[19] 中华人民共和国国家知识产权局

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

[12] 发明专利说明书

专利号 ZL 200480027177.5

[45] 授权公告日 2010 年 1 月 6 日

[11] 授权公告号 CN 100578453C

[22] 申请日 2004.9.29
[21] 申请号 200480027177.5
[30] 优先权
[32] 2003. 9.30 [33] US [31] 10/676,581
[86] 国际申请 PCT/US2004/032075 2004.9.29
[87] 国际公布 WO2005/033936 英 2005.4.14
[85] 进入国家阶段日期 2006.3.20
[73] 专利权人 英特尔公司
地址 美国加利福尼亚州
[72] 发明人 G·霍夫雷纳 廖世伟 田新民
王 宏 D·拉弗瑞 P·王
D·金 M·吉尔卡尔 J·申
[56] 参考文献
US20020083252A1 2002.6.27
US20030041228A1 2003.2.27
US6363410B1 2002.3.26
AN INEXPENSIVE INTER - PROCEDURAL REGISTERALLOCATOR. J. M. MULDER. MICROPROCESSING AND MICROPROGRAMMING, Vol. 27 No. 1/5. 1989
审查员 孟宪超
[74] 专利代理机构 上海专利商标事务所有限公司
代理人 李 玲

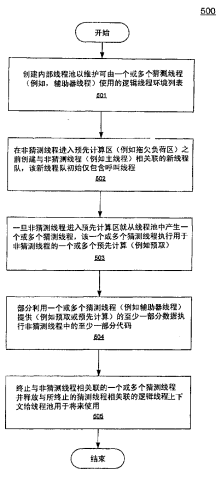
权利要求书 3 页 说明书 20 页 附图 15 页

[54] 发明名称

用于多线程的线程管理的方法和装置

[57] 摘要

这里描述了用于多线程的线程管理的方法和装置。 在一个实施例中，示例性处理包括在编译具有数据处理系统中可执行的一个或多个线程的代码期间，选择具有最底顺序的当前线程；确定分配给从所述当前线程中产生的一个或多个子线程的资源；以及考虑到分配给所述当前线程的一个或多个子线程的资源来分配用于所述当前线程的资源，以避免所述当前线程及其一个或多个子线程之间的资源冲突。 也描述了其它方法和装置。



1.一种线程管理方法，包括：

在编译具有主线程的代码期间，产生一个或多个可在数据处理系统中执行的线程，所述一个或多个线程中的每一个包括从所述主线程复制的一个或多个指令，所述一个或多个线程与线程依赖图相关联；

根据所述线程依赖图遵照从底向上的顺序从所述一个或多个线程中选择当前线程；

确定分配给从所述当前线程中产生的一个或多个子线程的资源；以及

考虑到分配给所述当前线程的一个或多个子线程的资源来分配用于所述当前线程的资源，以避免所述当前线程及其一个或多个子线程之间的资源冲突，所述方法还包括：

在为当前线程分配资源之前确定数据处理系统中是否有剩余资源；

如果根据上述确定，没有剩余的资源，则删除所述当前线程的至少一个子线程；以及

利用与所述至少一个删除的子线程相关联的资源分配用于所述当前线程的资源。

2.如权利要求 1 所述的线程管理方法，其特征在于，所述资源包括各线程使用的硬件寄存器和存储器中的至少一个。

3.如权利要求 1 所述的线程管理方法，其特征在于，分配给所述一个或多个子线程的资源被记录于当前线程可访问的数据结构中。

4.如权利要求 1 所述的线程管理方法，其特征在于，还包括更新关于分配给当前线程的资源的数据结构中的资源信息，所述数据结构可由当前线程的父线程访问。

5.如权利要求 1 所述的线程管理方法，其特征在于，还包括按所述自底向上的顺序重复所述选择、确定和分配操作，直到已处理所述一个或多个线程中的每一个。

6.如权利要求 5 所述的线程管理方法，其特征在于，在已处理所述一个或多个线程的每一个后对作为所述一个或多个线程的父线程的主线程分配资源，

根据分配给所述一个或多个线程的资源来分配所述主线程的资源。

7.如权利要求 1 所述的线程管理方法，其特征在于，还包括：

更新与分配给所述当前线程的资源相关的数据结构中的资源信息，所述数据结构可以由所述当前线程的父线程访问。

8.一种数据处理系统，包括：

用于在编译具有主线程的代码期间，产生一个或多个可在数据处理系统中执行的线程的装置，所述一个或多个线程中的每一个包括从所述主线程复制的一个或多个指令，所述一个或多个线程与线程依赖图相关联；

用于根据所述线程依赖图遵照自底向上的顺序从所述一个或多个线程中选择当前线程的装置；

用于确定分配给从所述当前线程中产生的一个或多个子线程的资源的装置；以及

用于考虑到分配给所述当前线程的一个或多个子线程的资源来分配用于所述当前线程的资源的装置，以避免所述当前线程及其一个或多个子线程之间的资源冲突，所述数据处理系统还包括：

用于在为当前线程分配资源之前确定数据处理系统中是否有剩余资源的装置；

用于如果根据上述确定，没有剩余的资源，则删除所述当前线程的至少一个子线程的装置；以及

用于利用与所述至少一个删除的子线程相关联的资源分配用于所述当前线程的资源的装置。

9.如权利要求 8 所述的数据处理系统，其特征在于，还包括用于更新关于分配给当前线程的资源的数据结构中的资源信息的装置，所述数据结构可由当前线程的父线程访问。

10.如权利要求 9 所述的数据处理系统，其特征在于，还包括用于按所述自底向上的顺序重复所述选择、确定和分配操作，直到已处理所述一个或多个线程中的每一个的装置。

11.如权利要求 10 所述的数据处理系统，其特征在于，还包括用于在已处理所述一个或多个线程的每一个后对作为所述一个或多个线程的父线程的主

线程分配资源的装置，根据分配给所述一个或多个线程的资源来分配所述主线程的资源。

12.如权利要求 8 所述的数据处理系统，其特征在于，还包括：

用于更新与分配给所述当前线程的资源相关的数据结构中的资源信息的装置，所述数据结构可以由所述当前线程的父线程访问。

13.如权利要求 8 所述的数据处理系统，其特征在于，所述资源包括各线程使用的硬件寄存器和存储器中的至少一个。

用于多线程的线程管理的方法和装置

技术领域

本发明的实施例涉及信息处理系统，尤其涉及用于多线程的线程管理。

背景技术

存储器等待时间已变成在现代处理器上实现高性能的关键瓶颈。现今的许多大型应用是存储器密集的，因为它们的存储器访问模式很难预测且它们的工作组正变得相当大。尽管高速缓存设计的持续进步以及预取技术的新发展，存储器瓶颈问题仍然存在。当执行指针密集的应用程序时该问题更糟，这些应用程序趋向于违背常规的基于跨步的预取技术。

一种解决方案是使一个程序中的存储器停顿与来自另一个程序的有用指令的执行相重叠，从而在总体吞吐量上有效地改善系统性能。改善单个处理器的多任务工作负荷的吞吐量已经是新兴的同时多线程（SMT）技术背后的主要动因。SMT处理器能在同一周期内发出来自多个硬件上下文或逻辑处理器的指令（也称作硬件线程）到超标量处理器的功能单元。SMT 通过在每个周期内经由独立线程之间的自然并行的利用来增加架构可用的总体指令级并行从而实现了更高的总吞吐量。

SMT 还可以改善多线程的应用的性能。但是，在减少等待时间方面，SMT 不直接改善单线程应用的性能。由于常规PC 环境中桌面应用的大多数仍是单线程的，所以研究 SMT 资源是否和如何可被用于通过减少其等待时间来增强单线程的代码性能是很重要的。此外，现有的编译器通常不能自动地为其创建的线程分配资源。

发明内容

本发明提供一种线程管理方法，包括：在编译具有主线程的代码期间，产生一个或多个可在数据处理系统中执行的线程，所述一个或多个线程中的每一个包括从所述主线程复制的一个或多个指令；从所述一个或多个线程中选择具有最底顺序的当前线程；确定分配给从所述当前线程中产生的一个或多个子线

程的资源；以及考虑到分配给所述当前线程的一个或多个子线程的资源来分配用于所述当前线程的资源，以避免所述当前线程及其一个或多个子线程之间的资源冲突。

本发明还提供了一种数据处理系统，包括：处理器，它能执行多线程操作；以及存储器，它与所述处理器耦合，其中处理器执行来自所述存储器的进程，以：在编译具有主线程的代码期间，产生一个或多个可在数据处理系统中执行的线程，所述一个或多个线程中的每一个包括从所述主线程复制的一个或多个指令；从所述一个或多个线程中选择具有最底顺序的当前线程；确定分配给从所述当前线程中产生的一个或多个子线程的资源；以及考虑到分配给所述当前线程的一个或多个子线程的资源来分配用于所述当前线程的资源，以避免所述当前线程及其一个或多个子线程之间的资源冲突。

附图说明

通过参考被用于说明本发明实施例的以下描述和附图最佳地理解本发明。图中：

图 1 示出了根据一个实施例的具有多线程能力的计算机系统。

图 2 示出了根据可选实施例的具有多线程能力的计算机系统。

图 3 示出了根据一个实施例的具有能产生辅助线程的编译器的计算机系统。

图 4A 示出了典型的对称多线程处理。

图 4B 示出了根据一个实施例的非对称多线程处理。

图 5 是示出根据一个实施例的用于执行一个或多个辅助线程的示例性处理的流程图。

图 6 是示出根据一个实施例的多线程系统的示例性软件架构的框图。

图 7 是示出根据一个实施例的用于产生辅助线程的示例性处理的流程图。

图 8 是示出根据一个实施例的用于并行化分析的示例性处理的流程图。

图 9A－9C 示出了根据一个实施例的用于应用程序、主线程和辅助线程的伪代码。

图 10 是示出根据一个实施例的示例性线程配置的框图。

图 11 是示出根据一个实施例的用于分配用于线程的资源的示例性伪代码的框

图。

图 12 是示出根据一个实施例的包含用于线程的资源信息的示例性资源数据结构的框图。

图 13 是示出根据一个实施例的分配用于线程的资源的示例性处理的流程图。

图 14A－14D 示出了使用技术实施例的多种基准测试的结果。

具体实施方式

将描述用于多线程系统的编译器创建的辅助线程的方法和装置。根据一个实施例，也称作 AutoHelper（自动辅助）的编译器在例如 Intel 公司的 Intel Pentium™ 4 超线程系统的多线程系统上执行基于线程的预取辅助线程。在一个实施例中，编译器使用于超线程处理器的辅助线程的生成自动化。这些技术关注识别和生成最小尺寸的辅助线程，它们可以被执行以实现及时和有效的数据预取，同时引发最小的通信开销。还执行运行时间系统以有效地管理辅助线程以及线程之间的同步。结果，辅助线程能发出用于顺序指针密集应用的及时预取。

此外，可以在编译器内为辅助线程管理诸如寄存器上下文的硬件资源。特别是，可以静态地或动态地在主线程和辅助线程之间以及多个辅助线程之间分割寄存器组。结果，当编译器用尽资源时或者在出现特定主线程事件的稀少情况的运行时间时，可以避免经由线程存储器的内驻/外驻寄存器拷贝并且可以在编译时间销毁所述线程。

在以下描述中，阐述了大量细节。但是，可以理解，本发明的实施例可在没有这些细节的情况下实施。在其它实例中，未详细示出公知电路、结构和技术，以便不影响对描述的理解。

以下详细描述的一些部分按照算法和对计算机存储器内数据比特的操作的符号表示来提供。数据处理领域内的熟练技术人员将这些算法描述和表示用于最有效地将其工作的实质传达给本领域的其它熟练技术人员。这里，算法被认为是获得期望结果的自一致操作序列。这些操作需要物理量的物理操作。通常，尽管不必要，但这些量采取能够存储、传输、组合、比较和其它处理的电子或磁性信号的形式。已证明，特别是出于普通使用的原因，将这些信号称作比特、值、元素、符号、字符、项、数量等有时是方便的。

但要记住，所有这些和类似的术语与合适的物理量相关联且仅仅是应用于这些量的方便的标记。除非另外特别声明或从以下讨论中显见，可以理解，使用诸如“处理”或“计算”或“计算”或“确定”或“显示”等的术语贯穿于描述中表示计算机系统或类似数据处理装置的动作和过程，它们将表示为计算机系统的寄存器和存储器内的物理（例如，电子）量的数据处理和转换成类似地被表示为计算机系统存储器或寄存器或其它这种信息存储、传输或显示装置内的物理量的其它数据。

本发明的实施例还涉及用于执行这里所述的操作的装置。一个装置可专门为所需的用途而构建，或者它可以包括由计算机中存储的计算机程序选择性地激活或再配置的通用计算机。这种计算机程序可存入计算机可读存储媒体，诸如但不限于任何类型的盘，包括软盘、光盘、CD-ROM 和磁光盘，只读存储器（ROM），随机存取存储器（RAM），诸如动态 RAM（DRAM）、可擦可编程 ROM（EPROM）、电可擦可编程 ROM（EEPROM）、磁或光卡，或者适于存储电子指令的任何类型的媒体，其中以上存储组件的每一种都耦合于计算机系统总线。

这里提供的算法和显示非固有地关于任何特殊计算机或其它装置。各种通用装置可与根据这里的教导的程序一起使用，或者可证明构建专门的装置来执行这些方法是方便的。用于这些系统的结构可从以下描述中显现。此外，本发明的实施例不参考任何特殊的编程语言进行描述。可以理解，各种编程语言都可用于实现这里描述的本发明实施例的教导内容。

机器可读媒体包括用于以机器（例如计算机）可读的形式存储和发送信息的任何机制。例如，机器可读媒体包括只读存储器（ROM）；随机存取存储器（RAM）；磁盘存储媒体；光学存储媒体；闪存装置；电气、光学、声学或其它形式的传播信号（例如，载波、红外线信号、数字信号等）；等等。

图 1 是可与实施例一起使用的示例性计算机的框图。例如，图 1 所示的示例性系统 100 可执行图 5－8 所示的处理。示例性系统 100 可以是多线程系统，诸如 Intel Pentium™ 4 超线程系统。示例性系统 100 可以是支持同时多线程（SMT）或芯片多处理（CMP）的系统。

注意：虽然图 1 示出了计算机系统的各种组件，但它不打算表示互连这些组件的任何特殊架构或方式，因为这些细节并非与本发明是密切联系的。还可以理解，网络计算机、手持计算机、蜂窝电话和具有更少或更多组件的其它数据处理系统也

可与本发明一起使用。

如图 1 所示，作为一种数据处理系统形式的计算机系统 100 包括耦合到微处理器 103 和 ROM 107、易失性 RAM 105 和非易失性存储器 106 的总线 102。微处理器 103（可以是 Intel 公司的 Pentium 处理器或者 Motorola 公司的 PowerPC 处理器）与高速缓存存储器 104 耦合，如图 1 的示例中所示的。总线 102 将这些组件互连在一起并且还将这些组件 103、107、105 和 106 耦合到显示控制器和显示器装置 108，以及输入/输出（I/O）装置 110，它们可以是鼠标、键盘、调制解调器、网络接口、打印机和本领域公知的其它装置。通常，输入/输出装置 110 通过输入/输出控制器 109 耦合到系统。易失性 RAM 105 通常作为持续地需要电能以刷新或维持存储器中的数据的动态 RAM（DRAM）实现。非易失性存储器 106 通常是磁性硬盘驱动器、磁光驱动器、光学驱动器或 DVD RAM 或者即使在系统断电后也能维持数据的其它类型的存储器系统。通常，非易失性存储器还将是随机存取存储器，尽管这是不需要的。虽然图 1 示出非易失性存储器是与数据处理系统中的剩余组件直接耦合的本地装置，但可以理解，本发明可以使用远程于系统的非易失性存储器，诸如通过诸如调制解调器或以太网接口的网络接口与数据处理系统耦合的网络存储装置。总线 102 可以包括通过各种桥、控制器和/或适配器相互连接的一条或多条总线，如本领域公知的。在一个实施例中，I/O 控制器 109 包括用于控制 USB 外围设备的 USB（通用串行总线）适配器或者用于控制 PCI 装置的 PCI 控制器，它们可以包含在 IO 装置 110 中。在另一实施例中，I/O 控制器 109 包括用于控制 IEEE 1394 装置（也称作火线装置）的 IEEE 1394 控制器。

根据一个实施例，处理器 103 可以包括一个或多个逻辑硬件上下文，也称作逻辑处理器，用于同时处理应用程序的多个线程，包括也称作非猜测线程的主线程以及也称作猜测线程的一个或多个辅助线程。处理器 103 可以是超线程处理器，诸如 Intel 公司的能执行多线程处理的 Pentium 4 或 Xeon 处理器。在应用程序的执行期间，并行地执行主线程和一个或多个辅助线程。辅助线程与主线程相关联地（但稍许独立）猜测执行，以执行主线程的一些预先计算，诸如地址或数据的猜测预取，从而减少由主线程引起的存储器等待时间。

根据一个实施例，辅助线程的代码（例如源代码和二进制可执行代码）由诸如 Intel 公司的 AutoHelper 编译器的编译器生成，由诸如处理器 103 的处理器执行

的操作系统（OS）在诸如易失性 RAM 105 的存储器中加载并执行。示例性系统 100 内运行的操作系统可以是 Microsoft 公司的 Windows 操作系统或者是 Apple Computer 公司的 Mac OS。或者，该操作系统可以是 Linux 或 Unix 操作系统。可以使用诸如嵌入式实时操作系统的其它操作系统。

当前的超线程处理器通常提供两个硬件上下文或逻辑处理器。为了改善单线程应用的性能，超线程技术可利用其第二上下文执行主线程的预取。具有分开的上下文允许辅助线程的执行与主线程的控制流断开，这与软件预取不同。通过远在主线程之前执行长程预取，辅助线程可较早地触发预取，并消除或减少主线程所经受的高速缓存不命中惩罚。

采用 AutoHelper，编译器能自动生成用于超线程机器的预取辅助线程。辅助线程目标在于向顺序工作负荷提供超线程的等待时间隐匿的好处。与常规并行编译器所产生的线程不同，辅助线程仅为主线程做预取，而主线程不重用来自辅助线程的计算结果。根据一个实施例，程序正确性仍由主线程的执行来维护，而辅助线程不影响程序正确性且仅用于性能改善。该属性允许在生成辅助线程时使用更进取形式的优化。例如，当主线程不需要帮助时，可以执行某些优化，这是常规吞吐量线程范例所不可能的。

在一个实施例中，如果预测在某一时间周期内不需要辅助器，该辅助器可以终止并释放与该辅助器相关的所有资源给主线程。根据另一实施例，如果预测立刻需要一辅助器，则该辅助器可处于暂停模式，这仍消耗超线程硬件上的一些资源。如果辅助器停留在暂停模式太久（例如，超过了可编程超时周期），则将调用指数补偿（通过暂停）。根据另一实施例，如果编译器不能预测何时将需要辅助线程，则该辅助器可处于打盹模式并可以放弃所占用的处理器资源给主线程。

此外，根据一个实施例，在辅助线程范例下使得性能监控和即时调节成为可能，因为辅助线程不贡献主程序的语义。当主线程需要辅助器时，它将唤醒主线程。例如，相对于失控辅助或落后（run-behind）线程，可调用上述处理之一来调节该失控辅助线程。

图 2 是能执行所揭示的技术的计算系统 200 的一个实施例的框图。在一个实施例中，计算系统 200 包括处理器 204 和存储器 202。存储器 202 可存储用于控制处理器 204 的操作的指令 210 和数据 212。处理器 204 可包括向执行核心 230 提供

指令信息的前端 221。该前端 221 可按程序顺序向处理器核心 204 提供指令信息。

对于至少一个实施例，前端 221 包括读取/解码单元 222，它包括用于多个线程上下文中的每一个的逻辑独立的序列器 220。逻辑独立的序列器 220 可包括标记逻辑 280 以便将用于猜测线程的指令信息标记为“猜测的”。本领域的熟练技术人员将认识到，对于在多处理器多线程环境中执行的实施例，读取/解码单元 222 内可以仅包括一个序列器 220。

如这里所使用的，术语“指令信息”意在表示执行核心 230 能理解和执行的指令。指令信息可存入高速缓存 225。该高速缓存 225 可作为执行指令高速缓存或执行跟踪高速缓存而实现。对于使用执行指令高速缓存的实施例，“指令信息”包括从高速缓存读取并被解码的指令。对于使用跟踪高速缓存的实施例，术语“指令信息”包括解码的微操作的跟踪。对于不使用执行指令高速缓存也不使用跟踪高速缓存的实施例，“指令信息”还包括用于可存入诸如 I 高速缓存 244 的指令高速缓存的指令的原始字节。

图 3 是示出根据一个实施例的包含用于生成一个或多个辅助线程的编译器的示例性系统的框图。参考图 3，示例性处理系统 300 包括存储器系统 302 和处理器 304。存储器系统 302 可存储用于控制处理器 304 的操作的指令 310 和数据 312。例如，指令 310 可包括编译器程序 308，它在被执行时使得处理器 304 编译存储器系统 302 内驻留的程序。存储器 302 保存要编译的程序、程序的中间形式以及所获得的编译程序。对于至少一个实施例，编译器程序 308 包括指令来生成用于相对于主线程的一个或多个辅助线程的代码。

存储器系统 302 意在作为存储器的一般化表示并可以包括各种形式的存储器，诸如硬盘驱动器、CD-ROM、随机存取存储器（RAM）、动态随机存取存储器（DRAM）、静态随机存取存储器（SRAM）和有关电路。存储器系统 302 可以存储指令 310 和/或由处理器 304 执行的数据信号所表示的数据。指令 310 和/或数据 312 可包括用于执行这里所讨论的任何或所有技术的代码。

特别是，编译器 308 可包括拖欠（delinquent）负荷标识符 320，它在由处理器 304 执行时标识主线程的一个或多个拖欠负荷区域。编译器 308 也可包括并行化分析器 324，它在由处理器 304 执行时对辅助线程进行一个或多个并行化分析。此外，编译器 308 可包括分片器（slicer）322，它标识要由辅助线程执行的一个或多

片以执行猜测预先计算。编译器 308 可进一步包括代码发生器 328，它在由处理器 304 执行时生成用于辅助线程的代码（例如，源代码和可执行代码）。

根据一个实施例，如图 4B 所示，SMT 机器中的执行辅助线程是非对称多线程的一种形式。常规的并行编程模型提供对称的多线程，如图 4A 所示。相反，诸如图 4B 中的辅助线程 451－454 的辅助线程作为具有轻量线程调用和切换的用户级线程（纤维（fiber））执行。此外，对称多线程需要对称线程（诸如图 4A 中的线程 401-404）上良好地调节的数据分解。在辅助线程模型中，根据一个实施例，主线程运行对整个数据组操作的顺序代码，而不带来数据分解开销。在不分解数据的情况下，编译器代替地集中于提供多个辅助器用于及时地预取主线程的数据。

图 5 是示出根据一个实施例的用于执行辅助线程的示例性处理的流程图。示例性处理 500 可通过处理逻辑执行，该处理逻辑可包括硬件（电路、专用逻辑等）、软件（诸如在通用计算机系统或专用机器上运行的）或者它们的组合。在一个实施例中，示例性处理 500 包括执行多线程系统中的应用程序的主线程，并在主线程进入具有一个或多个拖欠负荷的区域时从主线程中产生一个或多个辅助线程以执行用于该主线程的一项或多项计算，其中在主线程的编译期间创建一个或多个辅助线程的代码。

参考图 5，在框 501 处，处理逻辑创建内部线程池以维护一个或多个辅助线程所使用的逻辑线程上下文的列表。在框 502 处，可以在主线程进入由编译器标识的拖欠负荷区域（例如，预先计算区域）之前创建新线程队。在一个实施例中，新线程队初始仅包含呼叫线程。根据一个实施例，在主线程进入所述区域以激活一个或多个辅助线程之前，编译器可插入一语句，诸如 start_helper 语句。在框 503 处，当主线程进入所述区域时，主线程（经由诸如 invoke_helper 的函数调用）产生一个或多个辅助线程，它们利用来自线程池的资源被创建以执行主线程的一项或多项预先计算，诸如预取地址和数据。根据一个实施例，如果用于执行所产生的辅助线程的逻辑处理器不可用，则可用创建辅助线程并将其置入用于线程队的运行队列，用于后续执行。在一个实施例中，运行队列可与超时相关联。假定预取将不再是及时的，在超时周期到期后，简单地结束（终止）对调用辅助器的请求。这与用于其中需要执行每个任务的并行编程的常规任务队列模型不同。

在框 504 处，部分利用一个或多个辅助线程提供的数据（例如，预取的或预

先计算的）执行主线程的区域内的至少一部分代码。根据一个实施例，辅助线程计算的结果不被集成入主线程。辅助线程的好处在于其预取的副作用，而不在于重用其计算结果。这允许编译器进取地优化辅助线程的代码生成。主线程处理正确性问题，同时辅助线程把程序性能作为目标。这还允许辅助线程调用语句（诸如invoke_helper）结束请求，只要认为合适。最后，如果在辅助线程中用信号表示了异常，诸如预取指令的非故障指令可用于避免对主线程的中断。

在框 505 处，当主线程将要退出拖欠负荷区域时终止与主线程相关联的一个或多个辅助线程（通过函数调用，诸如 finish_helper），且与终止的辅助线程相关联的诸如逻辑线程上下文的资源被释放回给线程池。这使得将来的请求能立刻再循环来自线程池的逻辑线程上下文。可用包括本领域的普通技术人员显而易见的其它操作。

超线程技术良好地适于支持一个或多个辅助线程的执行。根据一个实施例，在每个处理器周期中，在共享的执行资源上可以调度并执行来自任一逻辑处理器的指令。这允许辅助线程及时地发出预取。此外，整个片载高速缓存层次结构在逻辑处理器之间共享，这有助于辅助线程有效地在高速缓存层次结构的所有等级处为主线程做预取。此外，尽管在逻辑处理器之间共享物理执行资源，在超线程处理器中复制该架构状态。辅助线程的执行将不改变执行主线程的逻辑处理器中的架构状态。

但是，在支持超线程技术的机器上，由于对存储器的写入，辅助线程仍会影响主线程的执行。因为辅助线程与主线程共享存储器，辅助线程的执行应保证不对主线程的数据结构进行写入。在一个实施例中，编译器（例如，AutoHelper）提供主线程和辅助线程之间的存储器保护。编译器去除对辅助线程中非本地变量的存储。

图 6 是示出根据一个实施例的编译器的示例性架构的框图。在一个实施例中，示例性架构 600 尤其包括前端模块 601、剖析器（profiler）602、过程间分析和优化模块 603、编译器 604、全局标量优化模块 605 以及后端模块 606。在一个实施例中，前端模块 601 提供用于诸如 C/C++和 Fortran 的各种编程语言编写的源代码的普通中间表示，诸如来自 Intel 公司的 IL0 表示。结果，可应用诸如 AutoHelper 604 的编译器，而不管源语言和目标平台。剖析器 602 进行剖析运行以检查该表示的特

征。过程间分析模块 603 可暴露程序呼叫边界上的优化机会。此后，调用编译器 604（例如，AutoHelper）以生成用于一个或多个辅助线程的代码。全局标量优化模块 605 利用部分冗余消除进行应用以最小化对表达式求值的次数。最后，后端模块 606 生成用于各种平台的辅助线程的二进制代码，诸如 Intel 公司的 IA－32 或 Itanium 平台。可以包括本领域普通技术人员显而易见的其它组件。

与常规方法不同，AutoHelper（例如，编译器）消除了剖析实行遍（pass）以使该工具更易于使用。根据一个实施例，编译器可直接分析来自剖析结果的输出，诸如由 Intel 的 VTune™ Performance Analyzer 所生成的那些，这能用于超线程技术。因为它是代替遍后工具的中间端遍，该编译器能使用几种产品质量分析，诸如数组依赖性分析和全局标量优化等。在编译器之后调用的这些分析对辅助线程的代码进行进取的优化。

根据一个实施例，编译器生成一个或多个辅助线程以预先计算和预取频繁不命中高速缓存的负荷（也称作拖欠负荷）所访问的地址。编译器还在主线程中生成产生一个或多个辅助线程的一个或多个触发器。编译器将该触发器作为调用函数实现，诸如 invoke_helper 函数呼叫。一旦达到该触发器，就期望负荷稍后出现于主线程的指令流中，因此猜测执行的辅助线程可减少主线程中高速缓存不命中的数量。

图 7 是示出根据一个实施例的由诸如 AutoHelper 的编译器执行的示例性处理的流程图。示例性处理 700 可由处理逻辑执行，该处理逻辑可包括硬件（电路、专用逻辑等）、软件（诸如通用计算机系统或专用机器上运行的）或者它们的组合。在一个实施例中，示例性处理 700 始于框 701，用以例如用 Intel 公司的 VTune 工具标识拖欠负荷，以执行用于辅助线程的并行化分析（框 702），以生成用于辅助线程的代码（框 703），并且向每个辅助线程和主线程分配诸如硬件寄存器或存储器的资源（框 704），这将在以下详细描述。

根据一个实施例，编译器利用一个或多个运行时间剖析标识应用程序源代码中最拖欠负荷。常规编译器按两个步骤收集这些剖析：剖析实行和剖析生成。但是，因为高速缓存不命中不是向编译器公开的架构特征，剖析实行遍不允许对编译器的高速缓存不命中的实行以标识拖欠负荷。用于每个高速缓存层次结构的剖析经由诸如 Intel 公司的 VTune™ Analyzer 的工具被收集。在一个实施例中，可以用在编译

器之前分开的剖析运行中的调试信息来执行应用程序。在剖析运行期间，采样高速缓存不命中且为应用程序中的每个静态负荷累积硬件计数器。

编译器标识用于基于线程的预取的候选。在特殊实施例中，VTune™ 以每个负荷为基概括高速缓存行为。因为用调试信息（例如，调试符号）编译用于剖析运行的二进制代码，可能使剖析源行号和语句相关。贡献大于一预定阈值的某些负荷可标识为拖欠负荷。在一特殊实施例中，贡献 90％高速缓存不命中的最大负荷被标注为拖欠负荷。

除了标识拖欠负荷指令之外，编译器生成精确计算拖欠负荷地址的辅助线程。在一个实施例中，生成用于辅助线程的分开代码。主线程和辅助线程的代码之间的分开阻止辅助线程的代码转换影响到主线程。在一个实施例中，编译器使用多入口线程，代替 Intel 产品的编译器中的常规出线，以生成用于辅助线程的分开代码。

此外，根据一个实施例，编译器以编译器选择的代码区域（被标注为预先计算区域）的粒度执行多入口线程。该区域包含一组拖欠负荷并定义用于猜测预先计算的范围。在一个实施例中，实现通常以循环区域为目标，因为循环通常是程序执行中的热点，且拖欠负荷通常是循环中多次执行的负荷。

图 8 是示出根据一个实施例的用于并行化分析的示例性处理的流程图。示例性处理 800 可由处理逻辑执行，该处理逻辑可包括硬件（电路、专用逻辑等）、软件（诸如通用计算机系统或专用机器上运行的）或者它们的组合。参考图 8，在框 801 处，处理逻辑构建依赖性图表，它获取主线程的数据和控制依赖性。根据一个实施例，为了滤出不相关的代码从而减少辅助线程的代码的大小，编译器首先构建获取数据和控制依赖性的的图表。过滤的有效性和合法性依赖于编译器精确地消除存储器引用歧义的能力。结果，编译器中的存储器消除歧义模块被调用以便对动态分配对象的指针消除歧义。因为指针可以是全局变量或函数参数，所以如果编译器按全程序模式进行编译，编译器执行的指向分析是过程间的。在一个实施例中，如果所有数组访问是有限表达，为了更精确地构建依赖性图表，可执行一系列数组依赖性测试，以使在构建依赖性图表时对数组中的每个元素消除歧义。否则，使用近似。此外，可以对结构中的每个域消除歧义。

参考图 8，在框 802 处，处理逻辑利用依赖性图表对主线程执行分片操作。在分片期间，根据一个实施例，编译器首先将拖欠负荷的负荷地址标识为片标准，它

指明中间分片结果。在构建了依赖性图表后，编译器计算所标识的片标准的程序片。片标准的程序片被定义为一组指令，它们有助于用于一个或多个辅助线程执行的存储器预取的地址的计算。分片可将代码减少到仅与地址计算相关的指令，从而允许辅助线程更快地运行并在主线程前面。编译器只需要将片中的指令复制到辅助线程的代码。

根据一个实施例，通过向后过渡地遍历依赖性边缘，编译器中的分片提取指令的最小序列以产生拖欠负荷的地址。可以将结果片的依赖性图表上的叶节点转换成预取指令，因为没有进一步的指令依赖于这些叶节点。由诸如 Intel 公司的 Pentium™ 4 的处理器执行的那些预取指令都是非阻塞和无故障的。不同的预取指令用于将数据带到存储器层次结构中不同等级的高速缓存。

根据一个实施例，可相对于给定的代码区域执行分片操作。当达到该区域外的代码时，给定区域中的依赖性图表上的遍历必须终止。因此，必须在遍历期间而非遍历之后终止分片，因为图表遍历可跨到区域之外且随后回到区域之内。在遍历后根据区域简单地收集片会损失精度。

在另一实施例中，编译器将每个拖欠负荷指令逐个分片。为了最小化辅助线程中的代码重复并减少线程调用和同步的开销，如果多个片在同一预先计算区域中，则编译器将它们合并成一个辅助线程。

参考图 8，在框 803 处，处理逻辑在线程上执行调度以重叠多个预取。在一个实施例中，由于超线程处理器支持具有大调度窗口的无序执行，当当前执行指令等待未决的高速缓存不命中时，处理器可搜索当前执行指令以外的独立指令。无序执行的这个方面可以提供优于有序处理器的实质性性能增益并减少链化猜测预先计算的需要。此外，编译器选择用于超线程处理器的基本猜测预先计算。即，一次仅调度一个辅助线程，以节省线程产生和通信开销。使用基本猜测预先计算的另一好处在于它不像链化猜测预先计算那样快速地充满超线程处理器上的存储器系统。当无序处理器搜索用于执行的独立指令时，这些指令会生成太多负荷请求并使存储器系统饱和。当辅助线程发出预取请求时，大量的未完成的不命中会快速地填满不命中缓冲器，结果使处理器停顿。因此，编译器在产生辅助线程方面需要有判断力。最后，为了确保及时进行预取，编译器将单个辅助线程和主线程固定在各自的逻辑处理器上。

参考图 8，在框 804 处，处理逻辑为线程选择通信方案。在一个实施例中，编译器提供一个为任何给定的片或者任何程序子集计算活跃性（liveness）信息的模块。活跃性信息提供了对通信成本的估计。该信息用于选择提供通信和计算之间的良好权衡的预先计算区域。该活跃性信息可帮助找到触发器或向后分片结束的点。

因为典型的超线程处理器在每个处理器周期发出三个微操作码并使用相同的硬分区资源，该编译器必须是有判断力的以便不使辅助线程减慢主线程的执行，特别是如果主线程已为每个周期的执行发出三个微操作码。对于包含拖欠负荷的循环嵌套，在选择用于进行猜测预先计算的循环级时，编译器进行再计算和通信之间的权衡。对于每个循环级，根据一个实施例，从最里面的一个开始，编译器选择基于通信的方案和基于计算的方案之一。

根据一个实施例，基于通信的方案在每次迭代时将内驻（live-in）值从主线程传送给辅助线程，使得辅助线程不需要再计算该内驻值。如果存在包含最拖欠负荷的内循环且如果用于该内循环的分片明显地减少了辅助线程的大小，则编译器将选择该方案。但是，如果用于内循环级的通信成本很大，则禁用该方案。如果较早计算出该内驻值且内驻数量较小，则编译器将给出通信成本的较小估计。

基于通信的方案将在运行时间创建主线程及其辅助线程之间的多个通信点。基于通信的方案对于超线程处理器很重要，因为通过再计算辅助线程中的片而仅依赖一个通信点会形成线程之间太多的资源争用。该方案与构建穿越（do-across）循环的类似之处在于在完成了计算用于该迭代的内驻值后主线程启动下一个迭代。该方案用通信交换较少的计算。

根据一个实施例，基于计算的方案假定起初两个线程之间的仅一个通信点传递入该内驻值。然后，辅助线程需要计算它需要的所有东西以生成精确的预取地址。如果没有内循环，或者如果用于该循环级的分片不明显增加辅助线程的大小，该编译器将选择该方案。一旦达到了单个通信点，基于计算的方案向辅助线程提供执行中的更多独立性。

根据一个实施例，为了选择用于猜测预先计算的循环级，编译器选择受益于基于通信的方案的最外面的循环。因此，一旦发现具有基于通信的方案的循环，上述方案选择算法可以终止。如果编译器没有找到任何具有基于通信的方案的循环，则最外面的循环将是用于猜测预先计算的目标区域。在编译器选择了预先计算区域

和它们的通信方案后，在主线程内定位良好的触发点将确保及时地进行预取，同时最小化主线程和辅助线程之间的通信。活跃性信息帮助定位触发器，其中触发器是向后分片结束的点。当内驻数量增加时，预先计算区域以外的分片结束。

参考图 8，在框 805 处，处理逻辑确定用于线程的同步周期以便在执行期间相互同步。根据一个实施例，同步周期用于表达辅助线程和主线程之间的距离。通常，辅助线程在同步周期的单元中执行所有其预先计算。这使得通信最小化并且限制了产生失控辅助器的可能性。因为编译器计算同步周期值并相应地生成同步代码，不再需要诸如 Outstanding Slice Counter 的特殊硬件支持。

如果同步周期太大，辅助线程引起的预取不仅会临时移动主线程要使用的重要数据还会潜在地移动未由主线程使用的较早的预取数据。另一方面，如果同步周期太小，预取可能太晚以至于不太有用。为了确定同步周期值，根据一个实施例，编译器首先计算片长度和主线程中的程序调度的长度之间的差。如果该差较小，则一个迭代中辅助线程引起的超过距离较小。辅助线程会需要多个迭代来保持足够的超过距离。因此，如果该差较小，则编译器增加同步周期，反之亦然。

此后，在代码生成阶段期间，编译器生成用于主线程和辅助线程的代码。在该代码生成阶段期间，编译器构建线程图表，作为分析阶段和代码生成阶段之间的接口。每个图表节点都标注一指令序列，或者代码区域。节点之间的调用边缘标注线程产生关系，这对于指定链化辅助线程是很重要的。具有线程图表支持代码重用，因为根据一个实施例编译器还允许用户将编译指示插入源程序以指定用于辅助线程和内驻的代码。基于编译指示的方法和自动方法两者共享同一图表抽象。结果，可以共享辅助线程代码生成模块。

辅助线程代码生成充分利用编译器中的多入口线程技术以生成辅助线程代码。与常规的公知概要相反，编译器不创建用于辅助线程的分开的编译单元（或例程）。相反，编译器生成辅助线程代码的线程入口和线程返回。编译器保持所有新生成的辅助线程代码完整或者内嵌在同一用户定义的例程中而不将它们分成独立的子程序。该方法提供更晚的编译器优化，其中有更多的机会用于对新生成的辅助线程进行最优化。辅助线程中指令更少意味着超线程处理器上的资源争用更少。这显示用辅助线程隐匿等待时间与常规对称多线程模型相比引起更少的指令和更少的资源争用，这是特别重要的因为超线程处理器在每个处理器周期中发出三个微操

作码并具有一些硬分区资源。

根据一个实施例，用于辅助线程的所生成的代码将通过编译器中的稍后阶段被重新排序和优化，诸如部分死存储消除（PDSE），部分冗余消除（PRE）和其它标量优化。在这种情况下，需要优化辅助线程代码以最小化由于辅助线程引起的资源争用。但，这些进一步的优化也会去除预取代码。因此，可以将叶拖欠负荷转换成编译器中的易变量赋值语句。片的依赖性图表中的叶节点暗示辅助线程中没有进一步的指令依赖于加载值。因此，易变量赋值语句的目的地变成寄存器临时，表示加速所获得的代码。使用易变量赋值可防止所有更晚的编译器全局优化去除用于拖欠负荷的所生成的预取。

根据一个实施例，编译器目的在于利用自计数机制确保辅助线程既不运行太超前也不落后主线程。根据一个实施例，为超前距离控制预设值 X。该 X 可以由用户通过编译器开关或者基于片（或辅助器代码）的长度和主代码的长度的程序分析进行修改。在一个实施例中，编译器使用用于主线程的初始值 X 生成 mc（M 计数器）并用用于辅助线程的初始值 0 生成 hc（H 计数器），并且该编译器生成计数器 M 和 H 以便计数主和辅助器代码中的同步向上周期。概念在于所有四个计数器（mc、M、hc、H）都执行自计数。辅助线程没有对主线程的推断。如果辅助线程太超前于主线程运行，它将发出一“等待”操作，如果辅助线程落后于主线程运行，则它将执行一“赶上”操作。

在特殊实施例中，对于所有的 X 个循环迭代，主线程发出一记入（post），以确保该辅助器不等待并可以前进以执行 non_faulting_load（无故障负荷）。此时，如果辅助线程在同步向上周期块中发出许多 non_faulting_load 后等待主线程，它将唤醒以执行 non_faulting_load。在另一特殊实施例中，对于所有 X 个循环迭代，辅助线程检查其 hc 计数器是否大于主线程的 mc 计数器且 hc 计数器大于辅助线程的同步向上周期 H*X，如果是，则辅助器将发出一等待并进入睡眠。这防止了辅助线程太超前于主线程运行。在另一实施例中，在另一同步向上周期块上进行迭代之前，辅助线程检查其 hc 计数器是否小于主线程的 mc 计数器。如果是，则辅助线程已落后，且必须通过更新其计数器 hc 和 H 以及来自主线程的所有捕获专用和内驻变量来“跟上并向前跳”。图 9A－9C 是示出根据一个实施例的应用程序、主线程和辅助线程的示例性伪代码的示图。参考图 9A－图 9C，编译器编译应用程序的

源代码901并使用前述技术中的至少一种生成用于主线程902和辅助线程903的代码。可以理解，代码901－903不限于C/C++。可以使用其它编程语言，诸如Fortran或汇编。

在创建了用于辅助线程的代码后，编译器可进一步静态或动态地分配用于每个辅助线程和主线程的资源，以确保主线程和辅助线程之间以及辅助线程之间没有资源冲突。可以为编译器内的辅助线程管理诸如寄存器上下文的硬件资源。特别是，可以在主线程和辅助线程之间以及多个辅助线程之间静态或动态地划分寄存器组。结果，当编译器用尽资源时，或者当出现特定主线程事件的稀少情况的运行时间时，可以避免经由用于线程的存储器的内驻/外驻寄存器拷贝且可以在编译时间销毁线程。

根据一个实施例，编译器可以按自底向上的顺序“走过”辅助线程并传达数据结构中的资源使用，诸如图12所示的资源表。可以是主线程的父辅助线程使用该信息并确保其资源不与线程资源重叠。当线程资源惩罚主执行线程时，例如通过迫使主线程溢出/填充寄存器，编译器可杀死先前创建的线程。

图10是说明根据一个实施例的线程的示例性配置的框图。在该实施例中，示例性配置1000包括主线程1001（例如，父线程）和可从主线程1001产生的三个辅助线程（例如，子线程）1002－1004，并且线程1003可从线程1002产生（例如，辅助线程1002是辅助线程1003的父线程）。可以理解，辅助线程不限于三个辅助线程，可以包括更多或更少的辅助线程。辅助线程可以通过产生指令生成并且线程执行可以在产生（spawn）指令之后继续。

在线程创建阶段期间由编译器创建线程，诸如图5－8所示的那些操作。根据一个实施例，编译器在线程创建阶段中创建线程，并在后续的线程资源分配阶段中分配用于线程的资源。动态地且通常地，当其父线程停顿时产生辅助线程。示例性配置1000可在页面故障或3级（L3）高速缓存不命中期间出现。

关键的是线程仅可以与父线程共享输入寄存器（或者一般资源）。例如，参考图10当主线程1001需要一寄存器时，在它产生辅助线程1002前它将一值写入寄存器R10并在辅助线程1002终止后使用寄存器R10。辅助线程1002和任何其子线程（本例中，辅助线程1003是辅助线程1002的唯一子线程，且辅助线程1002和1004是主线程1001的子线程）都不可以对寄存器R10进行写入。否则，它们

将破坏主线程 1001 中的值。这将导致不正确的程序执行。为了避免这种资源冲突，根据一个实施例，编译器可以静态或动态地划分资源。

根据一个实施例，编译器按自底向上的顺序分配用于辅助线程和主线程的资源。图 11 是示出根据一个实施例的用于分配线程资源的示例性伪代码的框图。即，在示例性算法 1100 中，编译器按自底向上的顺序分配用于辅助线程的所有资源（框 1101），此后基于辅助线程所使用的资源来分配用于主线程的资源（框 1102），以避免资源冲突。

出于说明目的，线程使用的资源被假定为硬件寄存器。但是，类似的概念可应用于本领域普通技术人员显而易见的其它资源，诸如存储器或中断。参考图 10，通过从线程链的领先线程自底向上行进，编译器动态地划分寄存器。在该示例中，辅助线程 1003 是包括辅助线程 1002 的第一线程链中的叶线程。辅助线程 1004 是第二线程链中的叶线程。编译器将每个辅助线程中的寄存器分配记录于数据结构中，诸如类似于图 12 的示例性资源表 1200 的资源表。随后，父线程读取其子线程的资源分配并进行其分配，并在其资源表中进行报告。

图 12 是示出根据一个实施例的示例性资源数据结构的框图。示例性数据结构 1200 可以实现作为存储器中存储的并可由编译器访问的表。或者，示例性数据结构 1200 可以在数据库中实现。在一个实施例中，示例性数据结构 1200 包括但不限于由线程 ID 1201 所标识的各线程使用的写入资源 1202 和内驻资源。也存在其它配置。

参考图 10 和 12，根据一个实施例，一开始，分配辅助线程 1003（例如，在自底向上方案中具有最底顺序的线程）的寄存器。内驻值是 V5 和 V6，并假定它们分别被分配给了寄存器 R2 和 R3。此外，V7 使得寄存器 R4 被分配，且 V9 使得寄存器 R5 被分配。用于辅助线程 1003 的资源表包括内驻＝（（V5,R2）,(V6,R3)）且写入的寄存器＝（R4,R5），如图 12 所示。在辅助线程 1002 中，编译器在分配期间用 R2 替换 V5 并用 R3 替换 V6 并在产生的指令中将寄存器 R4 和 R5（在辅助线程 1003 中被写入）标记为是活跃的。这防止了辅助线程 1003 的产生点上的 R4 或 R5 的寄存器使用，从而防止了辅助线程 1002 和辅助线程 1003 之间的资源冲突。对于辅助线程 1002，内驻值是 V3 和 V4 并分别被分配给寄存器 R6 和 R7。当 V8 和 V20 被分别分配给寄存器 R8 和 R9 时，用于辅助线程 1002 的资源表包括内驻

＝（（V3,R6）,(V4,R7)）以及写入的寄存器＝（R2,R3,R4,R5,R8.R9），如图 12 所示。写入的寄存器是用于辅助线程 1003 的内驻寄存器（例如，R2 和 R3）、辅助线程 1003 中的写入寄存器（例如 R4 和 R5）以及辅助线程 1002 中的写入寄存器（例如 R8 和 R9）。随后，编译器分配用于辅助线程 1004 的寄存器。当为所有的辅助线程分配了寄存器时，它分配用于主线程 1001 的寄存器。

此外，根据一个实施例，当编译器用尽寄存器时，它可以删除链内的一个或多个辅助线程。这例如可以出现于主线程用尽寄存器时，因为辅助线程链太深或者单个辅助线程需要太多寄存器且主线程必须溢出/填充寄存器。编译器可应用启发式方法以允许特定数量的溢出或者删除整个辅助线程链或该线程链中的一些线程。删除辅助线程的可选方案是明确地配置上下文保存/恢复的权重，以使在上下文切换时，通过辅助线程的执行被写入的父线程的活跃寄存器可以由硬件自动保存。尽管该上下文切换相对昂贵，但这种情况是稀少的。此外，这种微粒上下文切换与多数 OS 支持的线程切换中所使用的全上下文切换或者常规的基于硬件的全上下文线程切换相比仍开销较少。

此外，当存在对内驻寄存器的冲突时，例如如果辅助线程 1003 重写一内驻寄存器（例如，mov v5=...）且在产生辅助线程 1003 后该寄存器也在辅助线程 1002 中使用，将存在对分配 v5 的寄存器（本例中，寄存器 R2）的资源冲突。为了处理该信息，编译器将使用可用性分析并插入补偿码，诸如在产生辅助线程 1003 前插入一 mov v5'=v5 指令，并在该产生后用 v5'替换 v5。

图 13 是示出根据一个实施例的用于分配线程资源的示例性处理的流程图。示例性处理 1300 可以由处理逻辑执行，该处理逻辑可包括硬件（电路、专用逻辑等）、软件（诸如通用计算机系统或专用机器上运行的）或者它们的组合。在一个实施例中，示例性处理 1300 包括在编译具有数据处理系统中可执行的一个或多个线程的代码期间选择具有最底顺序的当前线程，确定向从该当前线程产生的一个或多个子线程分配的资源，并且考虑到分配给当前线程的一个或多个子线程的资源来分配用于该当前线程的资源，以避免当前线程及其一个或多个子线程之间的资源冲突。

参框图 13，在框 1301 处，处理逻辑标识一个或多个线程，包括主线程及其辅助线程，并选择具有最底顺序的线程作为当前线程。可以利用在编译的线程创建阶段期间创建的线程依赖性图表来标识这些线程。在框 1302 处，处理逻辑检索可从

当前线程中产生的任何子线程的资源信息。这些资源信息从与子线程相对应的数据结构中获得，诸如图 12 的资源表 1200。框 1303 处，如果没有更多的可用资源，处理逻辑可以从链中删除一个或多个线程并重新开始（框 1309）。如果有更多的可用资源，框 1304 处，处理逻辑考虑到其子线程所使用的资源来分配用于当前线程的资源，不引起资源冲突。此后，在框 1305 处，处理逻辑在诸如资源表 1200 的相关资源表中更新分配给当前线程的资源。以上处理继续，直到不剩余更多的辅助线程（例如，主线程的子线程）（框 1306 和 1308）。最后，在框 1307 处，处理逻辑基于所有辅助线程的资源信息分配用于主线程的资源（例如，所有辅助线程的父线程），而不引起资源冲突。可以包括其它操作。

已基于类似于以下配置的系统针对各种基准测试工具测试了上述技术：

具有超线程技术的处理器	
线程	2 个逻辑处理器
跟踪高速缓存	12k 微操作码，8 路相联，每行 6 个微操作码
L1 D 高速缓存	8k 字节，4 路相联，64 字节行大小 2 周期整数访问，4 周期 FP 访问
L2 统一高速缓存	256 字节，8 路相联，128 字节行大小，7 周期访问等待时间
负荷缓冲器	48
存储缓冲器	24

各种基准测试工具包括以下至少一种：

基准	描述	输入设定
nbody_walker	从 Nbody 图表中的任一节点遍历最近的体	20k 体
mst	计算用于数据聚类的最小生成树	3k 节点
em3d	解决 3D 中的电磁传播	20k 5 度节点
health	建模健康系统的体系数据块	5 级

mcf	用于总线调度的整数编程算法	Lite

图 14A 是示出 nbody_walker 基准测试工具上辅助线程的性能改善的示图。图 14B 是示出在同步周期的给定值下 nbody_walker 的加速结果的示图。图 14C 是示出相对于各种基准测试的自动处理对手动处理的示图。图 14D 是示出在给定的同步周期下使用 nbody_walker 自动处理优于手动处理的改善的示图。

因此，已描述了用于多线程的线程管理的方法和装置。在以上说明书中，已参考其特定的示例性实施例描述了本发明。显然，可以对其进行各种修改而不背离本发明的较宽精神和范围，如以下权利要求书中所阐述的。因此，说明书和附图被认为是说明性而非限制性的。

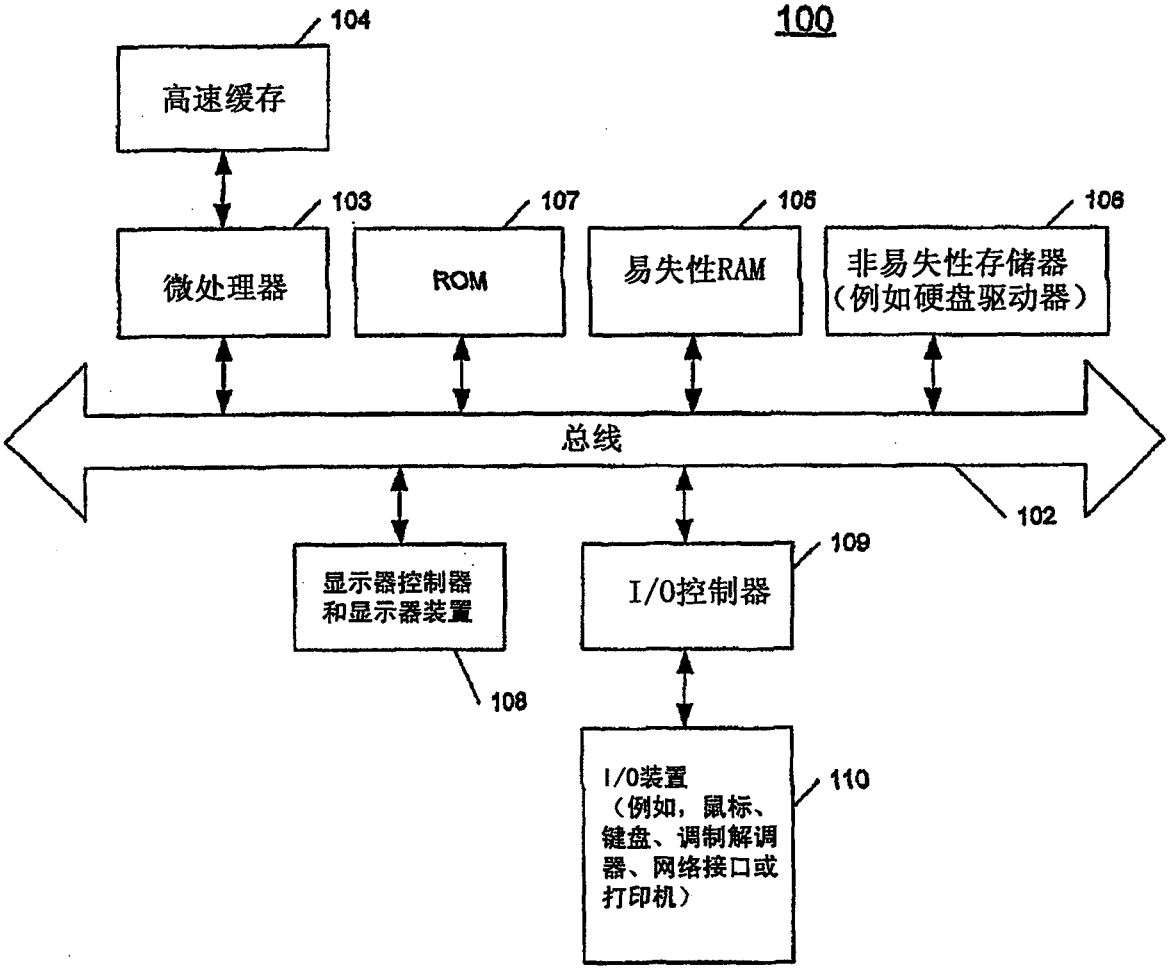

100
104
高速缓存
103
微处理器
107
ROM
105
易失性RAM
106
非易失性存储器
(例如硬盘驱动器)
总线
102
显示器控制器
和显示器装置
108
109
I/O控制器
I/O装置
(例如，鼠标、
键盘、调制解调
器、网络接口或
打印机)
110

图 1

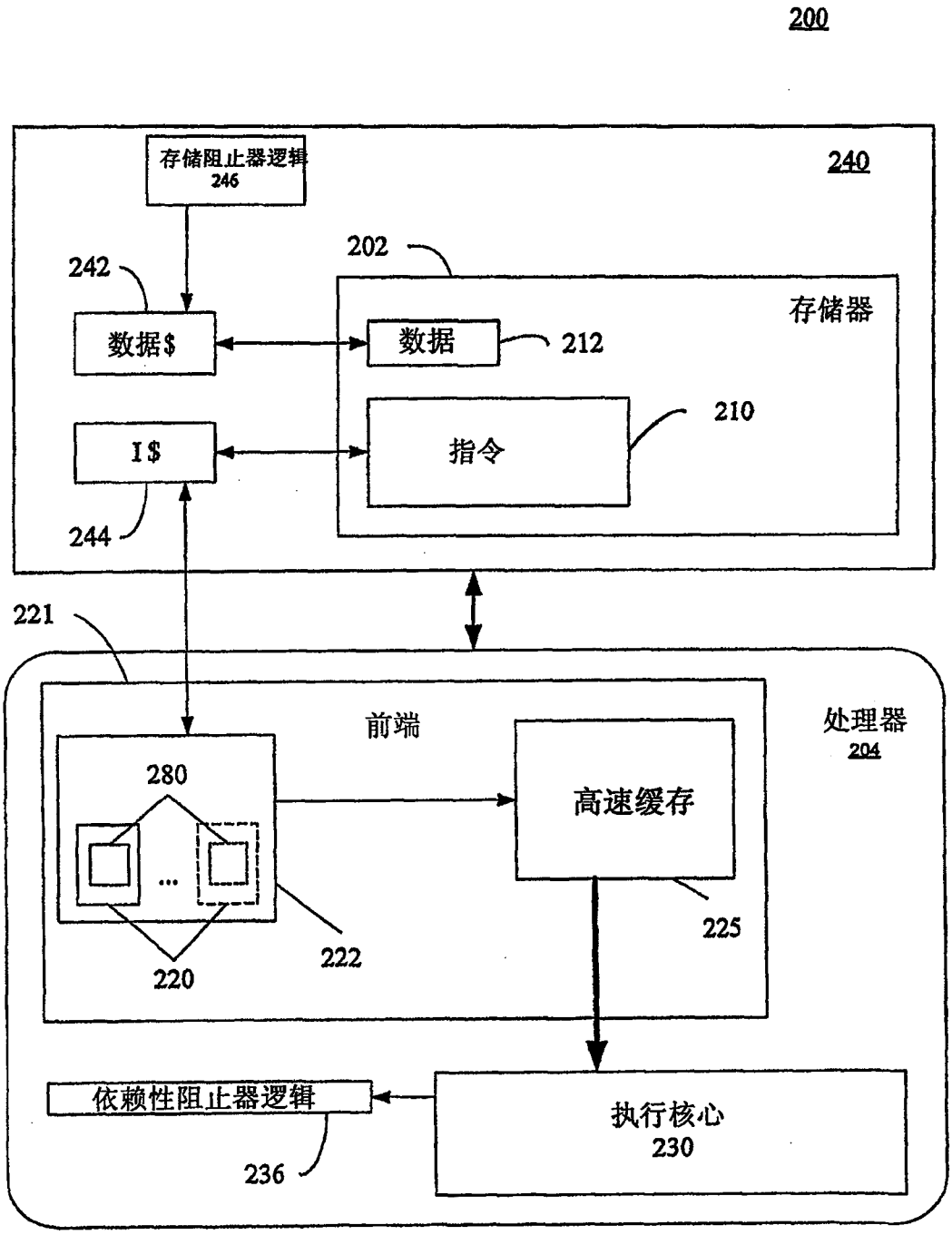

200
240
存储阻止器逻辑
246
202
242
存储器
数据$
数据
212
210
I$
指令
244
221
前端
处理器
204
280
高速缓存
...
225
222
220
依赖性阻止器逻辑
执行核心
230
236

图 2

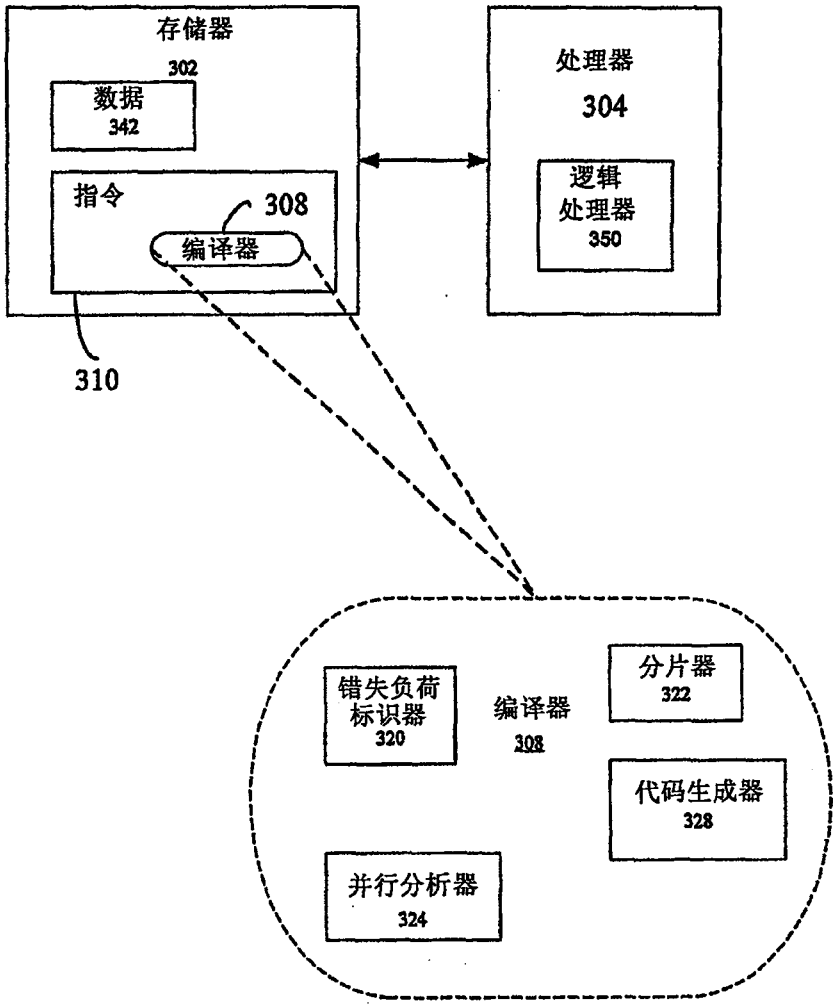

存储器
302
数据
342
指令
308
编译器
310
处理器
304
逻辑
处理器
350
错失负荷
标识器
320
编译器
308
分片器
322
代码生成器
328
并行分析器
324

图 3

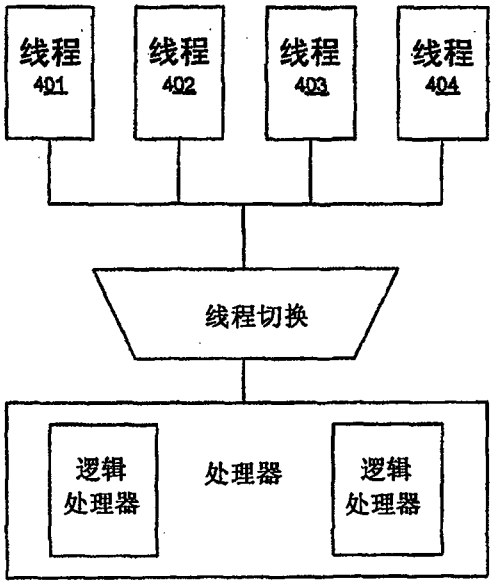

线程
401
线程
402
线程
403
线程
404
400
线程切换
逻辑
处理器
处理器
逻辑
处理器

图 4A

（现有技术）

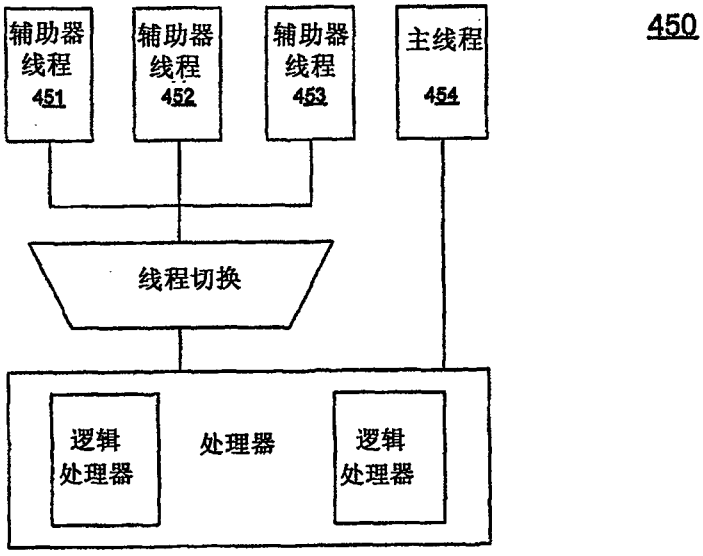

辅助器
线程
451
辅助器
线程
452
辅助器
线程
453
主线程
454
450
线程切换
逻辑
处理器
处理器
逻辑
处理器

图 4B

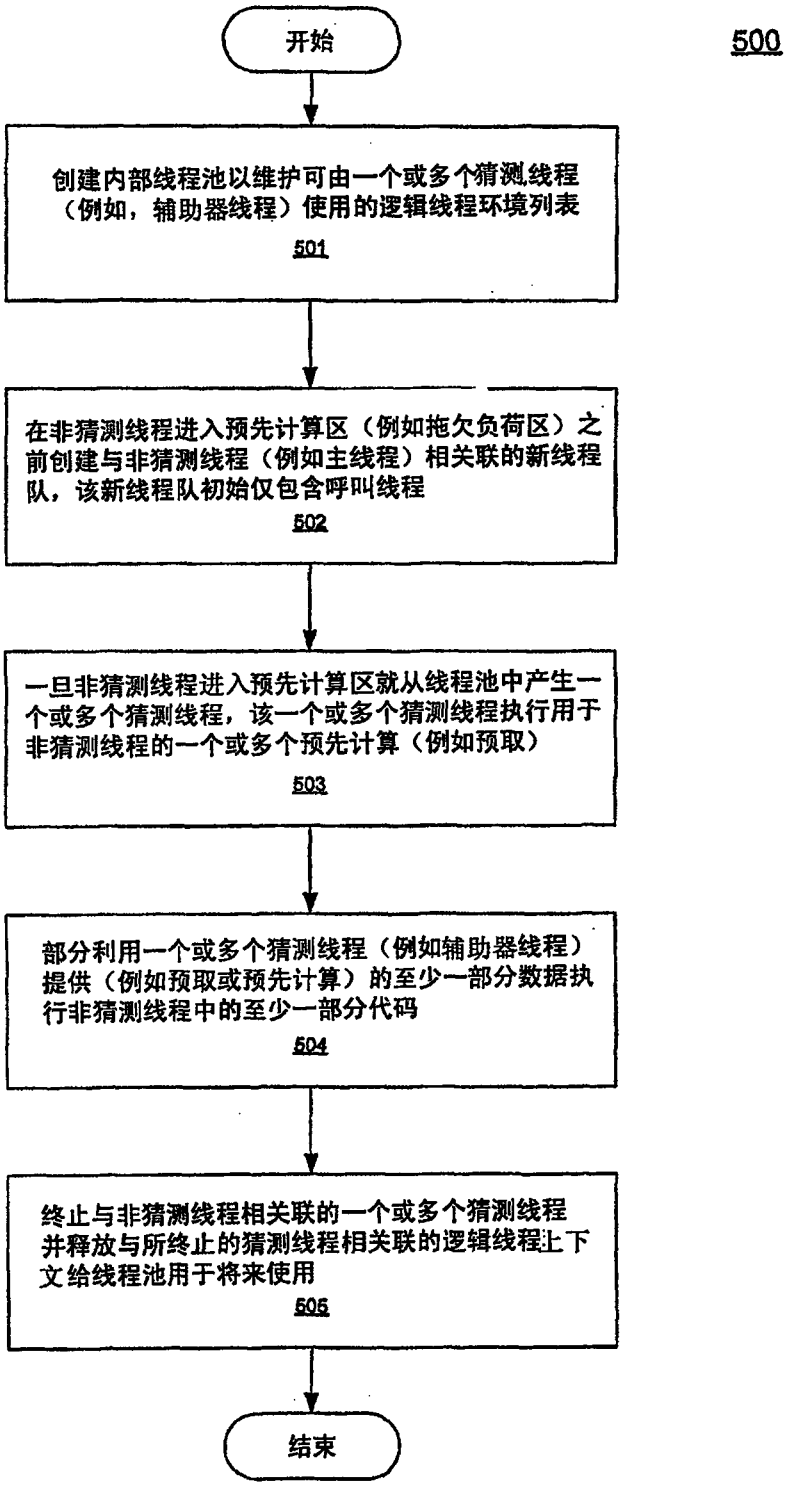

开始
500
创建内部线程池以维护可由一个或多个猜测线程（例如，辅助器线程）使用的逻辑线程环境列表
501
在非猜测线程进入预先计算区（例如拖欠负荷区）之前创建与非猜测线程（例如主线程）相关联的新线程队，该新线程队初始仅包含呼叫线程
502
一旦非猜测线程进入预先计算区就从线程池中产生一个或多个猜测线程，该一个或多个猜测线程执行用于非猜测线程的一个或多个预先计算（例如预取）
503
部分利用一个或多个猜测线程（例如辅助器线程）提供（例如预取或预先计算）的至少一部分数据执行非猜测线程中的至少一部分代码
504
终止与非猜测线程相关联的一个或多个猜测线程并释放与所终止的猜测线程相关联的逻辑线程上下文给线程池用于将来使用
505
结束

图 5

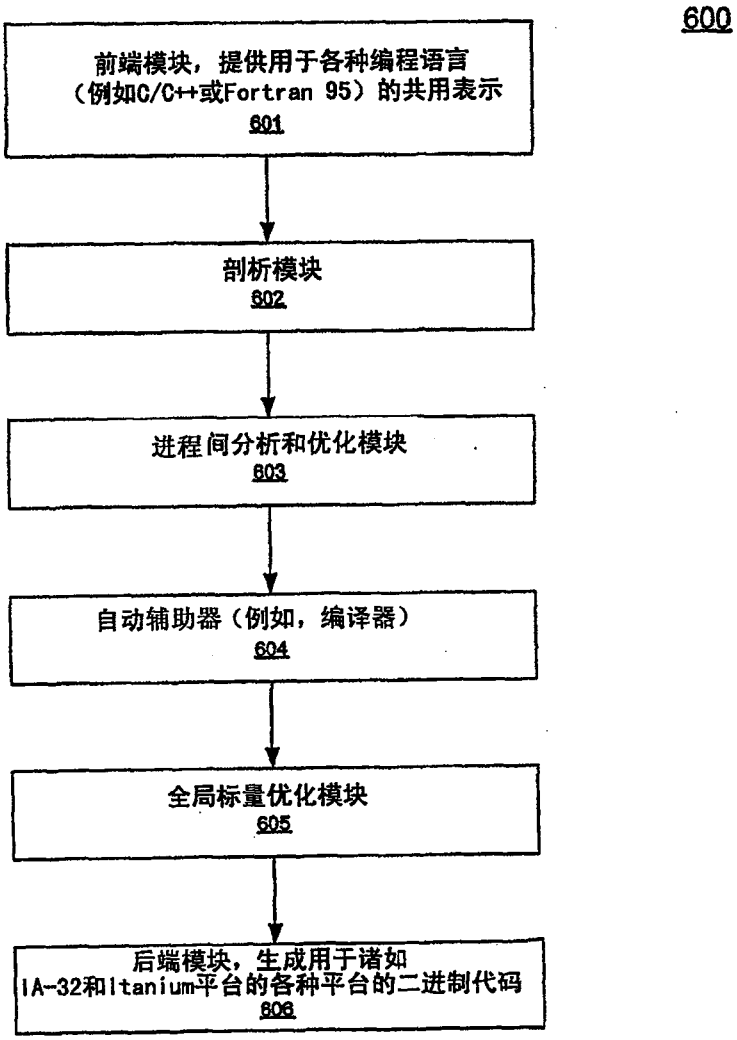

600
前端模块，提供用于各种编程语言
（例如C/C++或Fortran 95）的共用表示
601
剖析模块
602
进程间分析和优化模块
603
自动辅助器（例如，编译器）
604
全局标量优化模块
605
后端模块，生成用于诸如
IA-32和Itanium平台的各种平台的二进制代码
606

图 6

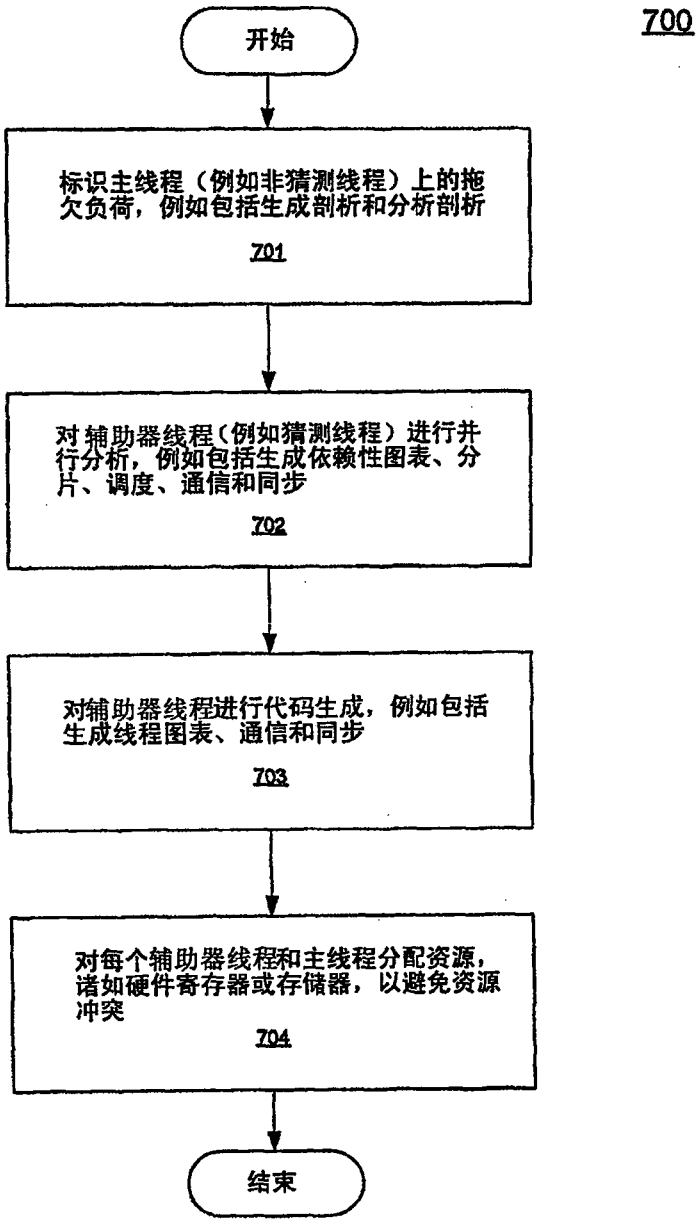

700
开始
标识主线程（例如非猜测线程）上的拖欠负荷，例如包括生成剖析和分析剖析
701
对辅助器线程（例如猜测线程）进行并行分析，例如包括生成依赖性图表、分片、调度、通信和同步
702
对辅助器线程进行代码生成，例如包括生成线程图表、通信和同步
703
对每个辅助器线程和主线程分配资源，诸如硬件寄存器或存储器，以避免资源冲突
704
结束

图 7

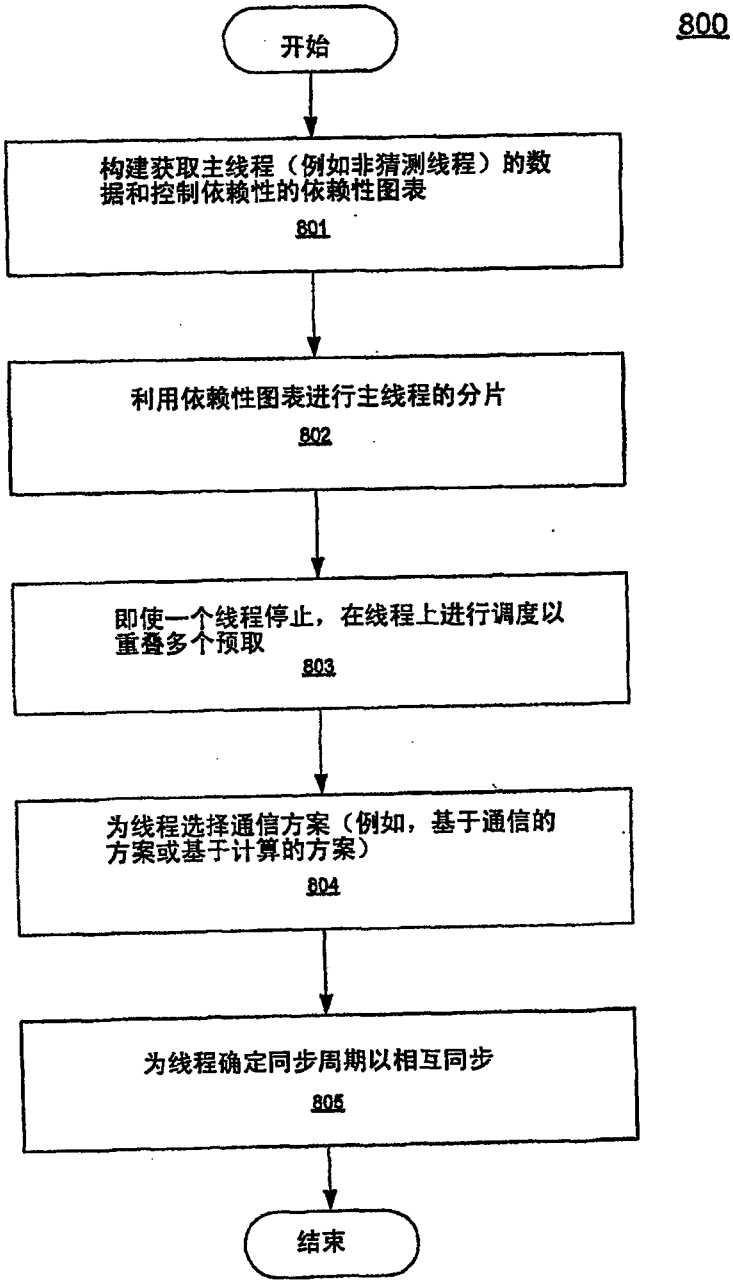

800
开始
构建获取主线程（例如非猜测线程）的数据和控制依赖性的依赖性图表
801
利用依赖性图表进行主线程的分片
802
即使一个线程停止，在线程上进行调度以重叠多个预取
803
为线程选择通信方案（例如，基于通信的方案或基于计算的方案）
804
为线程确定同步周期以相互同步
805
结束

图 8

```
void foo_main(LIST *p)
{
 ....
 { while (p!= NUL:L) {
       ...
       do_work1(p->data1, 10);
       do_work2(p->data2, 20);
       ...
       p= p->next;
   }
 }
}
```

(I)连续码

901

图 9A

```
void foo_main(LIST *p)
{ mc = X; M=1; _ssp_begin();
 _ssp_spawn_helper(... helper_foo, ...p...);
 while (p != NULL ) {
     do_work1(p->data1, 10);
     do_work2(p->data2, 20);
     mc = mc + 1;
     if (mc > M*X) {
        M++; _ssp_post(helper_tid);
     }
     p= p->next;
  }
  _ssp_end();
}
```

(II) 主线程码

902

图 9B

```
T-entry foo_helper: captureprivate(p)
{ hc = 0; H = 1; local_p = p;
  while (local_p!= NUL:L) {
     non_faulting_load(local_p->data1);
     non_faulting_load(local_p->data2);
     hc = hc + 1;
     if (hc > H*X && hc > mc)
     { H++; _ssp_wait(main_tid); }
     else if (hc <= mc) {
      (hc, local_p, H) = catchup(mc, p, M)
     }
     local_p = local_p->next;
  }
T-ret }
```

(III) 辅助器线程码

903

图 9C

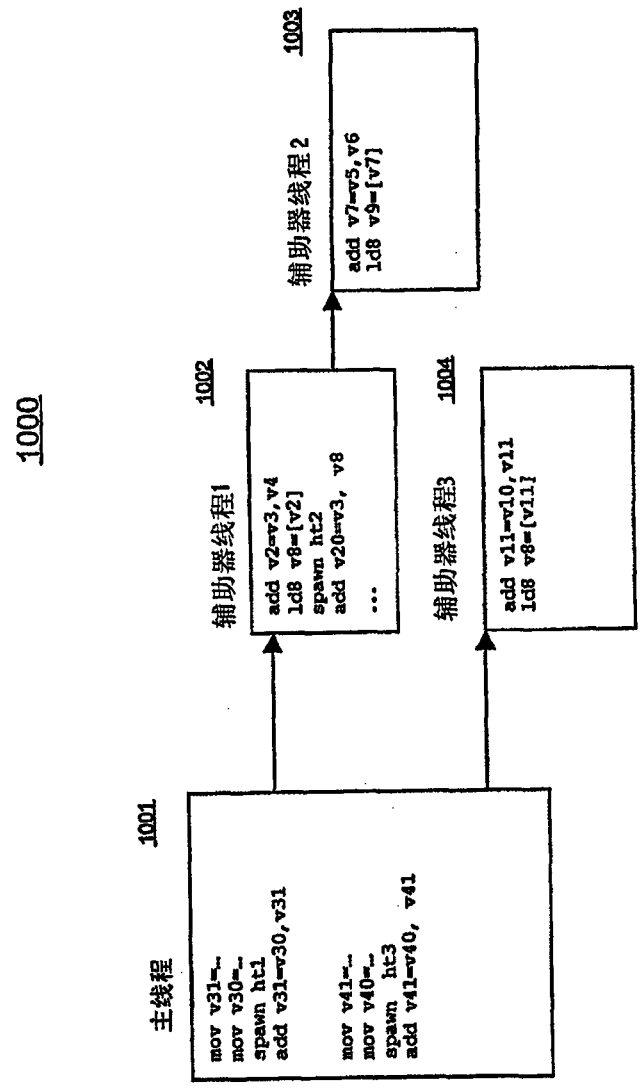

1000
1001
主线程
mov v31=_
mov v30=_
spawn ht1
add v31=v30,v31
mov v41=_
mov v40=_
spawn ht3
add v41=v40, v41
1002
辅助器线程1
add v2=v3,v4
ld8 v8=[v2]
spawn ht2
add v20=v3, v8
...
1003
辅助器线程2
add v7=v5,v6
ld8 v9=[v7]
1004
辅助器线程3
add v11=v10,v11
ld8 v8=[v11]

图 10

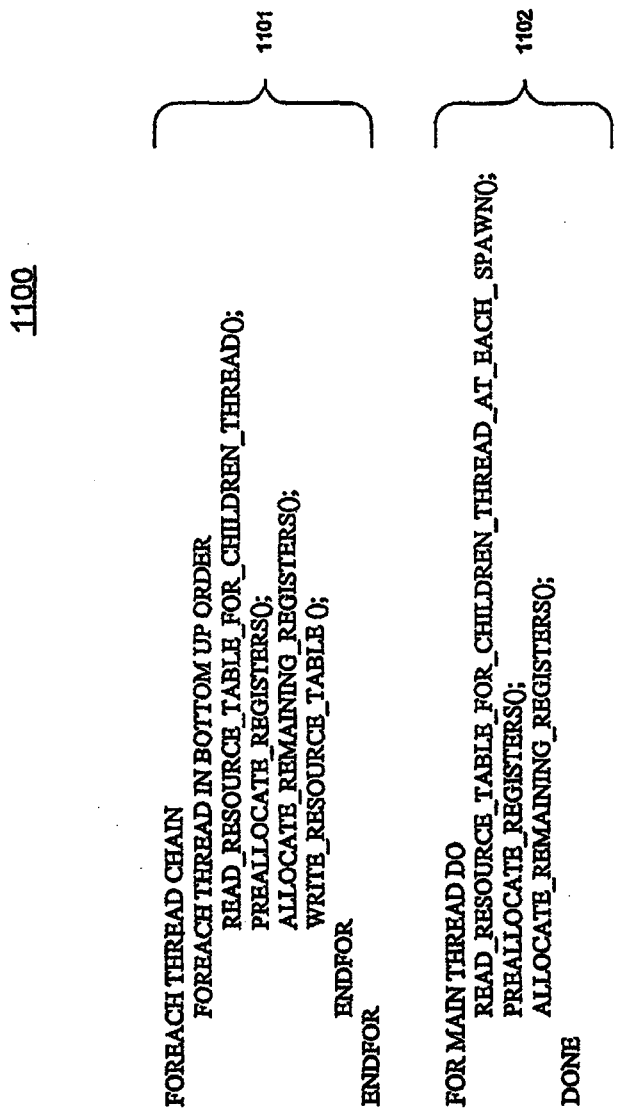

1100
1101
1102
FOREACH THREAD CHAIN
FOREACH THREAD IN BOTTOM UP ORDER
READ_RESOURCE_TABLE_FOR_CHILDREN_THREAD();
PREALLOCATE_REGISTERS();
ALLOCATE_REMAINING_REGISTERS();
WRITE_RESOURCE_TABLE ();
ENDFOR
ENDFOR
FOR MAIN THREAD DO
READ_RESOURCE_TABLE_FOR_CHILDREN_THREAD_AT_EACH_SPAWN();
PREALLOCATE_REGISTERS();
ALLOCATE_REMAINING_REGISTERS();
DONE

图 11

1200

线程ID 1201	写入资源 1202	内驻资源 1203
辅助器线程 2	R4, R5	(V5, R2) (V6, R3)
辅助器线程 1	R2, R3, R4, R5, R8, R9	(V3, R6) (V4, R7)
辅助器线程 3	...	...
...	...	...
辅助器线程 n	...	...
主线程	...	...

图 12

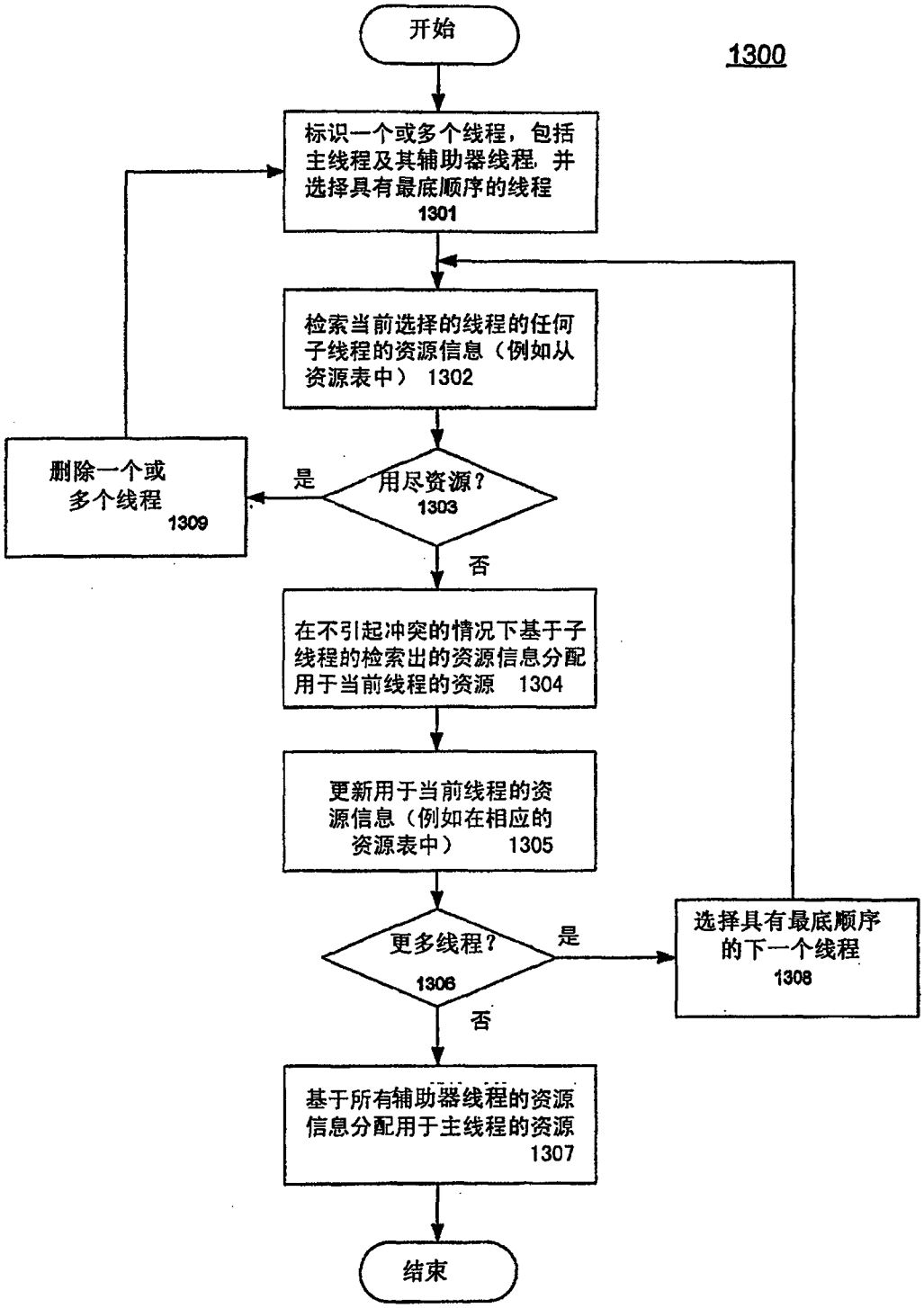

开始
1300
标识一个或多个线程，包括主线程及其辅助器线程，并选择具有最底顺序的线程
1301
检索当前选择的线程的任何子线程的资源信息（例如从资源表中） 1302
用尽资源？
1303
是
删除一个或多个线程
1309
否
在不引起冲突的情况下基于子线程的检索出的资源信息分配用于当前线程的资源 1304
更新用于当前线程的资源信息（例如在相应的资源表中） 1305
更多线程？
1306
是
选择具有最底顺序的下一个线程
1308
否
基于所有辅助器线程的资源信息分配用于主线程的资源
1307
结束

图 13

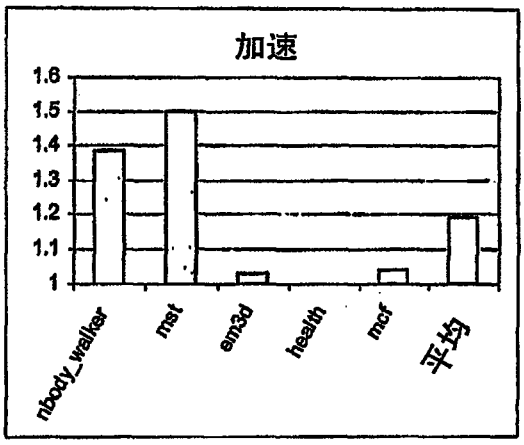

加速
1.6
1.5
1.4
1.3
1.2
1.1
1
nbody_walker
mst
em3d
health
mcf
平均

图 14A

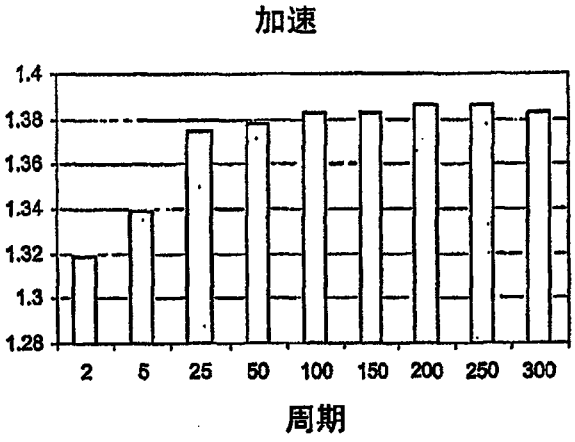

加速
1.4
1.38
1.36
1.34
1.32
1.3
1.28
2
5
25
50
100
150
200
250
300
周期

图 14B

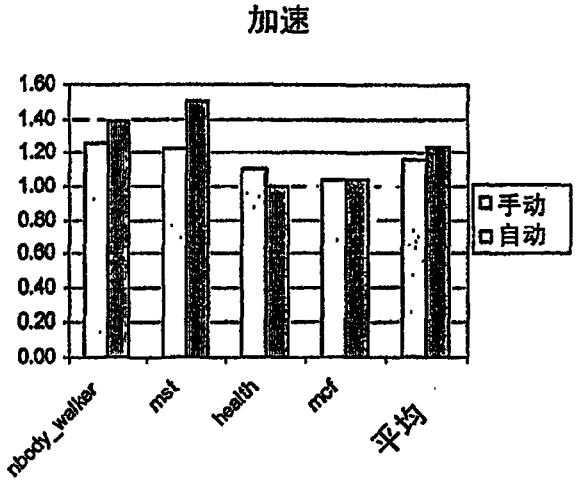

加速
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
手动
自动
nbody_walker
mst
health
mcf
平均

图 14C

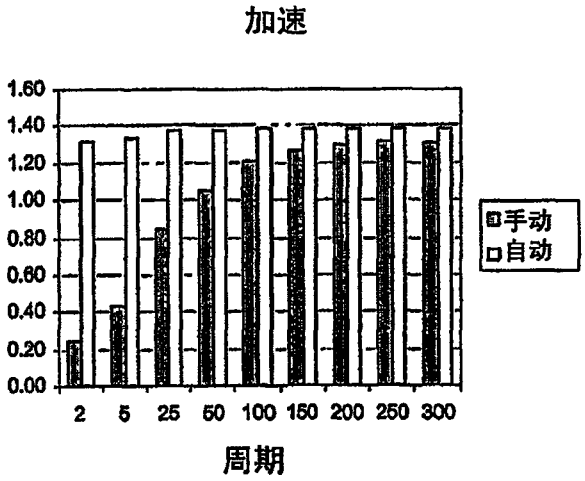

加速
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
手动
自动
2
5
25
50
100
150
200
250
300
周期

图 14D