

(12) 发明专利

(10) 授权公告号 CN 101329638 B

(45) 授权公告日 2011. 11. 09

(21) 申请号 200710109089. 5

郭龙, 陈闳中, 叶青

(22) 申请日 2007. 06. 18

. 构造串行程序对应的并行任务 (DAG)

(73) 专利权人 国际商业机器公司

图. 《计算机工程与应用》. 2007, (第 01
期), 41-43.

地址 美国纽约

审查员 孟祥岳

(72) 发明人 冯博 阎蓉 王鲲 王华勇

(74) 专利代理机构 北京市中咨律师事务所

11247

代理人 李峥 于静

(51) Int. Cl.

G06F 9/45(2006. 01)

G06F 9/455(2006. 01)

(56) 对比文件

CN 1123930 A, 1996. 06. 05, 全文 .

CN 1645339 A, 2005. 07. 27, 说明书第 1-4
页 .

US 6230313 B1, 2001. 05. 08, 说明书第
1, 7-24 栏以及附图 1-10.

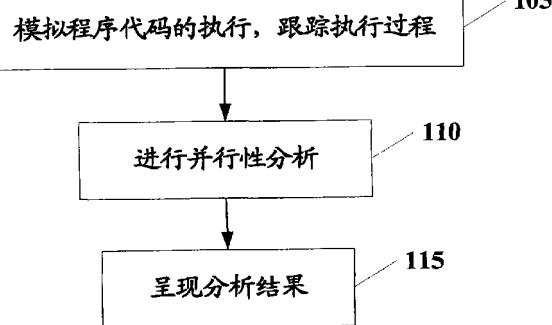
权利要求书 3 页 说明书 10 页 附图 11 页

(54) 发明名称

程序代码的并行性的分析方法和系统

(57) 摘要

本发明提供一种程序代码的并行性的分析方法和系统以及用于跟踪程序代码的执行信息的模拟器。该程序代码的并行性的分析方法包括：模拟程序代码的顺序执行，以跟踪该程序代码的执行过程；以及根据对上述程序代码的执行过程的跟踪结果，对上述程序代码进行并行性的分析。本发明通过模拟程序代码的顺序执行来收集其执行信息，并根据所收集的执行信息对该程序代码进行并行性的分析，以帮助程序设计者更有效地针对多核体系结构进行程序代码的并行任务的划分，从而提高并行软件开发的效率。



1. 一种程序代码的并行性的分析方法,包括:

模拟程序代码的顺序执行,以跟踪该程序代码的执行过程;以及

根据对上述程序代码的执行过程的跟踪结果,对上述程序代码进行并行性的分析,

其中上述模拟程序代码的顺序执行,以跟踪该程序代码的执行过程的步骤进一步包括:

为上述程序代码模拟其目标系统的执行环境;

将上述程序代码加载到上述所模拟的执行环境中;

在上述执行环境中,顺序执行上述所加载的程序代码中的各指令;以及

将上述程序代码在上述顺序执行过程中的执行信息记入日志,

其中上述将程序代码在上述顺序执行过程中的执行信息记入日志的步骤进一步包括:

判断上述程序代码中的当前指令的类型;

如果其是调用非系统 API 的函数调用 / 返回指令,则将当前执行周期以及被调用函数的标识记入日志;

如果其是调用系统 API 中的存储器分配或释放函数的函数调用 / 返回指令,则将当前执行周期以及被分配的存储器大小、存储器地址记入日志;以及

如果其是加载 / 存储指令,则将当前执行周期以及数据地址、数据大小、读 / 写类型记入日志。

2. 如权利要求 1 所述的程序代码的并行性的分析方法,其中上述对程序代码的执行过程的跟踪结果是记录有上述程序代码在上述顺序执行过程中的执行动作、所述动作的相应时间信息或存储器类型信息的日志。

3. 如权利要求 1 所述的程序代码的并行性的分析方法,其中上述日志被分类为函数调用日志、存储器分配日志以及存储器访问日志,分别用于记录上述程序代码在顺序执行过程中的函数调用信息、存储器分配信息以及存储器访问信息。

4. 如权利要求 1 或 2 所述的程序代码的并行性的分析方法,其中根据对上述程序代码的执行过程的跟踪结果,对上述程序代码进行并行性的分析的步骤进一步包括:

根据上述跟踪结果,分析上述程序代码在上述执行过程中的各次函数调用之间的依赖关系,以找出其中没有依赖关系的、可并行执行的函数调用。

5. 如权利要求 4 所述的程序代码的并行性的分析方法,其中分析上述程序代码在上述执行过程中的各次函数调用之间的依赖关系的步骤进一步包括:

根据上述跟踪结果所记录的上述程序代码在上述顺序执行过程中的各执行动作及其相应的时间信息和存储器类型信息,来进行上述依赖关系的分析。

6. 如权利要求 4 所述的程序代码的并行性的分析方法,其中分析上述程序代码在上述执行过程中的各次函数调用之间的依赖关系的步骤进一步包括:

根据上述跟踪结果,为上述程序代码在上述执行过程中的各次函数调用获得该次函数调用期间的读访问记录集合和写访问记录集合;

对于同一函数中的各函数调用,根据该各函数调用期间的读访问记录集合和写访问记录集合,确定其之间是否存在存储器访问冲突;以及

将上述同一函数中的各函数调用中存在存储器访问冲突的各函数调用确定为具有依

赖关系、不可并行执行，而将不存在存储器访问冲突的各函数调用确定为不具有依赖关系、可并行执行。

7. 如权利要求 4 所述的程序代码的并行性的分析方法，其中分析上述程序代码在上述执行过程中的各次函数调用之间的依赖关系的步骤进一步包括：

以上述程序代码在上述执行过程中的各次函数调用作为节点、以函数调用之间的相互关系作为节点之间的关系，生成该程序代码的调用树；

根据上述跟踪结果，为上述各节点获得其所对应的函数调用期间的读访问记录集合和写访问记录集合；

对于上述调用树中同一父节点下的各叶子节点，根据其读访问记录集合和写访问记录集合确定其之间是否存在存储器访问冲突；以及

将上述同一父节点下的各叶子节点中存在存储器访问冲突的各叶子节点进一步合并为一个节点。

8. 如权利要求 1 或 2 所述的程序代码的并行性的分析方法，其中根据上述跟踪结果，对上述程序代码进行并行性的分析的步骤进一步包括：

根据上述跟踪结果，对上述程序代码在上述执行过程中的各次函数调用进行代码大小和数据大小的分析。

9. 如权利要求 8 所述的程序代码的并行性的分析方法，其中对上述程序代码在上述执行过程中的各次函数调用进行代码大小和数据大小的分析的步骤进一步包括：

对于上述程序代码在上述执行过程中的各次函数调用，

将其所对应的被调用函数的本身代码大小与该函数调用期间进一步调用的各函数的代码大小之和作为该函数调用的代码大小；以及

将其在生命周期中进行各次存储器访问时所访问的存储器空间大小之和作为该函数调用的数据大小。

10. 一种用于跟踪程序代码的执行过程的模拟器，包括：

模拟的执行环境，用于模拟程序代码能够在其中顺序执行的目标系统的执行环境；以及

执行过程跟踪单元，用于跟踪上述程序代码在上述模拟执行环境中的顺序执行过程，以获得该程序代码的执行信息，

其中上述执行过程跟踪单元判断在上述模拟的执行环境中执行的当前指令的类型：

如果该指令是调用非系统 API 的函数调用 / 返回指令，则将当前执行周期以及被调用函数的标识记入日志；

如果该指令是调用系统 API 中的存储器分配或释放函数的函数调用 / 返回指令，则将当前执行周期以及被分配的存储器大小、存储器地址记入日志；以及

如果该指令是加载 / 存储指令，将当前执行周期以及数据地址、数据大小、读 / 写类型记入日志。

11. 权利要求 10 所述的用于跟踪程序代码的执行过程的模拟器，还包括：

跟踪预处理单元，用于获得与上述程序代码在上述顺序执行过程中的各执行动作相应的存储器类型信息，并将其记入日志。

12. 权利要求 10 所述的用于跟踪程序代码的执行过程的模拟器，其中上述模拟的执行

环境进一步包括：

模拟存储器,用作所模拟的上述目标系统的存储器；

存储器管理单元,用于在程序代码的存储器访问期间将虚拟地址转换为物理地址；

流水线,其包括指令取单元、指令解码单元和指令执行单元,分别用于实现从上述模拟存储器的指令的获取、指令的格式转换以及指令的执行；以及

模拟寄存器,用作所模拟的上述目标系统的寄存器。

13. 一种程序代码的并行性的分析系统,包括：

权利要求 10-12 中任意一项所述的用于跟踪程序代码的执行过程的模拟器；以及

并行性分析单元,用于根据上述模拟器对程序代码的执行过程的跟踪结果,对该程序代码进行并行性的分析。

14. 权利要求 13 所述的程序代码的并行性的分析系统,其中上述并行性分析单元根据上述模拟器获得的跟踪结果中记录的上述程序代码的各执行动作及其相应的时间信息和存储器类型信息,来进行并行性的分析。

15. 权利要求 13 所述的程序代码的并行性的分析系统,其中上述并行性分析单元进一步包括：

访问记录获得单元,用于根据上述模拟器所获得的跟踪结果,为上述程序代码在执行过程中的各次函数调用获得读访问记录集合和写访问记录集合；以及

依赖性确定单元,用于对于同一函数中的各函数调用,根据该各函数调用的读访问记录集合和写访问记录集合,确定其之间是否存在依赖关系。

16. 权利要求 15 所述的程序代码的并行性的分析系统,其中上述依赖性确定单元进一步包括：

调用树构造单元,用于根据上述模拟器所获得的跟踪结果,以上述程序代码在执行过程中的各次函数调用作为节点、以函数调用之间的关系作为节点之间的关系,生成该程序代码的调用树；以及

调用树优化单元,用于对于上述调用树中同一父节点下的各叶子节点,根据其所对应的函数调用期间的读访问记录集合和写访问记录集合确定其之间是否存在存储器访问冲突,并且将存在存储器访问冲突的各叶子节点进一步合并为一个节点。

17. 权利要求 13 所述的程序代码的并行性的分析系统,还包括：

代码 / 数据大小分析单元,用于对于上述程序代码在执行过程中的各次函数调用,将其所对应的被调用函数的本身代码大小与该函数调用期间进一步调用的各函数的代码大小之和作为该函数调用的代码大小,并将其在生命周期中进行各次存储器访问时所访问的存储器空间大小之和作为该函数调用的数据大小。

程序代码的并行性的分析方法和系统

技术领域

[0001] 本发明涉及数据处理领域,具体地,涉及用于多核体系结构的基于模拟的程序代码的并行性的分析方法和系统以及用于跟踪程序代码的执行过程的模拟器。

背景技术

[0002] 随着对于计算机的应用需求的不断提高,单个处理器的能力(主频、线宽等)也随之不断地增长。但是,可以预见,这样的单个处理器的能力提升终究会达到物理极限。从而,在单个处理器的能力增长到一定程度时,为了继续获得更高的微处理器性能,自然不得不向更宽的方向发展。推动微处理器性能不断提高的因素主要是半导体工艺技术的飞速进步和处理器体系结构的不断发展。目前的半导体工艺技术已可使微处理器集成的晶体管数目达到数亿个,能够确保微处理器的结构向更加复杂的方向发展。从而,在这样的技术发展和应用需求之下,多核(多处理器)体系结构便成为了必然产物。

[0003] 多核体系结构通过在一个芯片上集成多个微处理器核心来提高程序执行的并行性。每个微处理器核心实质上都是一个相对简单的单线程微处理器或者比较简单的多线程微处理器。在多核体系结构中,多个微处理器核心可以并行地执行任务,因而具有较高的线程级并行性。并且,多核体系结构通过采用相对简单的微处理器作为处理器核心,能够得到高主频、设计和验证周期短、控制逻辑简单、扩展性好、易于实现、功耗低、通信延迟低等优点。从而,在今后的发展趋势上,无论是移动与嵌入式应用、桌面应用还是服务器应用,都将采用多核体系结构。

[0004] 但是,在多核体系结构带来了如此多的好处的同时,它也对系统和程序设计等方面提出了挑战。也就是说,由于多核体系结构在单个处理器内封装了多个处理器“执行核”,所以只要软件设计合理,就能够支持软件的多个线程的完全并行执行。从而,这样的多核体系结构的设计概念迫使软件开发朝并行化方向发展,以便充分发挥多核体系结构的优势。

[0005] 但是,在x86架构下,应用程序的开发者还停留在单线程的开发模式下,随着多核体系结构逐渐应用在PC、Server、嵌入式系统和游戏控制台等中,x86架构下的传统的顺序程序设计概念终将被并发和同步所打破。尤其对于异构存储器限制系统那样的CELL多核体系结构(每个处理器核具有有限的256K字节本地存储器)的程序设计者来说,更要适应从顺序到并行的程序设计概念的转变。也就是说,程序设计者必须学会如何为CELL这样的多核体系结构设计应用程序,即学会如何进行并行程序设计。但是,在并行程序设计中,程序代码中的各函数的并行性识别及并行任务的划分通常被认为是一种极其依赖于程序设计者的领域知识、经验和对于体系结构的了解的技术。没有足够的工具支持,并行性分析及任务划分会极大地降低整个并行软件开发的效率。

[0006] 因此,需要设计出一种高效、准确的程序代码的并行性的分析技术,来帮助并行程序设计者更有效地针对多核体系结构进行程序代码的并行性的分析以及任务的划分,从而提高并行软件开发的效率。

发明内容

[0007] 本发明正是鉴于上述现有技术中的问题而提出的，其目的在于提供一种基于模拟的程序代码的并行性的分析方法和系统以及用于跟踪程序代码的执行过程的模拟器，以便通过模拟程序代码的顺序执行来收集其执行信息，并根据所收集的执行信息对该程序代码进行并行性的分析，来帮助程序设计者更有效地针对多核体系结构进行该程序代码的并行任务的划分，从而提高并行软件开发的效率。

[0008] 根据本发明的一个方面，提供一种程序代码的并行性的分析方法，包括：模拟程序代码的顺序执行，以跟踪该程序代码的执行过程；以及根据对上述程序代码的执行过程的跟踪结果，对上述程序代码进行并行性的分析。

[0009] 根据本发明的另一个方面，提供一种用于跟踪程序代码的执行过程的模拟器，包括：模拟的执行环境，用于模拟程序代码能够在其中顺序执行的目标系统的执行环境；以及执行过程跟踪单元，用于跟踪上述程序代码在上述模拟执行环境中的顺序执行过程，以获得该程序代码的执行信息。

[0010] 根据本发明的另一个方面，提供一种程序代码的并行性的分析系统，包括：上述的用于跟踪程序代码的执行过程的模拟器；以及并行性分析单元，用于根据上述模拟器所获得对程序代码的执行过程的跟踪结果，对该程序代码进行并行性的分析。

附图说明

[0011] 相信通过以下结合附图对本发明具体实施方式的说明，能够使人们更好地了解本发明上述的特点、优点和目的。

[0012] 图1是根据本发明实施例的程序代码的并行性的分析方法的流程图；

[0013] 图2是图1的方法中的执行过程跟踪步骤的详细流程图；

[0014] 图3是根据本发明一个实施例的执行信息的日志的示例；

[0015] 图4是图1的方法中的并行性分析步骤的详细流程图；

[0016] 图5(a)～(c)是根据本发明一个实施例的并行性分析过程的实例；

[0017] 图6是根据本发明实施例的程序代码的并行性的分析系统的方框图；

[0018] 图7是根据本发明实施例的用于跟踪程序代码的执行过程的模拟器的方框图；

[0019] 图8-10示出了图7的模拟器的工作过程；以及

[0020] 图11-13是程序代码的示例。

具体实施方式

[0021] 在多核体系结构中，对于计算强度很大的应用程序来说，会存在有大量的数据处理以及复杂的数据相关性，如果不进行数据分割的话，只能由一个低频的单核来串行处理，则执行时间会很长。从而，在多核体系结构中，需要正确地对这样的应用程序进行并行化。

[0022] 应用程序的并行化实际上就是将冗长的串行算法的时间复杂度通过增加空间复杂度的方式进行压缩，把以前一个周期一个操作去执行的算法结构改造成一个周期可以进行多个操作的并行算法，这就是并行化要做的主要工作。也就是说，并行化就是要找出一个应用程序内能够并行执行的任务，并将其分配给多个处理器核来并行执行，从而实现在一个时刻或者时间段内有一个以上的事件发生。但这种并行化却并非是一个简单的过程。因

为,即使多核体系结构中的多个处理器核能够同时执行一个程序中的多个任务,但这些任务之间可能会存在运行资源冲突的问题。多个处理器核同时运行期间很多资源其实是共享的,比如高速缓存、存储器、BUS 等。如果并行化不正确,例如不能够并行执行的任务被进行了并行化,则其结果可能会使该应用程序所得到的执行结果不正确;又例如,本来能够并行执行的任务未被并行化,则其结果便是处理器核的利用效率的降低。针对于此,本发明的目的便是要提供一种有效的程序代码的并行性的分析技术,以便将程序设计者从繁杂的并行性识别及任务划分中解脱出来。

[0023] 下面就结合附图对本发明的各个优选实施例进行详细的说明。

[0024] 图 1 是根据本发明实施例的程序代码的并行性的分析方法的流程图。如图 1 所示,首先,在步骤 105,模拟程序代码的顺序执行,以跟踪该程序代码的执行过程。具体地,在该步骤中,首先,为要进行并行性分析的已设计完成的程序代码模拟其目标系统的执行环境,然后在该模拟的执行环境中顺序执行该程序代码,并且在顺序执行的同时跟踪其执行过程。在该步骤中,模拟程序代码在目标执行环境中的顺序执行的目的是为了收集其执行信息,以用于对该程序代码的并行性的分析。在本发明的一个实施例中,上述执行信息包括但不限于:函数调用信息、存储器分配信息以及存储器访问信息。对于该步骤,将在后面结合附图 2 和 3 进行详细说明。

[0025] 在步骤 110,对上述程序代码进行并行性的分析。具体地,在该步骤中,根据对于上述程序代码的顺序执行过程的跟踪结果,即在步骤 105 中所收集的执行信息,分析该程序代码中的各函数调用之间的依赖关系。在该步骤中,根据执行信息,将上述程序代码中没有存储器访问冲突的各函数调用确定为没有依赖关系。对于该步骤,将在后面结合附图 4 和 5 进行详细说明。

[0026] 本发明实施例可选择地包括用于向用户呈现对上述程序代码的并行性的分析结果的步骤 115。具体地,在该步骤中,向用户呈现的是在步骤 110 中获得的上述程序代码中各函数调用之间的依赖关系,以帮助该程序代码的设计者找出该程序代码中没有依赖关系、能够由多个处理器核并行执行的任务。在一个实施例中,上述程序代码中的各函数调用之间的依赖关系是以树状图的形式来表现的。关于该树状图,在后面结合附图 4 和 5 的说明中有详细记载。此外,在其他实施例中,上述程序代码中的各函数调用之间的依赖关系也可以以列表、文本等任何形式来呈现。

[0027] 下面,结合图 2 详细描述上面图 1 的方法中的执行过程跟踪步骤 105。图 2 是该执行过程跟踪步骤 105 的详细流程图。

[0028] 如图 2 所示,首先,在步骤 205,为要进行并行性分析的程序代码模拟其目标系统的执行环境。具体地,该步骤是利用模拟器来实现的。也就是说,在该步骤中,利用模拟器在主系统(当前系统)上模拟将要实际使用该程序代码的目标系统,以便为该程序代码提供目标系统的执行环境。在本发明的一个实施例中,上述目标系统是 CELL 多核系统,也就是说,上述要进行并行性分析的程序代码是为 CELL 系统设计的。此外,在本发明的一个实施例中,上述的所模拟的执行环境包括模拟存储器、存储器管理单元、流水线以及模拟寄存器等对于程序代码的执行来说最基本的系统组件。但是,本发明并不限于此,只要能够确保程序代码的顺序执行并且所模拟的是目标系统的环境,该执行环境可以包括任何其他的系统组件。此外,对于上述模拟器,将在后面结合图 7-10 进行详细说明。

[0029] 接着,在步骤 210,将上述程序代码加载到所模拟的目标系统的执行环境中。具体地,在该步骤中,将上述程序代码加载到该模拟执行环境中的模拟存储器中,并且分析该程序代码中的符号表,以获得该程序代码中的各函数的地址。由于该符号表中记录着该程序代码中的各函数的名称、大小和存储地址,所以可以根据各函数的名称来获得其相应的地址。

[0030] 在步骤 215,获取上述程序代码中当前将要执行的指令并执行。具体地,由于所模拟的执行环境中的模拟指令寄存器的值指示着当前将要执行的指令的虚拟地址,所以在该步骤中,首先获取模拟指令寄存器中的虚拟地址,并将其传送给存储器管理单元,以便将该虚拟地址转换为物理地址;然后,根据所得到的物理地址,从模拟存储器的相应位置处获取将要执行的指令,将其解码为二进制格式,并执行。同时,模拟指令寄存器自动指向下一将要执行的指令。

[0031] 接着,下面的步骤 220-255 是跟踪程序代码的执行过程、记录执行信息的过程。

[0032] 在步骤 220,判断上述当前指令是否为诸如 call(x86) 或 bl(PPC) 的函数调用 / 返回指令。如果是,则该过程前进到步骤 225;否则,转到步骤 250。

[0033] 在步骤 225,对于在步骤 220 中被确定为函数调用 / 返回指令的当前指令,进一步判断其所调用的函数是否为系统 API (Application Programming Interface, 应用程序编程接口), 诸如 C 库函数。如果是,则该过程前进到步骤 230;否则,转到步骤 245。

[0034] 在步骤 230,对于在步骤 220 中被确定为函数调用 / 返回指令的当前指令,进一步判断其所调用的函数是否为存储器分配指令或释放指令,即是否为指令 malloc 或 free。如果是,则该过程前进到步骤 235;否则,转到步骤 240。

[0035] 接着,由于在步骤 230 中确定当前指令所调用的函数是系统 API 中的 malloc 或 free,所以在步骤 235,将该当前指令执行时的有关存储器分配 / 释放信息记录到存储器分配日志中。具体地,在该步骤中,将执行该当前指令时的系统周期(时间信息)、所分配的存储器大小、存储器地址记录到该存储器分配日志中。然后,该过程前进到步骤 240,执行该指令所调用的系统 API。

[0036] 接着,由于在步骤 225 中确定当前指令是函数调用 / 返回指令且其所调用的函数不是系统 API,所以在步骤 245,将该当前指令执行时的有关函数调用信息记录到函数调用日志中。具体地,在该步骤中,将执行该当前指令时的系统周期、该指令所调用的函数的标识记录到该函数调用日志中。为了便于用户识别,优选地,该函数的标识采用该函数的名称。但是,由于在指令的执行过程中其所调用的函数是以地址来指示的,所以在记录日志之前,首先要根据上面提到的该程序代码中的符号表,将所调用的函数的地址转换为相应的函数名称,然后再将该函数名称记录到日志中。

[0037] 接着,在步骤 250,对于在步骤 220 中被确定为非函数调用 / 返回指令的当前指令,进一步判断其是否为加载 / 存储指令。如果是,则该过程前进到步骤 255;否则,转到步骤 260,判断是否存在下一指令。

[0038] 接着,由于在步骤 250 中确定当前指令是加载 / 存储指令,所以在步骤 255,将该当前指令执行时的有关存储器访问信息记录到存储器访问日志中。具体地,在该步骤中,将执行该当前指令时的系统周期、该指令所访问的数据地址、数据大小以及读 / 写类型记录到该存储器访问日志中。然后,该过程前进到步骤 260,判断是否存在下一指令。

[0039] 在步骤 260,若存在下一指令,则返回到步骤 215,继续获取并执行下一指令。若不存在下一指令,则前进到步骤 265。

[0040] 在步骤 265,对上述步骤 215–260 所生成的日志进行预处理。具体地,在该步骤中,分析上述程序代码中的各函数的存储器访问操作或存储器分配操作的本地性或非本地性,并相应地设置上述日志中与该存储器访问操作或存储器分配操作对应的记录中的存储器类型项。举例来说,变量 i 是函数 A 中的局部变量,则对于函数 A 中针对变量 i 的存储器访问操作,在日志中将与该存储器访问操作对应的记录中的存储器类型项设置为本地,以表明该存储器访问操作不依赖于其他函数中的存储器操作,可以本地进行。

[0041] 下面,对于图 11–13 中所示出的情况进行说明。

[0042] 在图 11 中,从表面来看,函数 a() 和 b() 在栈 (i 和 j) 和堆 (p 和 q) 上可能会存在存储器操作的重叠部分,但实际上,根据这两个函数的主要代码部分可以分析出,其并没有冲突的存储器操作部分,是可以分配给不同的处理器核并在各处理器核的本地存储器的基础上执行的。所以,在这样的情况下,将日志中与这两个函数中的栈 (i,j) 和堆 (p,q) 操作相应的记录中的存储器类型设置为本地,以表明该堆、栈操作与其他函数中针对相同变量的堆、栈操作并不存在依赖关系,没有冲突,可以本地进行。

[0043] 再者,在图 12 和 13 中,分别地,可以看出,函数 a() 和 b() 均使用相同的锁 i 来保护其关键部分,但从这两个函数的主要代码部分可以看出,其并没有对相同的存储器变量进行操作,从而不存在冲突的存储器操作。所以,在这样的情况下,将日志中与这两个函数的锁操作相应的记录中的存储器类型设置为本地,以表明该锁操作并不与其他函数中针对相同变量的锁操作相冲突,可以本地进行。

[0044] 下面,举一个具体例子来说明利用图 2 的过程所得到的跟踪结果。图 3(a) 示出了两段简单的程序代码,其中的 main() 是主函数,Add() 是被主函数所调用的子函数。图 3(b) 示出了记录有图 3(a) 中的这两段程序代码的执行信息的日志。可以看出,该日志按时戳(执行周期)的顺序依次记录了这两段程序代码在执行过程中的所有存储器访问信息、存储器分配信息、函数调用信息等。并且,该日志中的各个字段的具体含义已在图 3(b) 中示出。

[0045] 以上,就是对图 2 的执行过程跟踪步骤的详细描述。需要说明的是,在图 2 所示的过程中,虽然将步骤 220–250 的记录执行信息的过程安排在步骤 215 的指令执行之后,但这只是为了便于说明而以这样的顺序来进行描述的,其并非是限定性的。在具体实现中,也可以在步骤 215 的指令执行的同时执行步骤 220–250。

[0046] 此外,还需要说明的是,在上面图 2 所示的过程中,虽然是根据指令的类型将程序代码的执行信息分类记录到不同的日志中的,但是,在实际实现中,也可以不分别设置函数调用日志、存储器分配日志以及存储器访问日志等这多种日志,而是仅利用一个日志来记录该程序代码的所有执行信息,并在进行并行性的分析时在该日志的基础上得到上述几类日志。并且,在具体实现中,无论是利用一个日志还是利用多个日志来记录执行信息,所记录的信息种类并不仅限于上述的函数名称、存储器地址、存储器类型等,而是,除了上述的信息类型之外,还可以记录其他更加详细的、与程序代码的执行有关的信息。

[0047] 下面,结合图 4 详细描述上面图 1 的方法中的并行性分析步骤 110。图 4 是该并行性分析步骤 110 的详细流程图。

[0048] 如图 4 所示,首先,在步骤 405,为当前进行并行性分析的程序代码生成调用树。具体地,在该步骤中,以该程序代码中的主函数作为根节点,以该程序代码在顺序执行过程中的各函数调用作为根节点之下的各节点,并以各函数调用之间的进一步调用关系作为相应的节点之间的父子关系,来生成该程序代码的调用树。如上面结合图 2 所说明的,与该程序代码的每一次非系统 API 的函数调用有关的信息都被记录在了函数调用日志中,所以在该步骤中,可以根据上面图 2 的过程所生成的日志来识别出该程序代码在执行过程中的各函数调用。该步骤所生成的调用树中的每一个节点都代表该程序代码在执行过程中的一次函数调用,并且每一个节点都包括下列各项:

[0049] 函数标识 func_id,其是该节点所对应的函数调用中被调用函数的标识。在本发明的一个实施例中,该标识是被调用函数的名称。

[0050] 调用函数列表 callee_list,其是到达该节点所对应的函数调用时的各级调用函数的列表,也就是说,该列表表示的是到达该函数调用时的堆栈状态。在本发明的一个实施例中,该列表是由各级调用函数的名称组成的。举例来说,如果主函数 main() 调用函数 tree(),而函数 tree() 在执行期间又调用了函数 node(),则对于函数 node() 的调用来说,其所对应的节点的调用函数列表 call_list 为 (main, tree)。

[0051] 调用号 call_number,其用于区别具有相同的函数标识和调用函数列表的不同的函数调用,也就是说,其用于区别同一函数的不同次调用。该调用号可以用数字来表示,也可以用其他标识来表示。

[0052] 以上各项均需通过分析图 2 的过程所生成的函数调用日志来获得。

[0053] 接着,在步骤 410,根据上面图 2 的过程所记录的执行信息日志,为上述调用树中的各节点获得其所对应的函数调用的读访问记录集合和写访问记录集合。也就是说,上述调用树中的各节点还包括下列各项:

[0054] 读访问记录集合 read_set,其是在该节点所对应的函数调用期间对存储器进行的读访问的记录的集合。

[0055] 写访问记录集合 write_set,其是在该节点所对应的函数调用期间对存储器进行的写访问的记录的集合。

[0056] 由于该程序代码的与每一次非系统 API 的函数调用及存储器访问有关的信息都被记录在了日志中,所以在该步骤中,根据该日志来识别出该程序代码在执行过程中的各非系统 API 的函数调用,并分别统计出该各函数调用期间的存储器读访问记录和写访问记录,以分别作为该函数调用所对应的节点的读访问记录集合和写访问记录集合。

[0057] 在步骤 415,对上述调用树进行优化。具体地,在该步骤中,假设节点 A 和 B 是该调用树中同一父节点下的两个叶子节点,如果:

[0058] 1)A 的读访问记录集合 read_set 中的记录 RA 与 B 的写访问记录集合 write_set 中的记录 R_B 涉及相同的非本地存储器地址 p;或

[0059] 2)A 的写访问记录集合 write_set 中的记录 R_A 与 B 的读访问记录集合 read_set 中的记录 R_B 涉及相同的非本地存储器地址 p;或

[0060] 3)A 的写访问记录集合 write_set 中的记录 R_A 与 B 的写访问记录集合 write_set 中的记录 R_B 涉及相同的非本地存储器地址 p;

[0061] 则将节点 A 和 B 合并为一个新的节点,以表示 A、B 的相互依赖性。其中,对于相同

的非本地存储器地址的判断,是根据各读、写访问记录中的存储器地址信息和存储器类型信息来进行的。

[0062] 也就是说,在本步骤中,对于上述程序代码中同一函数下的各函数调用,根据其读、写访问记录集合中各记录的存储器地址和存储器类型,判断其中一个函数调用的写访问记录集合与另一个函数调用的读或写访问记录集合是否存在关于同一个非本地存储器地址的记录,若存在,则确定这些函数调用之间具有依赖关系、不可并行执行;否则,确定这些函数调用间不具有依赖关系、可并行执行。

[0063] 并且,重复步骤 415,直到该调用树中不再存在能够进一步合并的上述那样的兄弟节点为止。

[0064] 下面,举一个具体例子来说明图 4 的过程。图 5(a) 示出了两段简单的程序代码,其中的 main() 是主函数,Add() 是被主函数所调用的子函数。图 5(b) 示出了记录有图 5(a) 中的这两段程序代码的执行信息的日志。可以看出,该日志按时戳(执行周期)的顺序依次记录了这两段程序代码在执行过程中的所有存储器访问信息、函数调用信息等。并且,由于这两段程序代码中不存在存储器分配 / 释放指令 malloc 和 free,所以没有记录与存储器分配 / 释放有关的信息。

[0065] 以上述图 5(a) 中的程序代码及图 5(b) 中相应的执行信息日志为例,利用上述图 4 的过程,可得到图 5(c) 所示的该程序代码的最终调用树的一部分。其中,节点 (add,main,0,read_set,write_set) 与节点 (add,main,1,read_set,write_set) 不能够合并,从而表示函数 add() 被主函数 main() 的第 1 次调用和第 2 次调用没有依赖关系、能够并行执行。

[0066] 从而,在上面图 1 的步骤 115 中便可向用户呈现利用该过程最终生成的调用树,以利用调用树中各节点之间的关系来表现程序代码中的各函数调用之间的依赖关系,使用户获得对于该程序代码的并行性的直观认识。

[0067] 此外,需要说明的是,上面图 4 的并行性分析过程所采用的树形分析方式仅是本发明的一个实施例,其并非要对本发明进行限制。在其他实施例中,也可以使用列表、文本等形式来进行程序代码的各函数调用之间的依赖关系的分析及呈现。

[0068] 返回到图 4,该并行性分析过程还可包括可选步骤 420。

[0069] 在可选步骤 420,对上述程序代码在其执行过程中的各次函数调用进行代码大小和数据大小的分析。具体地,该步骤根据下列算式来实现:

[0070] $\text{self_code_size}(A) = \text{the own codesize of function } A$

[0071]

$$\text{code_size}(\text{call}A) = \text{self_code_size}(A) + \sum_{\text{A 所调用的函数 } f} (\text{self_code_size}(f))$$

[0072]

$$\text{data_size}(\text{call}A) = \sum_{\text{A 的存储器访问 } i} (\text{sizeof}(i))$$

[0073] 也就是说,在该步骤中,对于上述程序代码在执行过程中的各次函数调用 callA,将其所对应的被调用函数 A 的本身代码大小与该函数调用期间进一步调用的各函数 f 的代码大小之和作为该函数调用 callA 的代码大小,并将该函数调用 callA 在生命周期中进行各次存储器访问 i 时所访问的空间大小之和作为该函数调用 callA 的数据大小。

[0074] 上面的步骤 420,对于当前进行并行性分析的程序代码是要应用于 CELL 那样每一

个处理器核都具有本地存储器限制(256KB)的多核体系结构的情况而言,是必须的。因为,在这样的系统中,每一个处理器核所能够承受的任务的大小(代码及数据大小)是受其本地存储器的限制的,所以在对各个处理器核进行任务的分配时,必须首先考虑该任务的大小是否适合于该处理器核。但是,对于处理器核并没有本地存储器大小限制的多核体系结构而言,在对其程序代码进行并行性的分析时,则无需考虑所分配的任务的大小是否适合于其处理器核,从而无需执行上面的步骤420。因而,步骤420在图4中是作为可选步骤、依情况来执行的。

[0075] 此外,在执行步骤420的情况下,在上面图1的步骤115中还要向用户呈现该步骤中所获得的各函数调用的代码大小和数据大小。各函数调用的代码大小和数据大小可以随调用树中的相应节点一起呈现。

[0076] 以上,就是对图4的程序代码的并行性分析过程的详细描述。需要说明的是,在上面图4所示的过程中,虽然将步骤410的读访问记录集合和写访问记录集合获取步骤安排在步骤405的调用树生成步骤之后,但这只是为了便于说明而以这样的顺序来进行描述的,其并非是限定性的。在具体实现中,也可以在步骤405的调用树生成的同时执行步骤410。

[0077] 以上,就是对本实施例的程序代码的并行性的分析方法的描述。从以上描述可知,本实施例首先通过模拟程序代码的顺序执行来收集对于该程序代码的并行性分析来说所需的执行信息,然后根据所收集的执行信息来进行各函数调用之间的依赖关系的分析,最终以直观的形式向用户呈现该分析结果。

[0078] 因而,在本实施例中,由于模拟程序代码的实际执行,所以能够收集到内核级的存储器访问信息以及由例如输入输出控制的系统调用所引入的潜在的依赖关系,从而在此基础上进行的并行性分析更加准确,能够帮助程序设计者更有效地针对多核体系结构进行程序代码的并行任务的划分。并且,利用本实施例,能够使程序代码的并行性分析独立于目标系统的硬件和操作系统。

[0079] 此外,以上虽然是针对于CELL系统的情况来描述本发明如何进行程序代码的并行性分析的,但是,并不限于此,本发明同样可应用于诸如Simple Scalar、Power系统芯片等其他多核系统的程序代码的并行性分析。

[0080] 在同一发明构思下,图6是示出根据本发明实施例的程序代码的并行性的分析系统的方框图。

[0081] 如图6所示,本实施例的程序代码的并行性的分析系统60包括:模拟器61、并行性分析单元62和呈现单元63。

[0082] 其中,模拟器61,用于跟踪要进行并行性分析的程序代码的执行过程,以收集其执行信息。

[0083] 图7是根据本发明实施例的用于跟踪程序代码的执行过程的模拟器的方框图。如图7所示,本实施例的模拟器61包括:模拟存储器611、存储器管理单元612、流水线613、模拟寄存器614、执行过程跟踪单元615和跟踪预处理单元616。

[0084] 其中,模拟存储器611,是从主系统(当前系统)的存储器中分配出的、用作模拟器61所模拟的目标系统环境中的物理存储器的存储器块。

[0085] 存储器管理单元(MMU)612,用于在程序代码的存储器访问期间将虚拟地址转换为

物理地址。该存储器管理单元 612 包含一转换后备缓冲器 TLB，其是高速缓存，并且存储有虚拟地址与物理地址之间的映射关系。该存储器管理单元 612 进行地址转换的过程是：搜索该转换后备缓冲器 TLB，找到匹配的虚拟地址与物理地址对，从而得到所需的物理地址。

[0086] 流水线 613，其包括指令取单元 6131、指令解码单元 6132 和指令执行单元 6133。指令取单元 6131 将多条指令从模拟存储器 611 中取到该流水线 613 上，指令解码单元 6132 分析每一条指令的二进制格式，而指令执行单元 6133 则执行每一条指令的动作。

[0087] 模拟寄存器 614，用作模拟器 61 所模拟的目标系统的寄存器，用于存储中间计算结果。其中的 PC 寄存器记录当前要执行的指令的位置（虚拟地址）。

[0088] 执行过程跟踪单元 615，用于跟踪程序代码在模拟器 61 中的顺序执行过程，以获得该程序代码的执行信息。具体地，其判断在该模拟器 61 中执行的当前指令的类型：如果该指令是调用非系统 API 的函数调用 / 返回指令，则将当前执行周期以及被调用函数的标识记入日志；如果该指令是调用系统 API 中的存储器分配或释放函数的函数调用 / 返回指令，则将当前执行周期以及被分配的存储器大小、存储器地址记入日志；以及如果该指令是加载 / 存储指令，将当前执行周期以及数据地址、数据大小、读 / 写类型和存储器类型记入日志。

[0089] 跟踪预处理单元 616，用于分析上述日志中所记录的各存储器访问操作和存储器分配操作的相应存储器类型，并将该存储器类型信息记入日志。

[0090] 下面，结合图 8-10 说明模拟器 61 的工作过程。

[0091] 如图 8 所示，在模拟器 61 启动后，指令取单元 6131 读取 PC 寄存器中的值，以获得下一条指令的地址，并将该地址传送给存储器管理单元 612 以进行虚拟 - 物理地址的转换。然后，指令取单元 6131 根据所获得的物理地址从模拟存储器 611 的相应位置处读取指令，同时 PC 寄存器的值自动指向下一指令。

[0092] 如图 9 所示，在指令执行单元 6133 执行一条加载 / 存储指令时，首先从该指令本身或寄存器中获取数据地址，并将该地址传送给存储器管理单元 612 以进行虚拟 - 物理地址的转换。然后，指令执行单元 6133 根据所获得的物理地址将数据加载或存储到模拟存储器 611 的相应位置处。与此同时，执行过程跟踪单元 615 记录相应的存储器访问信息。

[0093] 如图 10 所示，在指令执行单元 6133 执行一条分支指令时，首先将 PC 寄存器中的值更新为该分支指令的目标地址，并丢弃掉已取到流水线 613 上的所有指令。然后，指令取单元 6131 根据 PC 寄存器中的新值将相应指令取到流水线 613 上。与此同时，执行过程跟踪单元 615 记录相应的函数调用信息。

[0094] 以上，就是对本实施例的用于跟踪程序代码的执行过程的模拟器的详细描述。利用本实施例的模拟器，能够获得对于程序代码的并行性分析来说详细、有用的执行信息。

[0095] 返回到图 6，并行性分析单元 62，用于根据模拟器 61 对程序代码的执行过程的跟踪结果，获得该跟踪结果所记录的该程序代码在执行过程中的各执行动作及其相应的时间信息和存储器类型信息，来进行并行性的分析。如图 7 所示，该并行性分析单元 62 进一步包括：调用树构造单元 621、访问记录获得单元 622、调用树优化单元 623 和代码 / 数据大小分析单元 624。

[0096] 其中，调用树构造单元 621，用于根据模拟器 61 所获得的跟踪结果，以上述程序代码在执行过程中的各次函数调用作为节点、以函数调用之间的关系作为节点之间的关系，

生成该程序代码的调用树。

[0097] 访问记录获得单元 622,用于根据模拟器 61 所获得的跟踪结果,为上述调用树构造单元 621 所生成的调用树中的各节点获得其所对应的函数调用期间的读访问记录集合和写访问记录集合。

[0098] 调用树优化单元 623,用于对于上述调用树中同一父节点下的各叶子节点,根据其读访问记录集合和写访问记录集合确定其之间是否存在针对同一非本地存储器地址的访问冲突,并且将存在存储器访问冲突的各叶子节点进一步合并为一个节点。

[0099] 代码 / 数据大小分析单元 624,用于对于上述程序代码在执行过程中的各次函数调用,将其所对应的被调用函数的本身代码大小与该函数调用期间进一步调用的各函数的代码大小之和作为该函数调用的代码大小,并将其在生命周期中进行各次存储器访问时所访问的存储器空间大小之和作为该函数调用的数据大小。

[0100] 接着,呈现单元 63,用于向用户呈现对上述程序代码的并行性的分析结果。该分析结果可包括上述程序代码的各函数调用之间的依赖关系、代码大小和数据大小的图形、文字等的表示。

[0101] 以上,就是对本实施例的程序代码的并行性的分析系统的描述。从以上描述可知,本实施例首先利用模拟器来收集对于该程序代码的并行性分析来说所需的执行信息,然后根据所收集的执行信息来进行各函数调用之间的依赖关系的分析,最终以直观的形式向用户呈现该分析结果。

[0102] 因而,在本实施例中,由于利用模拟器模拟程序代码的实际执行,所以能够收集到内核级的存储器访问信息以及由例如输入输出控制的系统调用所引入的潜在的依赖关系,从而在此基础上进行的并行性分析更加准确,能够帮助程序设计者更有效地针对多核体系结构进行程序代码的并行任务的划分。并且,利用本实施例,能够使程序代码的并行性分析独立于目标系统的硬件和操作系统。

[0103] 本实施例的程序代码的并行性的分析系统及其各个组成部分可以由诸如超大规模集成电路或门阵列、诸如逻辑芯片、晶体管等的半导体、或者诸如现场可编程门阵列、可编程逻辑设备等的可编程硬件设备的硬件电路实现,也可以用由各种类型的处理器执行相应的软件的方式实现,也可以由上述硬件电路和软件的结合实现。并且这些各个组成部分也可以物理上集中在一起实施,也可以物理上相互独立而操作上互相协作。

[0104] 以上虽然通过一些示例性的实施例对本发明的程序代码的并行性的分析方法和系统进行了详细的描述,但是以上这些实施例并不是穷举的,本领域技术人员可以在本发明的精神和范围内实现各种变化和修改。因此,本发明并不限于这些实施例,本发明的范围仅以所附权利要求为准。

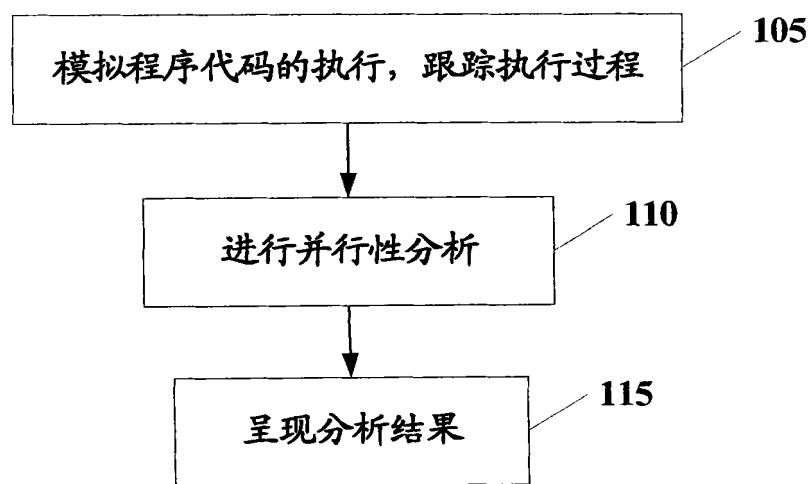


图 1

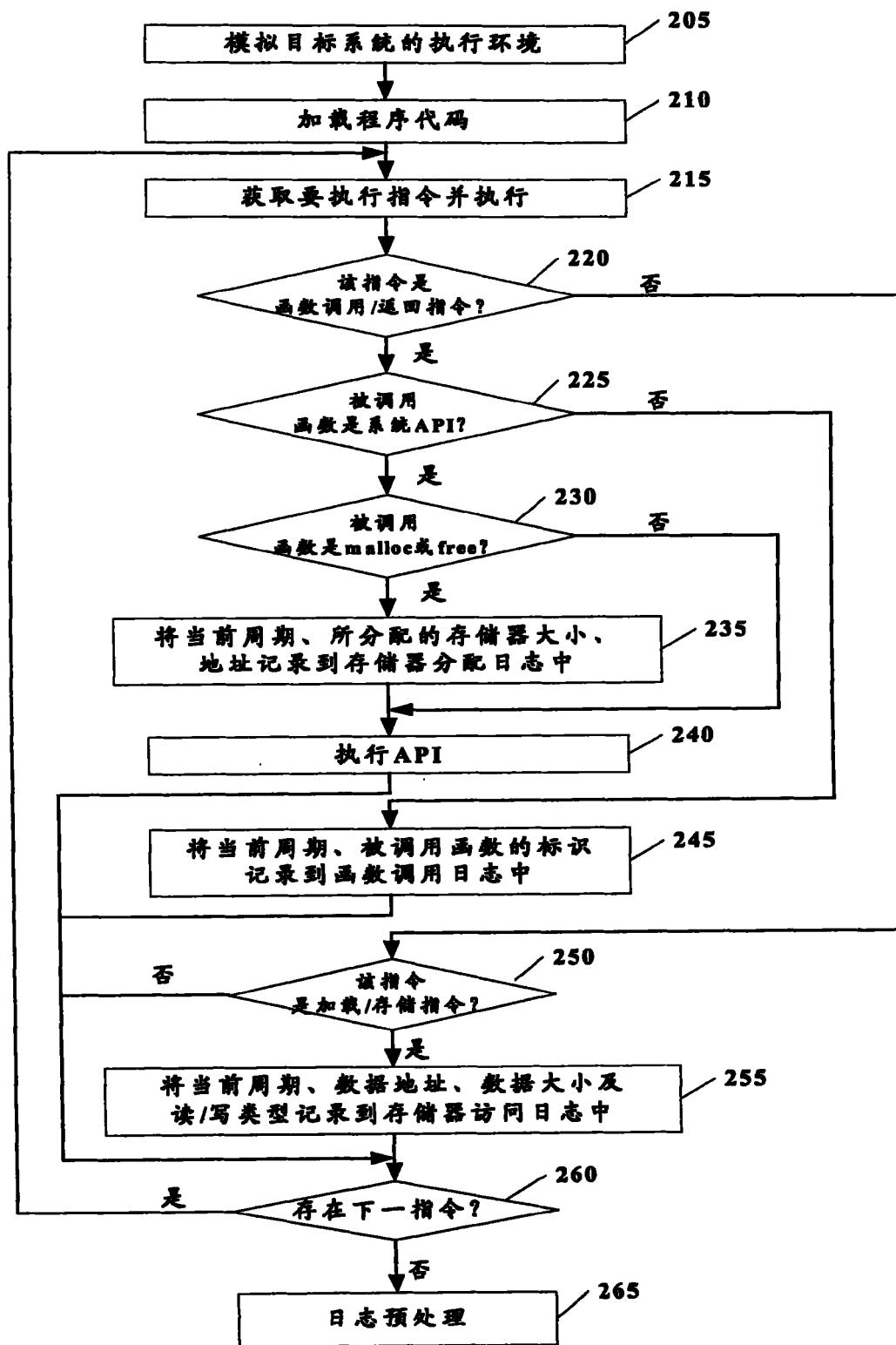


图 2

```
00001: #include <stdlib.h>
00002:
00003: #define MAX_LEN 4
00004: #define STEP 2
00005:
00006: int Add (int *a, int *b, int offset, int len)
00007: {
00008:     int i = 0;
00009:     int * p = malloc(len);
00010:
00011:     for (i = offset; i < offset + len; i++)
00012:     {
00013:         a[i] = a[i] + b[i];
00014:     }
00015:     free(p);
00016:     return 0;
00017: }
00018:
00019: int main()
00020: {
00021:     int i = 0, a_array[MAX_LEN], b_array[MAX_LEN];
00022:
00023:     for (i = 0; i < MAX_LEN; i += STEP)
00024:     {
00025:         Add(a_array, b_array, i, STEP);
00026:     }
00027:     return 0;
00028: }
```

图 3 (a)

时戳	函数名称	行 #	变量名	存储器地址	存储器类型	访问类型	调用信息
1	main	21	i	0x12fed4	local	WRITE	main() begins
2	Add	08	i	0x130ed8	local	WRITE	Add() begins
3	Add	09	p	0x218890	local	WRITE	malloc() called
4	Add	13	a[i]	0x12febC	non-local	READ	
5	Add	13	b[i]	0x12fea4	non-local	READ	
6	Add	13	a[i]	0x12febC	non-local	WRITE	
7	Add	13	a[i]	0x12fec0	non-local	READ	
8	Add	13	b[i]	0x12fea8	non-local	READ	
9	Add	13	a[i]	0x12fec0	non-local	WRITE	
10	Add	15	p	0x218890	local	WRITE	free() called, Add() ends
11	main	23	i	0x12fed4	local	READ	
12	main	23	i	0x12fed4	local	WRITE	
13	Add	08	i	0x130ed8	local	WRITE	Add() begins
14	Add	09	p	0x218890	local	WRITE	malloc() called
15	Add	13	a[i]	0x12fec4	non-local	READ	
16	Add	13	b[i]	0x12feac	non-local	READ	
17	Add	13	a[i]	0x12fec4	non-local	WRITE	
18	Add	13	a[i]	0x12fec8	non-local	READ	
19	Add	13	b[i]	0x12feb0	non-local	READ	
20	Add	13	a[i]	0x12fec8	non-local	WRITE	
21	Add	15	p	0x218890	local	WRITE	free() called, Add() ends
22	main	23	i	0x12fed4	local	READ	
23	main	23	i	0x12fed4	local	WRITE	main() ends

图 3 (b)

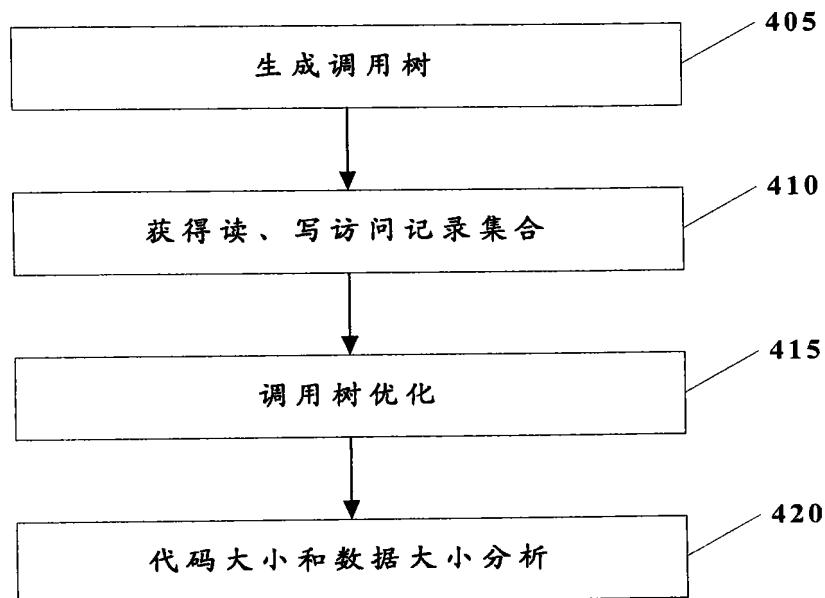


图 4

```

int Add( int* a, int* b, int offset, int len )
{
    int i = 0;
    for( i = offset; i < offset + len; i ++ )
    {
        a[i] = a[i] + b[i];
    }
    return 0;
}

int main()
{
    int i = 0;
    int i = 0, a_array[MAX_LEN], b_array[MAX_LEN];
    for ( i = 0; i < MAX_LEN; i += STEP )
    {
        Add( a_array, b_array, i, STEP );
    }
    return 0;
}

```

图 5(a)

时戳	函数名称	行 #	变量名	存储器地址	存储器类型	访问类型	调用信息
1	main	38	i	0x12fed4	local	WRITE	main() begins
2	Add	20	a[i]	0x12feb0	non-local	READ	Add() begins
3	Add	21	b[i]	0x12fea4	non-local	READ	
4	Add	22	a[i]	0x12feb0	non-local	WRITE	
5	Add	20	a[i]	0x12fec0	non-local	READ	
6	Add	21	b[i]	0x12fea8	non-local	READ	
7	Add	22	a[i]	0x12fec0	non-local	WRITE	Add() ends
8	main	42	i	0x12fed4	local	READ	
9	main	43	i	0x12fed4	local	WRITE	
10	Add	20	a[i]	0x12fec4	non-local	READ	Add() begins
11	Add	21	b[i]	0x12fec0	non-local	READ	
12	Add	22	a[i]	0x12fec4	non-local	WRITE	
13	Add	20	a[i]	0x12fec8	non-local	READ	
14	Add	21	b[i]	0x12feb0	non-local	READ	
15	Add	22	a[i]	0x12fec8	non-local	WRITE	Add() ends
16	main	42	i	0x12fed4	local	READ	
17	main	43	i	0x12fed4	local	WRITE	main() ends

图 5 (b)

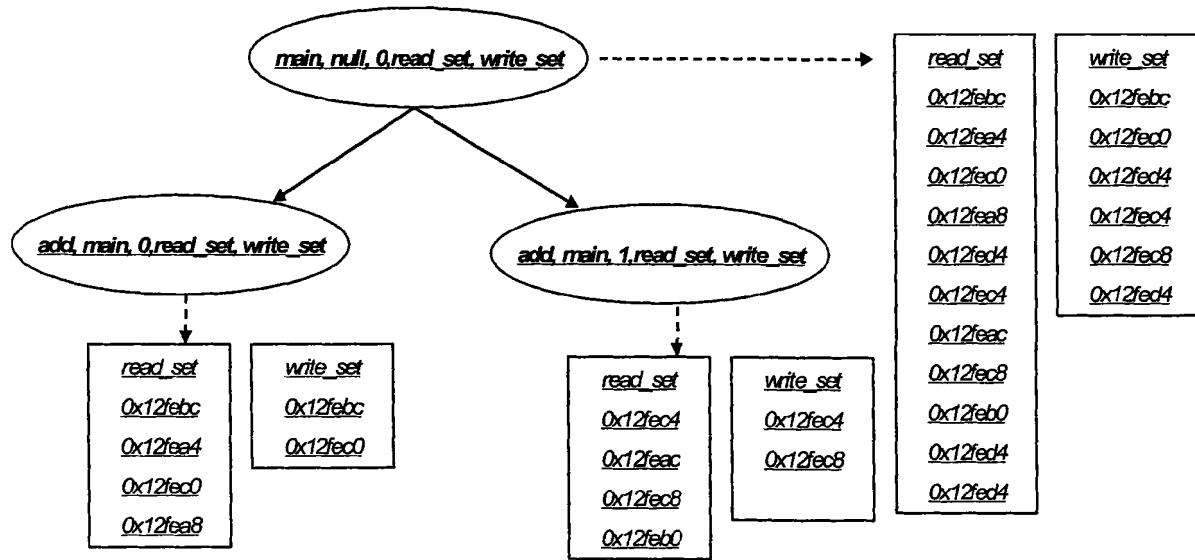


图 5 (c)

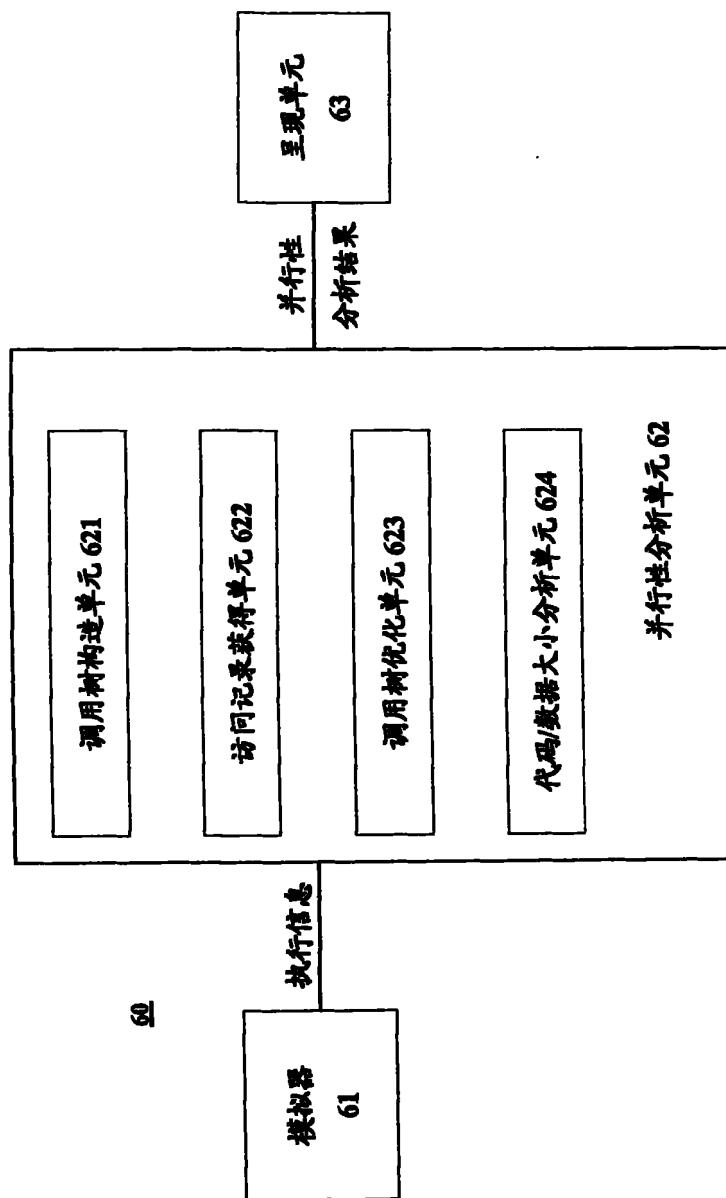


图6

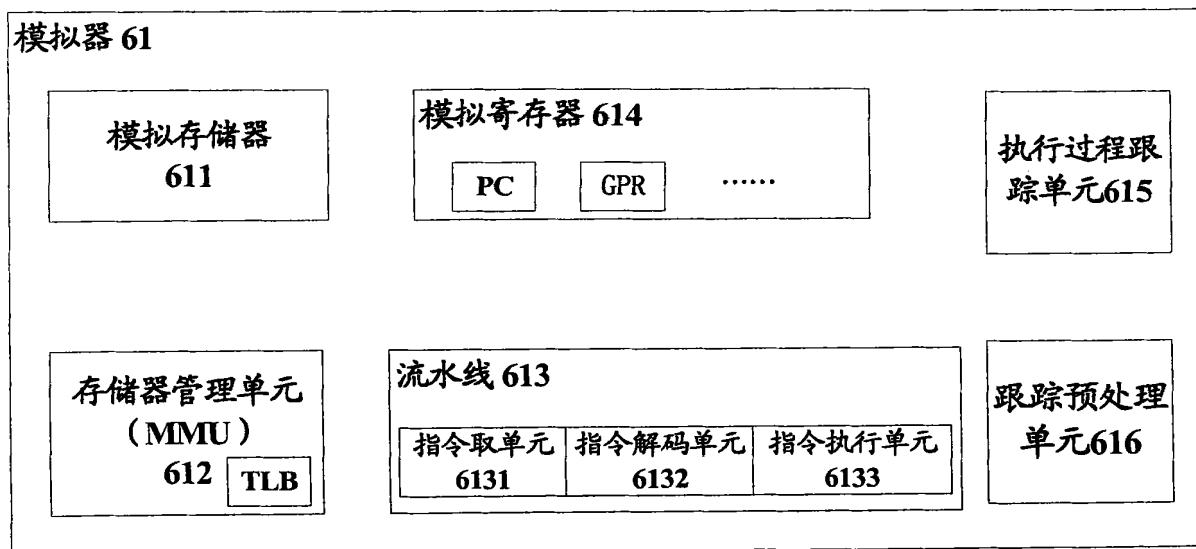


图 7

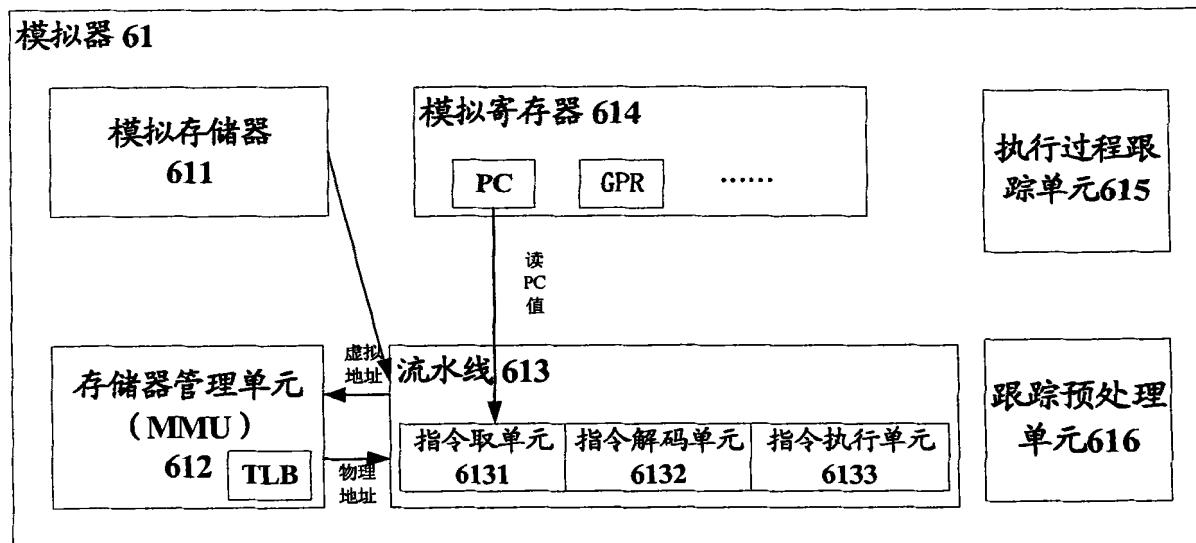


图 8

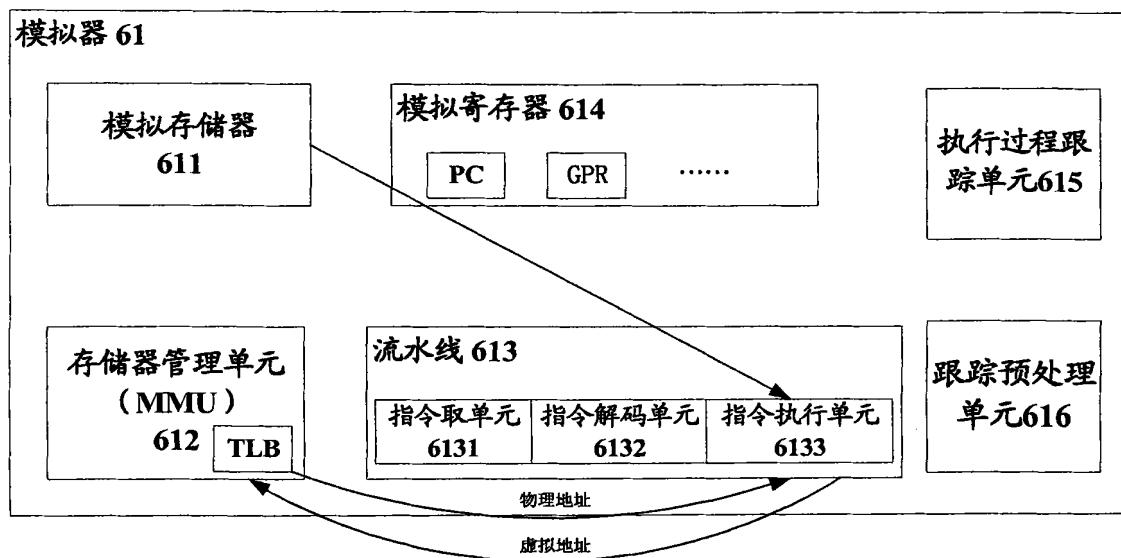


图 9

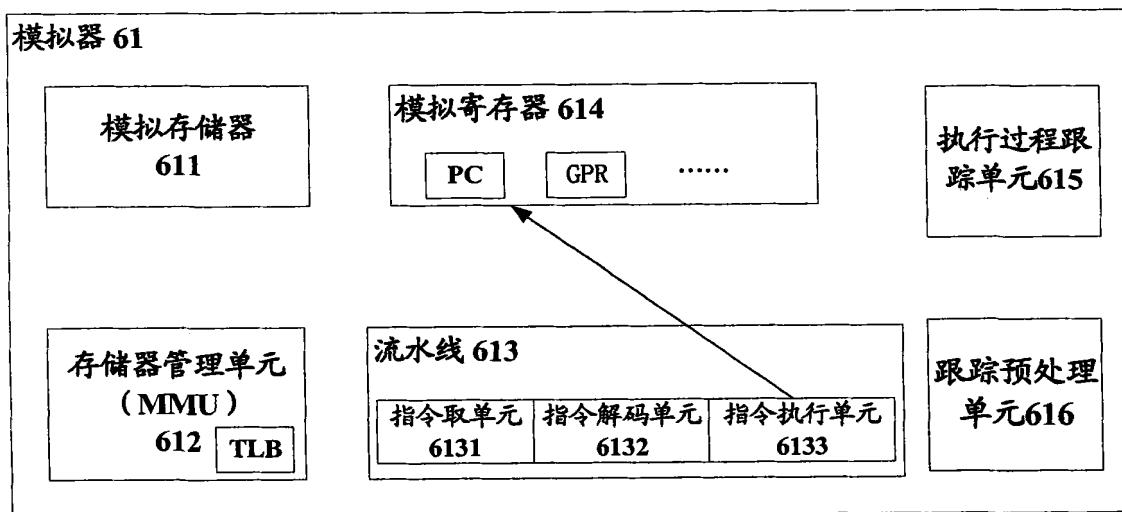


图 10

A0 { a(); b(); }	a0 { int i=10; char * p = malloc(i); free(p); }	b0 { int j=8; char * q = malloc(j); free(q); }
---------------------------	--	---

图 11

a0 { lock(&i); int j; j++; unlock(&i); }	b0 { lock(&i); int k; k++; unlock(&i); }
---	---

图 12

```
mutex i;
ITEM data[ARRAY_SIZE];
a()
{
    lock(&i);
    for (int j=0;j<ARRAY_SIZE;j+=2)
    {
        .
        data[j] = ...
    }
    unlock(&i);
}
b()
{
    lock(&i);
    for (int j=1;j<ARRAY_SIZE;j+=2)
    {
        .
        data[j] = ...
    }
    unlock(&i);
}
```

图 13