[19] 中华人民共和国国家知识产权局

[51] Int. Cl.
G06F 9/45 (2006.01)
G06F 9/46 (2006.01)

[12] 发明专利说明书

专利号 ZL 200480013989.4

[45] 授权公告日 2008 年 8 月 27 日

[11] 授权公告号 CN 100414503C

[22] 申请日 2004.3.19
[21] 申请号 200480013989.4
[30] 优先权
[32] 2003.5.20 [33] US [31] 10/441,357
[86] 国际申请 PCT/US2004/008589 2004.3.19
[87] 国际公布 WO2004/104823 英 2004.12.2
[85] 进入国家阶段日期 2005.11.21
[73] 专利权人 英特尔公司
地址 美国加利福尼亚州
[72] 发明人 吴甘沙 路奎元 石小华
[56] 参考文献
US2002040470A1 2002.4.4
审查员 董 鑫

[74] 专利代理机构 北京嘉和天工知识产权代理事务所
代理人 严 慎

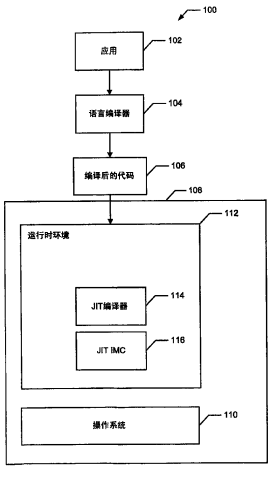
权利要求书 3 页 说明书 9 页 附图 8 页

[54] 发明名称

用于在受控运行时环境中恢复同步到面向对象的软件应用的装置和方法

[57] 摘要

公开了用于在受控运行时环境中恢复同步到面向对象的软件应用的装置和方法。所公开的装置和方法确定一个类对象是否已被动态加载，对程序代码执行逸出分析，并且确定在初始的逸出分析期间做出的假设是否还有效。另外，如果在初始逸出分析期间所做的假设不再有效，那么所公开的系统和方法恢复同步到因所述类对象的加载而受到影响的至少一部分程序代码。



1. 一种恢复同步到程序代码的方法，包括：

确定类对象是否已被动态加载；

当确定所述类对象已被动态加载时，对所述程序代码执行逸出分析；

确定在初始逸出分析期间做出的假设是否有效；以及

如果在初始逸出分析期间所做的假设不再有效，那么识别并恢复同步到因所述类对象的加载而受到影响的所述程序代码中的至少一部分。

2. 如权利要求 1 所述的方法，其中在初始逸出分析期间做出的假设与在某一具体时间上只对一个执行线程可访问的至少一个对象相关联。

3. 如权利要求 1 所述的方法，其中所述程序代码是建立在动态编程语言之上。

4. 如权利要求 3 所述的方法，其中所述动态编程语言是 Java。

5. 如权利要求 1 所述的方法，其中所述程序代码的所述至少一部分是调用地点。

6. 如权利要求 1 所述的方法，其中恢复同步到所述程序代码的至少一部分的步骤包括对与所述程序代码的所述至少一部分相关联的执行线程进行锁定/解锁补偿。

7. 如权利要求 6 所述的方法，还包括在对所述执行线程进行锁定/解锁补偿之前，暂停所述执行线程。

8. 如权利要求 6 所述的方法，还包括在对所述执行线程进行锁定/解锁补偿之前，修补与所述执行线程相关联的调用地点。

9. 如权利要求 6 所述的方法，其中对所述执行线程进行锁定/解锁补偿的步骤包括使用标签和 cookie 之一将与所述程序代码相关联的地址改变为桩代码地址。

10. 如权利要求 9 所述的方法，还包括执行与所述桩代码地址相关联的桩代码，以解锁所述程序代码。

11. 一种用于恢复同步到程序代码的设备，包括：

确定类对象是否已被动态加载的装置；

当确定所述类对象已被动态加载时，对所述程序代码执行逸出分析的装置；

确定在初始逸出分析期间做出的假设是否有效的装置；以及

如果在初始逸出分析期间所做的假设不再有效，那么识别并恢复同步到因所述类对象的加载而受到影响的所述程序代码中的至少一部分的装置。

12. 如权利要求 11 所述的用于恢复同步到程序代码的设备，其中在初始逸出分析期间做出的假设与在某一具体时间上只对一个执行线程可访问的至少一个对象相关联。

13. 如权利要求 11 所述的用于恢复同步到程序代码的设备，其中所述程序代码是建立在动态编程语言之上。

14. 如权利要求 13 所述的用于恢复同步到程序代码的设备，其中所述动态编程语言是 Java。

15. 如权利要求 11 所述的用于恢复同步到程序代码的设备，其中所述程序代码的所述至少一部分是调用地点。

16. 如权利要求 11 所述的用于恢复同步到程序代码的设备，还包括：通过对与所述程序代码的所述至少一部分相关联的执行线程进行锁定/解锁补偿，从而恢复同步到所述程序代码的所述至少一部分的装置。

17. 如权利要求 16 所述的用于恢复同步到程序代码的设备，还包括：在对所述执行线程进行锁定/解锁补偿之前，暂停所述执行线程的装置。

18. 如权利要求 16 所述的用于恢复同步到程序代码的设备，还包括：在对所述执行线程进行锁定/解锁补偿之前，修补与所述执行线程相关联的调用地点的装置。

19. 如权利要求 16 所述的用于恢复同步到程序代码的设备，还包括：使用标签和 cookie 之一将与所述程序代码相关联的地址改变为桩代码地址，从而对所述执行线程进行锁定/解锁补偿的装置。

20. 如权利要求 19 所述的用于恢复同步到程序代码的设备，还包括：执行与所述桩代码地址相关联的桩代码，以解锁所述程序代码的装置。

21. 一种恢复同步到程序代码的方法，包括：
对所述程序代码反复执行逸出分析；
识别所述程序代码中将为其恢复同步的至少一个部分；
暂停与所述程序代码相关联的至少一个执行线程；
将所述程序代码修补为原始的同步版本；以及
对与所述至少一个执行线程相关联的代码进行锁定/解锁补偿。

22. 如权利要求 21 所述的方法，其中将所述程序代码修补为原始的同步版本的步骤包括修补与所述程序代码的所述至少一个部分相关联的调用地点。

23. 如权利要求 21 所述的方法，其中对与所述至少一个执行线程相关联的代码进行锁定/解锁补偿的步骤包括将堆栈帧中的一个地址改变为指向桩代码。

24. 如权利要求 21 所述的方法，其中反复执行逸出分析的步骤包括每次加载一个新的类用于执行，就执行所述逸出分析。

25. 一种具有与之相关联的操作系统的计算机系统，所述计算机系统包括：

建立有运行时环境的装置，所述运行时环境可操作地耦合到所述操作系统；

可操作地耦合到所述运行时环境的即时编译器，其中所述即时编译器包括：

用于响应于所述运行时环境接收到新的类，执行程序代码的逸出分析的装置；

用于识别与将为之恢复同步的程序代码相关联的至少一个调用地点的装置；以及

用于恢复同步到所述至少一个调用地点的装置。

26. 如权利要求 25 所述的计算机系统，其中所述运行时环境是建立在虚拟机上。

27. 如权利要求 25 所述的计算机系统，其中所述用于恢复同步到所述至少一个调用地点的装置包括用于通过暂停至少一个执行线程，将所述至少一个调用地点修补为原始的同步版本，并且对所述至少一个执行线程进行锁定/解锁补偿的装置。

用于在受控运行时环境中恢复同步到面向对象的软件应用的装置和方法

技术领域

本公开总地涉及受控运行时环境，更具体地说，涉及用于在受控运行时环境中恢复同步到面向对象的软件应用的装置和方法。

背景技术

对日益增加的软件应用可移植性的需要（即，在具有不同的硬件、操作系统等的多种平台上执行给定的软件应用的能力）以及减少独立软件开发商（ISV）抢占市场的时间的需要已经导致日益增多的受控运行时环境的开发和使用。

受控运行时环境一般使用动态编程语言，例如 Java、C#等来实现。通常被称为运行时环境的软件引擎（例如 Java 虚拟机（JVM）、公共语言运行时（CLR）等）执行动态程序语言指令。运行时环境插入在将被执行的动态程序语言指令（例如，Java 程序或源代码）和目标执行平台（即，执行动态程序的计算机的硬件和操作系统）之间，或者说在两者之间起到接口的作用，使得动态程序可以以与平台无关的方式来执行。

动态程序语言指令（例如 Java 指令）不是被静态编译并直接链接到本机码或机器码以供目标平台（即，目标处理系统或平台的操作系统和硬件）执行的。相反，动态程序语言指令被静态编译为中间语言，而中间语言可以被即时（JIT）编译器解释或者接着编译为可由目标处理系统或平台执行的本机码或机器码。一般地，JIT 编译器是由容留在目标处理平台（例如计算机系统）的操作系统中的运行时环境来提供的。因此，运行时环境，具体地说是 JIT 编译器将与平台无关的程序指令（例如，Java 字节码、C#字节码等）翻译成本机码（即，可由底层的目标处理系统或平台执行的机器码）。

为了提高整体产率，很多动态编程语言以及它们提供支持的受控运行时环境都提供使并发编程技术（例如，多线程技术）能够被利用的基础结构。具体地说，很多动态编程语言都提供同步特征或操作，以使得多个并发的执行线程能够共享或者访问给定的对象及其变量，而不会引起冲突或争用。例如，在全局可访问对象（即，公共对象）的情形中，软件设计者一般假定在运行时期间会发生冲突或争用，并且在对象内包括适当的同步操作，以防止这样的冲突或争用。按照这种方式，软件设计者可以保证全局可访问对象是“线程安全的”（即，可以没有冲突或争用地被用于多线程运行时环境中）。

不幸的是，与对象同步相关联的处理开销导致执行时间的大幅减少。例如，在一些公知的 Java 应用和基准程序（benchmark）的情形中，同步开销可能会占用整个执行时间的

10%到20%。此外，不管在运行时期间是否真的需要这种同步，同步通常都被用作在运行时期间防止发生争用（特别是在对象库的情形中）的防护措施。

已知的逸出分析（escape analysis）技术可被用来提高包括不必要同步在内的代码的整体执行速度。总地来说，已知的逸出分析技术采用一种完整程序分析，以便能够去除与在程序的运行时执行期间将不会为之发生争用的全局对象和非全局对象相关联的同步操作。已知的逸出分析技术是基于静态链接代码模型的，该模型假定在运行时期间将不会加载任何新的对象类。然而，一些流行的编程语言，例如 Java 和 CLI 都提供了动态类加载特征，该特征允许将要在运行时上下文中被调用的函数或方法的动态链接。在一些情形中，将一个类动态地加载到正在执行先前已优化程序（例如，同步已被完全或部分去除的程序）的运行时环境中，这可能导致程序工作在非安全方式下（例如数据争用）或者发生故障。虽然已知的基于静态链接代码模型的逸出分析技术可被用来从静态链接代码中去除同步，但是当先前失去同步的（即，经过优化的）代码随后因被动态加载的类的影响又需要同步时，这些已知的技术不支持恢复同步到先前失去同步的代码。此外，其他开放世界的特征，例如反射和本机法可以使逸出分析无效，从而导致不安全的执行条件。

发明内容

为了解决上述技术问题，本发明的一方面在于提供一种恢复同步到程序代码的方法，所述方法包括确定类对象是否已被动态加载；当确定所述类对象已被动态加载时，对所述程序代码执行逸出分析；确定在初始逸出分析期间做出的假设是否有效；以及如果在初始逸出分析期间所做的假设不再有效，那么恢复同步到因所述类对象的加载而受到影响的所述程序代码中的至少一部分。

本发明的另一方面在于提供一种用于恢复同步到程序代码的设备，所述设备包括确定类对象是否已被动态加载的装置；当确定所述类对象已被动态加载时，对所述程序代码执行逸出分析的装置；确定在初始逸出分析期间做出的假设是否有效的装置；以及如果在初始逸出分析期间所做的假设不再有效，那么恢复同步到因所述类对象的加载而受到影响的所述程序代码中的至少一部分的装置。

本发明的再一方面在于提供一种恢复同步到程序代码的方法，所述方法包括对所述程序代码反复执行逸出分析；识别所述程序代码中将为其恢复同步的至少一个部分；暂停与所述程序代码相关联的至少一个执行线程；将所述程序代码修补为原始的同步版本；以及对与所述至少一个执行线程相关联的代码进行锁定/解锁补偿。

本发明的再又一方面在于提供一种具有与之相关联的操作系统的计算机系统，所述计算机系统包括：可操作地耦合到所述操作系统的运行时环境；可操作地耦合到所述运行时环境的即时编译器，其中所述即时编译器被配置为：响应于所述运行时环境接收到新的类，执行程序代码的逸出分析；识别与将为之恢复同步的程序代码相关联的至少一个调用地

点；以及恢复同步到所述至少一个调用地点。

附图说明

图 1 是可被用来实现这里描述的同步恢复装置和方法的示例性体系结构的框图。

图 2 图示了与在逸出分析之后已为之去除同步的类相关联的基于 Java 的代码示例。

图 3 图示了用于另一类的基于 Java 的代码示例，所述类如果被动态加载，则使对调用地点 A 的同步去除无效。

图 4 是图 1 中所示的即时编译器可被配置来恢复同步到受动态加载的类影响的调用地点的示例性方式的流程图。

图 5 是图 1 中所示的即时编译器可被配置来恢复同步到受影响的调用地点的示例性方式的更详细流程图。

图 6 是描绘一种图 1 中所示的即时编译器可被配置来为与受影响的调用地点相关联的每个线程执行锁定/解锁补偿的方式的示例性伪码。

图 7 图示了不管与调用地点 A 相关联的类的动态加载，仍保持对其的同步去除有效的基于 Java 的代码示例。

图 8 和 9 图解式描绘了一种示例性的方式，图 1 中所示的即时编译器可被配置来以此方式执行解锁补偿操作。

图 10 是描绘一种方式的示例性代码，以此方式，cookie COMP_UNLOCK_TAG 可被用来在堆栈展开上下文中恢复返回地址。

图 11 是可被用来实现这里描述的装置和方法的示例性处理器系统。

具体实施方式

图 1 是可被用来实现这里描述的同步恢复装置和方法的示例性体系结构 100 的框图。对于示例性的体系结构 100，由一种或多种动态编程语言和/或指令组成的一个或多个软件应用 102 被提供给语言编译器 104。应用 102 可以使用与平台无关的语言来编写，例如 Java 或 C#。然而，也可以使用其他动态的或与平台无关的计算机语言或指令。另外，应用 102 中的一些或全部可被存储在将要执行该应用的系统中。附加地或替换地，应用中的一些或全部可被存储在与将要执行应用 102 的系统相互独立（并且可能位于远端）的系统上。

语言编译器 104 静态编译应用 102 中的一个或多个，以生成编译后的代码 106。编译后代码 106 是以二进制格式存储在存储器（未示出）中的中间语言代码或指令（例如，当编译后（complied）的应用是用 Java 写的时，是字节码）。对于应用 102，编译后的代码 106 可被本地存储在将要在其上执行编译后代码 106 的目标系统 108 上。目标系统 108 可以是下面参考图 11 所详细描述的计算机系统等。目标系统 108 可以与一个或多个终端用户等相关联。附加地或替换地，编译后代码 106 可以经由一条或多条通信链路被传递到目标系统 108，所述通信链路例如是局域网、因特网、蜂窝系统或其他无线通信系统等。

编译后代码 106 的一个或多个部分（例如一个或多个软件应用）可以由目标系统 108 来执行。具体地说，操作系统 110，例如 Windows、Linux 等容留可执行编译后代码 106 的一个或多个部分的运行时环境 112。例如，如果编译后的代码 106 包括 Java 字节码，那么运行时环境 112 就是建立在执行 Java 字节码的 Java 虚拟机（JVM）等之上。运行时环境 112 将编译后代码 106 的一个或多个部分（即，中间语言指令或代码）加载到运行时环境 112 可以访问的存储器（未示出）中。优选地，运行时环境 112 将整个应用（或者有可能是多个应用）加载到存储器中，并验证编译后的或中间语言代码 106 的类型安全性。

在应用或者多个应用被运行时环境 112 加载到存储器中之后，与正在执行的应用所调用的、或者为执行该应用所需的方法或对象相关联的中间语言指令可以由即时（JIT）编译器 114 来处理。JIT 编译器 114 编译中间语言指令，以生成由计算机系统 108 内的一个或多个处理器（例如，图 11 中所示的处理器 1122）执行的本机码或机器码。

JIT 编译器 114 可以将本机码（即，与计算机系统 108 兼容的，或者可由计算机系统 108 执行的机器码）存储在 JIT 内存缓存（JIT IMC）116 中。按照这种方式，运行时环境 112 可以重复使用与先前编译过的、被调用一次以上的方法相关联的本机码。换言之，被编译为本机码并被存储在 JIT IMC 116 中的中间语言指令可以被运行时环境 112 重复使用并执行多次。

虽然 JIT IMC 116 被描绘为实现在运行时环境 112 内，但是 JIT IMC 116 的其他配置也是可能的。例如，JIT IMC 116 可以是操作系统 110 所容留的其他运行时模块、会话或环境（未示出）内的另一个数据结构的一部分。在其他实施例中，特别是涉及虚拟调用的实施例中，可以实现 JIT IMC 116，使得与将被调用的方法相关联的本机码被存储在公知的数据结构中，例如虚拟分派（dispatch）表。

总的来说，动态编程语言例如 Java 提供两种类型的使软件设计者能够生成线程安全码或软件对象（software object）的同步。同步后的软件对象一次仅可以由一个执行线程来访问，从而防止与该对象所使用的参数或变量相关的冲突或争用的发生。换言之，全局对象以及可由一个以上的执行线程访问的其他对象可以通过引入软件锁定和解锁机制而成为线程安全的，上述机制防止一个以上的线程同时访问对象。一种这样类型的同步使得代码块（即，一条或多条语句）被同步。另一种这样类型的同步使得方法（即，对代码块的调用）被同步。

动态编程语言为了同步代码块和方法，一般既提供高级或语言级同步语句，又提供低级或受控运行时级原语。例如，在 Java 的情况下，关键字“synchronized（已同步）”在语言级（即，高级）被用来宣告一个块或方法将受到同步保护。另外，在 Java 的情况下，对应于语言级关键字“synchronized”的低级或受控运行时原语是“monitorenter”和“monitorexit”。然而，为了简化以下的讨论，低级同步原语将被称为“lock（锁定）”和“unlock

（解锁）”，而高级或语言级同步语句将使用关键字“synchronized”来表示。

如上所述，已知的逸出分析技术一般采用完整程序分析，该分析基于的假设是：一旦程序执行已经启动（即，在运行时期间），将不会加载附加的对象类。具体地说，用于执行逸出分析的算法可以使用上下文敏感型方法，这些方法对编译后的程序执行过程间（inter-procedural）分析，以确定被传递给被调用者（例如，被调用的方法、对象等）的参数是否被指定为静态变量。这样的完整程序分析或过程间分析是基于以下假设：在运行时期间将不加载新的对象类，并且这些分析通常被称为封闭世界分析。这样的封闭世界分析经常可导致激进的同步码优化。换言之，与在这样的封闭世界分析时不发生争用的全局对象和非全局对象相关联的所有同步操作都可以被消除，从而大大加快 JIT 编译器 114 可执行代码的速率。

这里描述的同步恢复装置和方法使得受控运行时环境（例如 JVM 环境）能够即时编译以下代码：所述代码基于新的类将不被动态加载（即，在运行时期间加载）的假设，已在一开始被优化为激进地去除同步操作。因而，初始优化（即，去同步）可以基于传统的或已知的逸出分析，这种分析执行封闭世界类型的分析，导致例如与不发生争用的全局对象或非全局对象相关联的同步操作的去除。然而，与已知的技术相反，这里描述的同步恢复装置和方法可以用在动态类加载上下文中。具体地说，这里描述的重新同步装置和方法使得 JIT 编译器（例如，JIT 编译器 114）能够判定新加载的类是否以使在初始的封闭世界类逸出分析期间做出的优化判决（例如，同步操作去除）无效的方式改变当前正被执行的程序的行为。换言之，当 JIT 编译器 114 在运行时或者它当前正在处理的程序或代码的执行期间遇到新加载的类时，对包括新加载的类在内的完整程序执行逸出分析，并且同步操作被恢复到需要来防止在多个执行线程之间的冲突或争用的代码或程序。下面将更详细地描述，这里描述的同步恢复装置和方法可以恢复同步到（修补同步回到）作为初始的封闭世界逸出分析的结果先前已被去同步的调用地点。另外，这里描述的同步恢复装置和方法还分析运行时上下文，并生成补偿代码，例如锁定和解锁操作，以保持正确的同步语义。

动态加入的类（即，在程序的运行时执行期间加入的）可以无效初始的封闭世界类逸出分析的结果，为了更好地理解这种示例性的状况，在图 2 和 3 中提供了示例性的基于 Java 的代码。具体地说，图 2 描绘了示例性的基于 Java 的代码，在对图 2 所示代码的初始的封闭世界类逸出分析期间，为了所述代码可以从调用地点 A 去除同步。更具体地说，因为至少在一开始的时候，“obj”没有逸出方法“caller”，所以可以从调用地点 A 去除同步。换言之，对于在当前正在执行方法“caller”的执行线程之外的执行线程而言，“obj”不是可访问的或者说可用的。结果，在用“foo”的未同步版本完成初始的封闭世界类型的逸出分析之后，“foo”的同步版本可以被取代。

图 3 描绘了包括从“Class 1”导出的新类“Class 2”在内的基于 Java 的代码示例。如果在包含图 2 所示代码的程序正被执行时，对象“Class2”被动态加载，那么在 Class1 中定义的公共静态字段“global”也可以是 Class2 的一个实例。因此，因为在 Class2 内的虚拟方法

“do_nothing”指定“obj”为“global”，所以“obj”对多个执行线程变为可用（即，逸出所述线程）结果，在调用地点 A 处的同步去除不再有效，程序的继续操作可能需要与调用地点 A 相关联的代码的重新同步。

图 4 是图 1 中所示的即时编译器 114 可被配置来恢复同步到或者重新同步受动态加载的类（例如，结合图 2 和 3 所描述的 Class2）影响的调用地点的示例性方式的流程图。一开始，运行时环境 112 判断一些开放世界特征（例如，新的类被加载（框 400））是否存在并将有可能破坏或无效先前做出的去同步决定。如果 JIT 编译器 114 在框 400 处确定不存在新加载的类，那么 JIT 编译器停留在框 400 处。相反，如果 JIT 编译器 114 确定新的类已被加载（框 400），则 JIT 编译器 114 对当前正被执行的整个程序，包括新加载的类在内执行逸出分析（框 402）。可以使用任何需要的技术来实现逸出分析，以识别逸出线程的对象（即，同时可由一个以上的执行线程来访问的对象）。

在逸出分析（框 402）之后，JIT 编译器 114 确定在新的类加载（框 400）之前，在初始的完整程序分析期间所做出的假设是否保持有效（框 404）。换言之，JIT 编译器 114 判断逸出分析（框 402）是否已确定新加载的类改变了程序的行为，以致于多个执行线程现在可以访问原先是线程安全的（即，一次只由一个执行线程访问，因此没有逸出它当前的执行线程），但现在不是线程安全的对象。具体地说，新加载的类可以导致早先的逸出分析所做决定的失效以及用于去除与某一调用地点相关联的非必要同步语句的优化例程。在早先的逸出分析期间所执行的去同步通常都是基于以下假设，即一个或多个对象将会保持线程安全。例如，一旦加载了在图 3 中所示的代码内定义的新类（即，Class2），在对图 2 中所示的代码执行的初始逸出分析期间所做的假设将会失效。

无论如何，如果 JIT 编译器 114 确定在先前的逸出分析期间所做的假设不再有效（例如，一个或多个对象不再是线程安全的，并且可能需要重新同步），那么 JIT 编译器 114 识别因目前的错误假设受到影响或冲击的调用地点（框 406）。在已经识别出受影响的调用地点（框 406）之后，JIT 编译器 114 恢复同步到这些地点中的每一个（框 408），下面将结合图 5 来更详细地描述。

图 5 是 JIT 编译器 114 可被配置来恢复同步到受影响的调用地点（图 4 的框 408）的示例性方式的更详细流程图。首先，JIT 编译器 114 暂停所有当前活动的执行线程（框 500）。按此方式暂停执行线程保证了如以下描述所完成的修补（patching）和锁定/解锁补偿可以以安全的方式来实现（即，不损害当前正在执行的进程）。此外，因为实际当中，新类的动态加载在运行时期间不是很频繁的活动，所以与所有线程活动的完全暂停相关联的开销（即，时延）对整体产率的影响几乎可以忽略不计。

在当前正在执行的线程已经暂停（框 500）之后，JIT 编译器 114 将受影响的调用地点修补回它们原始的同步版本（框 502）。例如，如果目标代码地址已被改变为引用一个方法的去同步版本，则该目标代码地址可以被改变回（即，修改为）原始的目标调用地址。

在修补（框 502）之后，JIT 编译器 114 根据需要为每一个暂停的线程执行锁定/解锁补偿（框 504）。如果在修补（框 502）前完成的对非同步代码的调用之后，并且在返回到正在调用的程序之前，一个或多个线程已经被暂停，那么就需要锁定/解锁补偿。在这些情况下，在修补（框 502）后立即取消线程活动暂停，这样允许 JIT 编译器 114 调用一个方法的同步版本，而同一方法已经启动（即，在框 500 处暂停之前已启动）的非同步版本将继续执行。结果，与该方法相关联的对象可以同时被两个执行线程访问，这可能导致对于一个或多个资源的不安全条件。

可以参考图 2 和 3 中所示的示例性代码来理解使用锁定/解锁补偿的方式。举例来说，如果 JIT 编译器 114 确定在 Class2 已被加载后，在调用地点 A 处的同步去除不再有效（即，在图 4 的框 402 处的逸出分析已识别出一个或多个逸出其执行线程的对象），则 JIT 编译器 114 完成调用地点 A 的重新同步（即，恢复同步到调用地点 A）（图 4 的框 408）。如果在执行锁定/解锁补偿（图 5 的框 504）时，JIT 编译器 114 确定第一执行线程当前正与对“foo”的非同步版本的活动调用相关联，则需要锁定/解锁补偿。具体地说，如果第一线程在“global”变成 Class2 的一个实例的同时开始执行“global.do_nothing（this）”，其利用“do_nothing”方法将“obj”指定为“global”，那么第二线程可以同时执行“global.foo（）”。结果，JIT 编译器 114 可以在相同的“obj”上执行“foo”的同步和非同步两个版本，从而违反了同步语义，并建立了不安全的共享资源条件。

图 6 是描绘一种图 1 中所示的 JIT 编译器 114 可被配置来为与受影响的调用地点相关联的每个线程执行锁定/解锁补偿（图 5 的框 504）的一种方式的示例性伪码。具体地说，图 6 中所描绘的示例性方法从栈顶到栈底遍访所有的堆栈帧（即，与当前活动的线程相关联的帧）。更具体地说，图 6 中所描绘的示例性方法通过对 z 反复执行锁定操作，其中 z 是“<x,y>”的被调用者，并且它是堆栈上的最后一帧（即，在堆栈上是活动的），从而对于每个受影响的地点“<x,y>”（即，必须为其恢复同步的每个调用地点）保持正确的同步语义。在 z 是静态方法的情况下，对类对象执行锁定操作。相反，如果 z 是一个实例方法（instance method）（即，涉及虚调用），则 JIT 编译器 114 使用任何需要的技术来定位将对其执行锁定操作的对象。

函数“compensate_lock_unlock（）”可被用来执行锁定操作，以保持正确的锁定顺序。另外，这个函数还保证正确的解锁顺序被执行。例如，如果正由 JIT 编译器 114 处理的代码是基于 Java 的，那么锁定和解锁操作必须被正确地配对，以实现同步的方法。具体地说，有两个地方必须执行解锁对象：一旦退出同步方法之后，以及在例外处理中同步方法的破坏性展开期间。

图 6 中所描绘的示例性方法使得 JIT 编译器 114 能够仅仅补偿代码中受新加入的类影响的那些部分，而将代码中未受新加载的类影响的部分留着不动。换言之，简单地修补一个方法的非同步版本的尾部代码可能会超出必要地反优化代码，因为运行时环境对于未受

新加载的类影响的其他调用地点可能需要（或者在不引入争用的情况下能够使用）所述方法的去同步版本。具体地说，由 JIT 编译器 114 执行的逸出分析可以单独地识别受新加载的类影响的调用地点（即，可以安全去除同步的原始假设对其不再有效的调用地点）以及调用所述方法的去同步版本对其仍然安全的那些调用地点。

图 7 图示了不管 Class2（图 2）的动态加载，同步去除对其仍保持有效的基于 Java 的代码示例。具体地说，当 Class2 被新加载时，在调用地点 B 对非同步的"foo"的调用仍保持有效，这是因为逸出分析确定"obj"的类型不是 Class2 类型的。

图 8 和 9 描绘了一种示例性的方式，在图 6 的示例性伪码中所示的 compensate_lock_unlock 方法可以以此方式完成。总的来说，compensate_lock_unlock 函数使得 JIT 编译器 114 能够将位于与当前正被执行的非同步方法相关联的堆栈帧的底部的返回地址重定向到或"劫持"到执行适当的解锁操作的桩代码（stub code）。对于图 8 中所示的例子，JIT 编译器 114 识别需要为其执行解锁操作的非同步方法"foo"的返回地址的位置。如图 9 中所示，JIT 编译器 114 用桩代码的地址替换非同步方法 foo'的堆栈帧中的返回地址。因而，当非同步方法 foo'的执行结束时，控制权被转移给适当的桩代码。接着，桩代码执行需要的解锁操作，并使 JIT 编译器 114 能够跳回到原始的返回地址。桩代码包括硬编码的句柄地址，该地址代表了需要为其执行解锁操作的对象的实际地址。

图 10 是描绘一种方式的示例性伪码，以此方式，cookie COMP_UNLOCK_TAG 可以与图 8 和 9 中所描绘的示例性技术结合起来，用于在堆栈展开上下文中恢复返回地址。在堆栈展开上下文中，运行时环境（例如，虚拟机）一般依赖于返回地址找到堆栈内的下一帧。例如，公知的开放运行时平台（ORP）将每个方法的起始和结束地址存储在查找表中。因而，给定了一个代码地址，虚拟机就可以经由查找表找到包含该代码地址的方法。然而，如参考图 8 和 9 所述地重定向或劫持返回地址将使得查找操作不能找到正确的下一帧。

图 10 中所示的伪码包括 cookie COMP_UNLOCK_TAG，这意味着与这个标签相邻的代码是补偿解锁桩代码（stub code）。如图 9 所示，实际的返回地址和 COMP_UNLOCK_TAG 在桩代码之前被紧紧捆绑在一起。结果，如果 JIT 编译器 114 没有在查找表中找到关联的方法，则 JIT 编译器 114 检查与之相邻的字是不是 cookie COMP_UNLOCK_TAG。如果找到了 cookie，则 JIT 编译器 114 返回实际的返回地址，该地址紧跟在 cookie 之前。cookie COMP_UNLOCK_TAG 被定义为一个非法代码序列，使得 JIT 编译器 114 不将 cookie 与编译后代码相互混淆。

在破坏性堆栈展开过程的情形中，JIT 编译器 114 通过经由上面参考图 10 所描述的 COMP_UNLOCK_TAG cookie 来识别当前活动的堆栈帧，从包含在桩代码中的句柄地址提取实际对象引用，并对对象执行解锁操作，从而补偿缺失的解锁操作。

虽然前面以示例的方式将锁定/解锁补偿描述为结合方法或调用来使用，但是本领域

的技术人员将很容易意识到，这里所描述的技术和装置可以被容易地调整为与代码块一起使用，以取得类似的或相同的结果。

图 11 是可被用来实现这里描述的装置和方法的示例性处理器系统 1120 的框图。例如，这里所描述的方法可以被实现为存储在存储器上，并由耦合到存储器的处理器来执行的指令。如图 11 所示，处理器系统 1120 包括耦合到互连总线或网络 1124 的处理器 1122。处理器 1122 可以是任何适合的处理器、处理单元或微处理器，例如 Intel Itanium®系列、Intel X-Scale®系列、Intel Pentium®系列等当中的处理器。虽然未在图 11 中示出，但是系统 1120 可以是多处理器系统，因而可以包括与处理器 1122 相同或类似的一个或多个附加的处理器，它们被耦合到互连总线或网络 1124。

图 11 的处理器 1122 被耦合到芯片组 1128，该芯片组 1128 包括存储器控制器 1130 和输入/输出（I/O）控制器 1132。众所公知，芯片组一般提供 I/O 和存储器管理功能以及多个通用和/或专用寄存器、计时器等，它们可以由耦合到该芯片组的一个或多个处理器来访问或使用。存储器控制器 1130 完成以下功能：使处理器 1122（如果有多个处理器的话，就是多个处理器）能够访问系统存储器 1134，该存储器可以包括任何需要类型的易失性存储器，例如静态随机访问存储器（SRAM）、动态随机访问存储器（DRAM）等。I/O 控制器 1132 完成以下功能：使处理器 1122 能够经由 I/O 总线 1140 与外围输入/输出（I/O）设备 1136 和 1138 通信。I/O 设备 1136 和 1138 可以是任何需要类型的 I/O 设备，例如键盘、视频显示器或监视器、鼠标等。虽然存储器控制器 1130 和 I/O 控制器 1132 在图 11 中被图示为芯片组 1128 内的独立功能模块，但是这些模块完成的功能可以被集成在单个半导体电路内，或者可以使用两个或更多单独的集成电路来实现。

虽然这里已描述了某些方法、装置和制品，但是本专利的覆盖范围不限于此。相反，本专利覆盖正好落入所附权利要求书在字面上或者在等同原则下的范围内的所有方法、装置和制品。

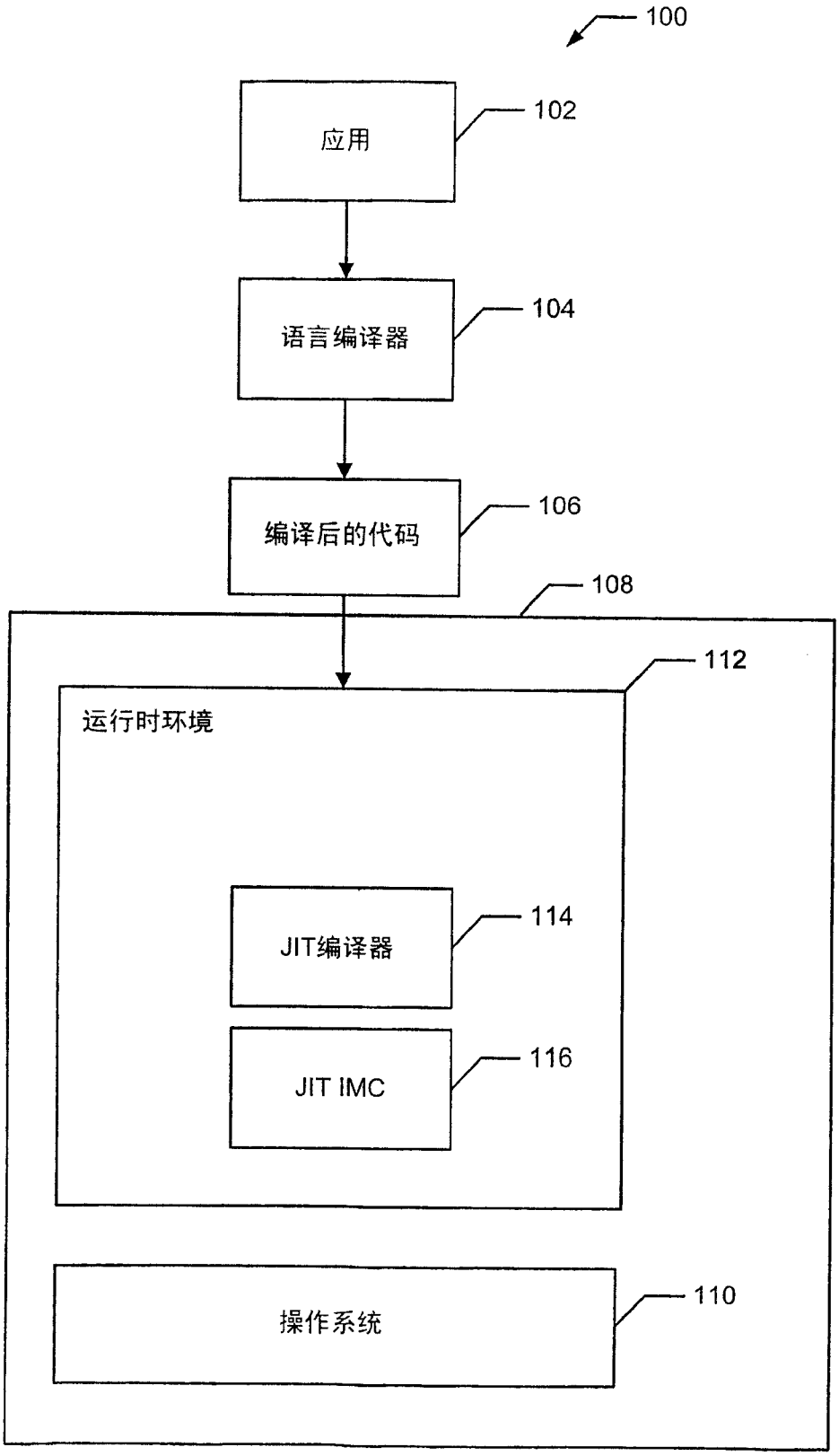

100
应用
102
语言编译器
104
编译后的代码
106
108
112
运行时环境
JIT编译器
114
JIT IMC
116
操作系统
110

图 1

```
Class Class1 {
        public static Class1 global;
        void do_nothing(Class1 obj) {}
        void synchronized foo() {
                global.do_nothing(this);
                … …
        }
}// Class1

void caller() {
        Class1 obj = new Class1();
        … …
        //调用地点A-同步被去除
        obj.foo();
}
```

图 2

```
Class Class2 extends Class1 {
        void do_nothing(Class1 obj) {
        global = obj; //escape thread
        }
} //Class2

void caller() {
        Class1 obj = new Class1();
        … …
        //调用地点A-同步去除无效
        obj.foo();
}
```

图 3

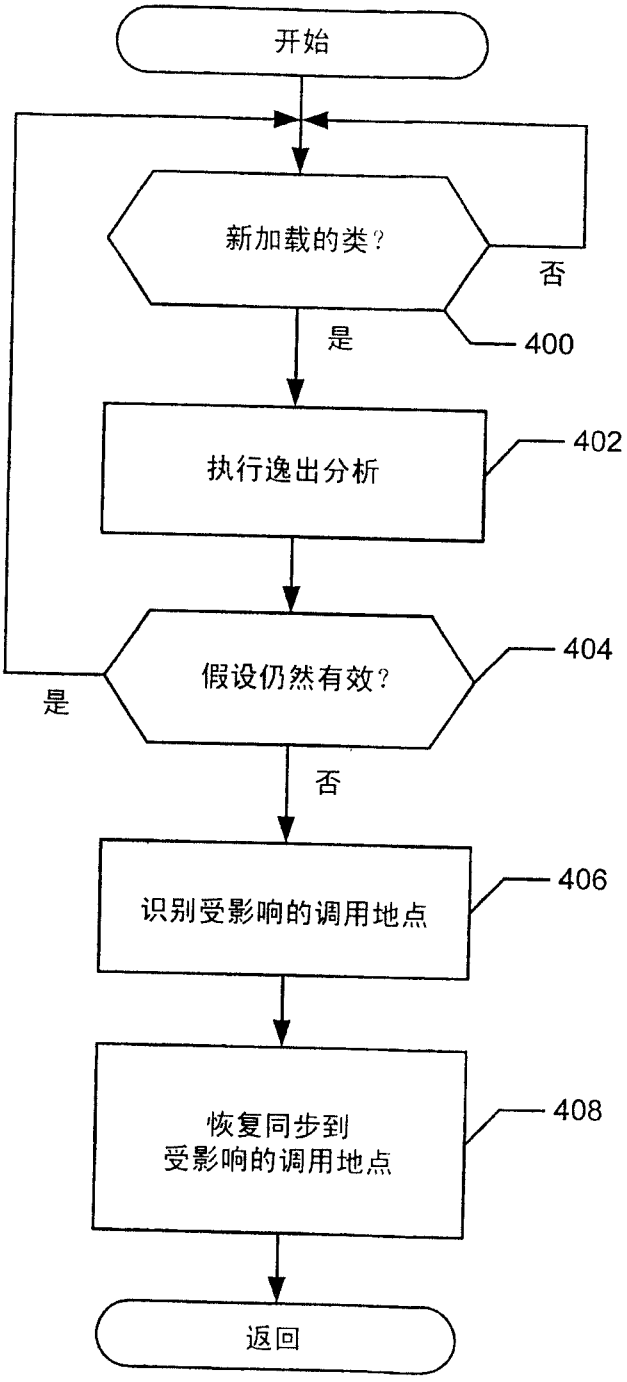

开始
新加载的类？
否
是
400
执行逸出分析
402
假设仍然有效？
404
是
否
识别受影响的调用地点
406
恢复同步到
受影响的调用地点
408
返回

图 4

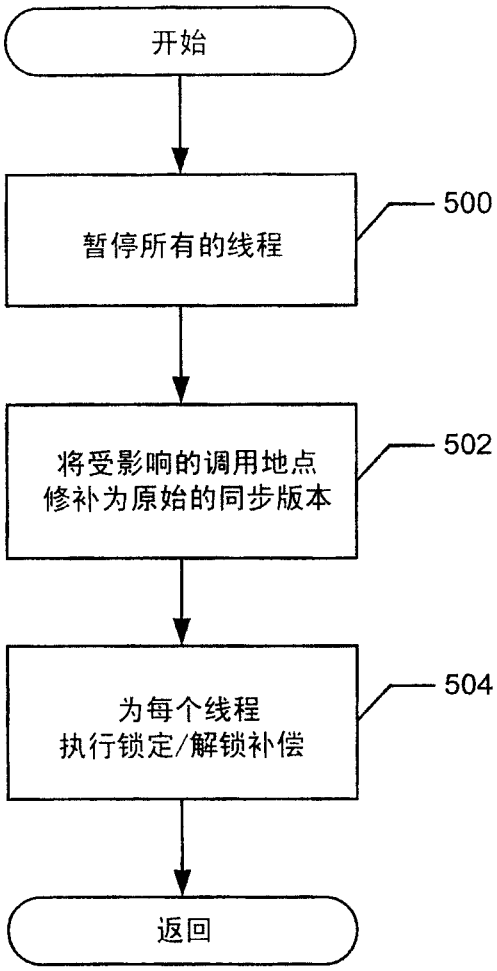

开始
暂停所有的线程
500
将受影响的调用地点
修补为原始的同步版本
502
为每个线程
执行锁定/解锁补偿
504
返回

图 5

```
//DS是逸出分析所提供的一组<caller, call_site>
re-synchronize_one_thread (thread, DS) {
        current_frame = current context of thread;
        while (current_frame is not bottom frame) {
                D= {<x,y>|<x,y> is a member of DS and current_frame.method ==x}
                for each <x.y> in D {
                        z = callee_of(<x,y>);//get the callee of call site y
                        if (last_frame.method == z)
                                compensate_lock_unlock(z);
                }
                last_frame = current_frame;
                current frame = unwind_to_next_frame();
        }
}
```

图 6

```
Static Class1 bar1() {
        return new Class1();
}

void caller1(){
        … …
        Class1 obj = bar1();
        //在逸出分析后，获知obj是Class1类型的
        obj.do_nothing();
        … …
        //调用地点B-同步去除仍然有效
        obj.foo();
}
```

图 7

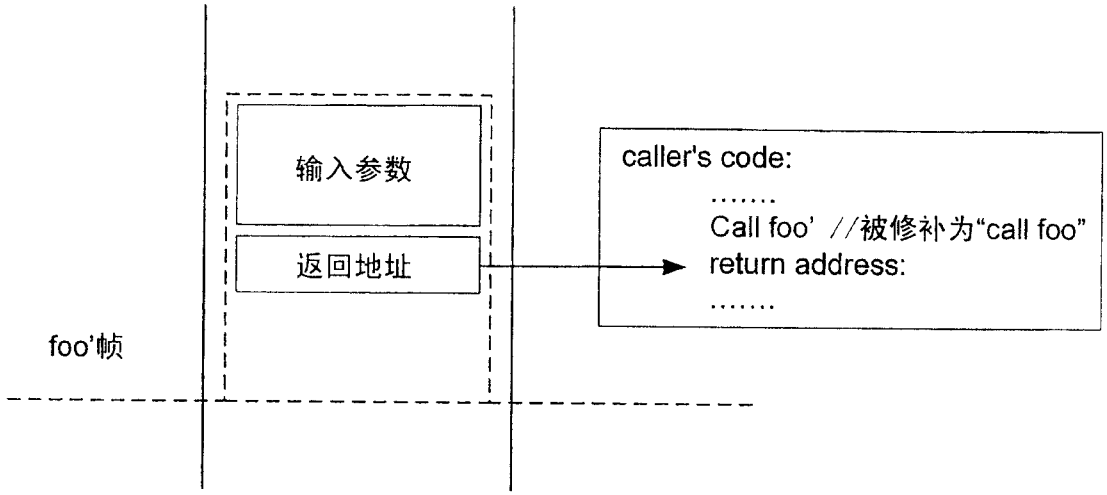

输入参数
返回地址
foo'帧
caller's code:
.....
Call foo' //被修补为"call foo"
return address:
.....

图 8

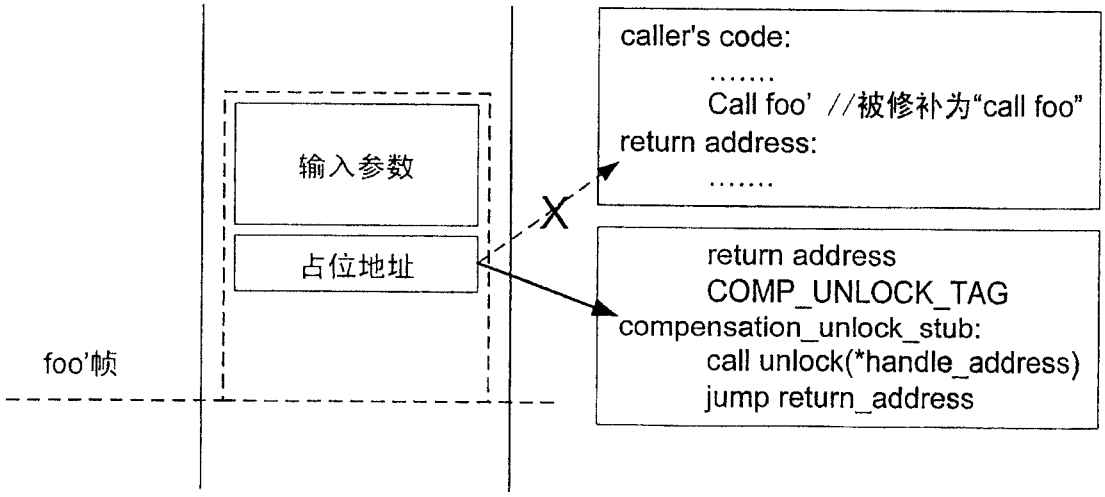

输入参数
占位地址
foo'帧
X
caller's code:
.....
Call foo' //被修补为"call foo"
return address:
.....
return address
COMP_UNLOCK_TAG
compensation_unlock_stub:
call unlock(*handle_address)
jump return_address

图 9

```
unwind_to_next_frame(current_frame) {
        return_address = current_frame.return_address;
        //return_address实际上是stub_address?
        //FAIL以下步骤，所以如果FAIL
        if (*(return_address -1) == COMP_UNLOCK_TAG {
                return_address = *(return_address -2); //实际返回地址
        }
        … …
}
```

图 10

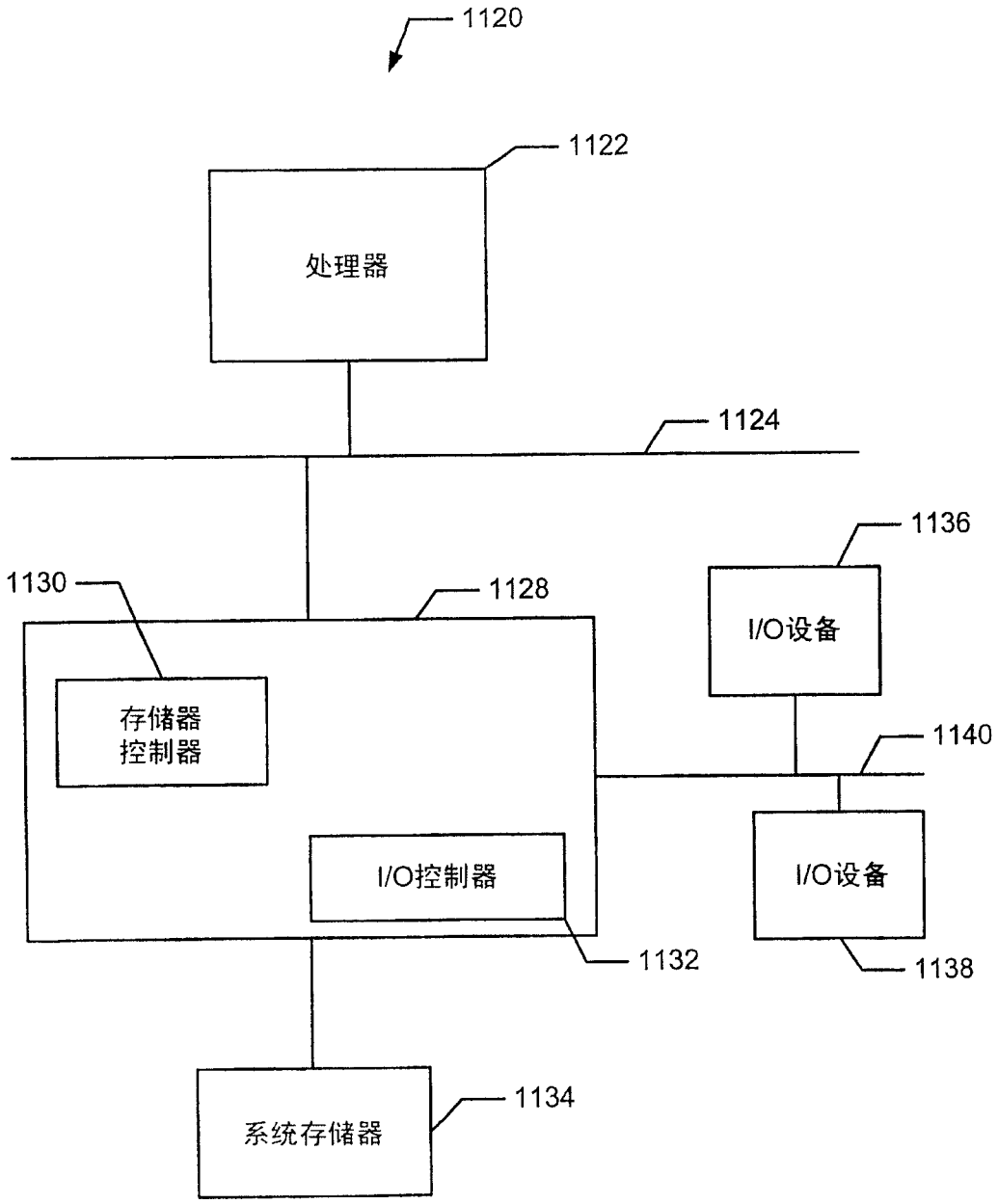

1120
1122
处理器
1124
1136
1130
1128
I/O设备
存储器
控制器
1140
I/O控制器
I/O设备
1132
1138
1134
系统存储器

图 11