(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 103119556 A
(43)申请公布日 2013.05.22

(21)申请号 201180046077.7
(22)申请日 2011.09.26
(30)优先权数据
12/890,639 2010.09.25 US
(85)PCT申请进入国家阶段日
2013.03.25
(86)PCT申请的申请数据
PCT/US2011/053285 2011.09.26
(87)PCT申请的公布数据
WO2012/040715 EN 2012.03.29
(71)申请人 英特尔公司
地址 美国加利福尼亚
(72)发明人 M·小布雷特尼茨 Y·吴 C·王 E·博林 S·胡 C·B·齐勒斯
(74)专利代理机构 永新专利商标代理有限公司 72002
代理人 张晰 王英
(51)Int.Cl.
G06F 9/30(2006.01)
G06F 9/305(2006.01)
G06F 9/06(2006.01)
G06F 15/76(2006.01)

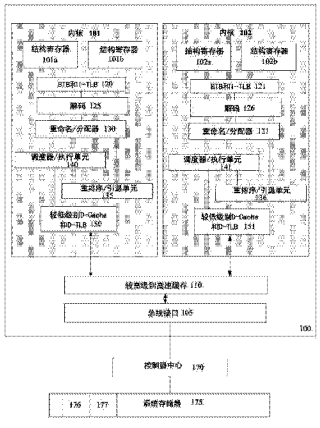
权利要求书3页 说明书30页 附图14页

(54)发明名称

用于提供在原子区内的条件提交的决策机制的装置、方法和系统

(57)摘要

在本文中描述了一种用于条件提交和/或推测检查点检查事务的装置和方法，其潜在地导致动态调整事务的大小。在二进制代码的动态优化期间，插入事务来提供存储器排序保护，这使得动态优化器能更积极地优化代码。并且条件提交使得能够有效执行动态优化代码，同时试图防止事务耗尽硬件资源。同时推测检查点使得能够在事务中止时快速并有效地恢复。处理器硬件适用于支持动态调整事务的大小，例如包括识别条件提交指令的解码器、推测检查点指令，或二者。并且处理器硬件还适用于执行操作来支持条件提交或执行推测检查点来响应解码这些指令。



CN 103119556 A

1. 一种用于优化代码的装置，包括：

保存程序代码的存储器；以及

处理器，包括

硬件资源，其适用于支持事务执行并且提供所述硬件资源的可用性表示；以及

执行逻辑，其适用于执行所述程序代码，以使所述处理器基于所述硬件资源的可用性表示而动态地调整包括所述程序代码的优化部分的事务区域的大小。

2. 根据权利要求 1 所述的装置，其中，适用于执行所述程序代码，以使所述处理器基于所述硬件资源的可用性表示而动态地调整包括所述程序代码的优化部分的事务区域的大小的执行逻辑包括：所述执行逻辑适用于执行在所述事务区域结束之前在所述程序代码内的条件提交指令，所述条件提交指令在所述事务区域结束之前提交所述事务以响应指示资源不足以完成所述事务区域的执行的所述硬件资源的可用性表示。

3. 根据权利要求 2 所述的装置，其中，所述存储器还适用于保存动态优化代码，并且其中，所述执行逻辑还适用于执行所述动态优化代码来优化部分代码以获得所述程序代码的优化部分，所述优化部分包括在运行时期间在所述事务区域的结束之前插入所述条件提交指令。

4. 根据权利要求 1 所述的装置，其中，适用于执行所述程序代码以使所述处理器基于所述硬件资源的可用性表示而动态地调整包括所述程序代码的优化部分的事务区域的大小的执行逻辑包括：所述执行逻辑适用于执行来自所述事务区域的事务写入，其中，所述事务区域将回滚到最近的检查点，并且响应于所述事务写入溢出硬件存储缓冲器而被提交。

5. 根据权利要求 1 所述的装置，其中，从组中选择所述存储器，所述组包括：处理器上的高速缓存存储器、直接与所述处理器耦合的系统存储器，以及间接地与所述处理器耦合的系统存储器。

6. 一种装置，包括：

适用于对与事务相关联的区域检查指令进行解码的解码逻辑；以及

适用于支持执行所述事务的硬件，其中，所述硬件还适用于提供使用度量存储元件的硬件使用度量的表示来响应对所述区域检查指令进行解码的所述解码逻辑，其中，所述使用度量存储元件适用于用来确定在所述事务结束之前是否执行所述事务的条件提交。

7. 根据权利要求 6 所述的装置，其中，从组中选择所述存储器，所述组包括：推测高速缓存存储器、存储缓冲器、推测寄存器文件，以及推测检查点寄存器文件。

8. 根据权利要求 6 所述的装置，其中，所述区域检查指令包括预期使用度量的表示；并且其中，所述使用度量存储元件适用于用来确定在所述事务结束之前是否执行所述事务的条件提交包括：所述硬件还适用于将所述期望使用度量的表示与所述待提供到所述使用度量存储元件的所述硬件使用度量的表示进行比较来获得使用比较结果；并且基于所述使用比较结果确定在所述事务结束之前是否要执行所述事务的所述条件提交。

9. 根据权利要求 8 所述的装置，其中，所述区域检查指令还包括分支目标地址；所述硬件适用于基于所述使用比较结果，使用预定义的算法，来确定在所述事务结束之前是否要执行所述事务的所述条件提交，并且其中，响应于基于所述使用比较结果、使用所述预定义的算法确定在所述事务结束之前将执行所述事务的所述条件提交，执行适用于跳转到所述分支目标地址。

10. 根据权利要求 8 所述的装置，其中，所述硬件适用于基于所述使用比较结果，使用预定义的算法，来确定在所述事务结束之前是否要执行所述事务的所述条件提交，并且将基于所述使用比较结果、使用预定义的算法确定在所述事务结束之前是否要执行所述事务的所述条件提交的表示展现给与所述事务相关联的条件代码，当执行所述条件代码时，用于：

响应于所述硬件展现确定在所述事务结束之前要执行所述事务的所述条件提交的表示，跳转到条件提交分支地址，

提交所述事务；以及

开始新的事务。

11. 根据权利要求 6 所述的装置，其中，所述使用度量存储元件包括适用于被与所述事务相关联的条件代码读取的寄存器，当执行所述条件代码时，评估关于硬件的软件期望使用度量的所述硬件使用度量，并且基于关于所述软件期望使用度量的所述硬件使用度量的评估，确定在所述事务结束之前是否要执行所述事务的所述条件提交。

12. 根据权利要求 11 所述的装置，其中，当执行所述条件代码时，响应于基于关于所述软件期望使用度量的所述硬件使用度量的评估确定在所述事务结束之前将执行所述事务的所述条件提交，所述条件代码还跳转到提交路径，在所述事务结束之前提交所述事务，并开始新的事务。

13. 一种装置，包括：

解码逻辑，其适用于对与在事务中的条件提交点相关联的条件提交指令进行解码，所述条件提交指令引用目的地地址；以及

硬件，所述硬件适用于响应于所述解码逻辑对所述条件提交指令进行解码，确定硬件资源是否包括足够可用空间来支持所述事务区域的执行，并且跳转执行到所述目的地地址来响应确定所述硬件没有包括足够可用空间来支持所述事务区域的执行。

14. 根据权利要求 13 所述的装置，其中，所述硬件资源包括存储器设备，并且其中，所述硬件还适用于基于在所述事务中的第二条件提交点之前将要使用的条目的期望数量和在所述存储器设备中的可用条目的数量，确定所述存储器设备是否包括足够的可用空间来支持所述事务区域的执行，其中，响应于可用条目的数量小于期望条目的数量，所述硬件适用于确定存储器设备没有包括足够的可用空间来支持所述事务区域的执行。

15. 根据权利要求 13 所述的装置，其中，所述条件提交指令还在所述事务中的所述第二条件提交点之前引用将要使用的条目的期望数量。

16. 一种系统，包括：

适用于保存程序代码的存储器，所述程序代码包括具有优化代码的原子区，所述优化代码包括在所述原子区结束之前的条件提交指令；以及

与所述存储器耦合的处理器，所述处理器包括解码逻辑和硬件，所述解码逻辑适用于识别所述提交指令，所述硬件适用于响应所述解码逻辑识别所述条件提交指令，来确定在所述原子区结束之前是否应提交所述原子区。

17. 根据权利要求 16 所述的系统，其中，所述硬件适用于响应所述解码逻辑识别所述条件提交指令来确定在所述原子区结束之前是否应提交所述原子区包括：所述硬件适用于跟踪在所述处理器中将用于执行所述原子区的硬件资源的可用数量，确定将要用于执行一

部分所述原子区的所述硬件资源的期望数量，以及响应于硬件资源的可用数量小于所述硬件资源的期望数量而确定在所述原子区结束之前应提交所述原子区。

18. 根据权利要求 16 所述的系统，其中，所述条件提交指令包括引用将用于执行一部分所述原子区的硬件资源的期望数量，并且其中，所述硬件适用于响应所述解码逻辑识别所述条件提交指令来确定是否在所述原子区结束之前应提交所述原子区包括：所述硬件适用于跟踪在所述处理器中将用于执行所述原子区的硬件资源的可用数量，以及响应于硬件资源的可用数量小于所述硬件资源的期望数量而确定在所述原子区结束之前应提交所述原子区。

19. 根据权利要求 18 所述的系统，其中，从组中选择所述硬件资源，所述组包括：高速缓存存储器、推测高速缓存存储器、存储缓冲器、寄存器文件、推测寄存器文件，以及推测检查点寄存器文件。

20. 根据权利要求 16 所述的系统，其中，从组中选择所述存储器，所述组包括：同步动态随机存取存储器（SDRAM）、只读存储器（ROM）和闪存。

21. 一种装置，包括：

适用于执行包括循环的事务的执行逻辑；

适用于对所述循环的迭代次数进行计数的计数器；以及

适用于响应所述迭代次数达到迭代阈值而在所述事务结束之前启动提交所述事务的硬件。

22. 根据权利要求 21 所述的装置，其中，所述迭代阈值是基于所述循环的执行分析而动态调整的。

23. 根据权利要求 21 所述的装置，其中，所述迭代阈值初始被设定为默认值，在所述硬件响应于所述计数器达到所述默认值而在所述事务结束前启动所述事务的提交之前，响应没有因为硬件限制而发生回滚而增加所述默认值；在所述硬件响应于所述计数器达到所述默认值而在所述事务结束前启动所述事务的提交之前，响应因为硬件限制而发生回滚而减小所述默认值；并且其中，所述默认值从一组值中选择，所述一组值包括：在所述循环中的推测存储的数量，由在所述循环的迭代期间将使用的估计条目的数量所划分的在存储器设备中的可用条目的数量，来自软件的开始值，以及在以前的所述循环执行期间因为硬件限制而发生回滚之前的循环迭代数量。

24. 根据权利要求 21 所述的装置，其中，所述计数器适用于对所述循环的迭代次数进行计数包括：所述计数器适用于被设定为默认值并且向下计数到零，并且其中，所述硬件适用于响应迭代的次数达到迭代阈值而在所述事务结束之前启动所述事务的提交包括：所述硬件适用于响应所述计数器达到零而在所述事务结束之前启动所述事务的提交。

用于提供在原子区内的条件提交的决策机制的装置、方法和系统

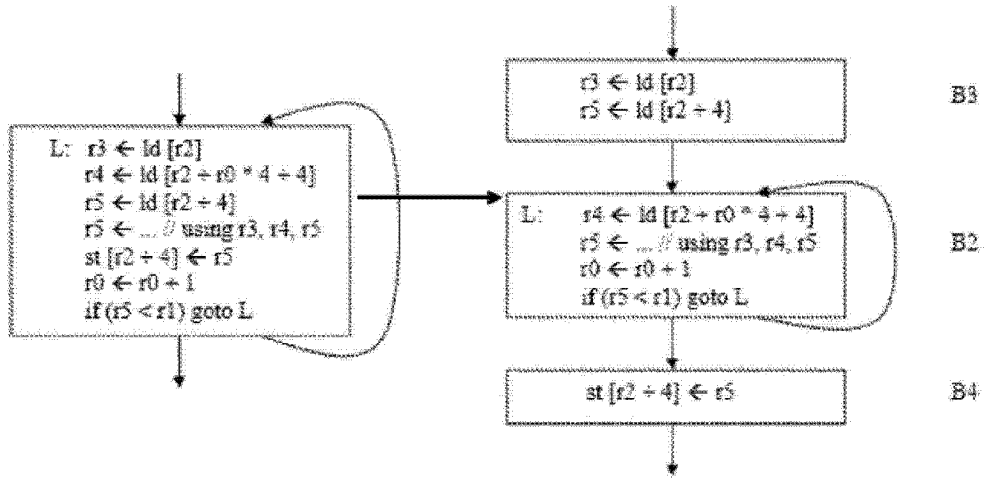
技术领域

[0001] 本发明涉及处理器领域，并且更具体地，涉及在处理器上的代码优化和执行。

背景技术

[0002] 半导体处理和逻辑设计的进步已允许存在于集成电路设备上的逻辑数量的增长。在以前，在单线程处理器上，因为不担心收到执行的其他线程的干扰，所以允许对代码(例如二进制代码)的优化可以是过度积极的。但是，计算机系统配置已经从在系统中的单个或多个集成电路发展到在独立的集成电路上存在多个内核、多个硬件线程，和多个逻辑处理器。处理器或集成电路通常包括单个物理处理器模具，其中处理器模具可以包括任意数量的内核、硬件线程，或逻辑处理器。在集成电路上的处理元件(内核、硬件线程，和逻辑处理器)数量的不断增长使得能够并行完成更多的任务。这种从单线程处理器到更多并行的、多线程执行的发展已导致对代码优化的限制。

[0003]



[0004] 伪代码 A：单线程代码优化的实施例

[0005] 例如，伪代码 A 示出了二进制代码的优化，其中通过部分冗余加载消除(PRLE)优化将从存储器在 [r2] 和 [r2+4] 处的加载提升到循环外至头部块(B3)。并且通过部分死存储消除(PDSE)优化将在存储器在 [r2+4] 处的存储下沉到循环外至尾部块(B4)。可以在单线程的环境中进行优化。但是，在多线程应用中，其他线程可以在循环执行期间从存储器的 [r2] 或 [r2+4] 读取或向其写入，这因为改变了存储器操作的执行顺序而潜在地导致无效执行。

附图说明

[0006] 通过示例方式来说明本发明，并且并非意图用附图中的图限制本发明。

[0007] 图 1 示出了适用于支持原子执行和原子区动态调整大小的多处理元件处理器的逻辑表示的实施例。

[0008] 图 2a 示出了包括基于硬件资源限制提供动态调整事务大小的优化代码的方法的流程图的实施例。

[0009] 图 2b 示出了用于插入条件提交代码的图 2a 的流程图的实施例。

[0010] 图 3a 示出了用于在执行期间动态调整事务大小的方法的流程图的实施例。

[0011] 图 3b 示出了用于确定在条件提交点是否存在足够的硬件资源来继续执行的图 3a 的流程图的实施例。

[0012] 图 4 示出了适用于支持动态调整事务大小的硬件的逻辑表示的实施例。

[0013] 图 5 示出了适用于支持动态调整事务大小的硬件的逻辑表示的另一实施例。

[0014] 图 6 示出了适用于支持动态调整事务大小的硬件的逻辑表示的另一实施例。

[0015] 图 7a 示出了包括提供用于在事务内的推测检查点的优化代码的方法的流程图的实施例。

[0016] 图 7b 示出了用于插入推测检查点代码的图 7a 的流程图的实施例。

[0017] 图 8 示出了在事务执行期间用于推测地在检查点检查(checkpoint)存储器的流程图的实施例。

[0018] 图 9 示出了适用于支持存储器的推测检查点的硬件的逻辑表示的实施例。

[0019] 图 10 示出了适用于支持寄存器文件的推测检查点的硬件的逻辑表示的另一实施例。

[0020] 图 11 示出了适用于支持高速缓存存储器的推测检查点的硬件的逻辑表示的另一实施例。

具体实施方式

[0021] 在下文的说明中，为了提供对本发明的完全理解而提出了多个特定细节，例如特定类型的处理器内核、特定的处理器配置、特定的指令类型、特定的硬件结构、特定的代码优化技术等的示例。但是，对于本领域的一个技术人员显而易见的是不采用这些特定细节也能实施本发明。在其他的实例中，为避免不必要地模糊本发明，没有详细描述公知的部件或方法，例如特定的和替换的处理器架构、用于所述算法的特定的逻辑电路 / 代码、特定的代码实现、特定的编译器细节，和微处理器的其他特定操作细节。

[0022] 在本文中描述的用于优化代码的方法和装置基于硬件限制使用动态大小的事务。具体地，代码的优化是讨论关于使用硬件限制的推测检查点和 / 或事务的条件提交。但是，在本文中描述的装置和方法并非如此有限，因为可以用任何形式的动态大小事务来实现它们。例如，代码优化可以动态地或静态地执行，并且可以在硬件、软件或其组合中执行。

[0023] 参考图 1，示出了包括多个内核的处理器的实施例。处理器 100 包括任意的处理器，例如微处理器、嵌入式处理器、数字信号处理器(DSP)、网络处理器，或执行代码的其他设备。在一个实施例中，处理器 100 包括至少两个内核 - 内核 101 和 102，其可以包括不对称内核和对称内核(示出的实施例)。但是，处理器 100 可以包括任意数量的处理元件，所述处理元件可以是对称的或非对称的。

[0024] 在一个实施例中，处理元件指的是线程单元、线程槽、处理单元、上下文、逻辑处理器、硬件线程、内核，和 / 或任意其他元件，所述元件能够保持处理器的状态，例如执行状态或架构状态。换句话说，在一个实施例中，处理元件指的是能够独立地与代码相关联的任意

硬件，所述代码例如为软件线程、操作系统、应用，或其他代码。一般地，物理处理器指的是集成电路，其潜在地包括任意数量的其他处理元件，例如内核或硬件线程。

[0025] 内核通常指的是位于集成电路上的能够维持独立的架构状态的逻辑，其中每一个独立维持的架构状态与至少一些专用的执行资源相关联。与内核相比，硬件线程通常指的是位于集成电路上的能够维持独立的架构状态的任意逻辑，其中独立维持的架构状态共享对执行资源的访问。由此可见，当某些资源被共享并且其他的资源被专用于架构状态时，线路在所命名的硬件线程和内核之间是重叠的。但是，操作系统经常将内核和硬件线程视为单独的逻辑处理器，其中操作系统能够独立地调度在每一个逻辑处理器上的操作。

[0026] 如图 1 中示出的物理处理器 110 包括两个内核，内核 101 和 102。在此处，认为内核 101 和 102 是对称内核，即具有相同配置、功能单元和 / 或逻辑的内核。在另一实施例中，内核 101 包括无序处理器内核，同时内核 102 包括顺序处理器内核。但是，可以独立地从任意类型的内核中选择内核 101 和 102，例如本地内核、软件管理内核、适用于执行本地指令集架构（ISA）的内核、适用于执行翻译的指令集架构（ISA）的内核、协同设计的内核，或其他已知内核。但是为了进行进一步的讨论，在下文进一步详细讨论了在内核 101 中示出的功能单元，同样的在内核 102 中的单元以类似的方式进行操作。

[0027] 如所述的，内核 101 包括两个硬件线程 101a 和 101b，其还可以被称为硬件线程槽 101a 和 101b。因此，在一个实施例中，软件实体（例如操作系统）潜在地将处理器 100 视为四个独立的处理器，即能够并发执行四个软件线程的四个逻辑处理器或处理元件。为了回避以上，第一线程与架构状态寄存器 101a 相关联，第二线程与架构状态寄存器 101b 相关联，第三线程与架构状态寄存器 102a 相关联，以及第四线程与架构状态寄存器 102b 相关联。如所示的，架构状态寄存器 101a 是架构状态寄存器 101b 的复制，因而能够存储用于逻辑处理器 101a 和逻辑处理器 101b 的独立的架构状态 / 上下文。在内核 101 中，其他更小的资源，例如在重命名分配器逻辑 130 中的指令指针和重命名逻辑也可被复制用于线程 101a 和 101b。一些资源，例如在重排序 / 引退单元 135 中的重排序缓冲器、ILTB120、加载 / 存储缓冲器和队列可以通过分区来共享。其他资源，例如通用的内部寄存器、页表基址寄存器、低级别数据高速缓存和数据 TLB115、执行单元 140，和无序单元 135 的一部分潜在地是完全共享的。

[0028] 处理器 100 经常包括其他资源，所述其他资源是完全共享的、通过分区共享的，或是处理元件专用的 / 专用于处理元件的。在图 1 中，示出了纯粹示例性的处理器的实施例，所述处理器具有具有说明性的处理器的逻辑单元 / 资源。注意处理器可以包括或忽略任意的这些功能单元，以及包括任意其他已知的未绘出功能单元、逻辑，或固件。如所示的，内核 100 包括简化的、代表性的无序（000）处理器内核。000 内核包括用于预测将被执行 / 采取的分支目标缓冲器 120 以及存储用于指令的地址转换条目的指令转换缓冲器（I-TLB）120。

[0029] 内核 101 还包括与取出单元 120 耦合以对取出的元素进行解码的解码模块 125。在一个实施例中，取出逻辑包括分别与线程槽 101a、101b 相关联的独立定序器。通常内核 101 与第一指令集架构（ISA）相关联，其定义 / 规定了在处理器 100 上的可执行指令。在此处，作为第一 ISA 一部分的机器代码指令通常包括指令（通常指操作码）的一部分，其引用 / 规定了将被执行的指令或操作。解码逻辑 125 包括电路，所述电路从这些指令的操作码中识别指令并将解码的指令在管道进行传递来用于如第一 ISA 所定义的处理。例如，如在下

面更详细讨论的，在一个实施例中，解码器 125 包括设计用来或适用于识别特定的、新的指令（例如条件提交指令和 / 或推测检查点指令）的逻辑。作为解码器 125 的结果或识别，架构或内核 101 采取特定的预定行动来执行与适当的指令相关联的任务。

[0030] 在一个示例中，分配器和重命名块 130 包括保留资源的分配器，所述资源例如为存储处理结果的指令的寄存器文件。但是，线程 101a 和 101b 是潜在能够无序执行的，其中分配器和重命名块 130 还保留其他资源，例如跟踪指令结果的重排序缓冲器。单元 130 还可以包括将程序 / 指令引用寄存器重命名为处理器 100 内部的其他寄存器的寄存器重命名器。重排序 / 引退单元 135 包括支持无序执行和无序执行指令在稍后顺序引退的部件，所述部件例如上面提到的重排序缓冲器、加载缓冲器，和存储缓冲器。

[0031] 在一个实施例中，调度器和执行单元块 140 包括调度在执行单元上的指令 / 操作的调度单元。例如，在具有可用浮点执行单元的执行单元的端口上调度浮点指令。也包括与执行单元相关联的寄存器文件来存储信息指令处理结果。示例性的执行单元包括浮点执行单元、整数执行单元、跳转执行单元、加载执行单元、存储执行单元，和其他已知的执行单元。

[0032] 较低级别数据高速缓存和数据转换缓冲器（D-TLB）150 与执行单元 140 相耦合。数据高速缓存用来存储最近使用的 / 操作的元素，例如数据操作数，其潜在地保持内存一致性状态。D-TLB 用来存储最近的虚拟 / 线性到物理地址的转换。如特定的示例，处理器可以包括用来将物理存储器分解为多个虚拟页面的页表结构。

[0033] 在此处，内核 101 和 102 共享对更高级别或更远离（further-out）高速缓存 110 的访问，高速缓存 110 用于高速缓存最近取得的元素。注意，更高级别或更远离指的是高速缓存级别增加或者更远离执行单元。在一个实施例中，更高级别高速缓存 110 是末级数据高速缓存—在处理器 100 的存储器层级中的末级高速缓存—例如第二或第三级别的数据高速缓存。但是更高级别高速缓存 110 并非限制于此，其可以与指令高速缓存相关联或包括指令高速缓存。跟踪高速缓存—一种指令高速缓存—相反可以在解码器 125 之后耦合来存储最近解码的跟踪。

[0034] 在绘出的配置中，处理器 100 还包括与在处理器 100 外部的设备进行通信的总线接口模块 105，处理器外部的设备例如为系统存储器 175、芯片组、北桥，或其他集成电路。存储器 175 可以专用于处理器 100 或被系统中的其他设备所共享。存储器 175 类型的常用示例包括动态随机存取存储器（DRAM）、静态 RAM（SRAM）、非易失性存储器（NV 存储器），和其他已知的存储设备。

[0035] 在一个实施例中，处理器 100 能够进行硬件事务执行、软件事务执行，或其组合或混合。事务也可以称为代码的临界或原子段 / 区，其包括将作为原子分组执行的指令、操作或微操作的分组。例如，可以使用指令或操作来划分事务或临界段。在一个实施例中，将在下面更详细地描述，这些指令是处理器 100 的硬件（例如，上述的解码器）所能识别的指令组的一部分，所述指令组例如指令集架构（ISA）。经常地，一旦将这些指令从高级语言编译为硬件可识别汇编语言，则包括解码器在解码阶段识别的操作代码（操作码），或指令的其他部分。

[0036] 一般地，在事务的执行期间，对存储器的更新在提交事务前不是全局可见的。作为一个示例，向位置的事务写入潜在地对本地线程是可见的，但是，响应于从另一线程的读

取，写入数据不会被转送直到提交了包括事务写入的事务。虽然事务仍然正在挂起，但是跟踪从存储器内加载或向存储器内写入的数据项 / 元素，这将在下文中更详细讨论。一旦事务到达提交点，如果没有检测到事务的冲突，则提交事务并使在事务期间做出的更新对全局可见。但是，如果在事务挂起期间事务是无效的，则中止事务并且潜在地重启事务而不使更新全局可见。结果是，如在本文中使用的事务的挂起是指已经开始执行并且没有提交或中止(即，挂起)的事务。

[0037] 软件事务存储器(STM)系统经常指执行访问跟踪、冲突解决，或通过执行软件代码之内或至少主要通过执行软件代码的其他事务存储器任务。在一个实施例中，处理器 100 能使用硬件 / 逻辑(即在硬件事务存储器(HTM) 系统内)执行事务。当实现 HTM 时，从架构和微架构角度二者来看存在大量的特定实现细节 ；本文中没有讨论大多数所述实现细节以避免不必要地模糊本发明。但是，为了说明目的公开了一些结构、资源和实现。但是，应注意这些结构和实现不是必需的并且可以使用具有不同实现细节的其他结构来进行扩增和 / 或替换这些结构和实现。

[0038] 作为组合，处理器 100 能够在无限的事务存储器(UTM)系统内执行事务，该系统试图使用 STM 和 HTM 系统二者的优点。例如，HTM 对于执行小的事务通常是快速并且高效的，因为其不依赖软件来执行所有的访问跟踪、冲突检测、验证和用于事务的提交。但是，HTM 通常仅能处理较小的事务，而 STM 能处理无限大小的事务。因此，在一个实施例中，UTM 系统使用硬件来执行较小的事务并且使用软件来执行对于硬件来说过大的事务。如能从下文的讨论中看出的，即使当软件正在处理事务时，也可以使用硬件来帮助并加速软件。此外，需要重点注意的是，也可以使用同一硬件来支持并加速纯 STM 系统。

[0039] 如上所述的，事务包括由在处理器 100 内的本地处理元件及潜在的其他处理元件二者进行的对数据项的事务存储器访问。在事务存储器系统中没有安全机制，这些访问中的一些将潜在地导致无效的数据和执行，即向数据的写入使读取无效，或无效数据的读取。结果是，如下文中所讨论的，处理器 100 潜在地包括逻辑来跟踪或监控存储器对数据项访问或数据项对存储器的访问以用于识别潜在的冲突，所述逻辑例如读监控器和写监控器。

[0040] 数据项或数据元素可以包括在任何粒度级别的数据，如由硬件、软件或两者组合的所定义的。数据、数据元素、数据项或其引用的示例的非详尽列表包括存储器地址、数据对象、类、一种动态语言代码类型的字段、一种类型的动态语言代码、变量、操作数、数据结构，和存储器地址的非直接引用。然而，对数据的任何已知的分组可以称为数据元素或数据项。上面示例中的一些，例如一种动态语言代码类型的字段和一种类型的动态语言代码是指动态语言代码的数据结构。为了说明，动态语言代码(例如来自 Sun 微系统有限公司的 Java™)是强类型语言。每一个变量均具有在编译时已知的类型。所述类型可以分为两类 ：原始类型(布尔的和数字的，例如，整数、浮点数)和引用类型(类、接口和数组)。引用类型的值是对对相的引用。在 JavaTM 中，由字段构成的对象可以是类实例或数组。对于类 A 的给定对象，习惯上使用符号 A::x 来指类型 A 的字段 x，并且 a.x 指类 A 的对象 a 的字段 x。例如，表达式可以表达为 a.x=a.y+a.z。在这里，加载字段 y 和字段 z，并且结果写入字段 x。

[0041] 因此，监控 / 缓冲存储器对数据项的访问可以在任意数据级别粒度执行。例如，在一个实施例中，在类型级别监控存储器访问数据。在这里，可以将向字段 A::x 的事务写入和字段 A::y 的非事务加载监控为对同一数据项(即类型 A)的访问。在另一实施例中，在字

段级别粒度执行存储器访问监控 / 缓冲。在这里，不将向字段 A::x 的事务写入和字段 A::y 的非事务加载监控作为对同一数据项的访问，因为它们引用了独立的字段。注意，可以考虑其他数据结构或编程技术来跟踪存储器对数据项的访问。作为示例，假设类 A 的对象的字段 x 和 y，即 A::x 和 A::y，指向类 B 的对象，被初始化来新分配对象，并且在初始化之后未被写入。在一个实施例中，向 A::x 所指向的对象的字段 B::z 的写入事务与 A::y 所指向的对象的字段 B::z 的非事务加载，没有被监控为对同一数据项的存储器访问。从这些示例推测，可以确定监控器可以在任意数据粒度级别执行监控 / 缓冲。

[0042] 在一个实施例中，处理器 100 包括监控器来检测或跟踪与数据项相关联的访问和潜在的随后冲突。作为一个示例，处理器 100 的硬件包括读监控器和写监控器来相应地跟踪被确定为受监控的加载和存储。作为示例，硬件读监控器和写监控器将在数据项粒度监控数据项，而不管在存储结构之下的粒度。在一个实施例中，通过跟踪在存储结构粒度关联的机制来使得数据项有界，以确保至少整个数据项是被适当地监控的。

[0043] 作为特定的说明性示例，读和写监控器包括与高速缓存位置相关联的属性来监控来自与这些位置相关联的地址的加载和向这些地址的存储，所述属性例如在较低级别数据高速缓存 150 内的位置（其可以包括推测高速缓存）。在此处，在发生对与高速缓存位置相关联的地址的读事件时，设置用于数据高速缓存 150 的高速缓存位置的读属性，来监控向同一地址的潜在冲突写入。在这种情况下，以类似方式操作用于写事件的写属性来监控潜在的冲突读取和向同一地址的写入。为深入这个示例，硬件能够基于监听对高速缓存位置的读和写来用读和 / 或写属性设置检测冲突，相应地指示高速缓存位置被监控。相反，在一个实施例中，设置读和写监控器或将高速缓存位置更新至缓冲状态，会导致监听（例如读请求或读所有权请求），其与在其他高速缓存中监控的地址的冲突被检测到。

[0044] 因此，基于设计，高速缓存一致性请求和高速缓存行的监控一致性状态的不同组合会导致潜在的冲突，例如高速缓存行在共享的读监控状态中保持数据项以及监听指示向该数据项的写请求。相反，高速缓存行保持在缓冲写状态中的数据项以及外部监听指示对该数据项的读请求可被认为是潜在冲突的。在一个实施例中，为检测访问请求和属性状态的这些组合，监听逻辑与冲突检测 / 报告逻辑以及状态寄存器相耦合以报告冲突，所述冲突检测 / 报告逻辑例如为用于冲突检测 / 报告的监控器和 / 或逻辑。

[0045] 但是，可以认为任意情况和场景的组合使事务无效。可被认为事务的非提交因素的示例包括：检测事务性访问存储器位置的冲突、丢失监控器信息、丢失缓冲数据、丢失与事务性访问的数据项相关联的元数据，以及检测其他无效事件，例如中断、环过渡，或明确的用户指令。

[0046] 在一个实施例中，处理器 100 的硬件以缓冲的方式保持事务的更新。如上所述的，在提交事务之前，事务写对全局是不可见的。但是，与事务写相关联的本地软件线程能够访问事务更新来用于随后的事务访问。作为第一示例，在处理器 100 中提供了独立的缓冲器结构来保持缓冲的更新，其能够提供对本地线程而非其他外部线程的更新。

[0047] 相比之下，作为另一示例，使用高速缓存存储器（例如数据高速缓存 150）来缓冲更新，同时提供相同的事务功能。在这里，高速缓存 150 能在缓冲一致性状态中保持数据项；在一种情况中，将新的缓冲一致性状态加入到高速缓存一致性协议（例如修改的专属分享无效（MESI）协议）中来形成 MESIB 协议。为响应用于缓冲数据项的本地请求—以缓冲一致

性状态保持的数据项，高速缓存 150 向本地处理元素提供数据项来确保内部事务的连续顺序。但是，响应于外部的访问请求，提供了错失响应来确保使事务性更新的数据项在提交前对全局是不可见的。此外，当高速缓存 150 的行被保持在缓冲一致性状态中并且被选中收回时，缓存的更新不会写回到更高级别的高速缓存存储器—缓冲更新不会扩散通过存储器系统，即在提交前不会对全局可见。相反，可以中止事务或将回收的行存储在数据高速缓存和更高级别的高速缓存存储器之间的推测结构中，例如牺牲高速缓存（victim cache）。在提交时，将缓冲行转换为已修改的状态来使数据项全局可见。

[0048] 注意，术语“内部的”和“外部的”通常相对于以下角度：与执行事务相关联的线程或处理共享高速缓存的元素的角度。例如，用于执行与执行事务相关联的软件线程的第一处理元素称作本地线程。因此，在上述的讨论中，如果对之前已被第一线程写入的地址进行存储或从其中加载，会导致接收到用于以缓冲一致性状态保持的地址的高速缓存行，接着因为第一线程是本地线程而将高速缓存行的缓冲版本提供给第一线程。相比之下，第二线程可以在同一处理器内的另一处理元素上执行，但是与负责在缓冲状态中保持的高速缓存行的事务执行无关—外部的线程；因此，从第二线程向地址的加载或存储错失了高速缓存行的缓冲版本，并且使用常用的高速缓存更换来从更高级别的存储器取回高速缓存行的未缓冲版本。

[0049] 在一个实施例中，处理器 100 能执行编译器 / 优化代码 177 来编译应用代码 176 以支持事务执行，以及潜在地优化应用代码 176。在这里，编译器可以插入操作、调用、函数和其他代码以启动事务执行。

[0050] 编译器经常包括将源文本 / 代码转化为目标文本 / 代码的程序或程序集。通常地，在多个阶段和多次遍数使用编译器完成程序 / 应用代码的编译并将高级别的程序语言代码转换为低级别机器或汇编语言代码。但是，仍然可以使用单遍编译器来用于简单编译。编译器可以使用任何已知的编译技术，并执行任何已知的编译操作，例如词法分析、预处理、解析、语义分析、代码生成、代码转换，和代码优化。如在本文中讨论的，事务执行和动态代码编译的结合潜在地导致启动更积极的优化，同时保持必要的存储器排序保护。

[0051] 较大的编译器经常包括多个阶段，但是经常大多数这些阶段被包括在两个通用阶段中：（1）前端，即通常可以进行句法处理、语义处理，和一些转换 / 优化，以及（2）后端，即通常可以进行分析、转换、优化，和代码生成。一些编译器是指中端，其示出了在编译器的前端和后端之间的描述模糊之处。结果是，关于插入、关联、生成，或编译器的其他操作可以用任何上述的阶段或遍数，及其任何已知的编译的阶段或遍数来进行。作为说明性的示例，编译器潜在地在编译的一个或多个阶段插入编译操作、调用、函数等，例如在编译的前端阶段插入调用 / 操作，并接着在事务的存储器转换阶段期间将调用 / 操作变为较低级别代码。注意，在动态编译期间，编译器代码或动态优化代码可以插入这些操作 / 调用，以及在运行时期间优化用于执行的代码。作为特定的说明性示例，可以在运行时期间动态地优化二进制代码（已编译的代码）。在这里，程序代码可以包括动态优化代码、二进制代码，或两者的组合。

[0052] 无论如何，在一个实施例中，不管编译器的执行环境和动态或静态特性，编译器编译程序代码来使得能够执行事务和 / 或优化程序代码段。因此，在一个实施例中引用程序代码的执行是指：（1）编译器程序或优化代码优化器的执行动态地或静态地编译主程序代

码以维持事务结构、执行其他与操作相关的事务，或优化代码；(2)主程序代码的执行包括事务操作/调用，例如已被优化/编译的应用代码；(3)与主程序代码相关联的其他程序代码(例如库)的执行，或(4)以上的组合。

[0053] 经常在软件事务存储器(STM)系统内，将使用编译器来插入一些操作、调用和与待编译的应用代码内联的其他代码，同时在库内提供了其他操作、调用、函数和代码。这潜在地提供了库分配器的能力来优化和更新库，而不必重新编译应用代码。作为特定的示例，在事务的提交点可以将提交功能的调用内联插入到应用代码内，同时在可更新的库中独立地提供了提交功能。此外，在何处放置特定的操作和调用的选择潜在地影响了应用代码的效率。

[0054] 如在上面的背景部分中陈述的，在多线程系统中的代码的积极优化对于存储器排序问题是潜在危险的。但是，在一个实施例中，代码优化与事务的存储器保护相结合来允许进行积极优化，同时保持存储器排序保护。在这里，可以将包括已优化代码的原子区插入到程序代码中，使得在执行已优化代码时，事务保护确保不会有存储器排序违例。结果是，能积极优化已优化的代码，并且原子区确保检测到存储器排序违例，所以不会提交该区。

[0055] 但是，原子区和代码优化的组合还可以从进一步修改中获益。因此，在一个实施例中，处理器 100 能基于在处理器 100 内的硬件资源的可用性动态地调整在程序代码 176 中的事务的大小。一般地，任一原子区是完全提交或中止的。但是，在这个示例中，当存在较低资源或不充分的资源来完成事务(或在事务内的一部分代码)的执行时，可以在事务的终点之前提交事务，即动态地调整为较小的事务。作为说明性的示例，假设使用高速缓存存储器 150 来保存试验、事务信息连同相关联的事务跟踪信息。在这个场景中，当高速缓存 150 在可用的高速缓存条目变低或溢出(选择事务性的访问行用于收回和牺牲高速缓存已满)时，可以在此时提交正在运行的事务，这致使试验信息变得全局可见。接着可以从提交点到原始事务的终点重新启动新的事务。因此，事务的硬件资源—在这个示例中高速缓存存储器 150—是不受约束的。并且事务能以两个更小的硬件事务完成，而不是回滚整个事务或将事务扩展到例如在 UTM 系统的软件中。

[0056] 因此，在一个实施例中，处理器 100 包括硬件来支持事务存储器系统，无论是 HTM、STM，或 UTM，例如以下任意硬件的组合：解码器、高速缓存存储器、推测存储结构、跟踪机构、存储缓冲器、寄存器文件、检查点存储机构，和任意其他已知支持事务执行的硬件。此外，处理器 100 还包括适用于跟踪、提供，或指示可用性、使用，或其表示的硬件/逻辑，使得硬件支持事务执行。作为第一示例，使用度量(硬件使用的表示)包括在存储结构(例如高速缓存存储器、牺牲高速缓存、存储缓冲器，或加载缓冲器)中多个可用的条目，或相反占用的条目。作为另一示例，使用度量可以包括事件的发生，例如存储器溢出、中断事件，或条目的收回。但是，可以使用任意的使用度量，无论其是实际的还是抽象的。

[0057] 作为更抽象的使用度量的示例，假设计数器对在代码内的循环迭代的数目进行计数，并且当计数器到达阈值时提交事务。在这里，阈值可以是基于代码分析随时间动态调整的，例如当因为硬件资源不足而发生事务中止时减小阈值。在那种情况下，不提供硬件资源的精确的、实际使用或特定事件。但是，通过计数器阈值的动态调整，硬件在硬件资源耗尽之前(即，在因为高的资源使用而执行中止或回滚之前)会必要地估算循环的数量。结果是，在一个实施例中，因为硬件估算正在估算用于代码执行的资源可用性，所以这样的硬件

估算是指用于硬件资源的使用或使用表示。

[0058] 在一个实施例中，在处理器 100 中的硬件能异步地确定何时要动态地重新调整事务大小。例如，当硬件资源的使用高时，处理器 100 可以提交事务并能够从在处理器 100 上执行的程序代码 176 的角度透明地重新启动另一事务。在这里，程序代码 176 包括执行逻辑 140 执行的事务。并且从程序代码的角度来看，事务是无缝执行的。但是，从硬件的角度来看，资源，例如具有高使用（溢出）的存储缓冲器，使得在事务终点之前硬件提交事务，在那个提交点重新启动第二事务，并接着在原始的事务终点提交第二事务。

[0059] 在另一实施例中，使用硬件和固件 / 软件的组合来执行事务的动态调整大小。在这里，处理器 100 包括支持事务执行和跟踪这些资源使用 / 可用的硬件资源。并且当执行来自程序代码 176 的条件代码时，致使执行逻辑 140 基于那些硬件资源使用 / 可用进行动态调整大小（在终点之前提交事务）。本质上，资源使用的检查和潜在的条件提交是同步执行的—作为软件指令的结果—而非独立于硬件执行特定的软件（异步地）。

[0060] 例如，执行逻辑 140/141 执行动态优化代码 177（其可以是动态编译器代码的一部分）来在处理器 100 的运行时期间动态地编译 / 优化程序代码 176。在这些编译 / 优化期间，将原子区连同在该原子区内的条件提交代码插入到程序代码 176 段中。执行逻辑 140/141 可以接着动态地优化代码 176 并在运行时期间动态地执行优化的代码 176。特别地，执行逻辑 140/141 执行原子区和在其中的优化代码。为响应解码阶段 125 遇到 / 解码条件提交代码，确定了硬件资源使用。注意，可能在之前已跟踪了所述使用 / 可用，但为响应条件提交代码，接着报告 / 评估所述使用。接着可以基于硬件资源的可用 / 使用对是否提交原子区做出决策。

[0061] 在一个实施例中，响应于条件代码，由硬件执行所述基于资源使用是否提交的决策。换句话说，硬件可以独立地评估硬件使用并确定使用是否足够高到致使早期提交。作为示例，条件提交代码包括解码器 125 可识别的条件提交指令。条件提交指令，或区域检查指令，包括分支目标地址。并且为响应使用解码逻辑 125 解码条件提交指令，硬件确定硬件资源使用是否过高，或可替换地，确定是否存在不充分的资源。如果使用过高或存在不充分的资源，则执行逻辑 140 将执行跳转到分支目标地址。

[0062] 在一个实施例中，硬件基于预确定的算法来确定是否使用过高或存在不充分的资源。例如，当硬件资源包括存储缓冲器时，使用过高包括使用了预确定数量的存储缓冲器条目或发生了溢出（没有可用的存储缓冲器条目）。硬件还可以基于以前的执行（代码分析）来估算代码的预期使用，并使用该估算连同当前的硬件使用来确定是否存在足够的资源来继续执行而不会进行条件提交。

[0063] 可替换地，条件提交指令还可以包括预期使用。并且硬件将预期使用与硬件使用进行比较来确定是否存在不充分资源。例如，假设将条件代码 176 插入到在程序代码 176 的原子区内的循环中。结果是，在每次循环的迭代中执行条件代码。在这里，条件提交指令在循环的迭代期间引用将要使用的预期数量的存储缓冲器条目，这可以基于由代码或代码的动态分析估算的将接触到的唯一存储缓冲器条目的数量。为响应解码逻辑 125 解码条件提交指令，条目的预期数量与处理器 100 的硬件确定的存储缓冲器中的可用条目的数量相比较。如果预期条目的数量大于可用条目的数量，则执行跳转到条件提交指令引用的分支目标地址。分支目标地址可以包括引用在程序代码 176 内的代码或其他代码（例如库代码）

的地址，来执行原子区的早期提交并重新启动第二原子区。

[0064] 在另一实施例中，软件确定何时硬件使用过高，或硬件可用过低。在这个示例中，处理器 100 包括存储元件，例如寄存器，其保存硬件使用的表示（硬件使用度量）。在这里，条件代码包括加载 / 读取使用度量、评估使用度量，并确定是否执行早期提交的各操作。如果将要执行早期提交，则条件代码包括跳转操作，当执行逻辑 140 执行跳转操作时将执行跳转到分支目标地址，其提交当前事务并可以开始另一原子区。

[0065] 注意，硬件和软件可以执行类似的评估。但是，硬件方案通过允许硬件完全地处理早期提交或仅接收条件提交指令而潜在地使得代码紧凑。但是，允许软件执行评估在确定何时执行早期提交时提供了更多的灵活性。结果是，可以基于实现和期望的优点来使用任意在硬件和软件之间的组合梯度来确定何时硬件使用 / 可用是过高或过低的。

[0066] 图 1 示出了具有表示不同模块、单元，和 / 或逻辑的示例性处理器的抽象的、逻辑的视图。但是，注意使用了在本文中描述的方法和装置的处理器不需要包括示出的单元。并且，处理器可以忽略一些或所有示出的单元。此外，图 1 仅绘出了两个内核；但是，处理器可以包括任意数量的内核，例如同一类型的多个内核，及多于两个的每个都为不同类型的内核。

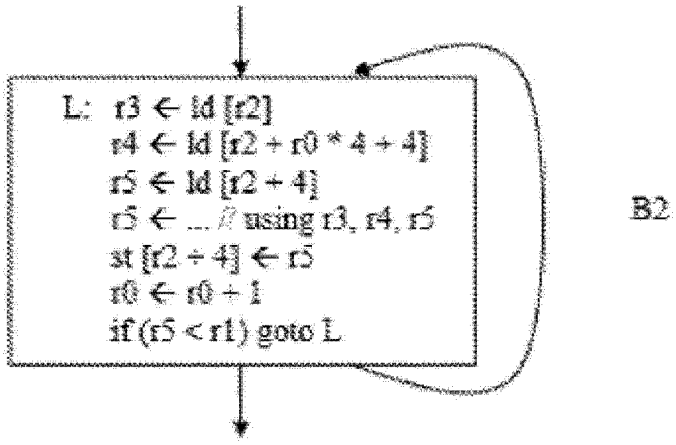
[0067] 图 1 示出了以点对点方式与外部存储器控制器（控制器中心 170）接口耦合的处理器的实施例。但是，许多当前的存储器已开始包括在处理器上的存储器接口模块—片上模块—具有互连多个内核的环配置，以及共享的高速缓存和其他接口。虽然没有示出，但是在一个实施例中，处理器 100 包括与内核、高速缓存和存储器控制器部件相耦合的环互连。

[0068] 在这里，使用高速缓存代理来管理一片物理分布的高速缓存。作为示例，每一个高速缓存部件管理用于分配内核的一片高速缓存，所述内核与高速缓存代理相关联并用于管理高速缓存的分布片。非常类似的高速缓存代理处理在环互连上的业务并与高速缓存片接口，内核代理 / 部件将处理业务并与内核接口。此外，环互连可以将存储器控制器接口逻辑（MCIL）和 / 或其它控制器耦合以与其他模块（例如存储器和 / 或图形处理器 0 接口。

[0069] 转到图 2a，描述了使用原子区优化代码的方法的流程图的实施例。虽然流程的块在图 2 中以大体上串行的方式示出，但可以用任何顺序来执行是示出的方法的流程，以及部分或完全并行地来执行。例如，可以在插入原子区开始和结束之前插入条件提交代码。此外，并不要求执行示出的块。并且未示出的其他块也可以与图示的块联合执行或取代图示的块。

[0070] 在块 205，识别出了将要优化的程序代码段。如上述的，程序代码可以指编译器代码，优化代码、应用代码、库代码，或任意其他已知的代码格式。作为特定说明性的示例，程序代码包括将在处理器 100 上执行的代码，例如准备用于执行的二进制代码、用于在处理器 100 上执行而动态编译的二进制代码，和 / 或动态优化来在处理器 100 上执行的二进制代码。此外，通过程序代码（例如编译器和 / 或优化代码）的执行来进行代码的插入（操作、函数调用等）和代码的优化。作为示例，优化代码是在运行时在处理器 100 上动态执行来仅在处理器 100 上程序代码的执行之前优化程序代码。

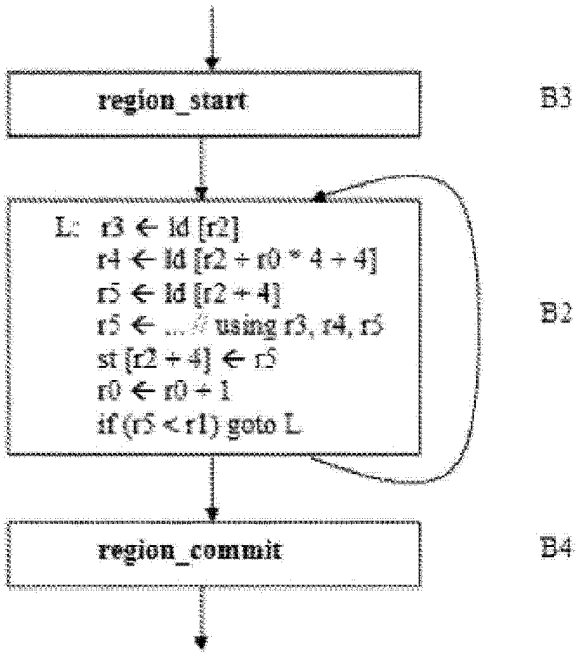
[0071]



[0072] 伪代码 B :将要优化的伪代码区域

[0073] 在一个实施例中，从例如伪代码 B 的区域中识别将要优化的程序代码段包括指示将要优化的程序代码段 / 区域的代码。例如，使用特定指令或划分来指示将要优化的或可能从优化中获益的代码段。作为另一选项，程序员提供了关于程序代码段的提示，优化代码利用所述提示来识别用于优化的段。在另一实施例中，基于分析信息识别 / 选择了区域。例如，程序代码在被硬件、运行在处理器上的软件、固件或其组合执行期间被分析。在这里，代码的分析生成提示或修改原始软件提示，以及提供了用于优化的区域的直接识别。此外，某些属性潜在地识别了代码段，所述属性例如为特定的类型、格式，或代码的顺序。作为特定说明性的示例，包括循环的代码针对潜在的优化。并且在执行期间循环的分析确定应当优化哪一个循环。还有，如果循环包括将要优化的特定代码（例如加载和存储），则识别包括这些代码的区域来用于优化。如能从伪代码 B 中看出的，能将包括加载和存储的区域提升和下沉到循环外部来优化循环执行。

[0074]

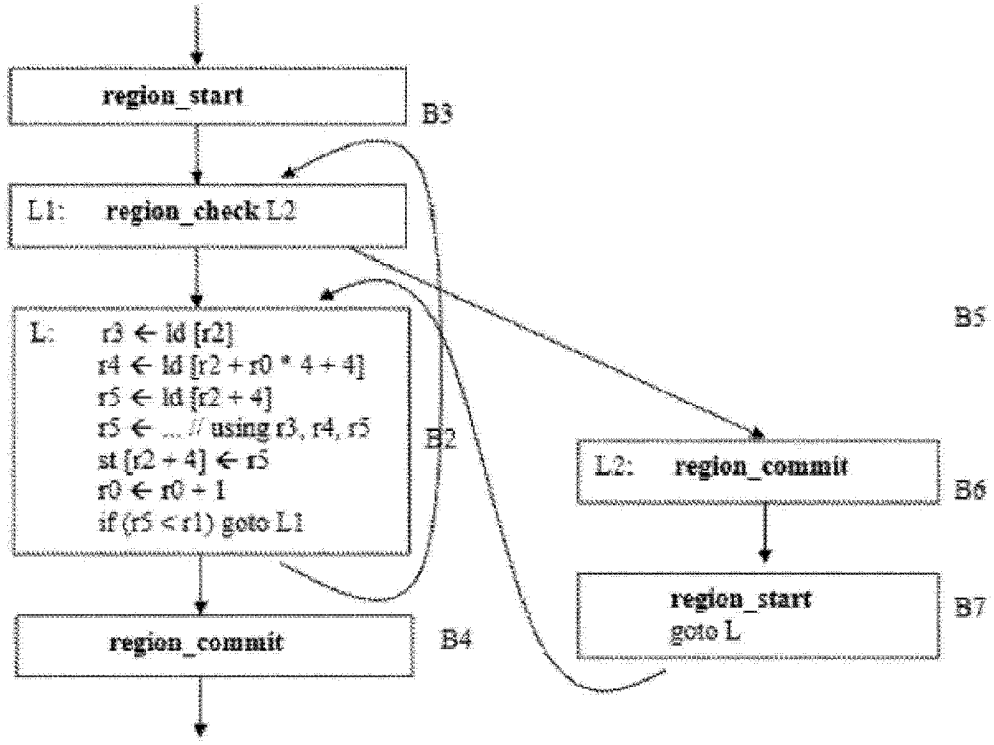


[0075] 伪代码 C :具有插入的开始和提交原子区指令的代码区域

[0076] 在一个实施例中，将被识别用于优化的代码段转换为原子区。或将至少一部分代码段转换为原子区域。在这里，通过开始和结束（提交）事务（原子区）指令来划分该部分代

码，如在块 210-215 中所示。如能从伪代码 C 中看出的，分别地在代码区域之前和之后插入区域开始和区域提交指令。在一个实施例中，代码包括多个入口和多个出口。结果是，可以在每一个入口点和每一个出口点分别插入开始原子区和结束原子区指令。但是，可以使用指示区域是原子的任何已知方法。

[0077]



[0078] 伪代码 D：具有插入的条件提交代码的代码区域

[0079] 在块 220，确定了条件提交点。注意，可以在要优化的代码区域内确定 / 指派多个条件提交点，但是为了讨论的简单，在下面仅更详细地讨论了一个提交点。可以基于任何已知的指派 / 确定算法来确定条件提交点以用于尝试避免在条件提交点之间耗尽硬件资源。作为第一示例，在循环中插入条件提交点，使得在每次迭代时遇到条件提交点。在这里，确定条件提交点将在循环的开始处。作为另一示例，代码的动态分析指示经常导致耗尽硬件资源的执行路径。所以，将条件提交点指派到这些执行路径来避免在执行这些路径期间耗尽资源。类似地，已知是独占的或耗资源重的执行路径可以具有指派给它们的条件提交点。

[0080] 在块 225，至少在条件提交点处插入条件提交代码。如果确定不存在足够的资源来支持执行进入到下一提交点，则条件提交代码将导致重新调整事务的大小（即，早期提交）。暂时转到图 2b，示出了用于插入条件提交代码的流程图的实施例。在流程 226 中，在条件提交点插入了条件提交指令。如能从伪代码 D 中看出的，在一个实施例中，条件提交指令包括在原子区内的提交点 L1 处插入的区域检查指令。

[0081] 作为第一示例，条件提交指令（例如区域检查指令或提交分支指令）包括对分支目标地址的引用。并且作为确定不存在足够的资源来支持执行进入到下一提交点的结果，响应于条件提交指令，执行将跳转到分支目标地址。在这里，条件提交代码还可以包括在分支目标地址处的提交代码。本质上，在这个示例中，条件提交指令将启动测试 / 查询来确定是否存在足够的资源。并且当不存在足够的资源时，在分支目标地址的代码将早期提交事务。

因此，在块 277 中，在分支目标地址插入提交指令。

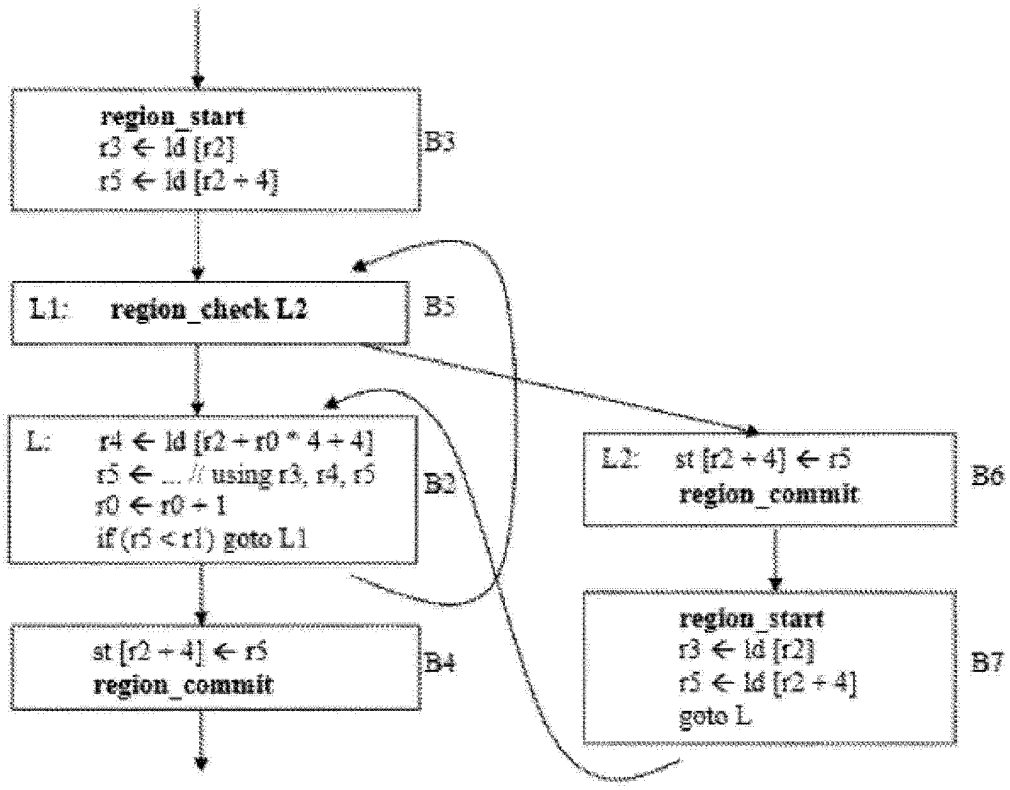
[0082] 在伪代码 D 中，在 B4 中的原子区的出口 / 结束点处插入第一 region_commit 指令，同时在块 B6 中的分支目标地址点 L2 处插入第二 region_commit 指令。在这里，如果响应于在 L1 处的 region_check 指令确定资源不足，则执行跳转到 region_check 指令引用的分支目标地址（B6）。执行 region_commit 指令（例如对事务库内的提交函数的调用或架构上识别的提交指令）来早期（在终点 B4 之前）提交原子区。并且进一步地，在块 228（来自伪代码 D 的 B7），在 region_commit 指令之后插入第二开始原子区指令（region_start）。结果是，在 B3 处开始原子区的执行（region_start）。并且继续进行直到在 L1 处的 region_check 指令确定资源不足或遇到终点 B4。但是，如果在 L1 处确定硬件资源不足，则重新调整原始的事务的大小，即在 B6 的提交点 L2 处提交。接着，在 B7 处开始第二原子区，并且区域执行作为第二事务继续进行，直到提交点 B4 或资源再一次在 L1 处受到限制。因此，可以动态地调整单个原子区的大小为更小的事务来避免耗尽硬件资源，其将预先致使中止或扩展到软件事务执行。

[0083] 注意，条件提交指令可以包括任何信息。例如，如以前所述的，在一个实施例中，条件提交指令包括在下一提交点前的预期硬件资源使用，例如在循环通过伪代码 D 中的代码 B2 期间使用的预期数量的条目。在这里，利用这个预期的使用来确定是否存在足够资源来支持用于循环 B2 的另一迭代的执行。作为特定的说明性示例，预期的使用包括在存储结构中的多个条目，所述存储结构为响应于在下一提交点前的代码区域的执行而唯一接触的结构。

[0084] 此外，条件代码并不限于条件提交指令。例如，在硬件确定使用并将该使用放置到寄存器中（将参考图 5 更详细地讨论）的实施例中，条件代码可以包括操作用来：从寄存器读取 / 加载使用、评估使用，并接着发布跳转或分支操作来转移以提交和重新启动与来自伪代码 D 的 B6 和 B7 相似的代码。在其他的实施例中，替代与硬件进行通信或查询硬件，软件可以估算硬件使用。在那个场景中，条件代码包括执行这些估算的代码。例如，通过软件的执行来保持计数（将参考图 6 更详细地描述）以在执行条件提交之前限制循环的迭代次数。在这个示例中，可将被执行以保持计数的代码认为是条件提交代码。

[0085] 在块 230，优化程序代码段。伪代码 E 示出了在优化之后的代码示例。

[0086]



[0087] 伪代码 E：在优化之后的代码区域

[0088] 虽然在一些实施例中认为原子区的划分和条件提交代码的插入优化了程序代码，但是在其他的实施例中进一步优化了程序代码段以获得执行益处，虽然这依靠了事务执行的存储器排序保护。作为特定的示例，使用部分冗余加载消除（PRLE）和部分死存储消除（PDSE）来使加载和存储的区域提升和下沉到循环外部。伪代码 E 示出了 PRLE 提升 [r2] 和 [r2+4] 的加载和 PDSE 下沉 [r2+4] 存储之后的原子区。注意，在一个实施例中，使用了进一步的存储器排序规则。在这里，确保没有违反存储器排序的存储器优化经过区域开始和提交。这个示例在伪代码 E 中能够看到，其中在 B6 处的 region_commit 之前插入 [r2+4] 存储，并且在 B7 处的 region_start 之后重新插入 [r2] 和 [r2+4] 加载。结果是，如果早期提交区域（调整到条件提交点），则在 B6 处的提交区域之前执行 [r2+4] 存储。并且在 B7 中重新启动新事务之后执行 [r2] 和 [r2+4] 加载。

[0089] 虽然以上已经探索了加载和存储优化，但是可以使用任意已知代码优化技术。代码优化的更多示例，其为非详尽的列表并且纯粹是说明性的，包括：循环优化、软件管道、数据流优化、代码生成优化、边界检查消除、分支偏移优化、死代码消除，和跳转线程。

[0090] 参考图 3a，示出了用于动态调整原子代码大小的流程图的实施例。在块 305 中，执行包括已优化程序代码的事务（原子区）。在一个实施例中，已优化的程序代码是在运行时期间动态调整的，即在运行时执行动态优化代码来在执行时立即优化程序代码。经常这种类型的动态优化或编译不会感知整个程序（例如在静态编译期间），但是能编译 / 优化部分代码段。

[0091] 在块 310 中，遇到区域检查指令。在一个实施例中，遇见区域检查指令包括解码指令的解码逻辑。但是，遇见可以指接收或服务指令（或指令的操作）的管道的任意阶段。例如，遇见区域检查指令可以相反指在缓冲器中分配条目用于与该指令相关联的操作、分派

操作、与执行单元一起实际执行指令来执行用于指令的操作或微操作，或任意其他已知的管道阶段。如上述的，在一个实施例中，区域检查指令是可被处理器的解码器识别以用于查询硬件的 ISA 的一部分。查询可以是简单地加载来查询使用的硬件。并且接着软件确定是否有足够可用的资源而不会更多地涉及硬件。相比之下，查询可以包括请求硬件来确定是否存在足够的资源来继续原子区的执行。在这里，如果存在资源不足，则查询提供硬件转移的目标地址。但是，在区域检查指令还提供硬件的期望使用来用于确定是否有足够的资源可用时，可能会更多地涉及查询。

[0092] 为响应区域检查指令，在块 315 中，在用于区域检查的点确定是否硬件单元具有足够的资源来完成一部分事务(例如直到下一条件提交点)。可以用任何方式实际地测量或估计是否存在足够的硬件资源的确定。作为第一示例，硬件本身跟踪和 / 或确定使用级别。并且硬件、固件、软件或其组合确定使用级别 / 度量是否包括足够的可用性来完成一部分事务。但是，应注意的是，使用级别还可以是仅在软件方向上的估计 / 测量。在这种情况下，测量可能不如上述硬件执行其自身的跟踪的示例准确，但是不必附加额外的硬件部件以支持测量。

[0093] 临时参考图 3b，示出了用于确定是否资源足够用于完成一部分事务的执行的流程图的实施例。在流程 350 中，确定了硬件单元或多个元件单元的期望使用。可以使用硬件、固件、软件或其组合来确定期望使用。

[0094] 在一个场景中，程序代码的执行包括分析事务。在分析期间，跟踪了因为有限的资源而发生的多个中止、没有中止的提交，和 / 或硬件使用。并且随后，例如编译器的代码基于过去的执行分析提供期望使用的提示或建议。在另一场景中，期望使用可以包括估算。例如，如果硬件单元是存储缓冲器，则在这个情况下期望使用包括在代码区域中的多个唯一存储(存储操作可能分配在新的存储缓冲器条目)。本质上，在代码中存储的数量估算在代码区域执行期间待被使用的存储缓冲器条目的数量。但是，确定期望使用并不限于软件分析或估算。而是，硬件可以执行类似的分析或估算，以及联合代码来确定期望使用。

[0095] 类似地，在流程 355 中，可以用硬件、软件、固件或其组合中的任一种来确定硬件单元的可用使用。继续以上的示例，假设条件提交指令通知硬件 32 个条目存储缓冲器的期望使用基于估算或过去的分析包括十个存储缓冲器条目。接着存储缓冲器利用其头部和尾部指针，能确定当前分配了 20 个条目(12 个可用条目)。从此确定中，在流程 360 中执行了比较。并且因为可用条目的数量大于期望使用，所以在流程 360 中确定存在足够的资源来继续执行。可替换地，如果仅有九个条目可用，则在流程 370 确定不存在足够的资源。

[0096] 但是，比较并不限于确切地确保在硬件资源中有足够空间可用。相反，可以通过使用阈值来提供缓冲区。例如，如果使用为高(高于阈值)或可用为低(低于阈值)，则可以在流程 365、370 中做出类似的确定。作为特定的说明性示例，假设六个条目的缓冲区被用于可用。在这个情况下，如果将使用的期望条目的数量是十并且在 32 个条目缓冲器中有二十个条目正在使用，则仅有十二个条目是可用的。所以如果将要分配十个条目的期望使用，则仅有两个条目将是剩下可用的。因为使用了剩下可用的六个条目的缓冲区，所以在流程 370 中确定资源不足。相反，如果二十个条目可用(仅使用了 12 个条目)，则存在足够资源(如在流程 365 中)，这是因为分配了用于代码区域的十个条目，将还剩下十个可用条目。

[0097] 此外，使用和可用可以考虑线程优先权和使用。在这里，如果多个线程共享对硬件

资源的访问，则资源可以是分区的或完全共享的。结果是，使用和可用与期望使用作比较可以考虑到这种共享，所以一个线程不会独占硬件资源（没有未另一线程留下足够可用的资源）。例如，如果两个线程通过分区共享对高速缓存的访问，那么来自一个线程的事务可以被限制到该高速缓存的一半条目。所以，使用和可用是与一半高速缓存相关的，而不是整个高速缓存。

[0098] 上述的公开已经参考了存储缓冲器并简要地参考了高速缓存；但是，使用 / 可用可以关于任意单个硬件资源或硬件资源的组合，例如存储缓冲器、加载缓冲器、高速缓存存储器、牺牲高速缓存、寄存器文件、队列、执行单元，或其他已知的处理器硬件。例如，如果硬件资源包括高速缓存存储器，则“期望使用”可以包括多个将要接触 / 使用的高速缓存行；“使用”可以包括保存数据的多个高速缓存行或在 / 不在特定一致性状态的多个高速缓存行，例如共享的、独占的，或修改的多个高速缓存行；并且“可用”可以包括在 / 不在特定一致性状态（例如无效一致性状态）的多个可用条目或行。此外，可用 / 使用已参考硬件测量可用 / 使用进行了进一步讨论。但是，如上文提到的，可以通过软件、硬件，或其组合直接地或间接地测量以及估算使用和可用。

[0099] 返回图 3a，如果确定有足够资源来完成区域，则执行流程返回到流程 305。换句话说，执行继续直到下一条件提交点；在此点可以再次执行硬件资源的评估。但是，如果确定不存在足够资源，则在流程 320 处在事务结束之前提交事务（动态调整大小）。并且在一个实施例中，为提供无缝执行，在流程 325 开始新的事务来继续原子执行。在这里，单个的较大事务必要地被动态分割为较小的可被硬件处理的事务，而不扩展到虚拟或系统存储器来提供更多的执行空间。

[0100] 转到图 4，示出了支持事务执行的逻辑块的实施例。如图所示，存储器 405 适用于保存程序代码 410，例如操作系统（OS）代码、管理程序代码、应用代码、动态编译器代码等。作为示例，程序代码 410 包括至少一部分将根据图 2a-2b 动态优化（在运行时或部分在程序编译时）的应用代码。在运行期间执行代码 410，代码 410 包括具有包含支持动态调整大小的代码的事务的已优化代码。在一个实施例中，程序代码 410 包括条件提交指令 415，例如之前讨论的区域检查指令。

[0101] 在这里，解码器 425（处理器的解码逻辑）适用于识别条件提交指令 415。例如，解码逻辑 425 被设计并互连来识别作为指令集架构的一部分的操作码 418。结果是，响应于解码器 425 解码包括操作代码（操作码）418 的条件指令 415，逻辑 430、硬件 435 和执行逻辑 440 将执行特定的（预定义）操作 / 微操作。如所示的，条件指令 415 包括引用期望硬件使用 416（硬件使用度量）416 和分支地址 420（地址位置，例如在代码内的基地址和到另一地址、目标地址的偏移）。

[0102] 在一个实施例中，当解码器 425 解码 / 遇见条件指令 415 时，硬件将确定是否有足够的硬件资源可用来容纳条件提交指令 415 所指示的执行硬件使用。可以使用任何已知的方法和装置来确定硬件资源使用，以及确定是否有足够的资源可用于容纳硬件资源的期望使用。但是，在下文讨论了特定的示例来提供用于实现这样的确定的一个实施例的说明。

[0103] 在这里，当解码器 425 接收条件指令 415 时，在处理器管道内解码器 425 和逻辑 430 执行其他的操作 / 微操作。例如，解码器 425 可以解码条件提交指令 415 为多个操作（微操作），例如待执行的操作的跟踪。注意从上述讨论来看，跟踪可以在解码之后存储 / 建

立在跟踪高速缓存中。并且例如，如果操作中的一个包括寄存器读取或从硬件 435 加载来指示硬件资源的使用级别，则逻辑 430 可以分配在加载缓冲器中的条目并调度在执行逻辑 440 上的加载执行。

[0104] 此外，硬件 435 适用于确定这样的使用级别 435，将响应于条件指令 415 而提供、确定或加载所述使用级别。从上可见，支持具有确定的使用级别的事务执行的硬件的多个示例包括：高速缓存存储器、推测高速缓存存储器、存储缓冲器、加载缓冲器、寄存器文件、推测寄存器文件、检查点寄存器文件等。结果是，指令 415 可以包括用于一个或多个硬件资源的一个或多个期望使用度量。并且独立的硬件或资源本身适用于跟踪其使用级别。例如，在一个实施例中，用于高速缓存存储器的高速缓存控制逻辑适用于跟踪高速缓存存储器的使用级别，例如在高速缓存存储器中保存的多个无效高速缓存行或在高速缓存存储器中的多个可用行。

[0105] 接着，基于与确定的硬件使用级别相比较的期望使用级别，确定是否存在足够的资源来继续事务的执行，而不用早期提交(动态调整大小)。并且如果要执行早期提交，则在示出的示例中，执行逻辑 440 跳转到如条件提交指令 415 所提供的分支目标地址 420。如上所述，分支目标地址可以包括代码，当执行所述代码时，在早期提交事务并重新启动另一事务来继续原子执行。

[0106] 作为特定的说明性示例，假设解码 425 接收条件提交指令 415。并且条件指令 415 包括十个存储缓冲器条目和十个高速缓存行的期望使用度量 416。硬件 435 (事务高速缓存存储器和存储缓冲器)确定它们的使用级别。注意这些硬件实体可以持续跟踪使用，并且它们在条件提交指令 415 查询 / 请求时提供这些使用。或者，当接收到来自条件提交指令 415 的请求时，逻辑可以实际上确定使用。不管怎样，向存储元件提供硬件使用级别 / 度量 436，所述存储元件例如为保存用于执行逻辑 440 的操作的信息的寄存器。接着，比较硬件使用级别 436 和期望硬件度量 416 来确定是否有足够的资源可用来继续执行而不用早期提交。注意从上面来说，可以基于设计者的任意优选算法，所述比较使用缓冲区、阈值，或直接比较来确定是否有足够的资源可用。

[0107] 在这里，假设存储缓冲器正在使用 32 个存储缓冲器条目中的 16 个(16 个可用条目)，并且事务高速缓存具有 64 个条目中的 60 个标记为已被事务访问(事务已经访问了这些行并且取代这样的行将导致信息的损失，致使中止或扩展到软件事务执行，即 4 个可用条目)。并假设设计者算法指定在考虑条目的期望数量之后还应有 4 个可用条目。在那种情况下，对于 10 个期望存储缓冲器条目和 16 个可用条目，在存储缓冲器内存在足够的可用空间来容纳原子执行直到下一条件提交点。但是，仅有四个高速缓存条目没有标记为已被事务访问的，所以没有足够的事务高速缓存空间。结果是，执行逻辑 440 (例如跳转执行单元)跳转到分支目标地址 420 来取得、解码和执行代码，以在早期提交事务(动态收缩事务)并重新启动另一事务。

[0108] 注意，已参考包括期望硬件使用度量和分支目标地址的条件提交指令讨论了上面的示例。但是，条件提交指令可以包括致使硬件来评估或估算是否有足够可用硬件来支持代码执行的任意指令。例如，条件提交指令可以仅是条件跳转指令，其中硬件相对于代码的过去硬件分析评估当前使用级别来确定是否应提交事务。并且硬件在做出评估之后能跳转到条件提交指令提供的分支地址。

[0109] 注意在另一实施例中，硬件可以异步地(不与特定的条件提交指令绑定或响应特定的条件提交指令)确定将要提交的事务。在这里，当处理器执行事务代码并且溢出事件(指示在硬件资源中没有空间剩下的事件，例如收回已被事务性标记的条目)发生时，则硬件可以提交硬件事务并且重新启动新的事务而不用很好地知晓代码。

[0110] 接下来参考图 5，示出了支持动态调整事务大小的硬件的另一实施例，之前(参考图 4)，讨论了可以将确定的硬件使用级别 / 度量放置在存储元件中，例如将用于执行逻辑 440 操作的寄存器。类似于那个示例，可以将确定的硬件使用级别 / 度量类似地加载到存储元件中，例如寄存器 550。但是，在一个实施例中，寄存器 550 可以包括模型特定寄存器(MSR)，寄存器 550 适用于被程序代码(例如用户级别的应用代码)访问(暴露给程序代码)，来执行硬件资源可用性的评估。在前面的示例中，硬件基于来自软件的预期使用来执行评估。但是在这里，软件能查询硬件(MSR550)并接收一个或多个硬件资源的使用级别。接着，软件能基于其自身的指南来确定是否存在足够的资源来继续原子执行到下一提交点。这可以为用户提供更多的灵活性，因为这允许用户确定使用多少硬件。结果是，处理器设计者可以选择处理器是否应该保留更多控制或用户是否能做出这种决定。

[0111] 作为特定的说明性示例，假设处理器执行动态编译器代码来优化并执行程序代码 510，代码 510 包括用于潜在地动态调整大小的已插入 / 已优化的事务。在这里，用于处理器的取得逻辑(未示出)基于用于处理器线程(也未示出)的指令指针来取得加载操作 515。将加载缓冲器条目分配到加载缓冲器中的逻辑 530/ 硬件 535 中。逻辑 530 调度和分派加载，并且执行逻辑 540 执行加载来从硬件 535 (支持事务执行的资源，例如高速缓存、缓冲器，或寄存器文件)加载确定的使用级别 536。注意，加载或前面的指令可以同步地致使硬件 535 来确定使用和 / 或将使用放置在 MSR550 中。可替换地，硬件 535 可以异步地将使用级别放置在 MSR550 中(基于事件或周期特性)。无论如何，操作 516 读取并评估使用级别。

[0112] 例如，操作 516 可以包括布尔表达式来用期望使用级别评估已加载的硬件使用级别或由软件定义的其他阈值级别。事实上，布尔表达式可以是条件语句和 / 或跳转指令 517 的一部分，其即将在下文中更详细地讨论。在这里，如果硬件使用级别的评估指示应在早期提交事务，则执行由解码器 525 通过操作码 517o 识别出的跳转指令 517 以转移到目的地地址 520，如上文讨论的，来提交事务并重新启动另一事务。

[0113] 现在转向图 6，示出了支持动态调整事务大小的硬件的另一实施例。在一个实施例中，计数器 650 适用于对通过循环 615 执行的多个迭代进行计数。注意，计数器 650 可以用硬件、软件、固件，或其组合来实现。例如，计数器 650 可以是保存值的寄存器。并且软件代码在每次迭代通过循环时读取当前值，修改(增加 / 减少)当前值为新的值，并将新的值存储在寄存器中。本质上，软件维护计数器。但是，硬件计数器也可以在每次循环的迭代开头或末尾增加 / 减少。

[0114] 在一个实施例中，可以使用计数器 650 的计数器值来确定何时发生事务的早期提交。例如，计数器 650 对循环的迭代数量进行计数直到其到达阈值。在到达阈值时，计数器过期或溢出致使早期提交。注意，计数器可以从零开始并且向上计数直到到达阈值(上溢)或在某个值开始并向下计数到零(过期或下溢)。

[0115] 可以用任何方式确定阈值(或用于计数器从开始值减少到零)。阈值可以是静态的预定义值，其包括在硬件中或由软件设定。在这里，软件或硬件可以基于任何已知的特性智

能地选择静态的预定义值，所述已知的特性例如为包括在处理器中的硬件类型或大小，以及执行的代码的类型或大小。或者，懈怠地选择所述静态的预定义值，例如保守值，来限制循环迭代次数为足够小的数以显著减少回滚。

[0116] 作为替换的实施例，阈值（开始值）是动态选择和可更改的。在这里，硬件、软件或其组合可以基于任何特性选择初始的开始值（阈值）。例如，可以将多个可用高速缓存行或存储缓冲器条目（例如，32）除以在代码的单个循环迭代中存储的数量的计数（例如，8）来确定阈值（例如，4）。在一个实施例中，存储数量的估算是合理保守的，因为多个存储可能仅接触单个高速缓存行。所以，可以使用更积极的初始值。作为另一示例，无论是代码还是硬件基于代码大小 / 类型、硬件单元大小 / 类型，或其他已知因素来选择积极的或保守的值来用于确保具有足够的资源来完成循环的执行。此外，硬件或软件可以分析代码并提供开始值 / 阈值。在这里，软件提示包括在循环开始执行之前提供阈值数量，并且基于该阈值数量设定计数器。或者，硬件可以类似地分析代码并且基于代码区域的分析历史来设定计数器。

[0117] 在一个实施例中，在初始化计数器之后，动态地修改阈值（开始值），例如基于循环的执行分析（代码分析或回滚确定）来调整。在一个实施例中，本质上使用计数器 650 来估算在回滚之前能执行的循环 615 的迭代次数的数量或因为有限的硬件资源而发生回滚的可接受数量。因此，如果多于可接受数量的回滚发生时，则通过在发生提交之前减少迭代的数量来减少之前的阈值从而减少回滚数量。在这个场景中，太频繁的回滚浪费了执行时间并潜在地致使延迟。但是，为确保阈值不会过于保守（当仍有足够资源可用时不必要地早期提交事务，这导致低效的原子执行），在一个实施例中，增加阈值直到确实发生回滚。注意增加 / 减少可以以单个、多个，或指数增加形式发生。作为示例，假设开始值首先被设置为 63（即，在提交前允许 64 次迭代）。如果因为硬件限制检测到多个中止 / 回滚，则阈值减少到 62。在发生随后的回滚时，其进一步减少 2（60）、4（56）、8（48）等，直到确定平衡的开始值来允许高效地完成执行。

[0118] 注意，已参考对循环迭代的数量进行计数的计数器而没有特定的参考条件提交指令进行了图 6 的讨论。这里，到达阈值的技术其可以通过硬件同步地启动提交。但是，在另一实施例中，计数器 650 连同来自图 5 和 6 的条件提交指令一起工作。作为示例，计数器可以是硬件机构，基于硬件计数器其可以确定 / 估算可用的硬件资源数量和致使跳转到提交代码的条件提交指令。例如，假设初始化计数器 650 为九（在条件提交之前应允许发生十次迭代）。并且在循环 615 的每次迭代时，如果计数器已过期（到达零），则执行条件提交指令（例如条件跳转指令）来跳转到分支目标地址。当完成十次迭代并且执行条件跳转时，执行流程跳转到目标地址来用于执行原子区的提交。

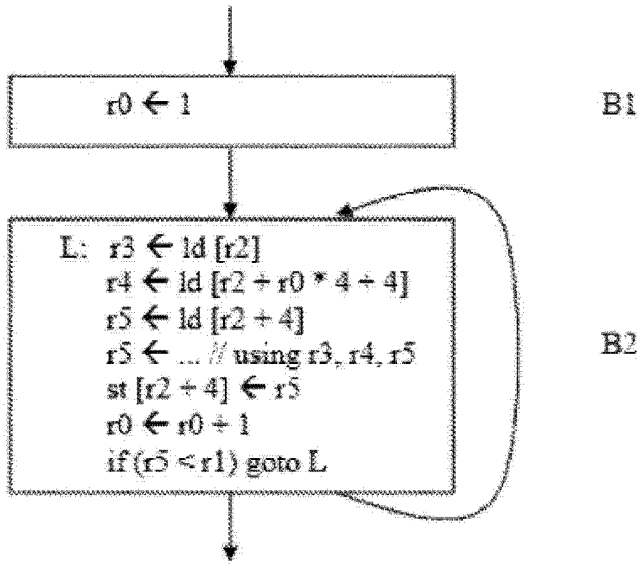
[0119] 虽然计数器不是单独的硬件单元的确切的、特定的使用级别测量，但是其潜在地是近似全包括估算。如上讨论的，通过阈值的动态调整，可以找到基于硬件限制的用于减少回滚的最优迭代数量。因此，如果任何以前未识别的硬件限制正在致使回滚，则动态计数器阈值能捕捉到，同时设计者或程序员单独的识别可以忽略这种未识别的原因。此外，计数器还可以连同特定的硬件单元使用测量使用来提供单元级别粒度以及过度的全局粒度。

[0120] 以前的讨论主要地集中在耗尽硬件资源之前的条件提交事务上。或判定硬件使用与期望使用来确定是否有足够的资源可用来支持执行。但是，在一些实施例中，通过原子区执行是有益的。并且周期性地（响应用户指令、事件，或硬件定义时间）来在检查点检查原子

区。所以，在遇见实际的硬件限制、异常、中断，或其他错误时；能回滚原子区到最近的、临时的检查点并提交原子区来释放资源。本质上，取代做出的前向查看估算并且主动地提交事务，在一个实施例中，仅在遇见实际的通常将要求中止或重新启动整个事务的资源限制时才重新调整事务大小。这通过执行多个在事务中的检查点来确保是否发生回滚来完成，其仅为回滚的执行的可接受数量。

[0121] 转到图 7a，示出了用于优化代码的方法的流程图的实施例，所述优化代码包括提供用于在事务内的推测检查点。在块 705 中，识别了将要优化的程序代码段。类似于图 2a 的讨论，可以基于用户识别 / 提示关于段、程序分析、代码分析、代码属性(特定的类型、格式、顺序，或区域的特性—循环或存储的数量)，或其他已知的用于识别将要优化的代码区域的方法来识别代码段 / 区域。在一个示例中代码包括二进制代码。并且在二进制代码的执行期间，动态地优化所述代码。结果是，在用于执行的优化期间，在二进制代码中遇见循环。所以，确定循环是将要优化的(插入推测检查点)来确保在检查点之间的距离足够小，使回滚不会导致执行的大量损失。作为示例，下面的伪代码 F 示出了将要优化的代码区域的示例。

[0122]



[0123] 伪代码 F: 将要优化的代码区域

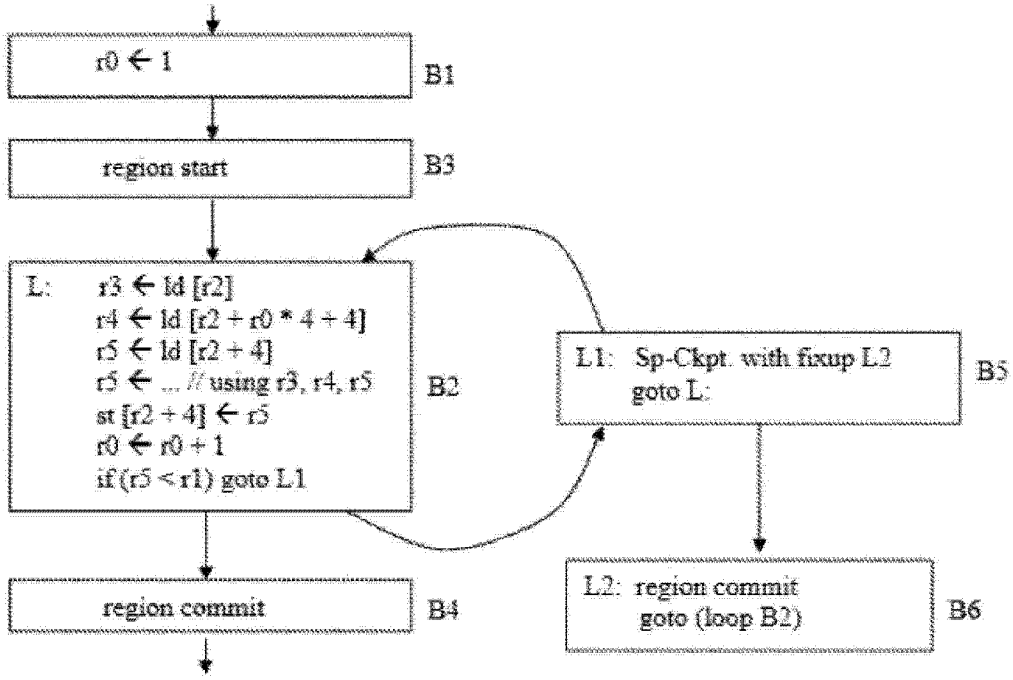
[0124] 还类似于图 2a 的讨论，在流程 710、715 中，代码区域是划分的(在段的开始处插入开始原子区域指令并在段的结束处插入结束原子区域指令)。如前面提到的，代码区域可以包括多个入口和多个出口。在块 720 中，确定了在程序代码内的推测检查点位置。注意，可以在将要优化的代码区域内确定 / 指派多个推测检查点，但是为了讨论简单，在下文仅更详细地讨论了一个检查点。

[0125] 可以基于任何已知的用于在原子区域内最小化回滚影响的指派 / 确定算法来确定推测检查点。作为第一示例，在循环内插入推测检查点，使得在每次迭代时遇见推测检查点。在这里，可以在循环的开始处或在循环的边界确定推测检查点。作为另一示例，代码的动态分析指示经常导致耗尽硬件资源或具有高指令计数的执行路径。所以，将推测检查点指派到这些执行路径，来避免因为在长执行路径内中止而回滚整个原子区。类似地，已知是独占或耗资源重的执行路径可以具有指派给它们的推测检查点。本质上，取代在整个事务

和关联的整个事务的回滚之前的先前技术检查点，通过使用在事务内（本地的、内部的，或临时的检查点）的检查点来执行更小的回滚（更少浪费执行）。结果是，可以优化更大的区域。并且如果遇见资源限制，执行小的回滚。此外，在回滚点，潜在地提交事务，处理最终的错误，并重新启动另一事务。

[0126] 在流程 725 中，在推测检查点位置插入推测检查点代码。推测检查点代码将在检查点检查推测硬件资源，例如存储器、高速缓存存储器，和 / 或寄存器文件。并且在一个实施例中，推测检查点代码还可以包括代码来恢复硬件资源到检查点以响应随后的中止条件。暂时转到图 7b，示出了用于插入推测检查点代码的流程图的实施例。伪代码 G 还示出了在插入推测检查点代码之后的代码的说明性示例。

[0127]



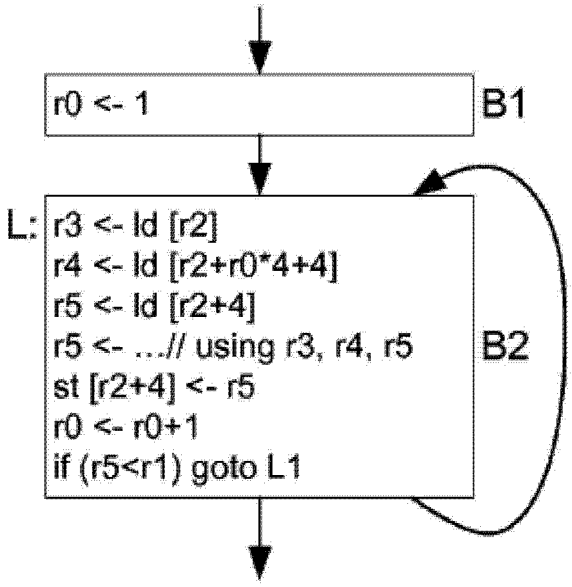
[0128] 伪代码 G：在插入推测检查点代码之后的代码区域

[0129] 在流程 726 中，在推测检查点（B2、B5 的循环后边界）处插入推测检查点指令 / 操作（在 B5 中的 L1）。在一个实施例中，推测检查点指令包括初始化推测硬件资源的检查点（当前快照）的任何指令。例如，处理器的解码器可识别推测检查点指令。并且一旦解码、规划，和执行推测检查点，致使推测寄存器文件、存储缓冲器，或其二者在检查点检查（当前状态的快照）到检查点存储结构，例如检查点推测寄存器文件和推测高速缓存。

[0130] 此外，在一个实施例中，推测检查点代码还包括一些其他代码来执行一些行动，潜在地避免致使当前回滚的同一回滚情况。例如，如果因为有限的硬件资源发生回滚，如果没有采取缓和行动，则可能每次都遇见相同的硬件限制，导致没有向前的进步。

[0131] 作为示例，假设将要优化在伪代码 H 中左边的代码。

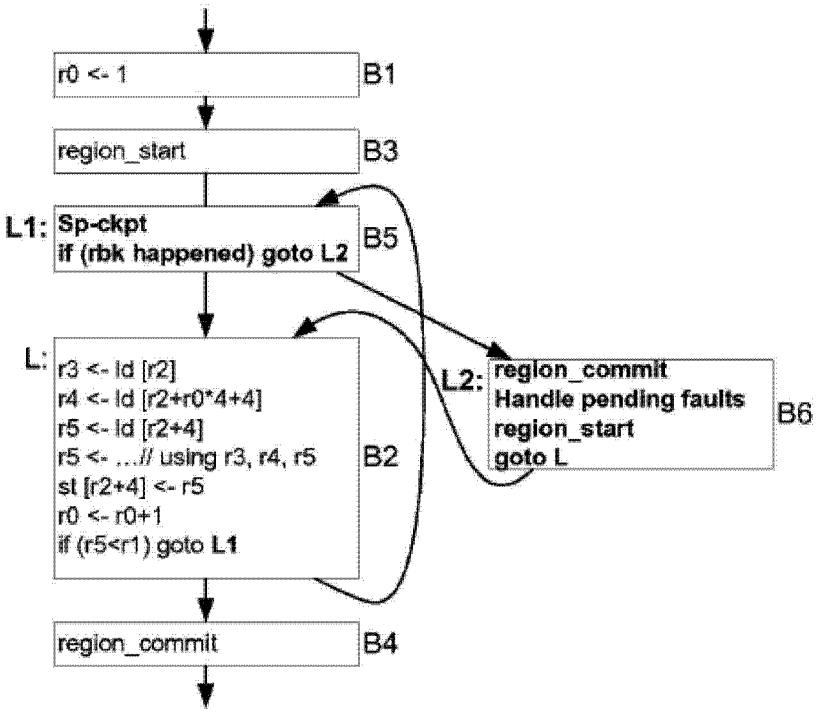
[0132]



[0133] 伪代码 H :将要优化的代码区域

[0134] 在这里，存在多个场景用于如何处理在已优化的代码内的回滚。作为第一示例，如下文伪代码 I 示出的，推测检查点指令与条件(分支)指令配对，所述条件指令能区分在回滚到推测检查点之后正常的循环执行或重新执行之间的区别。当执行分支指令时，其跳转到不同的执行路径来以任何已知的方式处理回滚以用于回滚到以前的检查点状态。

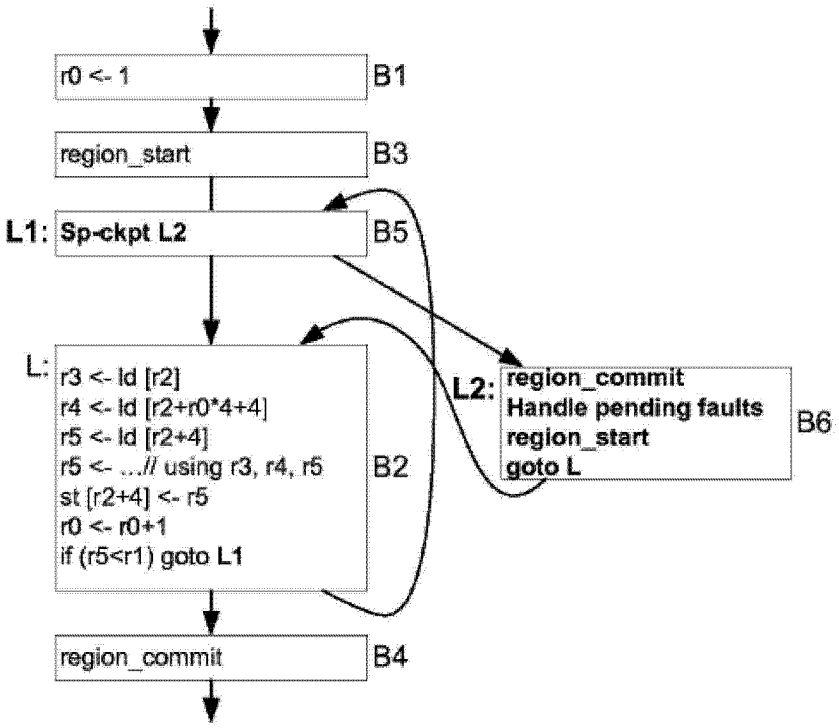
[0135]



[0136] 伪代码 I :分支指令与推测检查点指令配对

[0137] 在另一场景中，在下文示出了伪代码 J，推测检查点指令与分支指令组合到单个检查点中并且分支指令在上文的一些实施例中已做过讨论。

[0138]

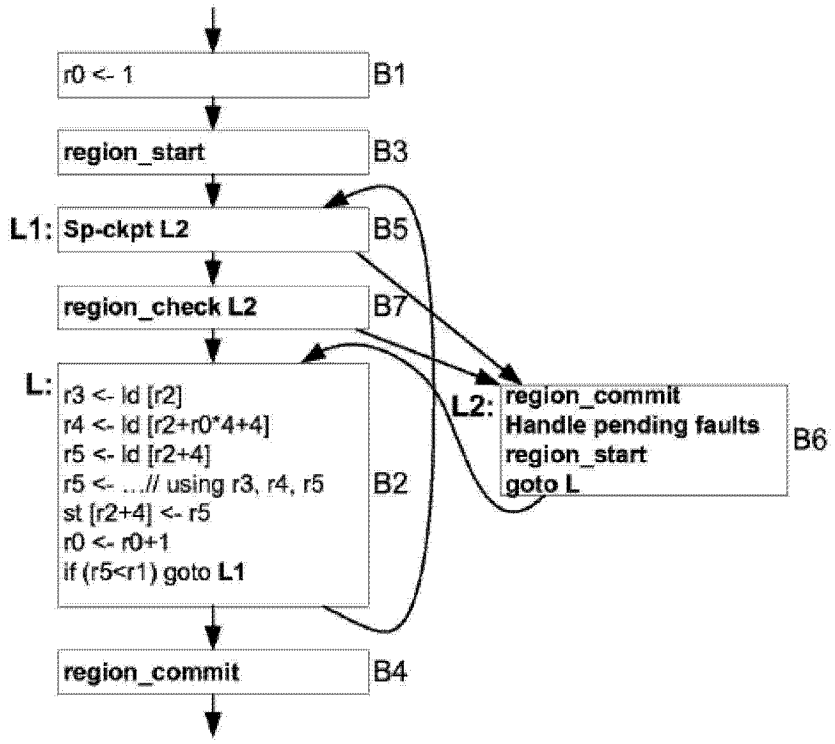


[0139] 伪代码 J :组合分支与推测检查点指令

[0140] 返回伪代码 G 的讨论，在一个实施例中，推测检查点代码还包括修补（还可以称为回滚、恢复、重构、还原等）代码（B5），当其执行时从检查点存储结构恢复或还原到最近的检查点的精确状态。注意，在一些实施例中，在上文描述的其他场景中还可以考虑推测检查点代码和 / 或修补代码。但是在这里，如伪代码 G 所示的，修补代码还可以包括推测提交代码（B6）来提交事务 / 原子区。本质上，推测检查点代码检查推测硬件。并且在耗尽推测硬件或遇见致使中止的事件（中断、动态断言、存储器别名检查等）时，检查点代码在推测硬件中恢复精确的状态，此外，为释放推测硬件来用于重新执行和继续执行，可选地提交事务。从这个讨论可见，达到修补代码可以用多种方法来完成。作为示例，修补代码可以来自以下输入 :（1）在检查点存储结构不具有足够空间来在检查点检查推测硬件情况下执行推测检查点操作 ;（2）推测硬件不能容纳时的推测操作的执行 ;（3）在推测执行期间的异常 ;或（4）任何其他导致回滚到事务内的临时的、内部的检查点的事件。

[0141] 此外，推测检查点可以与条件提交结合来提供进一步的好处，例如避免因为缺少推测资源的中止，同时检查点检查使任意的这类因为错误、异常，或其他未预测的原因导致的回滚 / 中止更加廉价（返回到检查点而不是整个原子区的开始，更少执行浪费）。下文的伪代码 K 示出了这种组合的一个示例。

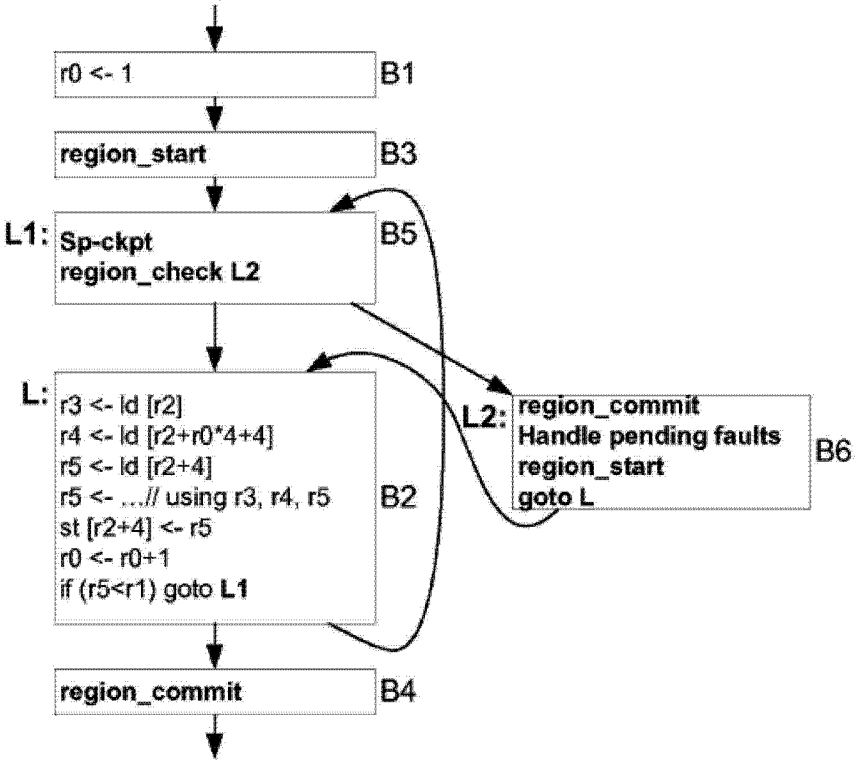
[0142]



[0143] 伪代码 K :推测检查点和条件提交的组合

[0144] 此外,在一个实施例中,使条件提交指令知晓推测检查点,如下文伪代码 L 所示的。

[0145]

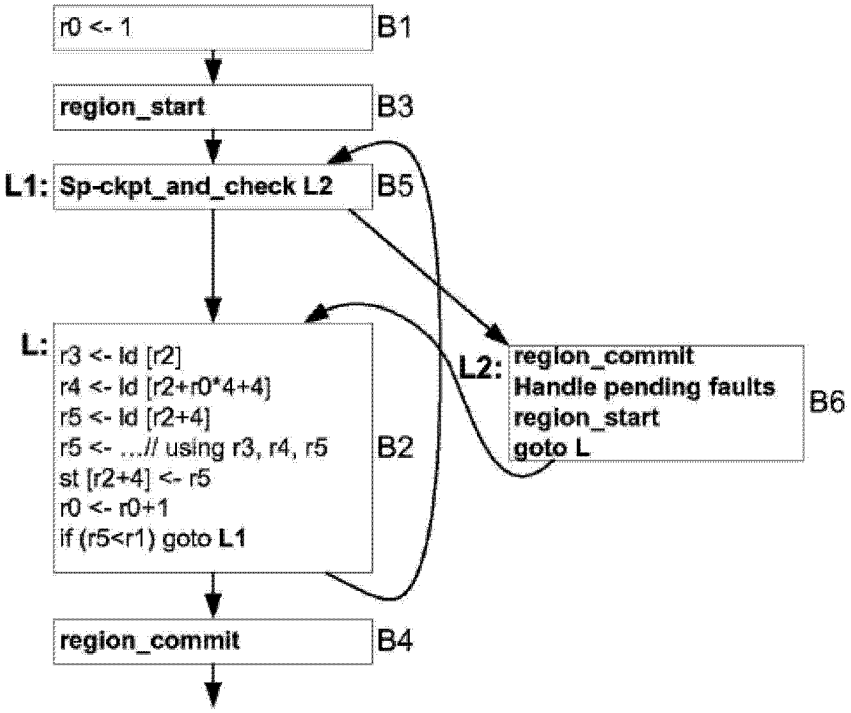


[0146] 伪代码 L :条件提交知晓推测检查点

[0147] 在这种情况下,如果系统将耗尽资源(如上所述)或执行是回滚重现(在执行回滚到推测检查点之后),region_check (条件提交)指令跳转到 L2。

[0148] 此外，不仅可以一起使用推测检查点和条件提交指令，在一个实施例中，指令本身在如下文所示出的伪代码 M 中组合。

[0149]



[0150] 伪代码 M：用于推测检查点和条件提交的一个指令

[0151] 在这里，当执行组合的指令时，执行检查点并评估条件提交（如果将耗尽硬件或在低资源上运行则提交）。在这里，如果系统以低推测资源运行或如果执行回滚到已记录的推测检查点，则执行跳转到 L2 并通过调节推测资源和维修最终错误来处理（在这个示例中）所述指令。

[0152] 虽然以前的讨论已参考如下内容：一旦硬件正在低运行时，回滚到以前（或最近的推测）的检查点，检测到错误，发生了异常，或其他未预测的事件致使中断，但是可以探索其他道路来响应这些中断。实际上，在一个实施例中，在中断时，硬件、软件，或其组合作出下一步将如何做的决策。例如，假设错误，例如在原子执行期间发生硬件生成错误 / 事件。原子执行中断。并且通常将控制转交到软件来确定错误的类型并维修错误。

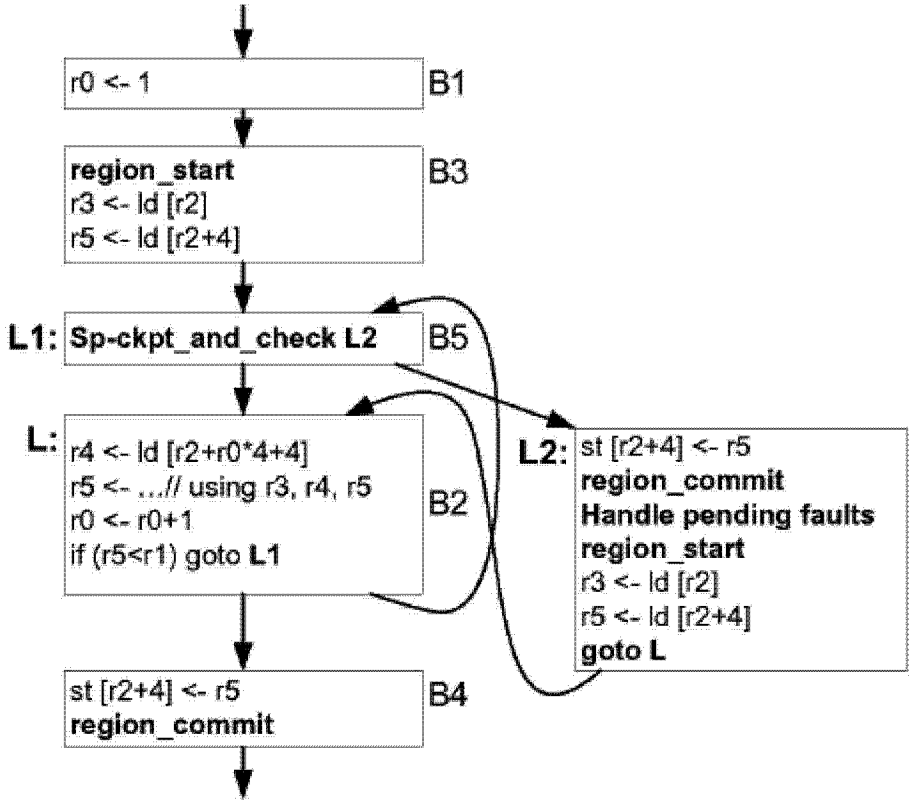
[0153] 在这里，软件（例如处理器）可以基于任意数量的因素，例如错误的类型、回滚到最近推测检查点的难度、通过回滚到最近提交点而不是最近推测检查点指令数量或指令损失量，或其他在选择程序中的执行点来用于返回的其他因素来决定下一步做什么。本质上，在这个说明性的示例中，软件（例如处理器）正在做出确定是否执行应该回滚到原子区的开始、在原子区内的最近提交点，或在原子区内的最近推测提交点。并且即使示例已集中在软件做出决策，但是硬件也可以做出确定。在一个实施例中，使用新的指令（speculative_rollback）来做出确定。在这里 speculative_rollback 指令包括任意信息，一旦所述信息被解码并执行会导致返回到程序代码中的适当位置（推测提交点或当前提交点）。

[0154] 重点注意的是，在本文中描述的代码不是必须分配到同一块、区域、程序，或存储器空间中。实际上，可以在主程序代码内的循环返回边界处插入推测检查点操作。并且修补代码可以位于库中或代码的其他部分中，所述修补代码包括回滚到最近检查点并可选地

提交事务的代码。在这里，正在执行主程序代码。并当输入修补代码时，执行对库代码的一个或多个函数的调用来执行回滚并提交。还有，硬件可以执行类似的检查点操作而没有软件的指导。在这里，硬件透明地检查推测硬件，例如在周期性的或事件驱动的基础上。并且为响应耗尽推测硬件，处理器回滚执行到最近的检查点、提交事务、重新开始事务，并重放在检查点和回滚点之间的指令。从软件角度来看，执行正常地继续，而硬件已处理了所有的资源约束和重新执行。但是，可以使用在硬件和软件之间任意级别的合作来达到本地的、内部的事务检查点。

[0155] 返回到图 7a，在块 730 中优化了程序代码段。下文的伪代码 N 示出了在优化之后来自伪代码 M 的代码区域的示例。

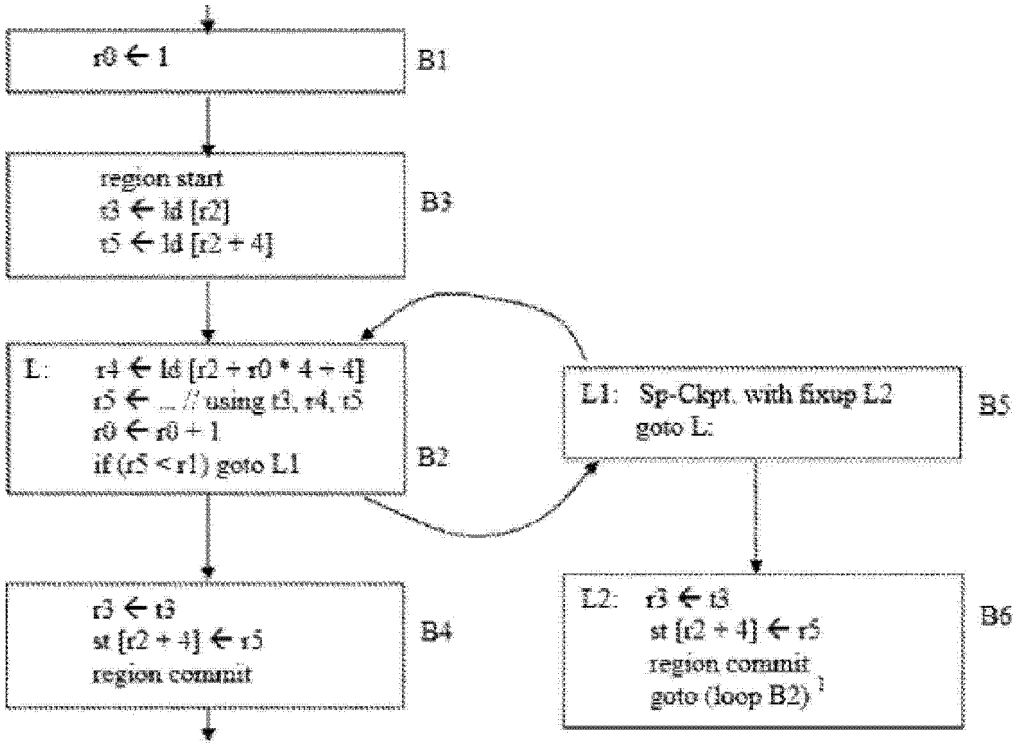
[0156]



[0157] 伪代码 N：包括用于推测检查点和条件提交的单个指令的已优化代码

[0158] 作为优化的另一示例，下文的伪代码 O 示出了优化的另一示例（来自伪代码 G 的代码区域的优化）。

[0159]



[1]: go to loop B2. In this example, both the cold code for B2 and L0: (at B3) are valid targets.

[0160] 伪代码 0 :在优化之后的代码区域

[0161] 如上面讨论的,可以在原子区内的代码上执行任意已知的优化技术。代码优化的几个示例,其为非详尽列表并且纯粹是说明性的,包括 :PRLE、PDSE、循环优化、数据流优化、代码生成优化、边界检查消除、分支偏移优化、死代码消除,和跳转线程。在伪代码 0 中,在 B6 中提交之后,在 B6 开始另一事务并且执行在 B2 处继续。在另一示例中(未示出),代码可以在 B3 处重新输入。在一些实施例中,注意也可以考虑原子区的划分和条件提交代码的插入来优化程序代码。

[0162] 接下来参考图 8,示出了在事务执行期间用于推测在检查点检查存储器方法的流程图的实施例。在流程 805 中,执行了事务。作为一个示例,事务是在运行时期间动态优化以在确保原子性的代码优化周围插入事务的二进制代码。

[0163] 在流程 810 中,遇见来自事务的推测检查点指令。如参考图 7a-7b 在上文讨论的,还可以在运行时期间由优化器在推测检查点插入推测检查点指令。处理器识别推测检查点指令(通常通过检测预指定 / 定义的操作代码的解码器)。并为响应推测检查点指令,在流程 815 中在检查点(推测的)寄存器文件中检查推测寄存器文件。另外,在流程 820 确定是否推测高速缓存包括足够的空间来在检查点检查存储缓冲器。如果有足够的空间可用,则在流程 825 中在推测高速缓存中在检查点检查存储缓冲器。但是,如果没有足够的空间,则执行修补 / 回滚程序(在下文更详细讨论的块 845-855)

[0164] 结果是,如果在事务执行期间遇见短期回滚事件,则用于当前执行的推测寄存器文件和存储缓冲器能还原到检查点状态。作为示例,在流程 830 中,遇见存储操作。在流程 835 中,确定存储缓冲器是否包括可用条目,例如将要分配用于存储操作的可用条目。并且如果有现成可用的条目,或有可以解除分配或重新分配的条目,则在块 840 分配条目。但

是，如果没有可用的存储缓存条目，则执行回滚程序（块 845-855）。

[0165] 回滚 / 还原程序 845-855 将从以前的检查点还原精确的架构状态。在这里，回滚处于没有提交（没有使得全局可见）的推测执行期间。因此，全局可见状态（非推测存储结构）应保持相同状态。但是，恢复支持当前推测执行的推测硬件结构。因为推测高速缓存已经保存来自存储缓冲器的推测更新直到最近的还原点，所以在块 850 清除存储缓冲器。换句话说，从事务开始到最近的检查点的存储保存在推测高速缓存中。并且从最近的检查点到当前执行点（回滚的初始化）的存储保存在存储缓冲器中。所以，已回滚的这些存储从存储缓冲器中被简单地丢弃了。

[0166] 此外，推测寄存器文件从检查点寄存器文件还原。在这里，推测寄存器文件保存所有的从事务开始时的更新（包括从最近检查点的那些更新），所以使用来自检查点寄存器文件的值来重新加载推测寄存器文件。注意，如果原始的检查点包括整个推测寄存器文件的拷贝（不仅是只在推测执行期间已修改的寄存器的选择性存储），则可以重新标记（已使用）检查点寄存器文件为推测寄存器文件并且可以清除以前的推测寄存器文件，并且随后用作检查点寄存器文件。或者将推测寄存器文件闪存拷贝（在一个或几个周期）到推测检查点寄存器文件。

[0167] 在流程 860 中，可以可选地提交事务。因为响应于异常、推测高速缓存中的空间缺乏，或存储缓冲器中的空间缺乏而达到回滚程序，所以可以提交事务来释放那些资源。在这里，将存储缓冲器和推测高速缓存更新提交到非推测高速缓存存储器，释放那些资源（在流程 875-880 中示出）。并且类似地将推测寄存器文件提交到非推测寄存器文件，释放其用于进一步的推测执行（在流程 875-880 中示出）。此外，如果将要执行事务的完全中止（865），则在块 870 中清除存储缓冲器和推测高速缓存来将其还原到事务前（非推测）状态。

[0168] 转到图 9，示出了适用于支持推测检查点的硬件的逻辑表示的实施例。解码器（解码逻辑）910 适用于或互连来识别推测检查点指令（SCPI）905。例如，用于包括图 9 的硬件的处理器的指令的预定义格式可以被指定并可以设计为硬件。并且具有特定位模式的一部分指令对应于特定指令；其中的一个是 SCPI905。

[0169] 接着，为响应 SCPI，在推测存储结构中保存的推测值 935 被在检查点检查作为在推测检查点存储结构中的推测检查点值 940。注意，执行逻辑 920 如示出的与解码器相耦合来执行 SCPI920。但是明显的，在解码和执行之间经常有管线的大量阶段。例如，可以将 SCPI 解码为在跟踪高速缓存内的操作的跟踪；并且可以在缓冲器中对那些操作进行排队、调度、分派和执行来执行在本文中描述的操作 / 方法。

[0170] 如在上文简短描述的，检查点包括在某时间点的状态快照，或至少足够的信息来恢复某时间点的值的状态。因此，在一个实施例中，检查点包括作为推测检查点值 940 的推测值 935 的整个拷贝。在这里，推测值 935 可以包括来自推测寄存器文件的推测寄存器文件值，并且推测检查点值 940 包括在时间上最近的检查点的推测寄存器的拷贝。作为另一示例，推测检查点值仅包括从最后检查点已修改的推测值 935。在这里，推测值 935 可以包括来自推测寄存器文件的推测寄存器文件值，并且推测检查点值 940 包括来自最后检查点且仅来自从最后检查点已修改的推测寄存器文件中的寄存器的推测寄存器文件值。还是另一示例，推测检查点值 940 包括从原子区的开头直到时间上的检查点的所有值，同时推测值 935 包括从检查点到当前执行点的所有推测值。在这里，存储缓冲器可以保存推测值 935，

将其加到在推测高速缓存中保存的较旧的值(从开始到最后的检查点)上。

[0171] 参考图 10,示出了适用于支持寄存器文件的推测检查点的硬件的逻辑表示的另一实施例。类似于上文的讨论,解码器 1010 和执行逻辑 1020 分别适用于接收、识别和执行 SCPI1005。为响应 SCPI1005,将推测寄存器文件 1035 在检查点检查到推测检查点寄存器文件 1040 中。如上所述,检查点可以包括寄存器文件 1035 到检查点寄存器文件 1040 中的闪存拷贝。或当要修改在文件 1035 中的寄存器时,将旧的值在检查点检查到寄存器文件 1040 中。在这里,取代响应 SCPI1005 的拷贝值,在文件 1035 中修改相对部分时将来自文件 1040 的旧检查点值清除或标记为无效。当执行继续时,在文件 1040 中再次在检查点检查修改的寄存器。

[0172] 为响应回滚(由于如图 8 的块 820 中的推测高速缓存的空间缺乏、如图 8 的块 840 的存储缓冲器中的空间缺乏、异常,或其他回滚事件)到最近检查点,将检查点的值(无论仅是已修改的还是全部的拷贝)从推测检查点寄存器文件 1040 还原到推测寄存器文件 1035 中。但是,如果有回滚到事务区域的最开始,例如事务的中止,则在一个实施例中,从非推测寄存器文件 1030 中重新加载推测寄存器文件 1035。本质上,推测文件 1035 是工作的推测寄存器文件。所以事务与推测寄存器文件 1035 共同工作(读取和写入)。所以如果在事务开头重新执行加载,如果没有重新加载非推测值,那么加载可能无意地加载在中止之前在推测寄存器文件 1035 中保存的后来修改的推测值。

[0173] 此外,为响应事务的提交,将推测寄存器文件 1035 提交(拷贝)到非推测寄存器文件 1030 中。在这里,使推测更新作为非推测结果而全局可见。再次,可以将整个的寄存器文件 1035 拷贝到非推测寄存器文件 1030 中。或将仅有那些在推测寄存器文件 1035 中的已被修改的寄存器拷贝到非推测寄存器文件 1030。

[0174] 转到图 11,示出了适用于支持高速缓存存储器的推测检查点的硬件的逻辑表示的另一实施例。如上文参考图 9-10,解码器 1110 和执行逻辑 1120 适用于解码和执行 SCPI1105。在这里,当执行来自原子区的推测存储时,执行逻辑 1120 使用存储缓冲器 1140 来保存推测更新。注意,从加载自以前存储的同一区域(本地线程)的加载从在存储缓冲器 1140 中保存的推测值加载。因此,可以使用在存储缓冲器 1140 和执行逻辑 1120 的加载执行单元之间的即时存储机构的类似加载。但是,对在存储缓冲器 1140 中存储的地址的非推测或非本地加载将接收在高速缓存 1130 中保存的非推测值,而不是在存储缓冲器 1140 中的值。此外,如果有来自原子区到被在检查点检查或移动到推测高速缓存 1135 的存储的地址的读取 / 加载,则应直接地或通过存储缓冲器 1140 将推测高速缓存值提供给加载。

[0175] 为响应 SCPI1105,将在缓冲器 1140 中的存储缓冲器更新移动到推测高速缓存 1135。结果是,推测高速缓存 1135 保存从原子区的开头到最近检查点的推测更新。并且存储缓冲器 1140 保存从最近检查点到当前执行检查点的推测更新。因此,在提交原子区时,将在推测高速缓存 1135 和存储缓冲器 1140 中的所有更新提交 / 移动 / 拷贝到非推测高速缓存 1130。如所示的,通过将存储缓冲器 1140 提交到推测高速缓存 1135 以及推测高速缓存 1135 提交到非推测高速缓存 1130 来执行提交。但是在另一个实施例中,将来自存储缓冲器 1140 和推测高速缓存 1135 的更新直接提供给非推测高速缓存 1130。在提交之后,更新是全局可见的并且通过存储器体系 1150 (更高级别的存储器并到起始单元)进行传播。

[0176] 此外,为响应到最近检查点的本地的内部回滚,清除了存储缓冲器 1140。如上所

述的，在这个实施例中，存储缓冲器 1140 必要地保存从最近检查点到当前执行点的更新。所以在回滚时，丢弃那些更新。注意，在一个示例中，可以启动本地回滚来响应存储缓冲器 1140 不能容纳来自原子区的新存储操作（图 8 的块 840）。并且在块 820 中，还可以启动回滚来响应推测高速缓存 1135 已满而不能基于 SCPI1105 高速缓存存储缓冲器 1140 更新。但是，当发生中止（整个原子区的回滚）时，则清除存储缓冲器 1140 和推测高速缓存 1135（从原子区的开头到当前执行点的更新）。

[0177] 在本文使用的模块是指任何硬件、软件、固件或其组合。通常被图示为分离的模块边界一般是变化的并且有可能重叠。例如，第一模块和第二模块可以共享硬件、软件、固件或其组合，同时潜在地保留一些独立的硬件、软件或固件。在一个实施例中，术语逻辑的使用包括硬件，例如，晶体管、寄存器或者其它硬件，例如可编程逻辑器件。然而，在另一实施例中，逻辑也可以包括与硬件集成的软件或代码，例如固件或微码。

[0178] 在这里使用的值包括数字、状态、逻辑状态或二值逻辑状态的任何已知表示。通常，对逻辑电平或者逻辑值的使用也被称为 1 或 0，这简单地表示二值逻辑状态。例如，1 指高逻辑电平而 0 指低逻辑电平。在一个实施例中，储存单元（例如晶体管或闪存单元）可以具有保持单个逻辑值或者多个逻辑值的能力。然而，已经使用了计算机系统中值的其它表示。例如，十进制数字 10 也可以被表示为二进制值 1010，以及十六进制字母 A。因此，值包括能够被保持在计算机系统中的信息的任何表示。

[0179] 此外，状态可以由值或者值的部分来表示。例如，第一值，例如逻辑 1，可以表示默认或初始状态；而第二值，例如逻辑 0，可以表示非默认状态。另外，在一个实施例中，术语重置和置位分别指默认的和更新的值或状态。例如，默认值可能包括高逻辑值，即重置，而更新的值可能包括低逻辑值，即置位。注意，值的任何组合可以被用来表示任何数量的状态。

[0180] 上面阐述的方法、硬件、软件、固件或代码集的实施例可以通过储存在可由处理部件执行的机器可访问或机器可读介质上的指令或代码来实现。机器可访问 / 可读介质包括提供（即储存和 / 或传送）机器（例如，计算机或电子系统）可读形式的信息的任何机制。例如，机器可访问介质包括随机存取存储器（RAM），如静态 RAM（SRAM）或动态 RAM（DRAM）；ROM；磁或光储存介质；闪存器件；电储存器件、光储存器件、声储存器件或其它形式的传播信号（例如，载波、红外信号、数字信号）储存器件等等。

[0181] 在说明书中通篇提到“一个实施例”或者“实施例”意指结合该实施例描述的特定特征、结构或特性被包括在本发明的至少一个实施例中。因此，短语“在一个实施例中”在说明书中各处出现并非必须全都指同一实施例。此外，所述特定的特征、结构或特性可以在一个或更多实施例中以任何适当的方式组合。

[0182] 在前述说明书中，已经参照特定的示例性实施例给出了详细的描述。然而显而易见的是，可以对本发明作出各种修改和改变而不偏离如所附权利要求书中所给出的本发明的宽泛精神和范围。因此，说明书和附图是要被视为说明性的而非限制性的。此外，前面对实施例和其它示例性语言的使用并非必须指同一实施例或同一实例，而是可以指不同的或相异的实施例，以及可能地指同一实施例。

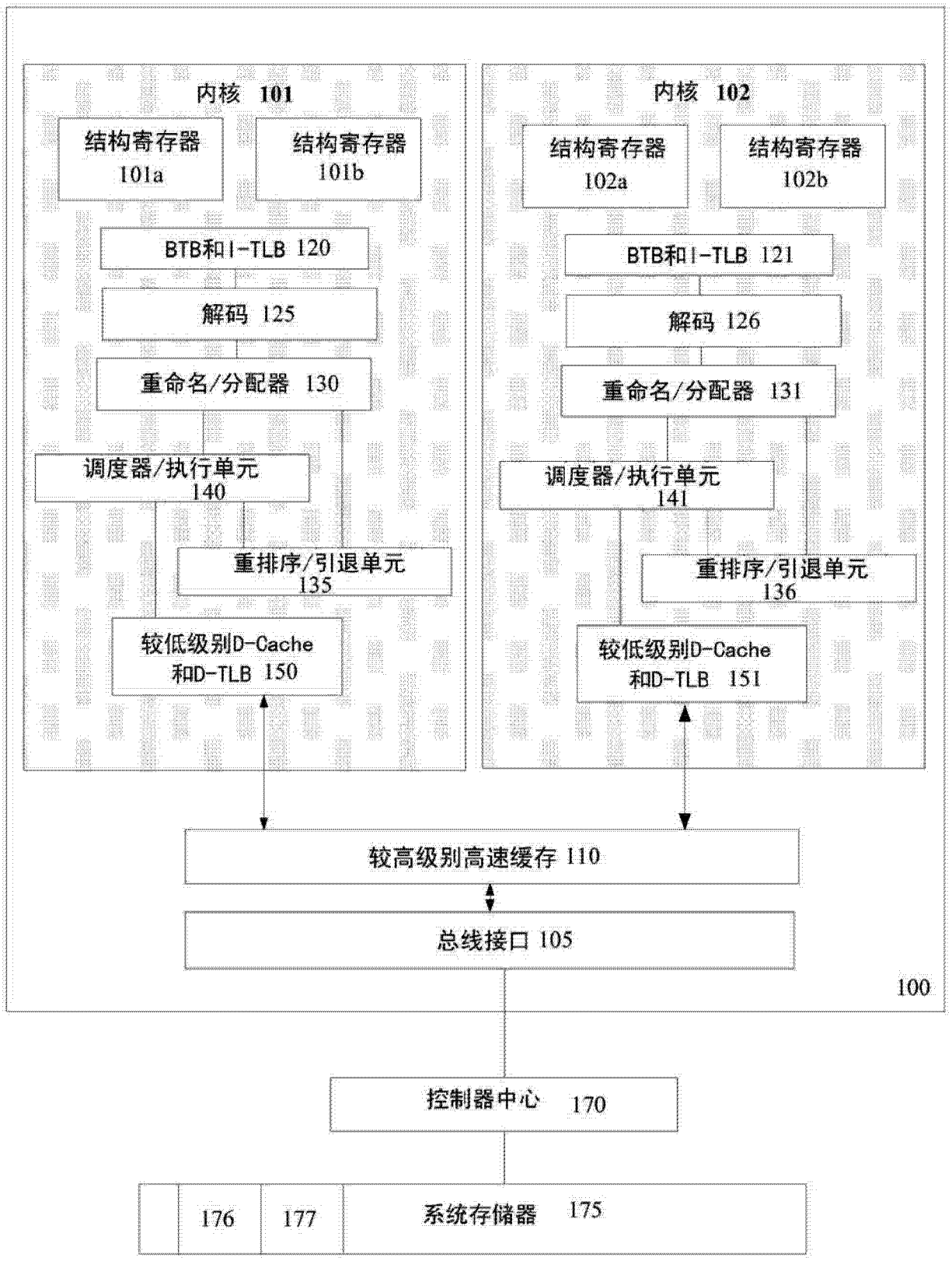

内核 101
结构寄存器 101a
结构寄存器 101b
BTB和I-TLB 120
解码 125
重命名/分配器 130
调度器/执行单元 140
重排序/引退单元 135
较低级别D-Cache 和D-TLB 150
内核 102
结构寄存器 102a
结构寄存器 102b
BTB和I-TLB 121
解码 126
重命名/分配器 131
调度器/执行单元 141
重排序/引退单元 136
较低级别D-Cache 和D-TLB 151
较高级别高速缓存 110
总线接口 105
100
控制器中心 170
176
177
系统存储器 175

图 1

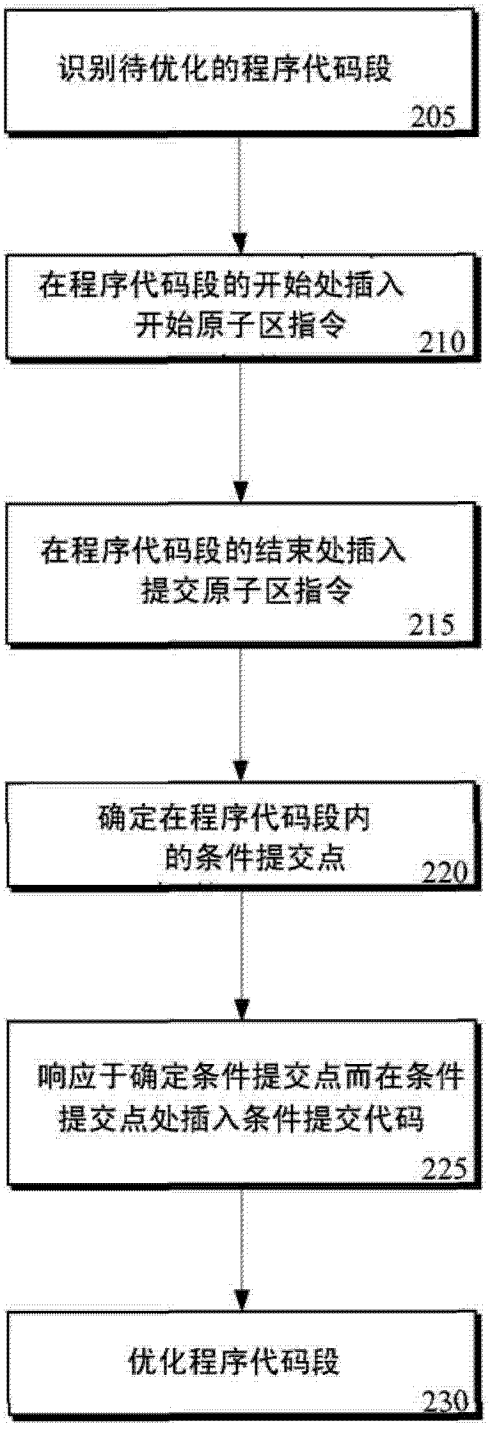

识别待优化的程序代码段
205
在程序代码段的开始处插入
开始原子区指令
210
在程序代码段的结束处插入
提交原子区指令
215
确定在程序代码段内
的条件提交点
220
响应于确定条件提交点而在条件
提交点处插入条件提交代码
225
优化程序代码段
230

图 2a

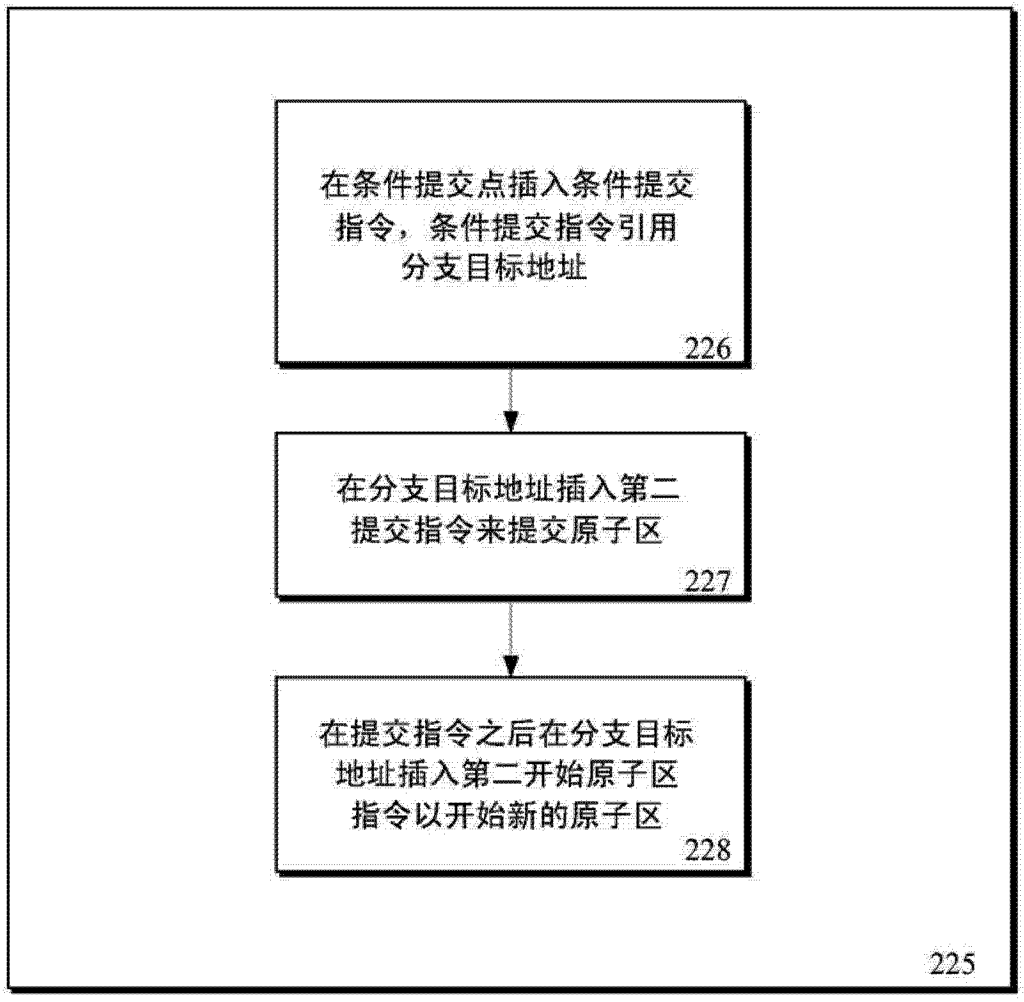

在条件提交点插入条件提交指令，条件提交指令引用分支目标地址
226
在分支目标地址插入第二提交指令来提交原子区
227
在提交指令之后在分支目标地址插入第二开始原子区指令以开始新的原子区
228
225

图 2b

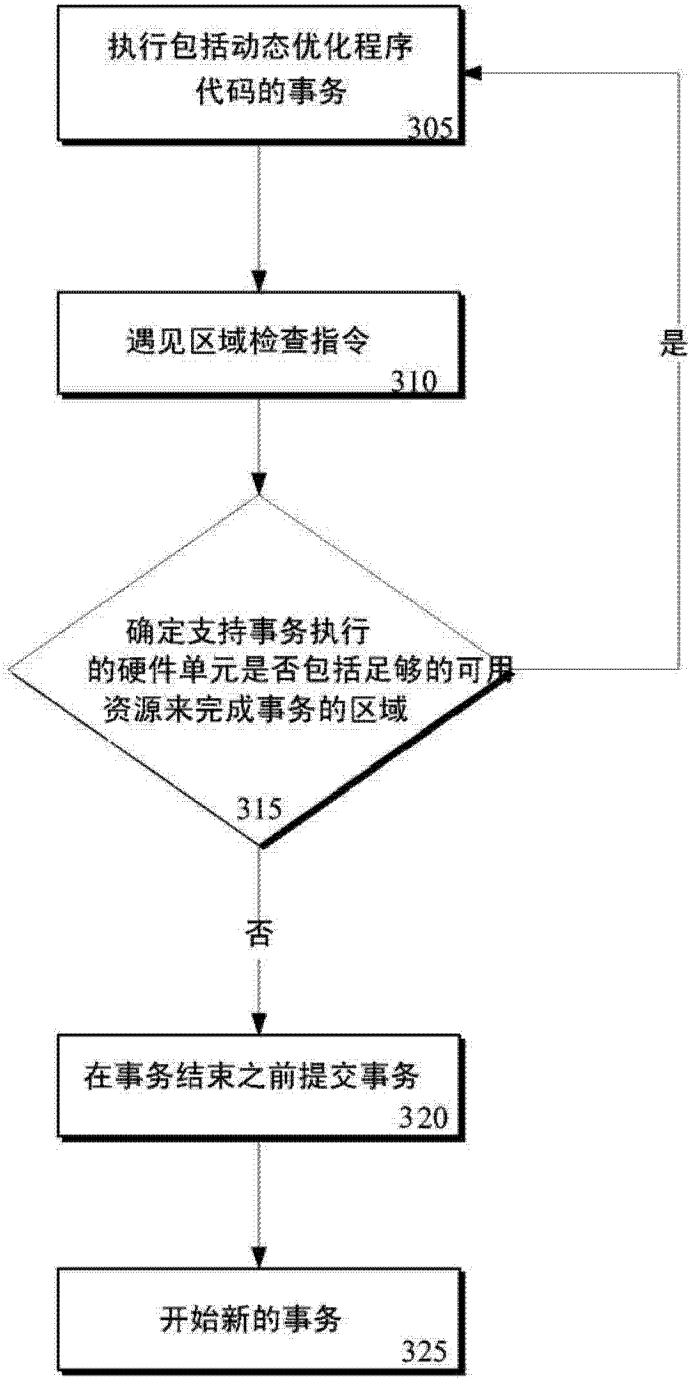

执行包括动态优化程序代码的事务
305
遇见区域检查指令
310
确定支持事务执行的硬件单元是否包括足够的可用资源来完成事务的区域
315
是
否
在事务结束之前提交事务
320
开始新的事务
325

图 3a

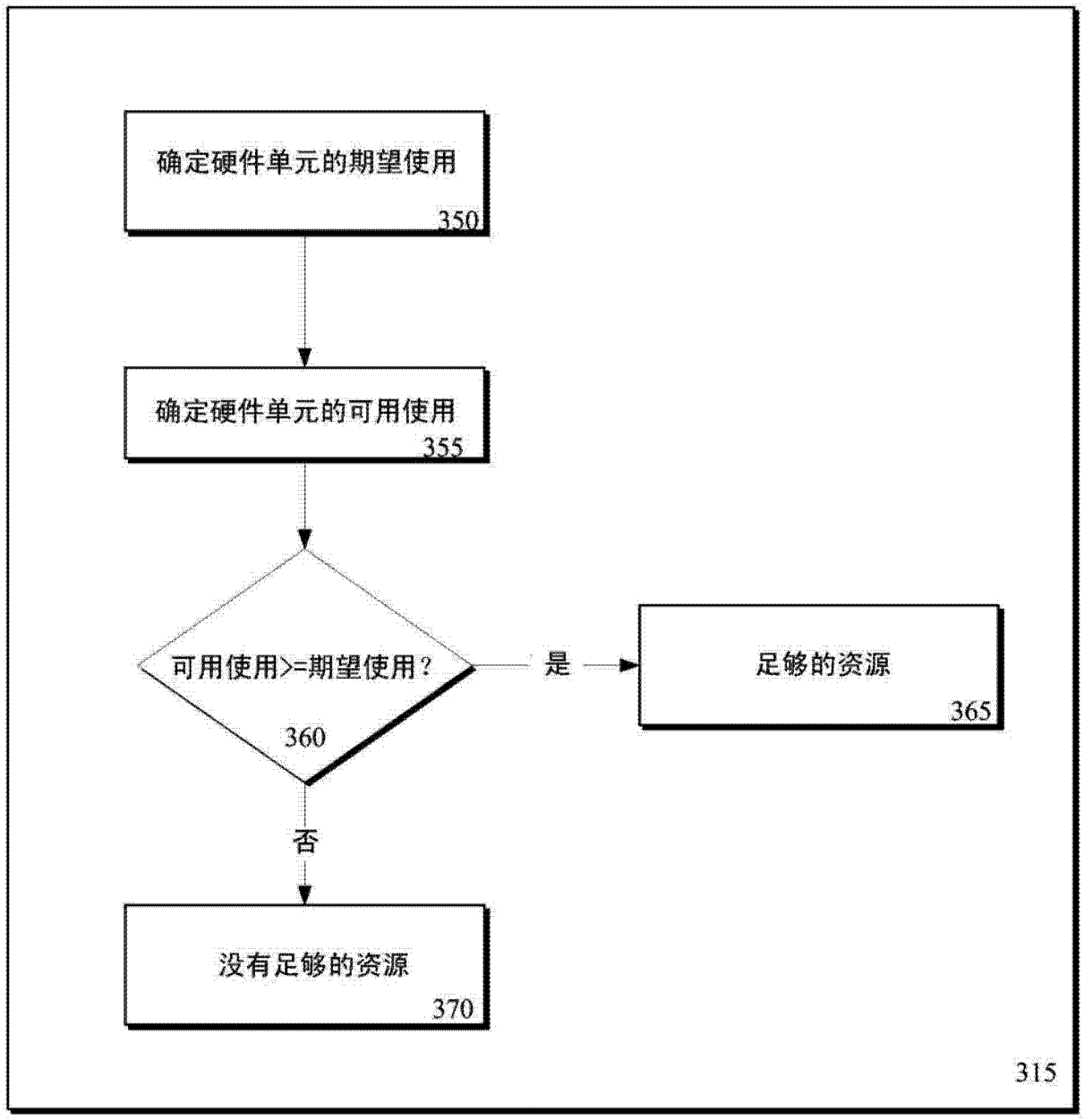

确定硬件单元的期望使用
350
确定硬件单元的可用使用
355
可用使用>=期望使用?
360
是
足够的资源
365
否
没有足够的资源
370
315

图 3b

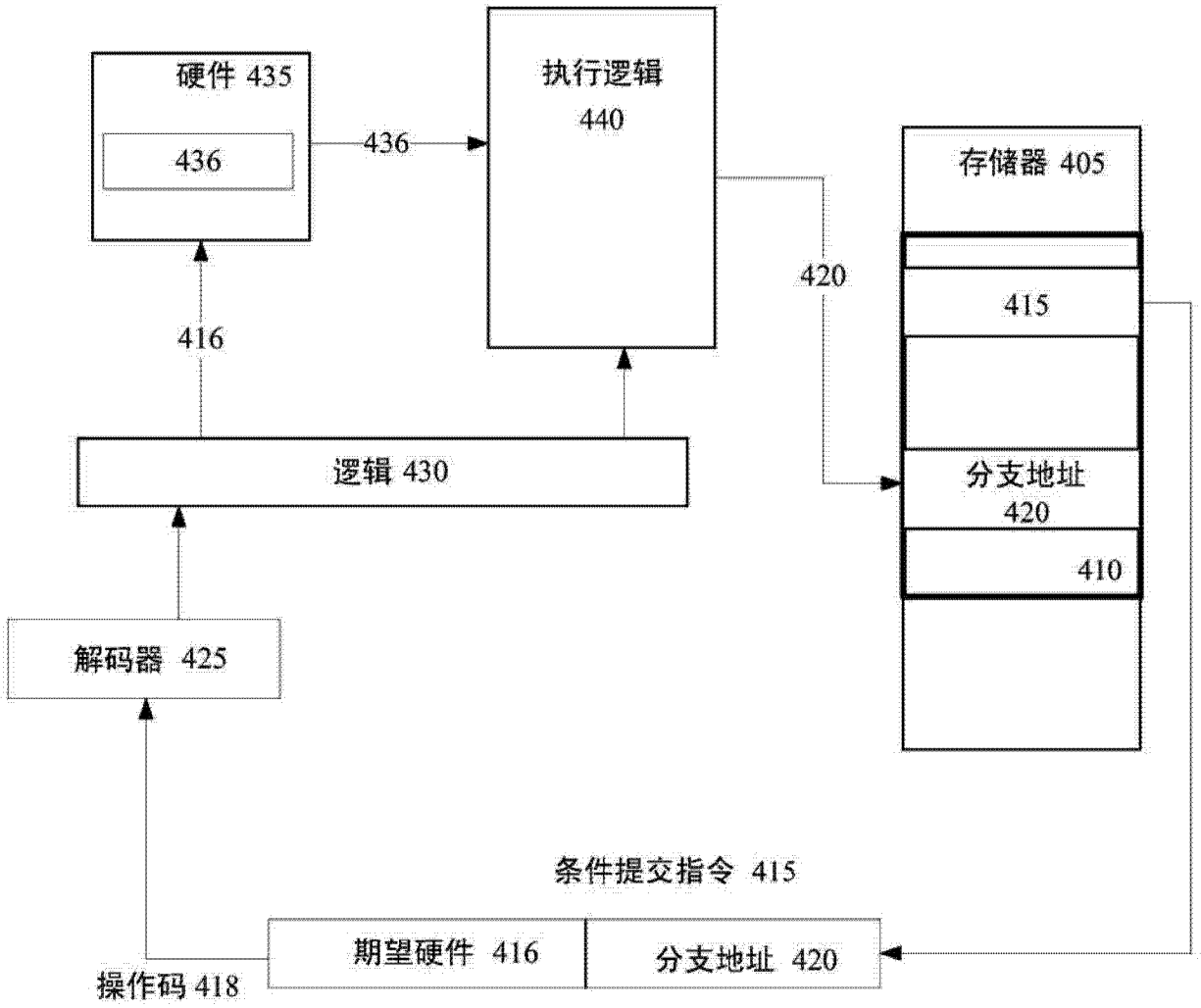

硬件 435
436
436
执行逻辑
440
存储器 405
420
415
416
逻辑 430
分支地址
420
410
解码器 425
条件提交指令 415
期望硬件 416
分支地址 420
操作码 418

图 4

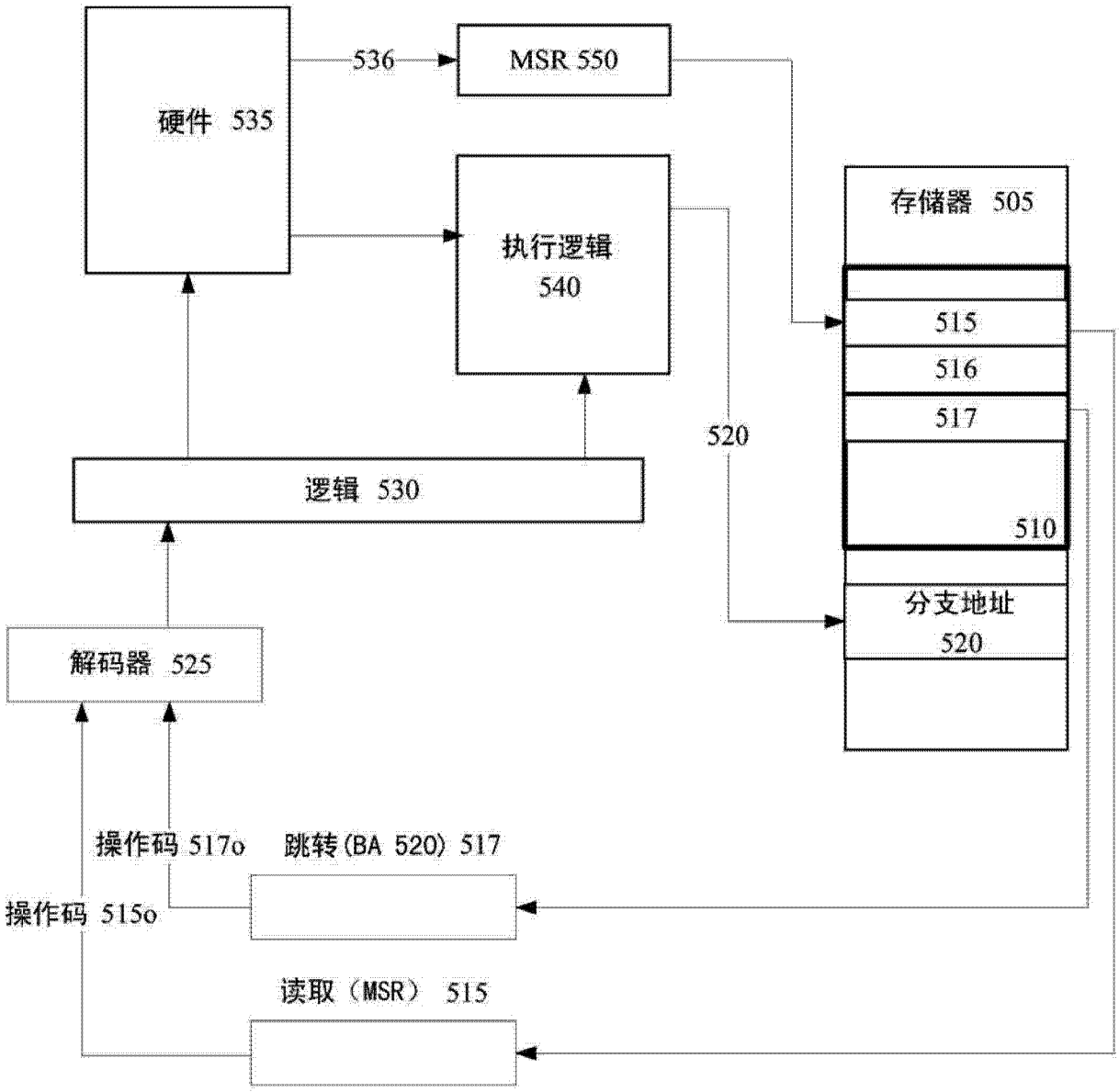

硬件 535
536
MSR 550
执行逻辑
540
存储器 505
515
516
517
510
520
逻辑 530
分支地址
520
解码器 525
操作码 517o
跳转(BA 520) 517
操作码 515o
读取（MSR） 515

图 5

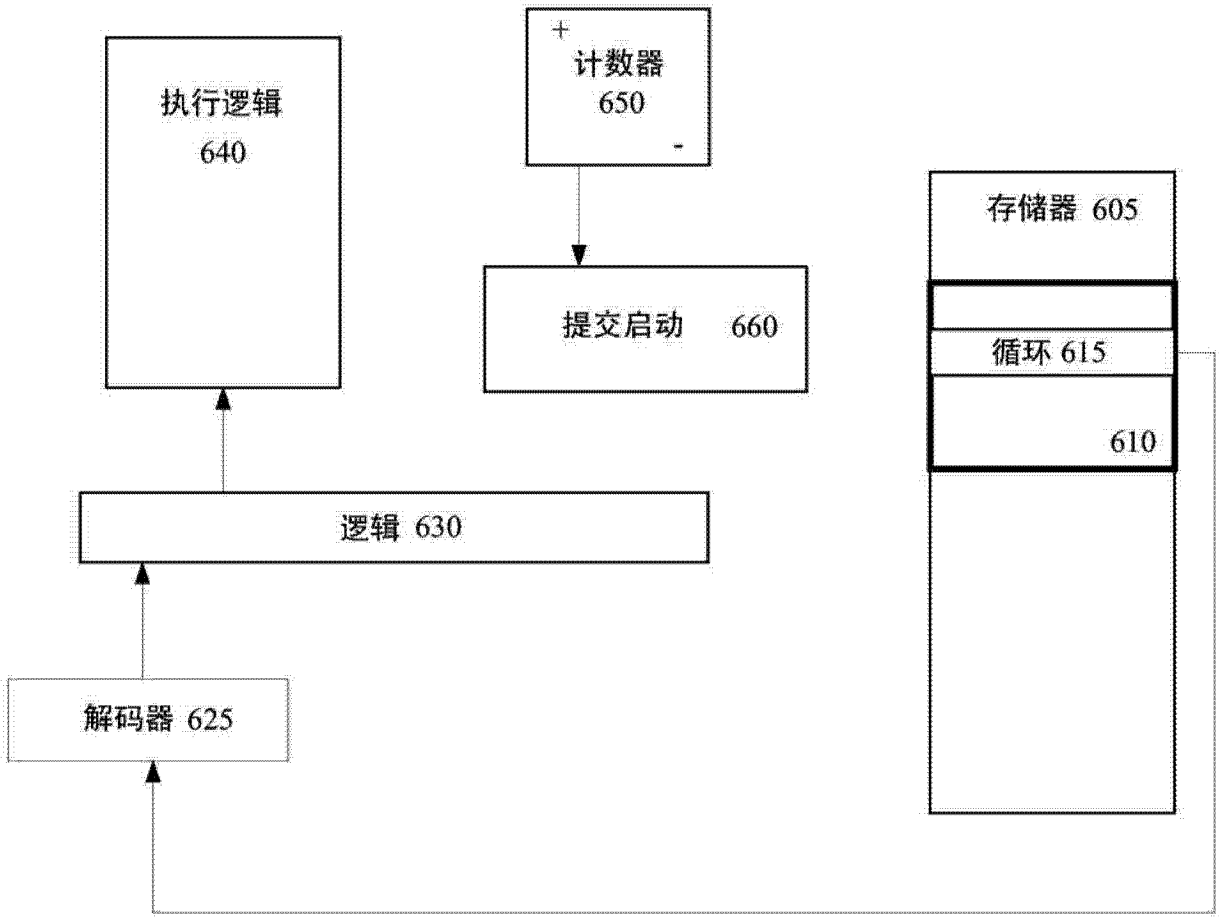


图 6

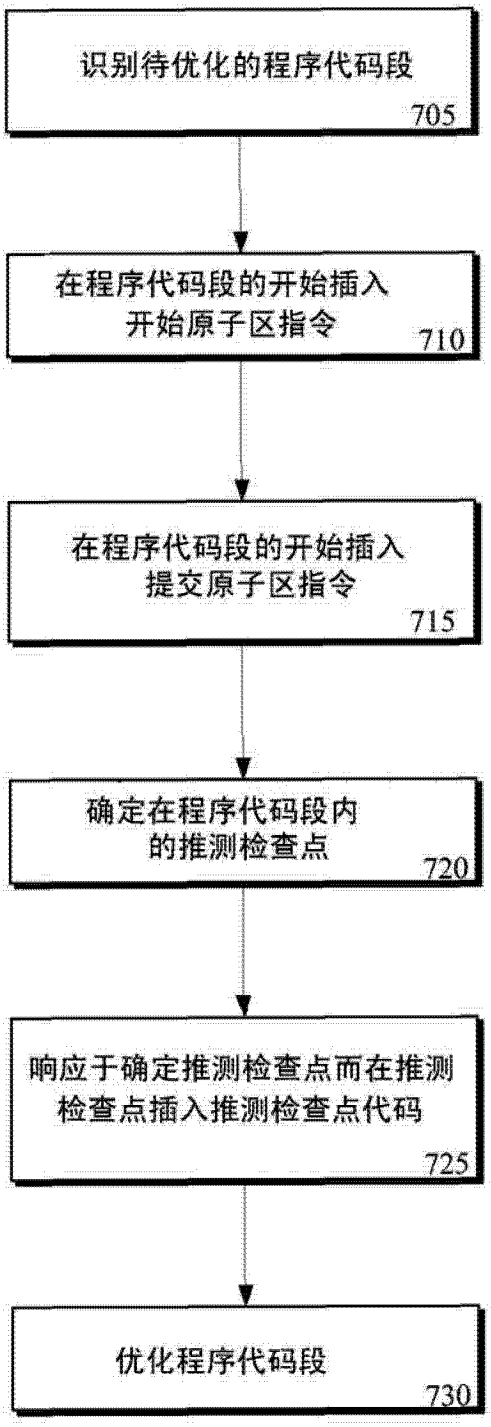

识别待优化的程序代码段
705
在程序代码段的开始插入
开始原子区指令
710
在程序代码段的开始插入
提交原子区指令
715
确定在程序代码段内
的推测检查点
720
响应于确定推测检查点而在推测
检查点插入推测检查点代码
725
优化程序代码段
730

图 7a

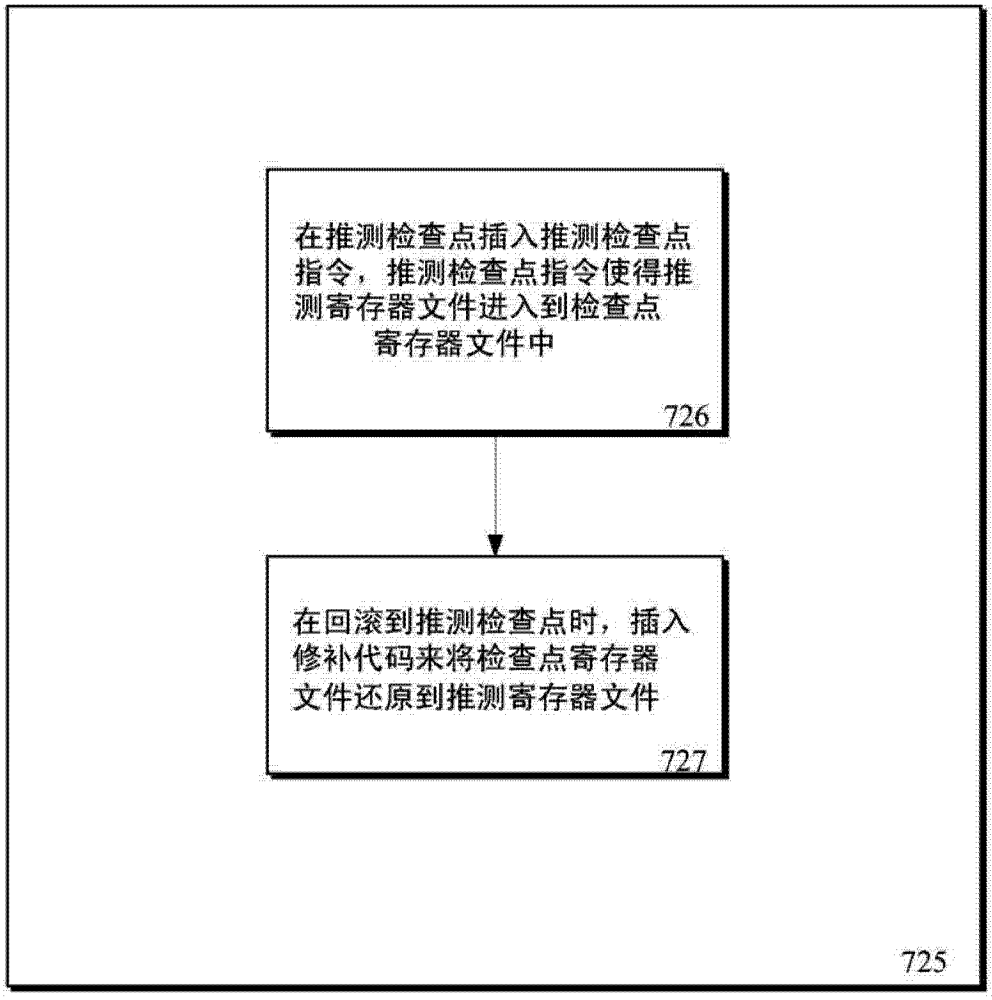

在推测检查点插入推测检查点指令，推测检查点指令使得推测寄存器文件进入到检查点寄存器文件中
726
在回滚到推测检查点时，插入修补代码来将检查点寄存器文件还原到推测寄存器文件
727
725

图 7b

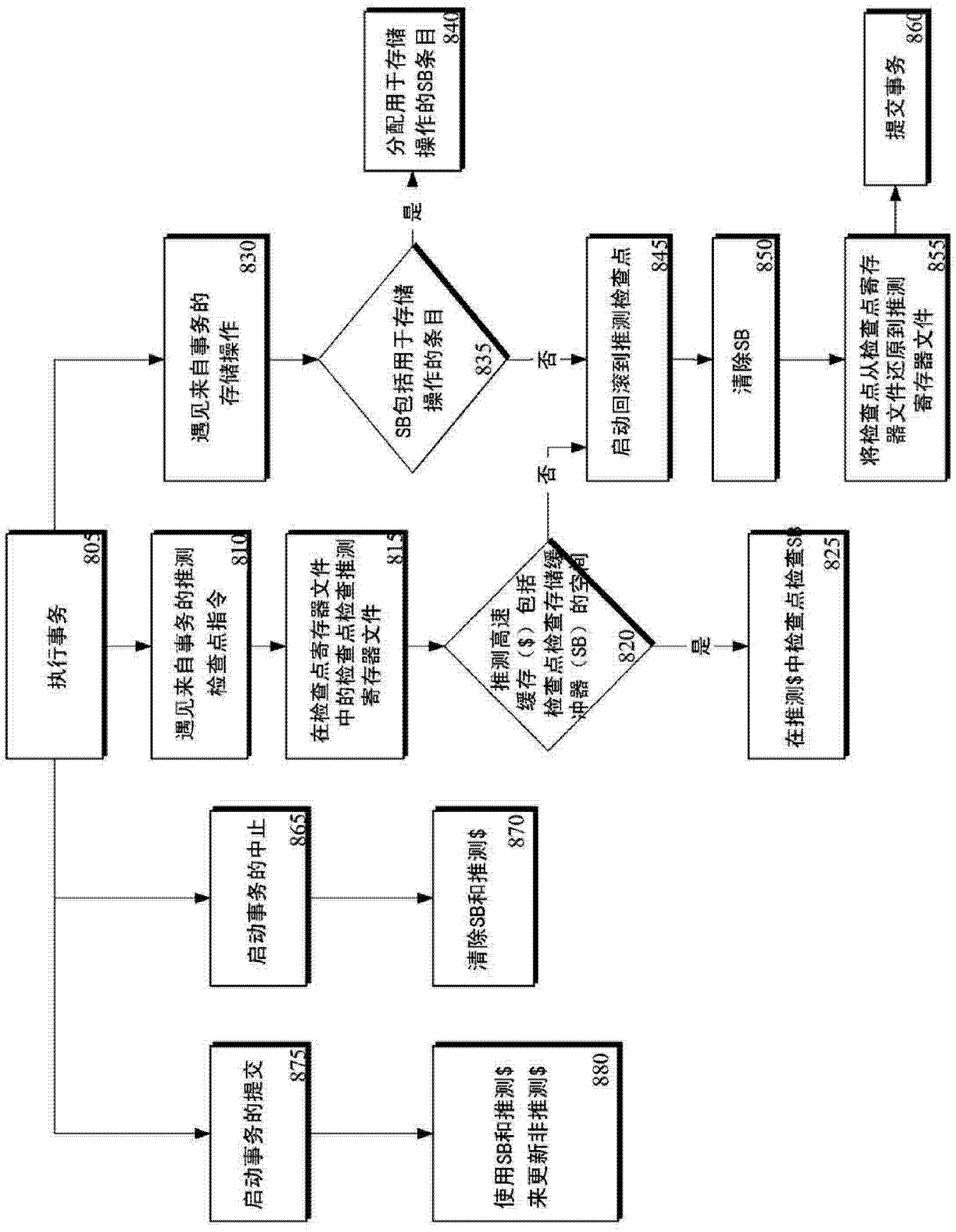

执行事务
805
遇见来自事务的推测检查点指令
810
在检查点寄存器文件中的检查点检查推测寄存器文件
815
推测高速缓存（$）包括检查点检查存储缓冲器（SB）的空间
820
是
否
在推测$中检查点检查SB
825
遇见来自事务的存储操作
830
SB包括用于存储操作的条目
835
是
否
分配用于存储操作的SB条目
840
启动回滚到推测检查点
845
清除SB
850
将检查点从检查点寄存器文件还原到推测寄存器文件
855
提交事务
860
启动事务的中止
865
清除SB和推测$
870
启动事务的提交
875
使用SB和推测$来更新非推测$
880

图 8

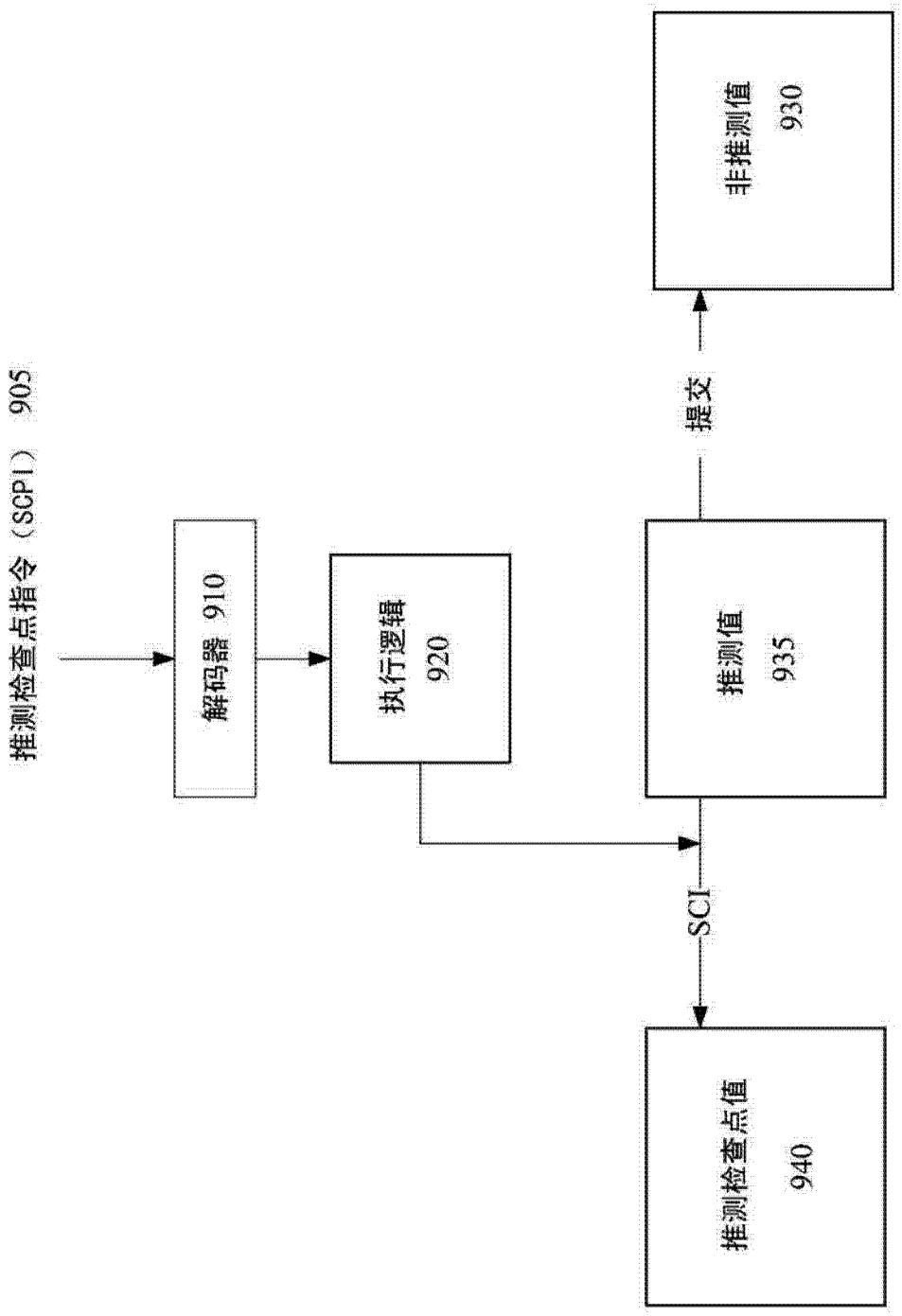

推测检查点指令（SCPI） 905
解码器 910
执行逻辑 920
推测值 935
非推测值 930
提交
SCI
推测检查点值 940

图 9

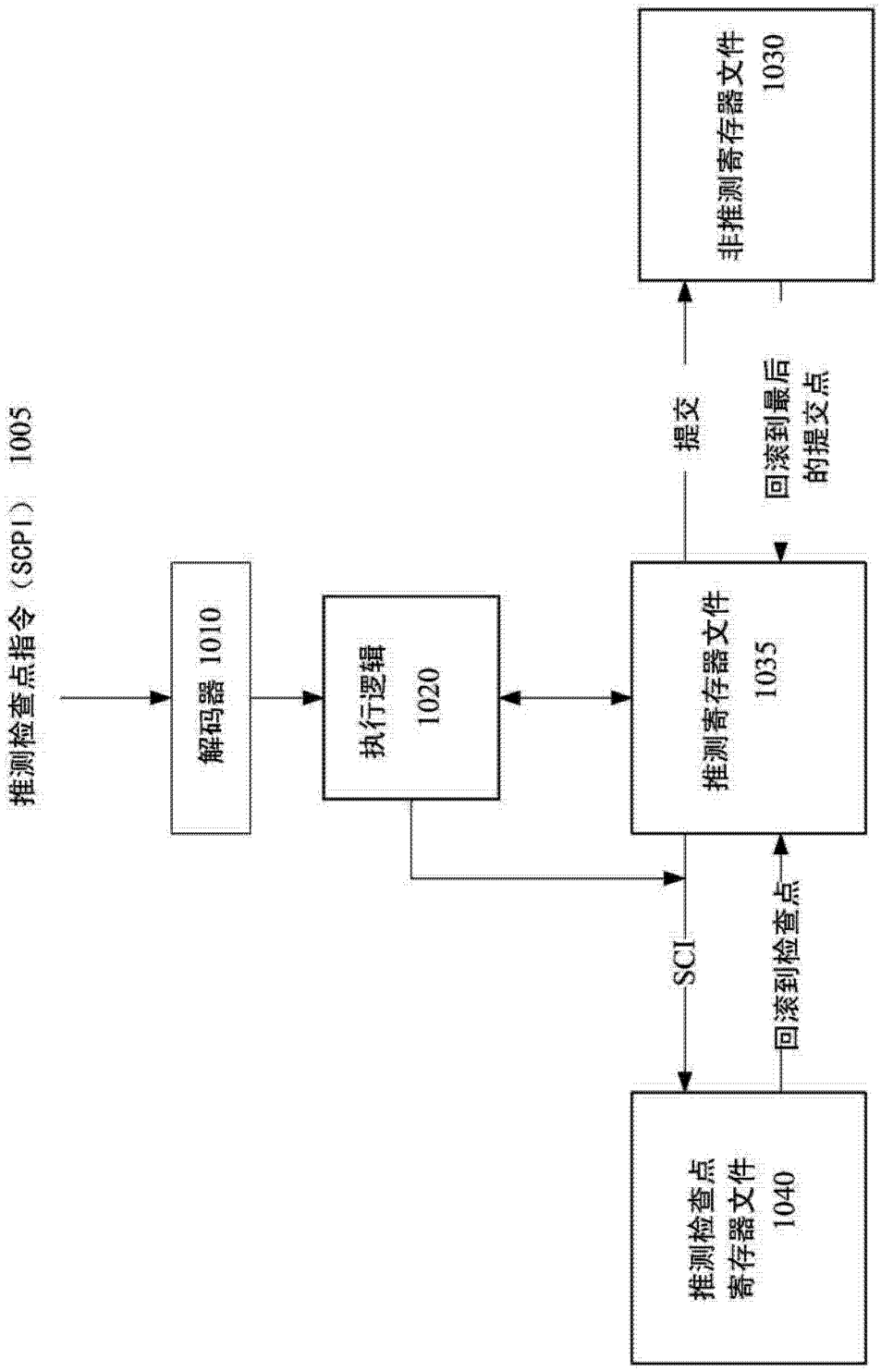

推测检查点指令（SCPI） 1005
解码器 1010
执行逻辑 1020
推测寄存器文件 1035
非推测寄存器文件 1030
提交
回滚到最后的提交点
SCI
回滚到检查点
推测检查点寄存器文件 1040

图 10

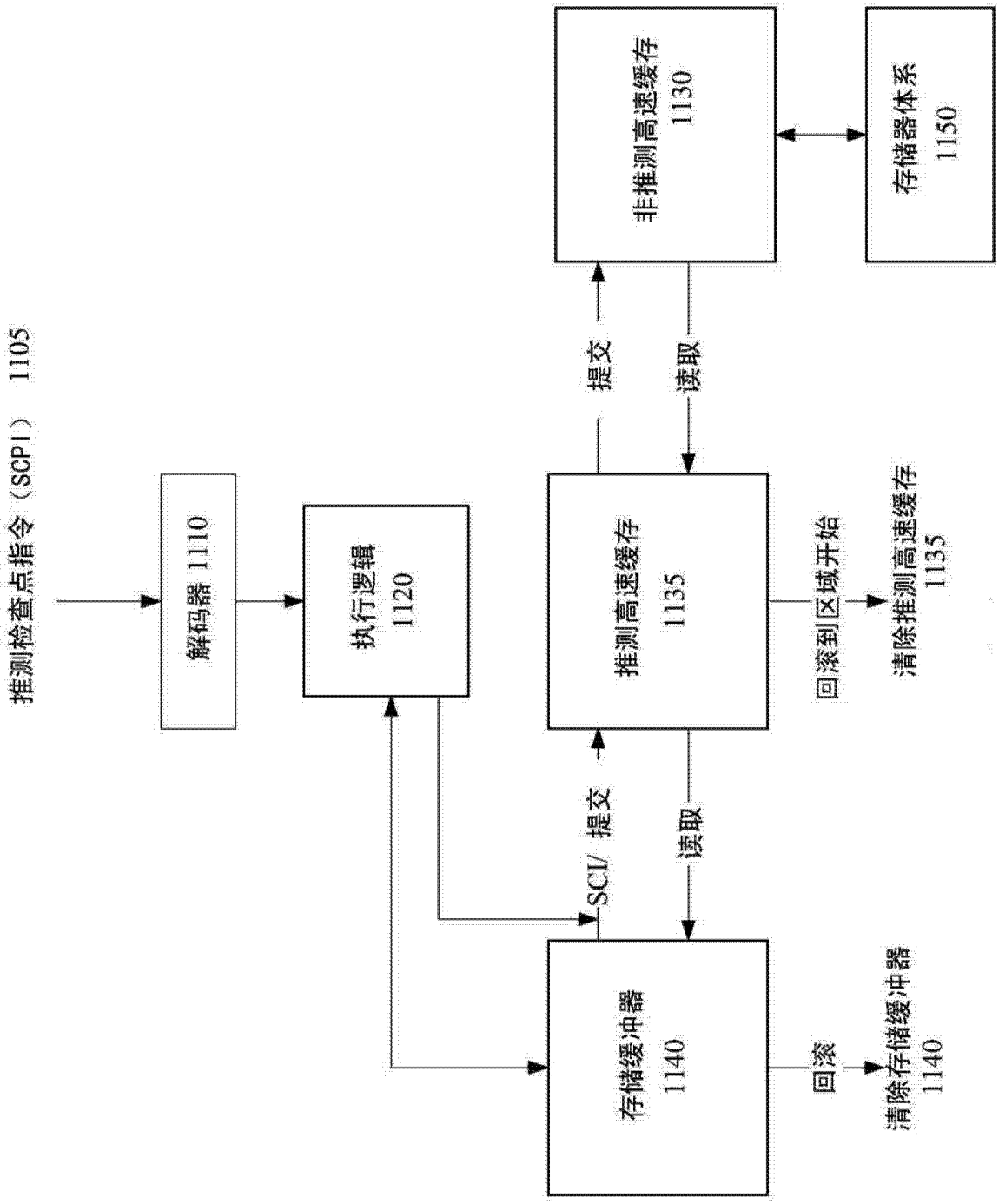

推测检查点指令（SCPI） 1105
解码器 1110
执行逻辑 1120
非推测高速缓存 1130
存储器体系 1150
推测高速缓存 1135
提交
读取
回滚到区域开始
清除推测高速缓存 1135
SCI/ 提交
读取
存储缓冲器 1140
回滚
清除存储缓冲器 1140

图 11