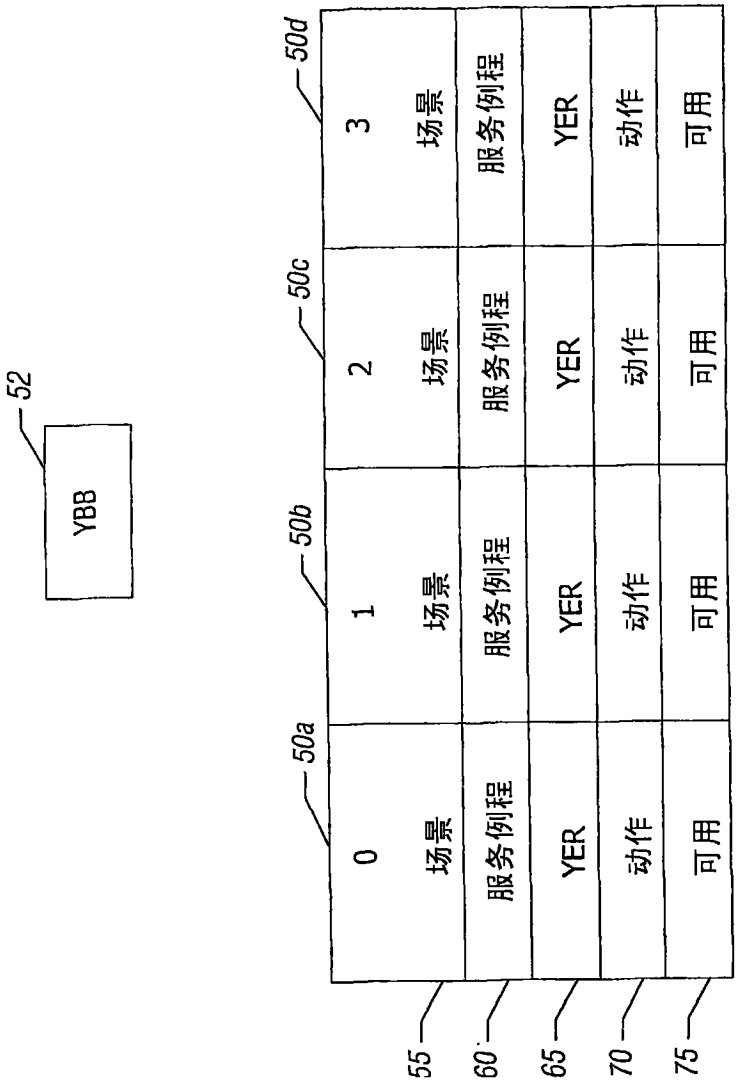

图2

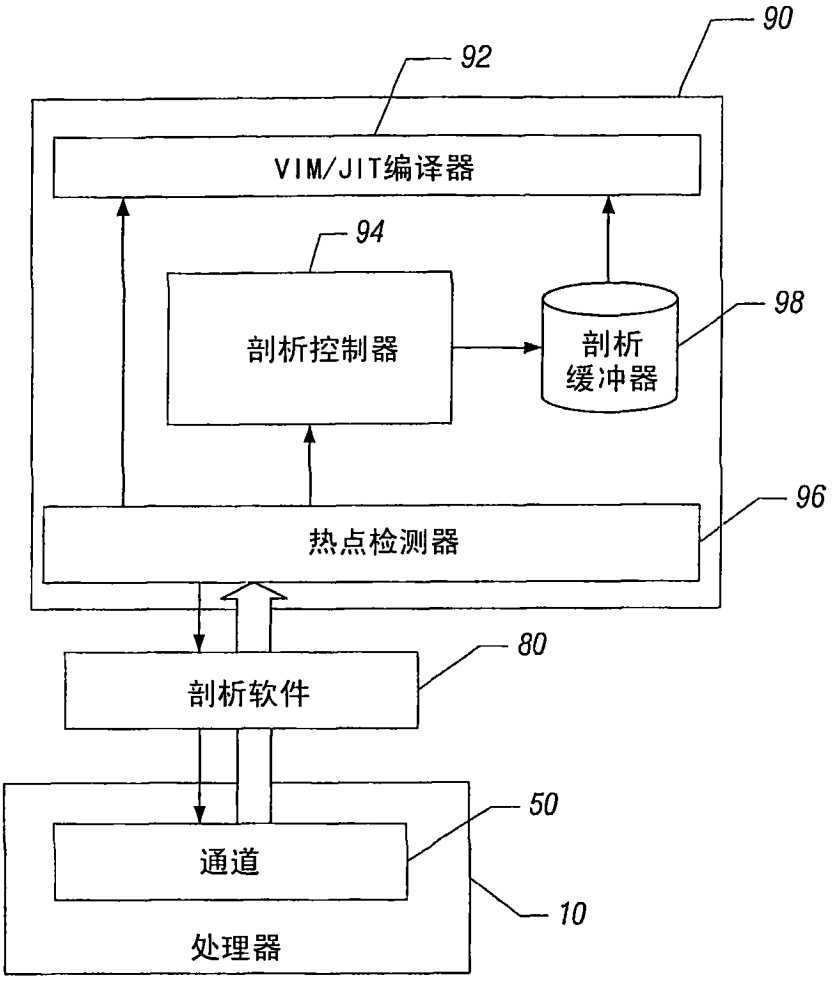

图 3

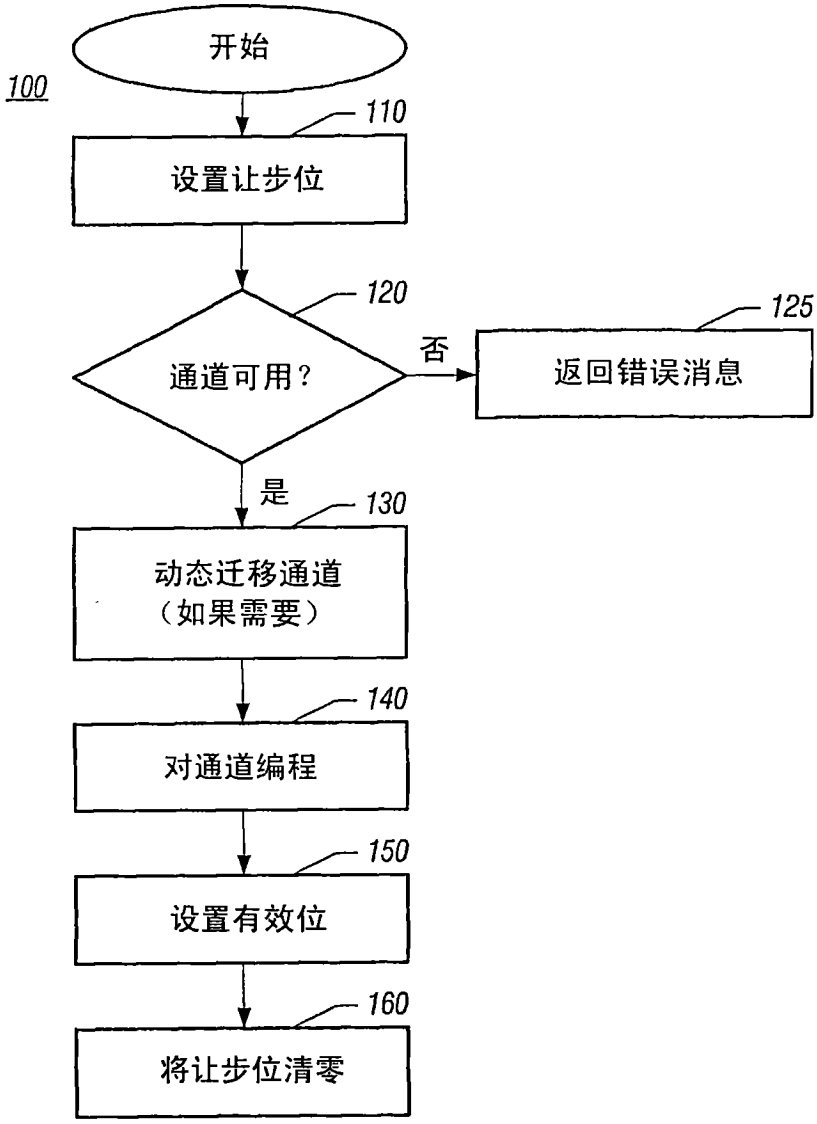
开始
100
110
设置让步位
120
通道可用？
否
125
返回错误消息
是
130
动态迁移通道
（如果需要）
140
对通道编程
150
设置有效位
160
将让步位清零

图 4

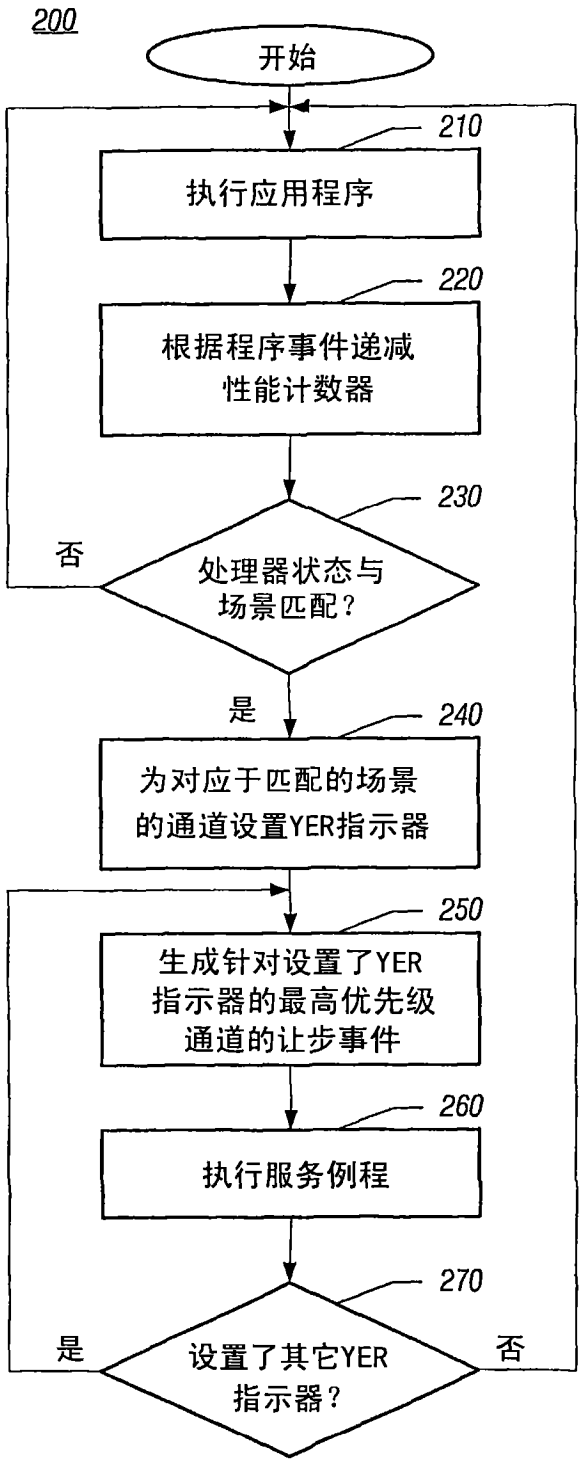
200
开始
210
执行应用程序
220
根据程序事件递减性能计数器
230
处理器状态与场景匹配？
否
是
240
为对应于匹配的场景的通道设置YER指示器
250
生成针对设置了YER指示器的最高优先级通道的让步事件
260
执行服务例程
270
设置了其它YER指示器？
是
否

图 5

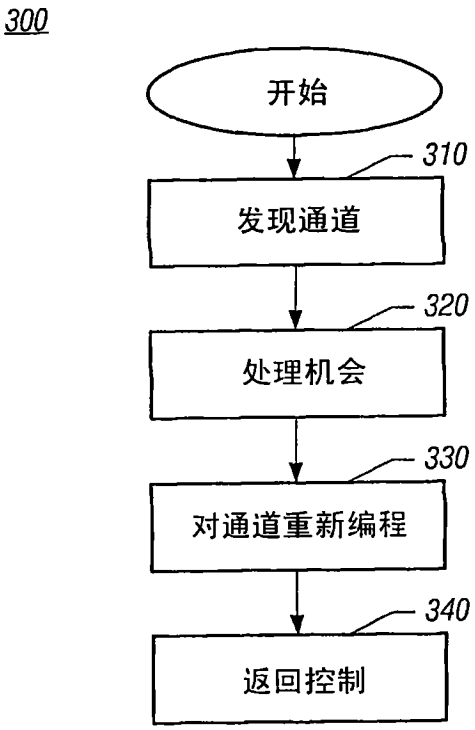
300
开始
310
发现通道
320
处理机会
330
对通道重新编程
340
返回控制

图 6

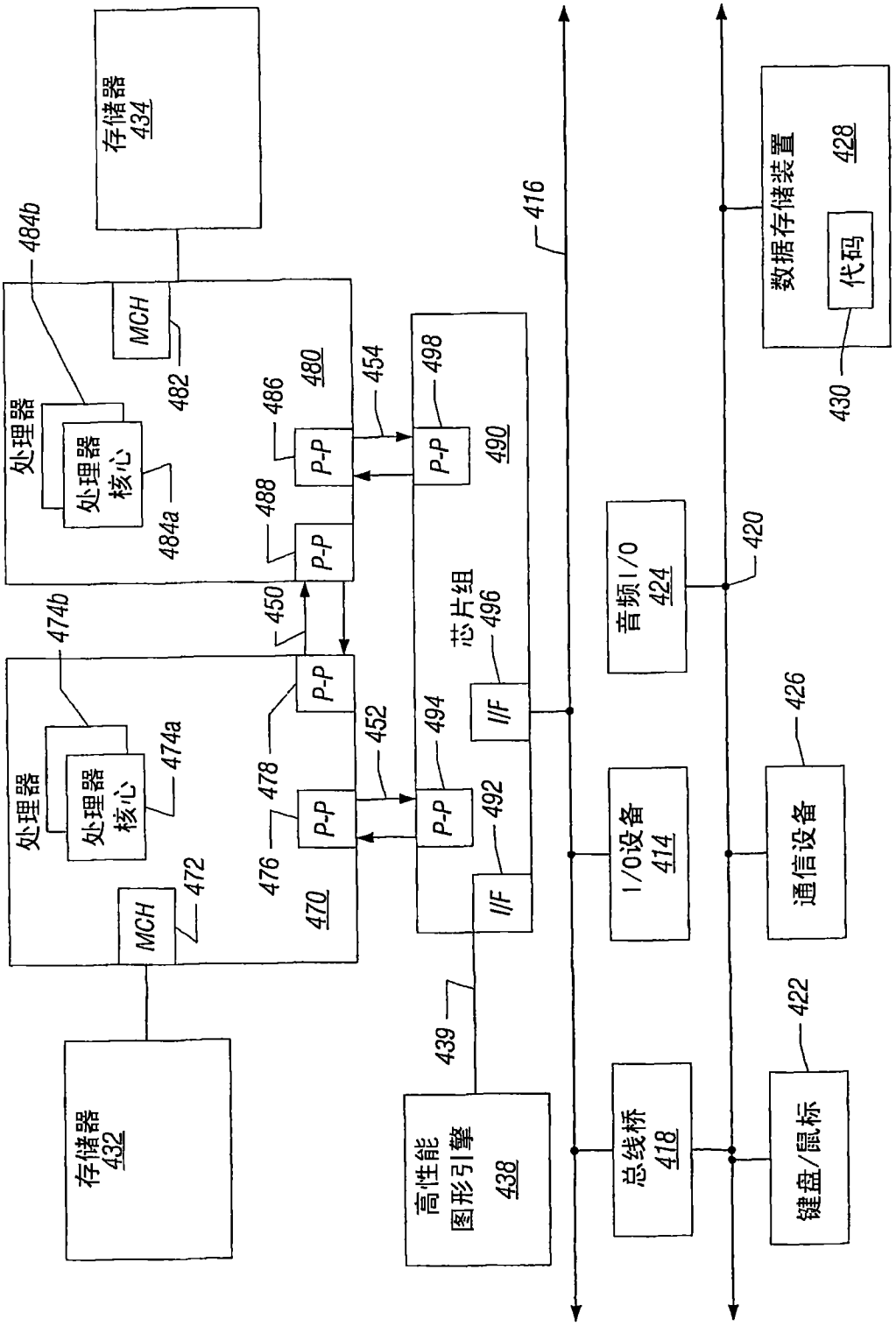
存储器
434
484b
MCH
482
处理器
处理器
核心
484a
486
480
P-P
454
498
P-P
490
488
P-P
450
474b
处理器
处理器
核心
474a
P-P
478
452
494
P-P
芯片组
496
I/F
492
476
P-P
470
I/F
MCH
472
存储器
432
439
高性能
图形引擎
438
416
音频I/O
424
420
数据存储装置
428
代码
430
I/O设备
414
通信设备
426
总线桥
418
键盘/鼠标
422

图7