软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(10):2432−2459 [doi: 10.3724/SP.J.1001.2013.04353]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种数据结构制导的线程划分方法与执行模型*

杜延宁[1], 赵银亮[1], 韩 博[1], 李远成[2]

[1](西安交通大学 电子与信息工程学院,陕西 西安 710049)

[2](西安科技大学 计算机科学与技术学院,陕西 西安 710054)

通讯作者: 赵银亮, E-mail: zhaoy@mail.xjtu.edu.cn

摘 要: 在对程序进行并行化时,为了保证结果的正确性,并行编译器只能采取一种保守的策略,也就是,如果它不能确定两段代码在并行执行时是否会发生冲突,它就不允许这两段代码并行执行.虽然这种做法保证了正确性,但同时也限制了对并行性的开发.在这种背景下,许多推测多线程方法被提了出来,这些方法通过允许可能冲突的代码段并行执行来把握更多的并行机会,同时,通过从冲突中恢复来保证结果的正确性.然而,传统推测多线程方法所使用的"沿控制流将串行程序划分为多个线程"的做法并不适合不同数据结构上的操作在控制流中相互交错的情况,因为如果沿控制流将程序线性地划分为多个线程,则同一个数据结构上的操作将被分到不同的线程中,从而非常容易发生冲突.为了有效地对这些程序进行并行化,提出了一种基于数据结构的线程划分方法与执行模型.在这种方法中,程序中的对象被划分成多个组,同一组中对象上的操作被分派到同一个线程中去执行,从而降低了在同一个数据结构上发生冲突的可能性.

关键词: 推测多线程;并行化;数据结构;划分方法;执行模型

中图法分类号: TP314 **文献标识码**: A

中文引用格式: 杜延宁,赵银亮,韩博,李远成.一种数据结构制导的线程划分方法与执行模型.软件学报,2013,24(10): 2432−2459. http://www.jos.org.cn/1000-9825/4353.htm

英文引用格式: Du YN, Zhao YL, Han B, Li YC. Data structure directed thread partitioning method and execution model. Ruan Jian Xue Bao/Journal of Software, 2013,24(10):2432−2459 (in Chinese). http://www.jos.org.cn/1000-9825/4353.htm

Data Structure Directed Thread Partitioning Method and Execution Model

DU Yan-Ning[1], ZHAO Yin-Liang[1], HAN Bo[1], LI Yuan-Cheng[2]

[1](School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China)

[2](School of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an 710054, China)

Corresponding author: ZHAO Yin-Liang, E-mail: zhaoy@mail.xjtu.edu.cn

Abstract: While parallelizing a program, to ensure the correctness of the result, a parallel compiler can only adopt a conservative policy that if it cannot accurately determine whether two pieces of code will conflict while being executed in parallel, it does not permit them to be executed in parallel. Although this approach can ensure correctness, it also limits the amount of parallelism that can be exploited. A number of speculative multithreading (SpMT) approaches gained wide popularity because they can get more parallelism by allowing two pieces of potentially conflicting code to execute in parallel while ensuring the correctness of the results by recovering from conflicts. However, the dividing-along-the-control-flow-path method employed by the traditional SpMT approach is unsuitable for a number of programs in which operations on different data structures are interleaved in the control flow path. If the program is linearly divided into threads along the control flow path, the operations on the same data structure will be divided among the different threads that are doomed to be prone to conflict when being executed concurrently. To effectively parallelize these programs, this paper proposes a data structure

* 基金项目: 国家自然科学基金(61173040); 国家高技术研究发展计划(863)(2008AA01Z136)
收稿时间: 2012-03-12; 修改时间: 2012-09-29; 定稿时间: 2012-12-03

centric approach, in which the objects of a program are organized into different groups and the operations on the objects within the same group are dispatched to the same thread for execution, thereby reducing the likelihood of conflict on the same data structure.

Key words: speculative multithreading; parallelization; data structure; partitioning method; execution model

并行化,即将串行程序划分为多个并发线程,是加速串行程序执行的一种重要手段.为保证得到的并行程序的正确性,我们必须确保这些线程的并行执行不会违犯由程序的串行语义所确定的代码间的数据依赖关系.如果一个线程违犯了它与另一个线程之间的数据依赖关系,我们就说这两个线程发生了冲突.

为了避免冲突,需要我们能够对代码间的依赖关系进行准确的分析.然而,由于输入的不可预知性,准确的依赖分析几乎是不可能的.在这种情况下,编译器只能采取一种保守的策略,即在无法准确判断两段代码在并行时是否会发生冲突的情况下,就按会发生冲突来处理,也就是不让这两段代码并行执行.这种保守的做法虽然能够确保程序结果的正确性,但同时也限制了对程序并行性的开发.因为也存在这样的可能,即这两段只是"可能冲突"的代码在并行执行的时候并不真正发生冲突.在这种情况下,就相当于我们为了回避风险而放弃了并行执行的机会.

与这类避免冲突发生的保守方法相对的是另一类方法,即允许冲突产生,冲突产生之后再来解决冲突(即消除违犯数据依赖关系的线程所造成的影响).按照这种做法,即便两段代码在并行执行时有可能发生冲突,我们也让它们并行.系统会对冲突进行检测,如果幸运没有发生冲突的话,并行就算是成功了;但如果不幸发生了冲突,则违犯依赖关系的线程将被撤销,并不会对程序的正确性造成任何影响.通过这种方式,既保证了程序结果的正确性,也使得那些在保守方法下无法被利用的并行性得到了有效的开发.因为这个特点,这类方法常被称作推测多线程或投机多线程(speculative multithreading,简称 SpMT)[1−4].

按照传统观点,花费宝贵的处理器资源来执行极有可能被撤销的线程成本过于昂贵.但是随着多核处理器的迅猛发展,这种做法也逐渐开始显现其吸引力.根据 Berkeley 大学的预计[5],将在 2025 年前出现上千核的处理器.而一个典型的 Windows 系统中线程的数目大约在 1 000 左右,而且绝大部分处于睡眠状态,也就是说,届时,与系统中活动线程的数量相比,硬件核的数量可能会更多.这就意味着,如果没有足够多的线程来利用这些核,则其将处于闲置状态.随着摩尔定律的失效,核数量的增速会远高于单个核性能的提高速度[6].在这种背景下,如何利用多核的数量优势来加速程序的执行,已成为并行化领域的一个主要研究热点.

虽然在推测多线程方法中,冲突并不会妨害程序的正确性,但冲突一旦发生,就需撤销线程,频繁的撤销发起会对系统性能造成严重的影响,从而抵消并行所带来的好处.所以,如何生成不易冲突的线程是推测多线程研究中最为关键的问题之一.乍看之下,这个问题似乎与非推测框架下如何生成绝对不会冲突的线程是同一个问题,但二者之间其实存在着根本的差别:对于非推测方法来说,如何对线程进行同步以避免冲突,是划分时需要考虑的主要问题;而对于推测方法而言,划分则更像是一种在充分把握并行机会与尽量降低冲突发生后的开销之间的权衡.

两个线程间冲突的可能性与这两个线程共享数据的多少有关,所以,为了降低线程间冲突的可能性,在划分线程时就需要谨慎考虑,以使线程间的共享数据尽可能地少.在传统的推测多线程划分方法中,串行程序被沿着控制流划分成多个线程:a) 对于循环区域,一般是在各次迭代的边界处进行划分;b) 而对于非循环区域,在决定划分位置时,编译器则会选择一个能够使得前后两段代码之间的依赖下降到最低的位置.因为循环的各次迭代一般工作在不同的数据集之上,所以沿控制流进行划分的做法对于循环区域很有效.然而对于非循环区域,这种在控制流中进行划分的方法有时就难以奏效.如图 1 所示,对对象 *g* 的操作(*g.init* 和 *g.adjust*)与对对象 *p* 的操作(*p.prepare* 和 *p.layout*)在控制流中相互交错.如果我们按照图 2(a)所示进行划分,线程 *t*1 与线程 *t*2 就很可能在对象 *g* 上发生冲突,这是因为 *g.init* 和 *g.adjust* 都对对象 *g* 进行了操作;如果我们按照图 2(b)所示进行划分,对象 *g* 和对象 *p* 上都有可能发生冲突;如果按图 2(c)进行划分,则很可能在对象 *p* 上发生冲突.总之,无论在什么地方进行划分,产生的两个线程都难免冲突.

本文提出了对此问题的一种解决办法:即一种基于数据结构来进行线程划分的方法.如图 2(d)所示,如果我们能把 *g.init* 与 *g.adjust* 放在同一个线程中去顺序执行,而把 *p.prepare* 与 *p.layout* 放在另一个线程中去顺序执

行,然后让这两个线程并行,就可以避免两种方法在同一个对象上发生冲突的情况出现,从而获得较好的并行性.与直接在控制流中进行划分的做法(如图 2(a)~图 2(c)所示)相比,图 2(d)的做法是先按对象划分程序的数据空间,然后再按数据空间的划分将原本在控制流中相互交错的操作划分到不同的线程中去.

```
1:  class Fixer {
2:      void tune (Graph g, Page p)
3:      {
4:          …
5:          … =g.init();
6:          … =p.prepare(…);
7:          g.adjust(…);
8:          p.layout();
9:      }
10: }
```

Fig.1 Operations on different data structures interleaved in the control flow path

图 1 不同数据结构上的操作在控制流中相互交错

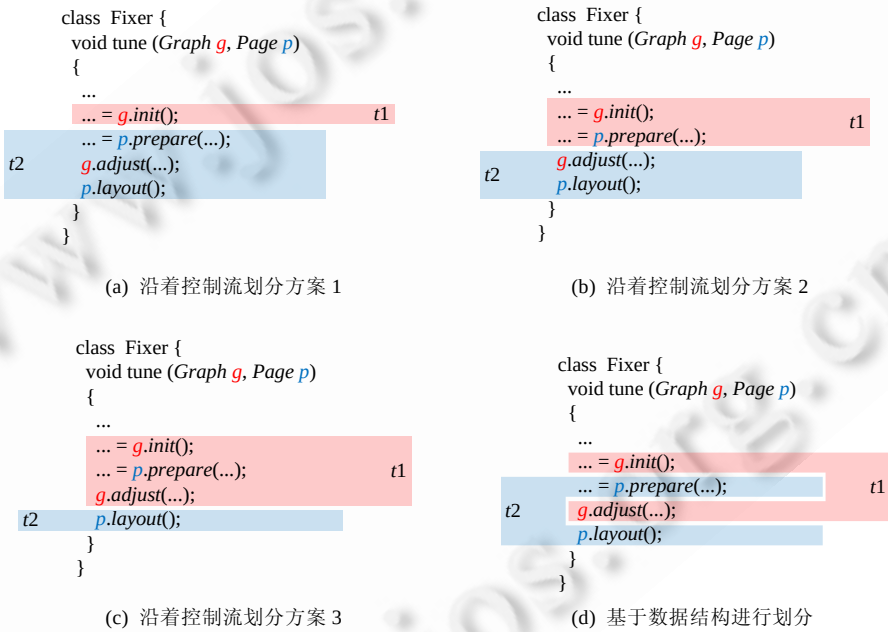

Fig.2 Different schemes for dividing a program into multiple threads

图 2 将一个程序划分为多个线程的不同方案

这种划分思想依赖于这样一个启发法(heuristics),即隶属于同一个对象的方法间冲突的可能性比较大,而隶属于不同对象的方法间冲突的可能性比较小.注意,我们并不是说隶属于同一个对象的两种方法并行一定会发生冲突,因为它们完全有可能操作在对象中彼此无关的两个部分上,从而避免冲突发生.但考虑到现代程序中数据结构的复杂性,两种方法在互不知情的情况下并行操作同一数据结构还能保证数据结构不变性(invariant)的可能非常小[7],而且即便在某个数据结构中存在着彼此无关从而可以并行操作的两个部分,这些部分也往往被封装成独立的对象.

在本文中我们展示出:

- 如果把关注点放在数据结构而不是控制结构上,一些在基于控制流划分框架下无法被利用的并行性就能得到有效的开发[8];
- 只需在编程时进行简单的人工标注,就可将一些只有程序员才知道的高层信息传递给运行时系统,从

而完成对串行程序的并行化工作;

- 通过异步消息传递来解耦串行程序的各个部分以加速程序执行,同时用确定的同步消息来验证推测性质的异步消息,以保证并行化后程序的正确性和串行语意[9].

本文第 1 节对并行化技术发展中面临的挑战以及各种技术路线进行介绍,将本文方法镶嵌到整个技术图景之中.第 2 节介绍 ROPE 方法,即对象分组以及基于对象分组的线程划分方法和执行模型.第 3 节对实现中的一些关键问题进行说明.第 4 节是正确性证明.第 5 节通过对 JOlden 基准程序集[10]中几个程序的研究来说明如何使用 ROPE 方法,并对相关的性能数据进行分析、解释.第 6 节检视相关工作以及它们与 ROPE 方法各自的特点.第 7 节对 ROPE 方法的创新进行总结,并给出下一步工作的可能方向.

1 并行化技术的路线、挑战与发展

在将串行程序划分为多个并发线程的过程中,必须尊重代码间的数据依赖关系,因为只有这样才能保证最终程序的正确性.但就如何划分来说,历来有两种路线:a) 一种是在控制流中依附特定的控制结构(如循环)来寻找潜在的线程,然后再根据这些线程是否违犯了数据依赖关系来判断如此划分是否合法[1−4,11],甚至还可以根据循环迭代间的数据依赖关系对循环做出等效的变换以方便划分[12];b) 另一种路线则完全摆脱了程序控制结构的束缚,直接从数据出发来进行线程划分,即首先将程序中的数据划分为若干不相交的数据集,然后再根据对数据空间的这种划分,将操作不同数据集的代码从串行程序中肢解出来,然后重新组合形成不同的线程.因为这些线程所操作的数据集本来就不相交,所以线程之间就不会再有任何数据依赖关系.

第 1 种路线在处理规则程序(如处理稠密矩阵的数值计算程序)时比较有效,相关理论已经相当成熟[12−15],所以多为自动并行化编译器所采用[11,16,17](这是第 1 种路线的全自动化实现).但对于非规则程序,由于其复杂的数据依赖模式,编译器在缺少人工干预的情况下往往很难对其中的循环做出有效变换,于是催生了很多并行编程模型(如 OpenMP[18],Cilk[19]).使用这类并行编程模型并不需要重新编写并行程序,只需在原有串行程序的基础上通过人工添加特定标注的方式,来指导编译器或运行时系统如何对诸如循环这类结构进行划分,对程序结构的影响相对较小(这是第 1 种路线的半自动化实现).

相比第 1 种路线,第 2 种路线[20−22]能够比较好地处理非规则程序,但因为它要求从数据划分出发来肢解并重构程序,所以得到的并行程序其结构与原来的串行程序相去甚远.进行此类划分,非常依赖一些关于程序行为的高层知识,而这些高层知识往往只有程序员才知道,编译器无从得知更无法利用,所以这条路线多为一些激进的并行编程模型(如 Actor[23],Object Assembly[24])所采用.也就是说,主要是用于指导程序员从头编写并行程序(这是第 2 种路线的纯手工实现).

以龙书(第 2 版)第 11 章为代表的经典方法[12−15]属于第 1 种路线,这类方法主要是针对处理稠密矩阵运算的 Fortran 数值程序而发展出来的.它们依赖于程序的仿射访问(affine access)模式来对循环进行并行化和数据局部性优化,所以它们的适用范围非常有限.比如,同样是数值计算中常见的稀疏矩阵,就因为其元素只能通过另一数组间接访问,故而不具备仿射访问模式,所以也就无法进行这类并行化和数据局部性优化.然而,通过多级指针而进行的复杂访问模式在非规则程序中非常常见,这就使得经典方法几乎无法应用于非规则程序.

随着计算机应用领域的推广,以交互式应用为代表的非规则程序数量已经远远超过了以数值计算为代表的规则程序,近 30 年来,面向对象程序的爆炸式增长尤为引人瞩目.这类程序一般是由围绕若干核心数据结构(称为对象)的若干过程(称为方法)所组成,其特点是:a) 对象多为动态产生,对象之间通过指针联结构成更复杂的数据体;b) 程序行为受输入变化的影响较大;c) 过程体较小,对同一对象的操作可能分布在多个过程之中,而在同一个过程中又可能同时(即短时间内)对多个对象进行操作.以上这些特点使得传统的基于程序分析的技术捉襟见肘[10].动态产生的数据、复杂的指针应用和易受输入影响的程序行为,使得静态分析难以为继(复杂的控制流使得静态地确定一个指针指向哪些对象都非常困难,所以推导出哪些对象是线程间共享的,从而得到哪些对象必须被加锁,都变得不够精确[25]);而较小的过程体则限制了在过程内进行划分的可选择空间;对不同对象的操作相互交织则破坏了程序的数据局部性,使得 Cache 的性能急剧下降[10].

因为传统的程序分析技术在处理非规则程序时面临的种种困难,推测多线程技术[1−4]开始获得广泛应用.因为推测执行机构的支持,线程即使违犯了数据依赖关系也不会导致程序错误(对数据依赖关系的违犯会被检测到,无效执行会被撤销),在一定程度上弥补了程序分析技术的不足,使得那些不怎么安全的划分中所蕴含的并行性也可以得到利用(而在保守方法中只能串行).但是,推测执行并不是没有代价,特别是在那些缺乏硬件支持的纯软件推测多线程环境[24,26]中处理面向对象程序时(如在Java虚拟机中实现的推测多线程支持[27,28]),推测执行失败带来的开销是惊人的[29],很有可能抵销并行带来的好处,使得推测并行得不偿失.所以,对划分的质量又重新提出了要求.对于这类程序,一个好的划分必须要保证两点:第一,线程之间较少发生冲突[30];第二,线程必须是粗粒度的[31].只有这样,才能保证推测并行的收益不会被开销所湮没.

要得到不易冲突的划分,就必须将数据空间的划分作为首要问题来考虑.既然对于面向对象程序,现有的程序分析技术尚无法提供所需的支持,那我们就干脆放弃程序分析,不再依靠程序分析技术提供的信息在较低的语意层面上来讨论数据空间的划分,而转而依靠包含高层语意信息的数据,即数据结构,来在较高的语意层面上讨论数据空间的划分(在相关工作一节我们可以看到,对于高层语意的应用乃是目前并行化技术发展的趋势,无论哪种路线都如此).又考虑到面向对象程序过程体较小,单个过程内缺少划分空间的特点,为了保证线程粒度,就需要我们突破过程边界来进行划分.这两种需求综合起来,启发我们选择一条直接从数据划分出发来划分线程的道路(将操作不同数据集的代码从串行程序中肢解出来,然后重新组合形成不同的线程),即选择并行化的第2种路线.目前,我们离第2条路线的全自动化实现尚有一定距离,要实现高质量的划分还需要少量的人工介入(即由程序员来标注分组策略),但并行化的大部分工作都是由运行时系统自动完成的(第2节中各个小节所占的篇幅反映了各部分的分量),可以说是第2种路线的一种半自动化实现.

2 ROPE方法

传统的推测多线程划分方法将控制流中的片段作为线程,然而这种做法对于图1所示的那种不同数据结构上的操作在控制流中相互交错的情况却不是很有效.是这种程序缺乏并行性吗?答案取决于我们从什么角度去观察.如果能够从数据结构的角度而不是控制流的角度进行观察,我们往往可以发现,在这类程序中存在这样一类并行性,它来源于不同数据结构上发生的操作之间的相对独立性.如果把关注焦点放在数据结构而不是控制流上,就能比较容易地并行化这类程序.通过把每个数据结构上的操作分派到各自的线程中去执行,就可以在很大程度上避免冲突的发生.

根据这一思想,我们设计了名为ROPE(rushing objects parallel environment)的基于数据结构的推测多线程划分方法和执行模型.ROPE方法的核心思想是,我们应当基于数据结构来划分数据空间,然后按照数据空间的划分来划分程序代码形成线程.

面向对象程序中存在两类数据结构:抽象数据结构(abstract data structure)和具体数据结构(concrete data structure).类(class)作为用户自定义类型(UDT),属于抽象数据结构;而对象(object)则属于具体数据结构.我们划分数据空间所依据的数据结构是指具体数据结构,即程序运行过程中出现的对象.

因为对象的存在,程序的数据空间实际上已经有了一个天然的划分,所以一个朴素的想法是,让程序中的每个对象都拥有自己的线程.然而我们发现,这种做法并不能获得最佳效果.原因是有些对象之间的联系非常紧密,如果我们硬把它们分在不同的线程中,会导致这两个线程执行的操作紧密耦合,从而难以并行.在这种情况下,最好把这些紧密耦合的对象看作是一个整体(即一个大的数据结构)来处理,也就是说,应当将这些对象放在同一个组中,由同一个线程来对其进行操作.即,需要在这种天然划分的基础上再进行重组,将那些紧密耦合的对象重新结合起来.

然而,在编程或编译时对象并未真正产生,还无法对其进行分组,所以能做的仅仅是确定分组的原则,即分组策略.但这些编译时确定的分组策略又必须与运行时才会出现的对象联系起来才能指导运行时系统对对象进行分组.编译时就存在的类,正好可以将两者联系起来.也就是说,可以通过在类定义上附加分组策略,来给该类的对象指定分组策略.

程序运行时,每当一个新对象产生时,运行时系统都要根据其分组策略来决定是为这个新对象创立新的组,还是将其加入已经存在的组.对象的分组情况,就是对程序数据空间的一种划分,这种划分是不断变化着的,每当有新对象产生,都要对划分做出少许调整.这样,随着新对象的不断产生,新的对象组也不断产生,而不断产生的对象则根据分组策略进入不同的组,从而动态地完成对程序数据空间的划分.

在创建一个对象组时,运行时系统还会为该组创建对应的线程.线程与对象组之间存在一一对应的关系,对象身上发生的操作只能由所属组对应的线程来执行.至于线程将执行哪些操作,以及按照怎样的次序来执行这些操作,只能随着程序的运行逐步确定,而在编译时,甚至在程序结束之前都是无法预知的(因为输入是无法预知的).这样,就根据数据空间的划分动态地完成了线程构造.

在传统的推测多线程划分方法中,线程包含哪些代码是在编译时就确定下来的;而在我们的方法中,线程包含哪些代码只有在运行的过程中才能逐步确定.具体说,在我们的方法中,线程划分包含了 3 个步骤,即:a) 编译时静态地确定数据空间的划分准则(分组策略);b) 运行时动态地划分数据空间(对象分组);c) 然后在运行时根据数据空间的划分动态地构造线程(操作分派).按照这种想法,程序并行化的过程分为两个阶段 3 个部分:

- 编程时,由程序员对程序中的类进行标注以确定分组策略;
- 运行时,由运行时系统根据分组策略对对象进行分组,并为新诞生的组创建对应的线程;
- 运行时,由运行时系统根据被操作对象所属的对象组,将操作分派到不同的线程中去执行.

可以看到,只有第 1 部分即分组策略的指定需由程序员来完成(我们只是提供相应的语言机制),后两部分(即对象分组和操作分派)都是由运行时系统自动完成的.其中,对象分组比较简单,只是单纯地按照由程序员所确定的分组策略来进行;而操作分派部分因为涉及到把一个串行程序转化成一个相互配合的多线程并行程序,较为复杂.这一点读者从稍后各部分所占的篇幅中可以体会到.

2.1 分组策略

为了让程序员能在程序中指定分组策略,我们提供了一个名为*@GroupingPolicies* 的 Java 标注(annotation),其定义如图 3 所示.*@GroupingPolicies* 带两个参数,其中,*policy* 用来为对象自身指定分组策略,*foreignPolicy* 用来为由该对象创建的其他对象指定分组策略.二者皆可取下面表 1 所示的 4 种数值之一.

```
import java.lang.*;

@annotation.Retention(annotation.RetentionPolicy.RUNTIME)
@annotation.Target({ annotation.ElementType.TYPE })
public @interface GroupingPolicies {
    GroupingPolicy policy();
    GroupingPolicy foreignPolicy();
}
```

Fig.3 Definition of the annotation *@GroupingPolices*

图 3 标注*@GroupingPolices* 的定义

Table 1 *Grouping* policies

表 1 分组策略

NEW_GROUP	为新对象创建新的对象组
CURRENT_GROUP	将新对象加入当前组
INITIAL_GROUP	将新对象加入初始组
UNSPECIFIED	不特别指定,由系统决定

在编程时,程序员将此标注应用在类定义上,从而为该类的对象指定分组策略.如图 4 所示,类 *Graph* 和类 *Page* 上的标注表明,当这些类的对象产生之后,运行时系统将为其创建一个新的对象组.值得说明的是,我们并不需要对每个类都逐一地进行标注以确定其分组策略.一般来说,程序员只需选出那些他认为最有并行价值的对象,将其类的分组策略标注为 *NEW_GROUP*,以便这些类的对象产生之后,运行时系统为之创建新的线程.程序员的选择虽然会影响并行化的效果,但对正确性丝毫没有影响.

```
1:  @GroupingPolicies(policy = NEW_GROUP)
2:  public class Graph {
3:      …
4:  }
5:
6:  @GroupingPolicies(policy = NEW_GROUP)
7:  public class Page {
8:      …
9:  }
```

Fig.4 Examples of applying annotation to specify grouping policies

图 4 通过标注指定分组策略的例子

分组策略的指定者是程序员,使用者是运行时系统,所以我们将代表分组策略的这些 Java 标注定义为运行时可见标注.这意味着运行时系统可以检测到一个类是否带有该标注,从而采取相应的措施.

此外,因为在 Java 中类本身也是对象,为了标记类对象本身而不是类的实例,我们还提供了另一个类似的标注*@ClassGroupingPolicies*.*@ClassGroupingPolicies* 和*@GroupingPolicies* 一样都是普通的 Java 标注,而不是对 Java 语言的扩充,这意味着经过标注的 Java 程序仍然可以通过普通 Java 编译器进行编译,并在普通的 Java 虚拟机上运行,只是没有加速效果而已.

2.2 对象分组

在这个阶段,运行时系统根据程序员在编程时所指定的分组策略,将遇到的对象划分到不同的对象组中.开始时,整个系统中只有一个初始组,对应于程序的初始线程.每当有新的对象产生,运行时系统就会根据新对象上(实际上是其所属类上)带有的*@GroupingPolicies* 标注信息来决定其分组:要么加入为之创建的新组,要么加入已经存在的老组.每个对象中都有一个额外的字段,用于记录该对象所属的对象组.新组产生时运行时系统会创建相应的线程,而当对象组中的所有对象都被垃圾收回之后,对象组和对应的线程也会被销毁.

2.3 操作分派

在对象分组的基础上,就可以将控制流中交错的操作分派到各自的线程中去执行了.下边我们通过一个具体的例子来说明程序在 ROPE 模型下的执行过程,以帮助读者建立起一个关于 ROPE 模型如何工作的直观印象,然后再对其中所涉及的各个关键问题进一步加以说明.

2.3.1 例子:执行模型

如图 5 所示,有 *a*,*b*,*c* 这 3 个对象,假设它们分属于 3 个不同的组.这样一来,将由 3 个不同的线程(*t*1,*t*2,*t*3)来负责执行其上的操作.图 6 显示了这 3 个线程之间相互协作以完成这个程序的过程.如图 6(a)所示,线程 *t*1 在执行方法 *a*.*foo* 的过程中遇到对方法 *b*.*bar* 的调用.因为对象 *b* 是由另一线程 *t*2 负责的,所以线程 *t*1 把确定控制转交给线程 *t*2,由线程 *t*2 在确定模式下执行 *b*.*bar*.但此时,线程 *t*1 并不是停下来等待线程 *t*2 把 *b*.*bar* 执行完,而是转入推测模式继续执行 *b*.*bar* 之后的代码.因为 Part II 依赖于 *b*.*bar* 的返回值,所以线程 *t*1 在继续执行之前会首先执行一个对应于 *b*.*bar* 的返回值预测方法来获得一个推测性质的返回值,然后再在推测模式下继续执行.随后,*t*1 碰到对 *c*.*qux* 的调用,因为对象 *c* 由另一线程 *t*3 负责,所以线程 *t*1 并不执行 *c*.*qux*,而是像刚才一样调用一个对应于 *c*.*qux* 的返回值预测方法,与此同时,线程 *t*1 还会给线程 *t*3 发送一条消息通知线程 *t*3,说“等会我可能会调用你负责的 *c*.*qux*,你有空的话就提前执行一下”.然后,线程 *t*1 继续在推测模式下执行 Part III.而线程 *t*3 现在处于空闲状态,于是就开始推测性地提前执行 *c*.*qux*.

如图 6(b)所示,当线程 *t*2 执行完 *b*.*bar* 之后,它把确定控制交还给线程 *t*1.线程 *t*1 得到确定控制后转入确定模式,将线程 *t*2 给出的 *b*.*bar* 的返回值与自己先前调用返回值预测方法得到的值进行对比,发现二者一致,这表明 Part II 是在正确的返回值的基础上进行的,所以因执行 Part II 而导致的对程序状态的改变将被提交.而 Part III 因为不光依赖于 *b*.*bar* 的返回值,还依赖于 *c*.*qux* 的返回值,所以其执行所产生的影响目前还不能提交.

如图 6(c)所示,在提交了 Part II 之后,线程 *t*1 的 PC 指针前进到对 *c*.*qux* 的调用处.然后,线程 *t*1 将确定控制转交给线程 *t*3,线程 *t*3 将之前推测执行 *c*.*qux* 时所用的参数与线程 *t*1 此时给出的参数进行对比,二者完全相同,

这表明线程 *t*3 之前对 *c.qux* 的推测执行是有效的,其产生的影响将被提交.在 *c.qux* 提交之后,线程 *t*3 现在由 *c.qux* 返回,如图 6(d)所示,此时的情形与从 *b.bar* 返回时是一样的.

以上都是验证成功时的情形,现在我们再来看一看验证失败时发生的情况.如图 6(e)所示,当 *b.bar* 返回时,线程 *t*1 发现线程 *t*2 给出的 *b.bar* 的返回值与线程 *t*1 之前通过执行返回值预测方法得到的返回值不一样,这意味着,之前 Part II 和 Part III 的执行都是在错误的返回值基础进行的.所以,线程 *t*1 此时就不得不重新执行 Part II.

方法调用验证失败与返回值验证失败的处理基本相同.如图 6(f)所示,当线程 *t*1 把确定控制交给线程 *t*3 之后,线程 *t*3 发现线程 *t*1 调用方法 *c.qux* 所用的参数与它自己之前推测性地执行 *c.qux* 时所用的参数不同,或者干脆它提前执行的根本就不是 *c.qux* 而是其他方法,那线程 *t*3 此时就不得不在确定模式下用正确的参数重新执行一遍 *c.qux*.

```
1:  a.foo()
2:  {
3:      … … …   // Part I of a.foo
4:      b.bar();
5:      … … …   // Part II of a.foo
6:      c.qux();
7:      … … …   // Part III of a.foo
8:  }
```

Fig.5 Example: Object *a*, *b*, and *c* is under the charge of thread *t*1, *t*2, and *t*3 respectively

图 5 例子:对象 *a*,*b*,*c* 分别由线程 *t*1,*t*2,*t*3 负责

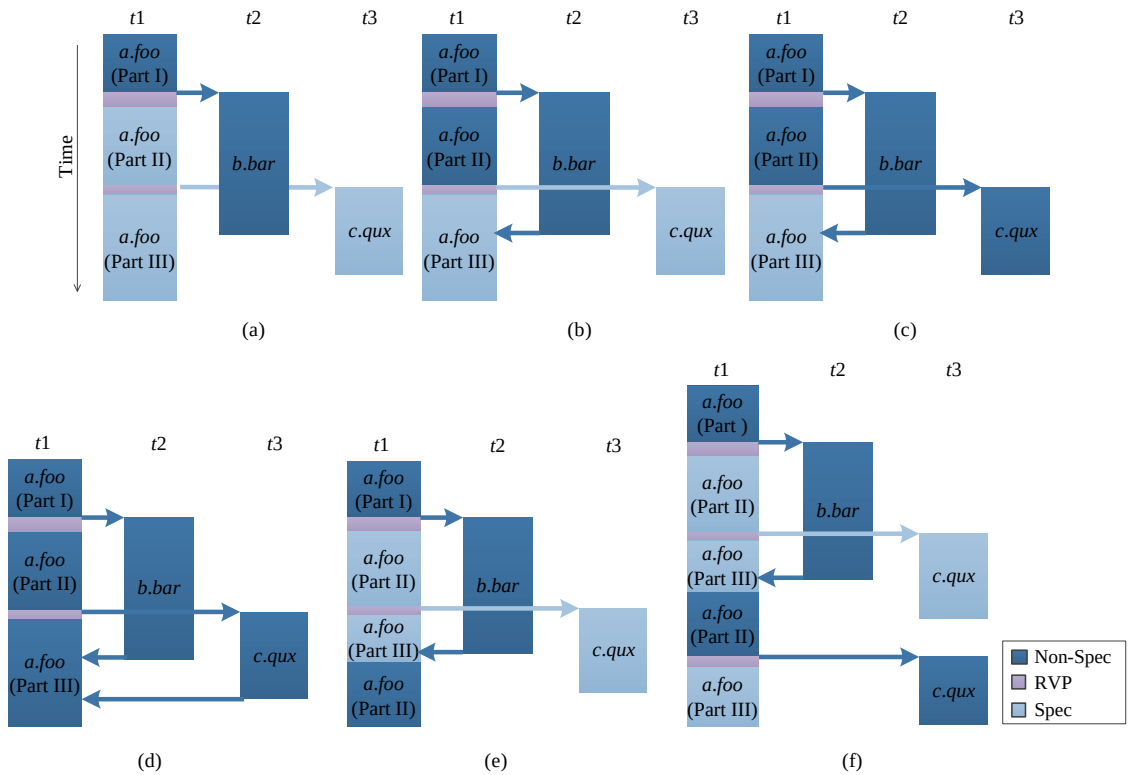

Fig.6 Example of the ROPE execution model

图 6 ROPE 执行模型的例子

2.3.2 确定模式与推测模式

按照程序的串行语义,控制从一种方法转移到另一种方法,从一个对象转移到另一个对象.我们把这个控制

称为确定控制,因为还有与之相对的推测控制.伴随着确定控制从一个组中的对象转至另一个组中的对象,获得确定控制的组所对应的线程转入确定模式,而失去确定控制的组所对应的线程转入推测模式.在任意时刻,只有一个线程处于确定模式,而所有其他线程都处于推测模式.在确定模式下,线程直接进行读写;推测模式下,对内存的写操作会被缓存起来,待到验证成功时再提交.同一个线程在不同的模式下有独立的上下文(context).按照Java 虚拟机规范[32],线程上下文可由三元组(PC,FP,SP)表示,其中,PC 持有当前指令的地址,FP 指向当前栈帧,SP 指向当前栈帧中操作数栈(operand stack)的顶部.

2.3.3 确定消息与推测消息

每个对象都有一个额外的字段记录着其所属的线程,检查该字段就可以知道对象属于哪个线程.线程遇到方法调用时,如果目标对象属于本线程,就直接执行该方法;如果属于其他线程,就将调用信息打包,以消息的形式发送给相应的线程(即目标线程),由目标线程去处理.根据线程遇到方法调用时所处的模式,线程发出的消息分为确定消息和推测消息.

确定消息达到目标线程后,无论目标线程此时处于何种状态,都必须立刻转入确定模式来处理确定消息.如果是推测消息,则会被放入目标线程的推测消息队列,等待处理.目标线程会在空闲时从推测消息队列中取出待处理的推测消息进行处理.

2.3.4 推测消息队列与 Effect 结构

如图 7 所示,推测消息队列分为前后两个部分:前边是已经处理并等待验证的推测消息(即待验证消息),后边是尚待处理的推测消息(即待处理消息).指针 verifyp 和 nextp 分别指向这个两部分的第 1 条消息.其他线程发来的推测消息被放在待处理部分的尾部.一旦线程开始处理某条推测消息,该消息就从队列的待处理部分进入待验证部分(nextp++).处理推测消息会改变程序的状态,然而在推测消息的正确性最终得到证明之前,这些更改并不能真正算数,它们都被暂存在与消息关联的 Effect 结构中,对于程序的其他部分是不可见的.当线程收到确定消息后,会将其与消息队列中的第 1 条待验证消息进行对比,如果相同,就说明推测模式下提前进行的消息处理是有效的,处理该消息对程序状态造成的影响将被提交;否则,这些影响将被丢弃.

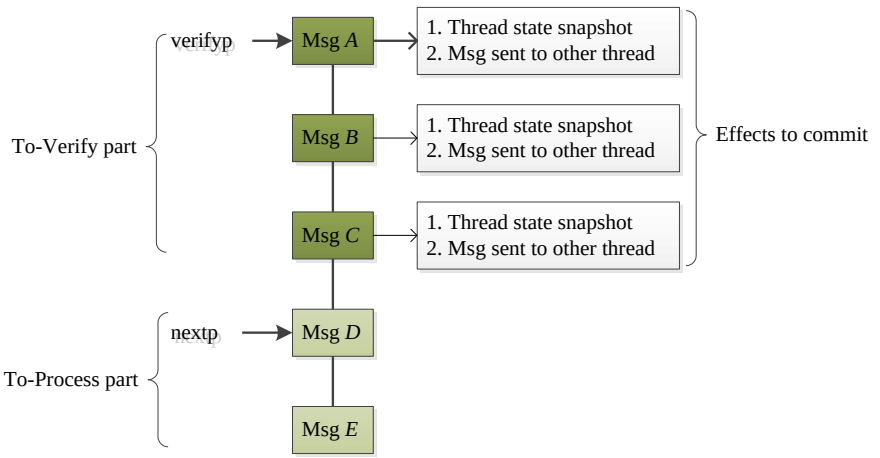

Fig.7 Speculative message queue

图 7 推测消息队列

每个处于待验证部分的推测消息,都关联着一个 Effect 结构.Effect 结构中记录着处理该消息对程序状态造成的影响.这种影响包括以下两个方面:

- 线程状态快照:即处理该消息对线程自身状态造成的影响;
- 发出的推测消息:即处理消息的过程中产生的发往其他线程的推测消息.

线程状态快照是在线程开始处理下一条推测消息之前的一刻才产生的,代表的是线程因处理上一条推测

消息最终所到达的状态.如果将来该推测消息验证成功,那么,这个快照所反映的状态将被提交,处理该推测消息的过程中发往其他线程的推测消息也会升级为确定消息,再次发往相应的线程.如果验证失败,那么,Effect 中的快照将被丢弃,其间发出的推测消息也会被从其他线程召回(见第 2.3.7 节).

2.3.5 被动消息与主动消息

在处理推测消息的过程中,如果遇到对其他线程所负责对象的方法调用,而本线程后续的执行又依赖于该方法的返回值(准确地说,后续执行是基于线程下一步将得到一个返回值消息这一假设),为了能继续执行,线程会执行相应的返回值预测方法,以获得一个推测性质的返回值.这个推测出来的返回值(return 消息)将来同样需要进行验证.这样一来,消息队列的待验证部分中就有了两种性质的待验证消息:一种是来自其他线程的推测性质的调用消息,另一种是本线程自己通过返回值预测构造出来的推测性质的返回值消息.前一种消息在被线程处理之前要先在消息队列的待处理部分排队;后一种消息一经产生,就立即得到处理.根据这两种消息的特点,我们把前一种称作被动消息,把后一种称作主动消息.对被动消息和主动消息的处理,在很多方面都是不同的,我们会在有关的地方指出.除调用消息(invoke 消息)、返回值消息(return 消息)外,还有一些其他类型的消息.表 2 列出了这些消息以及相关的指令.如表 2 所示,invoke 指令和 return 指令各自只对应一条消息;而 put,get,astore,aload 指令每个都对应两条消息.其中,put 与 get 指令与对象字段的读写有关,astore 和 aload 则与数组元素的读写有关.带 ret 后缀的消息表明指令执行完毕.

Table 2 Content of messages

表 2 消息的内容

指令	消息	消息内容
INVOKE	invoke_msg	目标对象引用 方法名 实参 源线程 ID 源线程的(PC, FP, SP)
RETURN	return_msg	返回值 方法调用方所在线程的(PC, FP, SP)
PUT	put_msg	目标对象引用 字段名 写入的数值 源线程 ID
	put_ret_msg	(无)
GET	get_msg	目标对象引用 字段名 源线程 ID
	get_ret_msg	读到的数值
ASTRORE	astore_msg	数组对象引用 元素大小 下标值 写入的数值 源线程 ID
	astore_ret_msg	(无)
ALOAD	aload_msg	数组对象引用 元素大小 下标值 源线程 ID
	aload_ret_msg	读到的数值

2.3.6 推测消息的验证

线程收到确定消息之后,会挂起正在进行的推测执行,并转入确定模式来处理确定消息.此时,如果消息队列中有待验证的推测消息,验证逻辑就将 verifyp 所指的推测消息与确定消息进行对比(确定消息若是被动消息,比较指针即可;若是主动消息,则需要对消息的内容进行比较),如果二者相同,则验证成功;否则,验证失败.如

果消息队列中没有待验证的推测消息,也认为是验证失败.验证成功后,线程会提交该推测消息所关联的 Effect,使线程在推测模式下对程序状态所作的更改变得对其他线程可见,然后从消息队列中移除该消息,并恢复被挂起的推测执行.如果验证失败,线程则会丢弃所有未提交的 Effect.这是因为,线程在获得确定模式从而能够对推测消息进行验证之前,可能已对多条推测消息进行了处理,每条推测消息的处理都是在前一条推测消息处理的基础上进行的,所以一旦一条推测消息被证明是错误的,不光与其直接关联的 Effect 要丢弃,其后所有待验证消息所关联的 Effect 都要丢弃.验证失败时,待验证部分的主动消息将被直接移除,被动消息则重新进入消息队列的待处理部分(移动 nextp 即可).

2.3.7 推测消息的召回

被动消息并不会永远呆在消息队列中直到它们验证成功,消息队列中的这些被动消息都是其他线程在推测模式下发出的,所以一旦发出这些被动消息的线程意识到自己是在错误的状态下发出了它们,就会将其从目标线程的消息队列中召回,以避免目标线程把宝贵的时间浪费在对错误消息的处理上.线程验证失败时就会发生召回,此时,所有未提交的 Effect 结构都会被丢弃,而 Effect 结构中所记录的发往其他线程的推测消息就会被召回.每个线程都有一个被召回消息集合,被召回的消息将被记入这个集合,线程在推测执行的过程中会经常检测这个集合.一旦有被召回的消息,线程就会将其从消息队列中清理出去.此时,被召回的消息可能位于消息队列的待处理部分,也可能位于待验证部分.如果处于待处理部分,则直接将其移除即可;如果处于待验证部分,则首先将与之关联的 Effect 结构丢弃,并且如同验证失败时那样,被召回消息之后的其他待验证消息所关联的 Effect 结构也都要扔掉,然后将被召回的消息从队列中移除.它之后所有受到影响的其他待验证消息,如果是被动消息,则将重回消息队列的待处理部分;如果是主动消息,则被移除.消息召回会引起连锁反应,即消息召回会导致 Effect 被丢弃,Effect 被丢弃又会导致更多的消息被召回.

2.3.8 状态快照与多版本状态缓冲区

状态快照中只记录线程上下文(PC,FP,SP)以及一个状态版本号.与该版本号相关的数据则存储在线程的多版本状态缓冲区中.为了不影响其他线程,线程在推测模式下的读写都将被导向多版本状态缓冲区.当进行快照时,多版本状态缓冲区会冻结当前版本,使其不受此后任何读写操作的影响.如果因为处理多条推测消息而多次快照,状态缓冲区中就会有线程状态的多个版本.多版本状态缓冲区采用一种增量的方式来保存这些不同的状态版本.也就是说,新版本中只记录那些相对于老版本发生了变化的数据,所以冻结操作是非常廉价的.读写操作的时间则分摊在推测执行之中.当快照被提交时,状态缓冲区中的对应版本将被提交至内存,变得对其他线程可见.

2.3.9 返回值预测方法与返回值预测模式

虽然其他组中对象上的方法调用被分派到其他线程去执行,但方法调用之后的代码却往往依赖于该方法的返回值.为了消除线程间的这种耦合,我们通过调用相应的返回值预测方法来为方法调用之后的代码提供一个推测性的返回值.返回值预测方法具有与原来方法一样的签名与返回值类型,但所包含的指令要少得多,从而也只需要较短的执行时间.返回值预测方法给出的返回值不必是正确的(虽然我们希望如此),因为不正确的返回值最终会被验证机制发现并更正.返回值预测方法既可以由编译器自动生成,也可以由程序员提供.如果程序员在源代码中为某种方法指定了返回值预测方法,编译器将采用程序员提供的版本;否则,编译器将自动产生一个版本.自动生成的返回值预测方法是通过逆向遍历原方法的控制流图来构造的,只有那些与返回值的形成直接相关的指令才会被选入.返回值预测方法的执行由返回值预测临时缓冲区与推测模式的一个特殊子模式即返回值预测模式(RVP 模式)来支持.返回值预测方法在返回值预测模式下执行.在执行期间,所有的写操作都被导向至返回值预测临时缓冲区,这样做的目的是为了避免返回值预测方法的执行改变其他线程所负责对象的状态.当返回值预测方法结束时,返回值预测临时缓冲区中的内容将被丢弃,这是因为执行返回值预测方法的目的只有一个,即获得返回值的预测值,其他副作用在得到这个返回值之后就没有用了.有关回调方法的内容,我们将在第 2.5 节中加以讨论.

2.3.10 结构图

在以上的讨论中,我们介绍了推测消息队列、Effect 结构、被召回消息集合(recalled msg set)、多版本状态缓冲区(state buffer)、返回值预测临时缓冲区(RVP buffer)等关键部分,现在我们把这些部分和系统的其他部分(共享内存)放在一起,构成一幅更大的图景.图 8 展示了 ROPE 虚拟机的各个部分以及它们之间的关系.

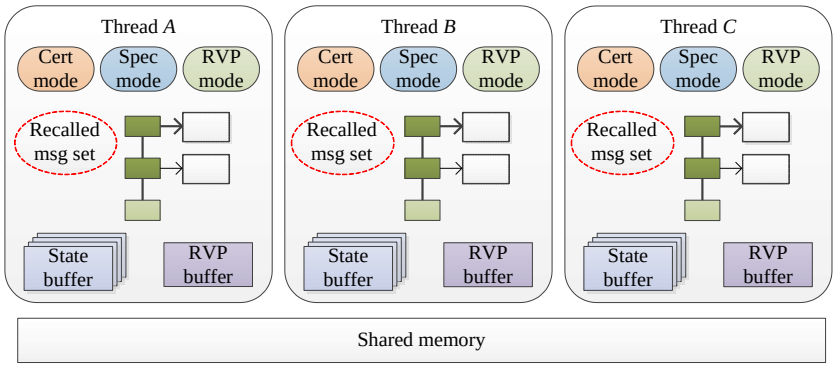

Fig.8 Overall architecture

图 8 总体结构图

2.3.11 关于名字

现在,读者一定不难理解为什么我们要将这种方法命名为 ROPE(rushing objects parallel environment)了:对象在推测模式下匆匆前行,线程不是首尾相接(在逻辑上并没有先后次序),而是像若干根 thread 相互缠绕成一条 ROPE.

2.4 用于方法的标注

无论*@GroupingPolicies* 还是*@ClassGroupingPolicies*,它们标注的对象都是类.此外,我们还提供了一些可用于方法的标注,它们是*@InvokerExecute*,*@Irrevocable* 和*@SpecSafeNative*.

2.4.1 *@InvokerExecute*

这个标注用来标记调用方执行方法.如果我们把遇到 invoke 指令的线程称作方法的调用方,把执行方法的线程称作方法的执行方,那么在串行模型中,因为只有一个线程,所以它既是方法的调用方,也是方法的执行方.在 ROPE 模型中,虽然方法的调用方可能有多个,但方法的执行方只有一个,即目标对象所在的线程.也就是说,方法的调用方与执行方是分离的,对象的状态只能由其所属线程来改变.

但是存在着这样一类方法,其所属对象并没有自己的状态,或者所属对象虽有自己的状态,但方法的执行并不涉及该状态,这样的方法根本不会影响其所属对象的状态,让多个线程同时执行它们也不会导致(该对象上的)冲突,此时就没有必要限制它们只能在某个特定的线程中执行.在这种情况下,可将这些方法标记为*@InvokerExecute*.当线程在推测执行的过程中遇到带有这个标注的方法时,将直接执行它们,而不是将其分派到目标对象所属的线程去执行.标准库中,java.lang.Math 所属的各种方法就属于此类.

2.4.2 *@Irrevocable*

这个标注用来标记不可召回方法.一般情况下,当验证失败时会召回发出的推测消息,因为一般错误的状态下发出的消息也是错误的.但有些方法其调用并不会受之前错误的影响.如图 9 所示,无论对方法 *foo* 返回值的预测是否正确,都不会影响对方法 *bar* 的调用.如果仅仅因为 *foo* 的返回值有错误而将 *bar* 召回(此时,消息 invoke *bar* 可能已被目标线程处理),并不是一种好的做法.在这种情况下,我们可以将 *bar* 方法标记为*@Irrevocable*,即不可召回方法.这样,即便 *foo* 的返回值验证失败,我们也不会将消息 *invoke bar* 从其他线程的消息队列中召回.

```
1:  void example()
2:  {
3:      if (b.foo()) {
4:          …
5:      }
6:      else {
7:          …
8:      }
9:      …
10:     c.bar();
11:     …
12: }
```

```
1:  class SomeClass {
2:      …
3:      @Irrevocable
4:      public void bar()
5:      {
6:          …
7:      }
8:      …
9:  }
```

Fig.9 Example: The invocation of *c.bar* is not affected by the return vaule of *b.foo*

图 9 例子:无论方法 *b.foo* 的返回值是什么,都不会影响对 *c.bar* 的调用

2.4.3 *@SpecSafeNative*

这个标注与推测模式下原生(native)方法的处理有关.因为原生方法的执行超出了虚拟机的控制范围,所以,为了安全起见,当线程在推测模式下遇到对原生方法的调用时,必须停下来等待确定模式.但如果程序的作者知道某种原生方法的执行并不会对 ROPE 模型造成危害,那么他就可以通过*@SpecSafeNative* 将该方法标注为推测安全的原生方法.在遇到这种方法调用时,即便处于推测模式,线程也会执行方法调用.*@SpecSafeNative* 常常与*@InvokerExecute* 一起使用,用来标记标准库中的一些方法,如 VMMath.log 和 VMMath.exp.

2.5 关于回调方法

在进行返回值预测时(参见第 2.3.9 节),我们并没有考虑方法回调的问题.如图 10 所示,当线程 *t*1 发出确定消息 *invoke b.bar* 后,它会调用相应的返回值预测方法 *b._rvp_bar* 以获得一个推测性质的返回值,并在此基础上继续执行.但当线程 *t*2 执行 *b.bar* 时,却发生了对方法 *a.qux* 的调用.这意味着线程 *t*1 在得到 *b.bar* 真正的返回值之前,需要首先处理来自线程 *t*2 的确定消息 *invoke a.qux*.而线程 *t*1 之前的推测执行却是在发出消息 *invoke b.bar* 之后会首先得到 *b.bar* 的返回值消息这个假设的基础上进行的,这样一来,验证必然失败.

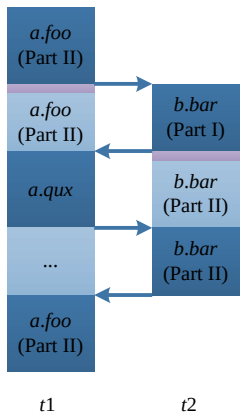

Fig.10 Example: Call back method

图 10 例子:回调方法

对于这个问题,我们曾经试图这样解决:让对应 *b.bar* 的返回值预测方法 *b._rvp_bar* 在给出推测性的返回值之前,先产生一个推测性的 *invoke a.qux* 消息,然后让线程 *t*1 从 RVP 模式转入推测模式,对该消息进行处理,并作为待验证消息记录下来,等处理完该消息之后再重新转入 RVP 模式继续执行 *b._rvp_bar* 余下的部分.这样一来,在 RVP 方法给出推测性的返回值之前,线程 *t*1 将会首先得到一个推测性的 *invoke a.qux* 消息.

但经过实验我们发现,这样做并不能如预想的那样可以提高加速比.因为即便知道了会发生回调而通过

RVP 方法在返回值消息之前先产生代表回调的 invoke 消息,也很难保证 invoke 消息其他参数的正确性,除非我们能事先知道程序的输入,并根据这种先验知识对 RVP 方法进行专门的“优化”.最终还是放弃了这条途径,我们意识到:提高消息推测成功率的关键并不在于复杂的预测技术,而在于数据空间的划分要使得对消息的推测更加容易.换言之,对象组的划分才是关键所在,而再高级的返回值预测技术也无法挽救一个差劲的划分.偶尔的回调并不会妨碍 ROPE 发挥作用,频繁的回调则暗示着对象间的紧密耦合,而这正是对象分组需要解决的问题.

2.6 关于负载平衡

负载平衡的目的是充分利用处理器资源,避免大量工作积压在某些处理器上,而另一些处理器却因为早就完成了自己的工作而无事可做[33].负载平衡固然很好,但却不是所有的并行程序都有条件实现负载平衡,这一方面与所用的划分算法有关,另一方面则与被并行化的程序有关.举一个极端的例子,一个完全无法并行化而只能串行执行的程序是根本无法为其实现负载平衡的,除了一个核从头到尾忙碌外,其他核只能闲着.

为了实现负载平衡,划分算法必须将程序划分为若干大小相当的线程,但除了那些处理稠密矩阵的规则程序之外,其他程序实际上很难在编译时静态地做到这点.所以,负载平衡一般都不是由编译器来做,而是由运行时的调度器来做.典型做法是将程序的工作分解为若干任务,然后根据这些任务之间的数据依赖关系来动态地调度满足数据依赖关系的任务去空闲的处理器上执行.在执行过程中会有任务不断产生,当某个处理器产生一个任务之后,如果该处理器还不能立即执行该任务,而此时还有其他处理器空闲的话,调度器就会将该任务分配给空闲处理器去执行.工作窃取(work stealing)[19,34]是这样一种调度方式:它不是在有新任务之后就立即寻找空闲处理器然后将任务分配给它执行,而是由空闲处理器自己主动去申请任务来执行.

在 ROPE 方法中,“任务”就是对象组身上发生的操作.在运行过程中遇到一个操作之后,我们按照数据依赖关系把可与本线程正在执行的操作并行执行的操作分派到其他线程去(排队等待)执行,而那些因为数据依赖不能与本线程正在执行的操作并行执行的操作则由本线程串行执行.也就是说,“任务”在线程之间如何分配是有既定策略的,而无需空闲线程提出申请.而且分配给某个线程的那些“任务”,因为数据依赖关系,其他线程想帮它执行也是不可能的.但就负载平衡来说,我们的方式所能达到的效果与工作窃取所能达到的效果完全一样.

3 实 现

下面我们对有关实现的一些关键问题加以说明,这些问题及其解决方案虽然不是 ROPE 模型自身的一部分,但却关系到 ROPE 模型与系统其他部分之间如何协作.

3.1 推测模式与异常处理

现代编程语言大都提供了某种形式的异常机制:一旦检测到异常,将进行堆栈解开(unwinding)、异常处理器查找等一系列异常处理流程.简言之,异常处理对于程序状态的影响是非常大的.对于串行程序来说,异常处理是实现程序功能应尽之义务;但对于 ROPE 中的推测线程来说,推测性地处理异常却仅仅是为了加速程序执行.但是,因为执行环境的推测性质,异常发生的几率要比串行环境下高得多,而这大量异常中只有极少一部分会最终转化为确定的异常,而处理异常又会严重影响整个系统的推测运行状态.所以,一旦在推测模式下发生异常,我们就让推测线程进入睡眠状态,等待确定模式的到来.

在确定模式下的异常处理与在串行模型中差不多,所不同的是,在 ROPE 模型中,调用链上的各种方法所属的对象往往是由不同的线程来负责的(如图 11 所示).在这种情况下,每个线程只负责解开调用链上自己所负责的那段,并在其中查找异常处理器.如果找到了,就执行该处理器;否则,就将异常抛给上级方法所在的线程,再由上级方法所在的线程继续进行异常处理.因为异常处理伴随着控制转移,所以上级方法所在线程在接到异常之后,必须丢弃所有未提交的 Effect,转入确定模式来处理异常.当前线程则在失去确定控制后转入推测模式,继续处理推测消息.

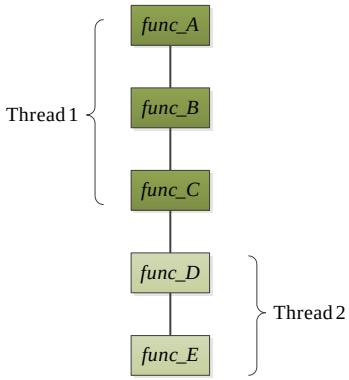

Fig.11 Different frames are under charge of different threads

图 11 栈中的不同栈帧由不同的线程负责

3.2 多线程Java程序的推测并行化

前面讲的都是如何将一个串行程序划分为多个 SpMT 线程.如果一个程序本身就已经是一个多线程程序,则仍然可以在此基础上进一步将其推测并行化.此时,每个线程都相当于是一个需要推测并行化的串行程序.在我们对这个问题作进一步说明之前,有必要先对一些概念加以界定.我们会涉及到 3 类线程,它们是:

- Java 线程;
- SpMT 线程;
- 操作系统线程.

之前所提到的线程都是指 SpMT 线程,它们之间的关系如图 12 所示.

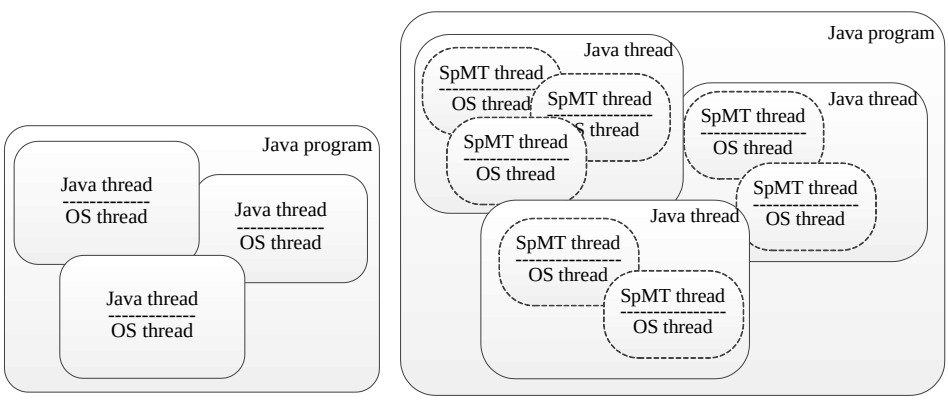

Fig.12 Traditional environment vs. ROPE environment. Relation among three kinds of thread

图 12 传统 Java 环境 vs.ROPE 环境中 3 类线程之间的关系

一个 Java 程序(即进程)由一个或多个 Java 线程组成.如图 12(a)所示,在传统 Java 环境中,每个 Java 线程都对应一个操作系统线程;而在 ROPE 环境中(如图 12(b)所示),Java 线程并不直接对应操作系统线程,它仅仅是一个概念意义上的线程.在每个 Java 线程的背后是若干 SpMT 线程(在图中用虚线表示),真正对应操作系统线程的是这些 SpMT 线程.此时,每个 Java 线程都相当于是一个可以推测并行化的“串行程序”,就像前面讲的那样,我们会对其进行对象分组,然后根据分组将其划分为多个 SpMT 线程.

但是,因为共享同一个进程空间,这些“串行程序”之间并不是完全隔离的,也就是说,在这些“串行程序”之间有共享对象存在.共享对象会被多个“串行程序”,即 Java 线程访问.对于每个访问它的 Java 线程来说,在第 1 次遇

到该对象时,都要决定此对象在该 Java 线程中的分组.这样一来,同一个对象就会处于多个对象组中,并由多个 SpMT 线程来负责执行其方法调用.乍看之下,这似乎与我们的“对象的方法只能由专门负责该对象的 SpMT 线程来执行”相矛盾.但是,只要注意到共享对象所属的多个组必然位于不同的 Java 线程之中,便可消除这些担心——共享对象就是被设计来让多个 Java 线程并发访问的,保证并发安全是程序员的责任.只要在串行模型下不会发生问题,在 ROPE 模型下就同样不会发生问题.

但共享对象被多个 SpMT 线程访问也带来了一个新的问题:原来对象中有一个额外的字段用来记录对象所属的 SpMT 线程(见第 2.3.3 节),但现在因为共享对象隶属于多个 SpMT 线程,对于每个访问该对象的 Java 线程,对象都必须记录它在该 Java 线程中由哪个 SpMT 线程负责.而且,到底有多少个 Java 线程会访问该对象也无法事先确定.所以,一个字段显然不够用.虽然可以通过在对象中增加一个从 Java 线程到 SpMT 线程的哈希表的方式来解决这个问题,但其缺点也是显而易见的:这样会把对象的存储模型复杂化;而对于非共享对象来说,仅仅因为共享对象的存在就增加这么多代价,违反了“What you don’t use, you don’t pay for”的原则.为了解决这一问题,我们采用了一种侵入式(intrusive)与哈希表相结合的杂交方案,即对象中只保留两个字段,一个用来记录创建该对象的 Java 线程,另一个用来记录该对象在创建它的 Java 线程中所属的 SpMT 线程.至于该对象与其他 SpMT 线程间的对应关系,则记录在附属各 Java 线程的小哈希表(从对象到 SpMT 线程)中.当查询对象属于哪个 SpMT 线程时,如果当前 Java 线程就是对象中记录那个 Java 线程,那么对象中记录的 SpMT 线程就是对象在该 Java 线程中所属的 SpMT 线程;否则,才去当前 Java 线程的哈希表中以对象为 key 去查找.

另外,多 Java 线程的存在对分组时机也有影响.原来在对象产生后立即对其进行分组,现在则是在对象首次被一个 Java 线程访问的时候才进行分组.

3.3 虚拟机重入与推测执行

假设 Java 虚拟机中用于解释执行 Java 字节码的函数为 execute_java_method,当 Java 虚拟机(这是一个 C/C++程序)初始化完成之后,便会立即调用 execute_java_method(这是一个 C/C++函数)来解释执行一个名为 Main 的静态 Java 方法.当 Main 返回时,函数 execute_java_method 结束,随后,Java 虚拟机退出.因此从某种意义上来说, execute_java_method 就代表着 Java 虚拟机.我们可以把 execute_java_method 的内部想象成一个循环,其中包含着一个巨大的 switch-case 结构,针对不同的 Java 指令来执行不同的操作.所谓虚拟机重入,就是指函数 execute_ java_method 的重入.因为虚拟机自身的需要,或应客户程序的要求,都可能导致虚拟机重入的发生.

- 因虚拟机自身的需要:解释某条 Java 指令的 C/C++代码需要另一种 Java 方法来辅助实现其功能.如在解释 NEW 指令时,实现该指令的 C/C++代码不是自己去加载 class 文件,而是通过调用标准库中的 Java 方法 loadClass 来完成该工作,为了解释 loadClass,就会导致函数 execute_java_method 重入;
- 应客户程序的要求:客户用 C/C++编写的原生方法在执行过程中调用某种 Java 方法,此时也会导致函数 execute_java_method 的重入.

函数 execute_java_method 提供了在 C/C++代码中调用 Java 方法的能力:被调 Java 方法参数的提供者是 C/C++代码,返回值的使用者也是 C/C++代码.而 C/C++代码的执行超出了 ROPE 模型所能掌控的范围,这意味着,C/C++代码为 Java 方法提供的参数以及 C/C++代码所使用的 Java 方法的返回值都无法像在 Java 方法调用 Java 方法时那样被记录下来,所以也无法对其进行验证,更无法撤销其影响.所以,当在推测模式下发生虚拟机重入时,遇到 execute_java_method 的线程必须停下来等待确定模式.此时,对指令的解释可能已经进行了一半,所以还要将(PC,FP,SP)等恢复到解释该指令之前的状态.不过,这种恢复完全是在 C/C++层面通过编程来实现的,而不是作为 ROPE 模型的一部分.在虚拟机重入的两种情况中,后一种因为原生代码必然是在确定模式下运行(第 2.4.3 节),所以不涉及到推测执行;而在前一种情况中,当遇到 execute_java_method 调用时,线程有可能处于推测模式,此时,线程就必须停下来等待确定模式.

3.4 Java虚拟机之外的实现

Java 字节码包含的高层语意信息较多,其指令与 Java 语言的语句在一定程度上有某种对应关系.如果是编

译成与高级语言语句没有直接对应关系的更低级的指令,则可以由编译器在方法调用之前插入对特定运行时函数的调用以达到知会运行时系统的目的.

4 正确性证明

为了证明程序在 ROPE 模型下与在串行模型下具有相同的执行效果,我们引入一个处于这二者之间的第 3 种模型,即退化的 ROPE 模型.在退化的 ROPE 模型下,线程在失去确定控制后并不推测执行,而是在原地等待,所以也不会产生推测消息.退化的 ROPE 模型将作为我们证明串行模型与 ROPE 模型等价性的桥梁.

公理 1. 在初始状态相同、外部输入相同的情况下,同一段代码的两次执行具有相同的效果.

引理 1. 在外部输入相同的情况下,退化的 ROPE 模型中确定线程的执行效果与串行模型中唯一线程的执行效果相同.

证明:在退化的 ROPE 模型中,虽然是多个线程共同负责执行程序,但这些线程之间是按串行顺序进行的,所以,其执行效果与串行执行模型中唯一线程的执行效果相同. □

引理 2. 在外部输入相同的情况下,ROPE 模型中确定线程的执行效果与退化的 ROPE 模型中确定线程的执行效果相同.

证明:两种模型中,需要确定线程执行的代码都是一样的,唯一的差别在于 ROPE 执行模型中,一段将要执行的代码段可能已经在推测模式下提前执行过了,现在就要说明这段代码提前执行的效果与确定执行的效果相同.为了证明这一点,需要说明两次执行的是同一段代码,并且初始状态一样,外部输入也一样.代码段相同,这一点可由消息验证加以保证;初始状态相同,可如前面已执行过部分的效果一样加以保证;外部输入相同,则可由推测执行不作外部输入,从而外部输入必然相同来保证. □

定理 1. 在外部输入相同的情况下,程序在 ROPE 模型下与在串行模型下的执行效果相同.

证明:从引理 1 和引理 2 可知,ROPE 模型中确定线程与串行模型中唯一的线程具有相同的执行效果.又因为在串行模型下,程序的执行效果就是该唯一线程的执行效果;而在 ROPE 模型下,因为只有确定线程才能产生可见的影响.所以,程序在 ROPE 模型下的执行效果完全由确定线程决定.所以,程序在 ROPE 模型下的执行效果与在串行模型下是相同的. □

5 测 评

我们对来自 JOlden Benchmarks[10]的 5 个基准程序,在不同的输入规模与划分方案组合下进行了测试:在进行了手工标注之后,这些程序由 OpenJDK 的 Java 编译器编译生成 Java 字节码,然后在 ROPE Java 虚拟机(ROPEVM)上运行并收集性能数据.程序中所有的返回值预测方法皆为手工编写——不过,它们并不包含任何神秘的魔法代码,事实上,大多数返回值预测方法中只有一条 return 语句.

与 OpenMP 类似,利用 ROPE 对程序进行并行化要求程序员(一般为程序作者)对于程序的行为有着足够的了解.如何对程序进行标注,以及能从 ROPE 中获得多大的加速效果,这些都与程序自身有着密切的关系,所以我们一方面介绍这些程序,另一方面来介绍我们是如何对其进行并行化的,并对影响并行化的各种因素进行分析.我们采用加速比来刻画程序并行化后的加速效果.加速比定义为程序在串行模型下的执行时间与其在 ROPE 模型下的执行时间之比.加速比为 1,意味着无任何加速,小于 1,则意味着并行的开销超过了并行所带来的好处.

图 13~图 15 分别显示了这些基准程序在 ROPE 模型下所获得的加速比、推测消息的验证成功率以及各种模式占总 CPU 时间的比例.所有的基准程序用相同模式命名,即:基准程序名-划分方案编号-输入规模编号.另外,这些程序的运行过程可分为两个阶段:第 1 阶段(构造阶段)构造程序中所用的数据结构,第 2 阶段(计算阶段)进行计算.所以即便同一个程序,在不同的阶段也会表现出来不同的特质,通过 ROPE 获得的效果也不尽相同.所以在必要的时候,我们还会进一步分阶段进行分析.在进行分阶段分析时,程序名称中的最后一位如果是 a,则表示构造阶段,如果是 b,则表示计算阶段.

从图 13~图 15 的对比中我们可以发现,加速比较高的程序一般是那些验证成功率较高的程序,这些程序处

于推测模式下的执行时间也较长.不过,这种对应关系并不是严格的,如 power-1-1 与 power-1-2 的验证成功率几乎相同,但它们的加速比却相差甚大.这是因为加速比不但与验证成功率有关,还与一次成功验证背后的代码量以及程序的执行时间长短有关.另外,特别需要注意的是,这种比较只对同一个基准程序的各种配置有意义.

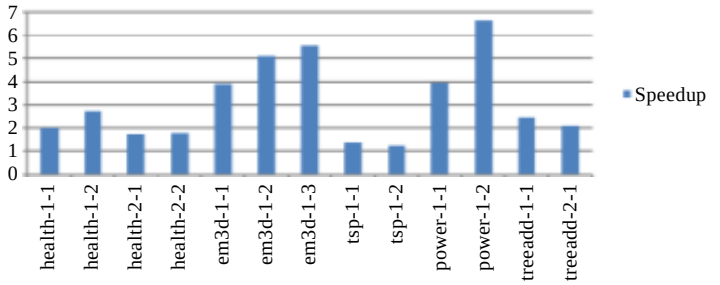

Fig.13 Speedup

图 13 加速比

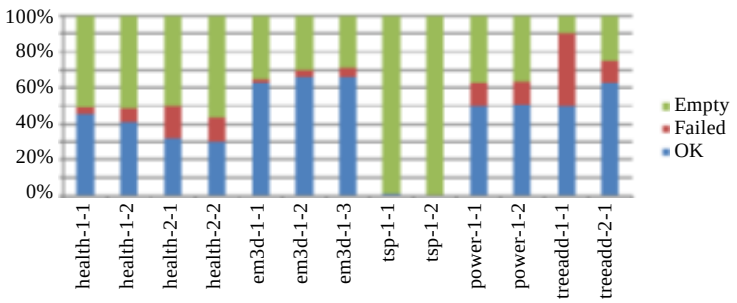

Fig.14 Verification success ratio, Empty means there are no messge to verify

图 14 验证成功率,Empty 表示验证时消息队列中无待验证消息

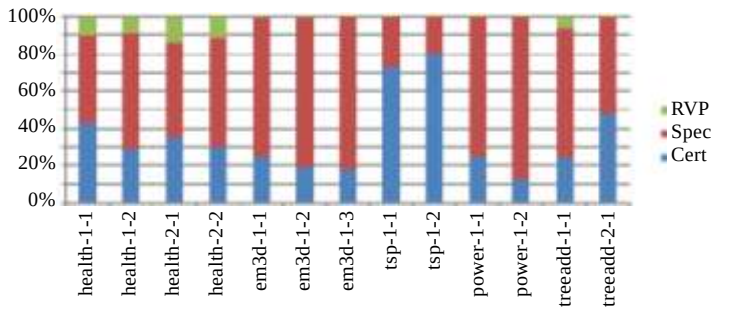

Fig.15 Breakdown of execution time in various modes

图 15 程序在各种模式下的运行时间的比例

5.1 Health

程序 Health 模拟一个多级医疗系统,该系统由处在不同层次上的多家医院构成.医院所在的社区对象会源源不断地产生病人,病人在医院排队就诊,然后由医生进行诊断,根据诊断结果,或者对其进行治疗,或者将其转往上级医院排队就诊.医院的工作流程由其所在社区对象驱动.医院与其所在社区对象之间的这种密切关系提示我们,宜将二者划分在同一个对象组中.另外,在该系统中,级别较高的医院因为要从下级医院接收转诊病人,故而活动会受下级医院的影响.为了成功地推测执行,上级医院所在线程就必须能够正确地预测出什么时候会

从哪个下级医院接收什么样的病人,而要做到这一点非常困难,所以,在这个程序中,并行性主要来源于最基层的那些医院,这些医院因为不接收转诊病人,所以其执行不会受到其他医院活动的影响,是最佳的并行对象.

我们对两种分组方案(即社区对象与医院是否同组)与两种输入规模共 4 种组合(见表 3)进行了测试.

Table 3

表 3

	同组	不同组
2 级 5 个医院	health-1-1	health-2-1
3 级 21 个医院	health-1-2	health-2-2

从图 13 我们可以看到:当医院对象与社区对象同组时,加速比较高;而且随着医院数量的增加,加速比也在增加.但把它们分开之后,不但加速比下降了,而且加速比也没有随着医院数量的增加而明显增加.这一现象可从图 14 得到解释:当医院对象与社区对象同组时,消息预测的成功率要稍高一些.

5.2 Em3d

程序 Em3d 模拟电磁波通过三维空间中物体的传播过程.代表电场和磁场的两个数组包含数量相同的节点对象.每个节点对象都持有一个数值,电场和磁场交替更新各自的节点值,而每个节点值的更新又依赖于另一个场中某几个节点的值.我们对 3 种输入规模进行了测试:em3d-1-1,em3d-1-2 和 em3d-1-3 具有相同的节点数目,但两个场之间的耦合程度(即节点依赖另一个场中节点的数目)不同.其中,em3d-1-1 的耦合度最低,em3d-1-3 最强,而em3d-1-2 处于两者之间.如图 13 所示,随着耦合度的增加,加速比非但没有下降,反而越来越高.乍看之下,这似乎与我们先前形成的“耦合越强,推测并行越不容易进行”的认识相左,但仔细分析后会发现:这是因为,在 Em3d 这个程序中,并行性主要来源于同一个场中各个节点间的并行更新,而不是不同场中节点的并行更新.从 em3d-1-1 到 em3d-1-3 越来越强的耦合度是指节点与另一个场中节点之间的关系.虽然随着这种耦合度的增加,与另一个场中节点的并行更新变得困难了,但却并未影响场内节点间的并行更新,因为同一个场中的各个节点之间并无任何依赖关系.至于加速比的上升则可解释为:由于依赖于另一个场中更多的节点,计算节点新值的代码量变多了,使得并行获得的收益与开销相比更加明显.

Em3d 的构造阶段,与其计算阶段相比,要复杂得多,以至于构造阶段几乎没有获得任何加速效果(如图 16 所示).相反地,计算阶段却获得了非常大的加速:em3d-1-3 计算阶段的加速比甚至超过了 33.

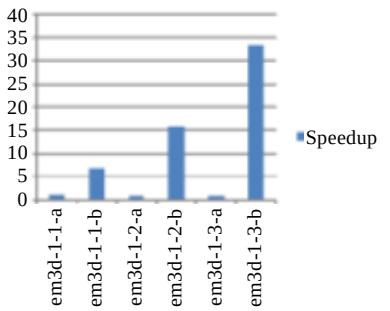

Fig.16 Construction phase and calculation phase of Em3d have significantly different speedups

图 16 Em3d 在构造阶段和计算阶段的加速比明显不同

5.3 TSP

程序 TSP 用来求解旅行推销员问题.我们对两种不同的输入规模进行了测试:tsp-1-2 的城市数量是 tsp-1-1 的两倍.这个程序的计算阶段因为各个对象之间的联系非常紧密,所以很难通过 ROPE 发掘其并行性,加速比主要来自构造阶段.如图 17 所示:tsp-1-2 在构造阶段的加速比几乎为 tsp-1-1 的两倍,差不多正比于城市数量;但在计算部分,无论 tsp-1-1 还是 tsp-1-2,都几乎没有获得任何加速.虽然相比 tsp-1-1,tsp-1-2 在构造阶段获益更多,但

随着输入规模的扩大,构造阶段的代码量并不及计算阶段的代码量增加得快.所以,虽然 tsp-1-2 构造阶段的加速比几乎为 tsp-1-1 的两倍,但是总的加速比却较 tsp-1-1 要低.

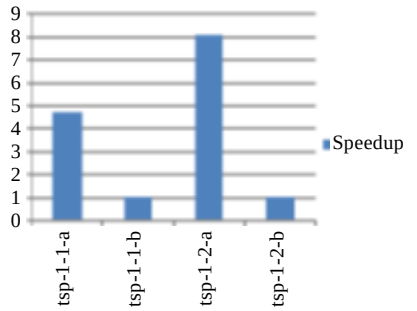

Fig.17 Acceleration of TSP almost come from the construction phase

图 17 TSP 的加速比几乎全部来自构造阶段

5.4 Power

程序 Power 用来求解电力系统定价优化问题.程序中包含变电站、主馈线、横向节点、分支节点、客户端等不同类型的对象.这些对象处在同一个树形结构的不同层次上.这个程序的特点是上层对象持有指向下层对象的引用,而下层对象却没有上层对象的引用.这意味着下层对象不依赖于上层对象,而上层对象却依赖于下层对象.让上层对象独立于下层对象自行活动并不见得有什么好处,所以我们选择赋予第 3 层上每个节点即每个横向节点一个线程,而让横向节点以下的其他对象与横向节点共享同一个线程.针对两种输入规模,我们进行了测试:power-2-1 的主馈线数目为 power-1-1 的两倍,其他配置都相同.从图 13 我们可以看到,power-2-1 的加速比接近 power-1-1 的两倍.

5.5 TreeAdd

程序 TreeAdd 用递归的方式求取二叉树上所有节点值之和:二叉树节点值之和=根节点值+左子树节点值之和+右子树节点值之和.我们测试了两种不同的分组策略:

- 在 treeadd-1-1 中,每个节点都单独成组;
- 在 treeadd-2-1 中,根节点入初始组,根节点左子树上的所有节点单独成一组,右子树上的所有节点另成一组.

从图 14 来看,节点单独成组的 treeadd-1-1 推测成功率较低.这是因为对于每个节点来说,其左子树节点值之和的计算是由其他线程负责的,所以当要用到左子树节点值之和时,它只能通过返回值预测方法进行推测,而我们绝无可能在真正计算出左子树节点值之和之前就神奇地预测出这个值来.所以,这些推测都以失败告终.所有的节点线程都如此.虽然 treeadd-2-1 中根节点在预测其左子树的节点值之和时面临同样的问题,但因其左子树上的节点都由同一个线程负责,所以这种失败只在根节点上发生 1 次.

图 13 显示,虽然推测成功率不如 treeadd-2-1,但 treeadd-1-1 的加速比却高于 treeadd-2-1.这是因为,虽然在计算阶段 treeadd-1-1 的推测成功率不及 treeadd-2-1(如图 18 所示).但在构造阶段,二者的成功率相当(如图 19 所示),而 treeadd-1-1 的线程数目远远多于 treeadd-2-1,所以获得了很高的加速比,从而弥补了计算阶段的不足.

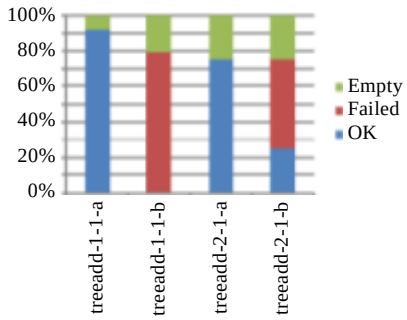

Fig.18 Verification success ratios of TreeAdd in various phases

图 18 TreeAdd 分阶段验证成功率

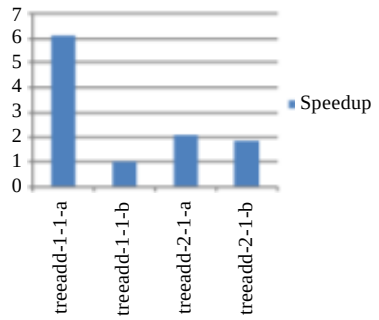

Fig.19 Speedups of TreeAdd in various phases

图 19 TreeAdd 分阶段加速比

5.6 讨 论

在我们了解了这些程序的具体内容之后,并行化看起来似乎是一件比较容易的事情.如在 Health 中,各家医院的活动很明显是可以并行的.如果我们赋予每家医院一个线程,一个包含 N 家医院的系统就会获得接近 N 的加速比.但从实验数据来看,ROPE 与此相距甚远.其中原因与一个问题有关,这就是“我们用什么标准来判断并行化之后程序的正确性”.当我们认为这样的并行化可行时,所凭借的知识实际上已经远远超出了源程序中所蕴含的信息:编译器可不知道各家医院的活动可以并行,更不知道交换两个病人的就诊次序并不意味着错误.虽然通过在程序中添加标注的方式能够告诉运行时系统“医院的活动可以并行”,但是“交换两个病人的就诊次序并不为错”这种知识却无法告诉运行时系统.也就是说,一些对于人来说显而易见的事情,对于一个自动并行化系统或者 ROPE 这样的半自动并行化系统来说,根本就是不可想象的(除非它具有人工智能).对于 ROPE 来说,对于“并行是否正确”的判断是建立在它对于“怎样才是正确的”这个问题的理解之上,而在它看来:只有与串行时一模一样,才算是正确的.ROPE 能做到的,通过人直接进行并行编程肯定也能做到,而且毫无疑问效果会更好.但人工方式要求更多的脑力投入,而且对于正确性没有保证.与其他全自动化的并行化方法相比,ROPE 的优势则在于,它只需要人通过简单的标注,提供给运行时系统一些程序之外的知识(这些知识对于人来说是显而易见),就可以对程序进行并行化.

最后有一点我们必须指出,读者也应该能意识到其必然性,那就是我们的统计数据是来源于模拟计数(我们模拟了一个具有 100 个核的系统),而非真实的秒表读数.正如我们在引言中所说,ROPE 的目标定位于未来具有大量硬件核的系统,而我们的测试工作是在一个只具有两个硬件核的 Linux 系统中进行的.虽然产生了大量的线程,但因为系统并没有足够多的硬件核来运行这些线程,多出来的线程只能按时间片来分享较少的硬件核,而在这些线程中,又有相当大一部分因为推测错误而在做无用功,所以如果用秒表去计时的话,会发现加速比不但没有提高,反而会随着线程数的增加而急剧下降.

另外,影响虚拟机性能的因素是多种多样的,限于我们有限的知识,有些因素对于最终性能的影响并未在我们的评估模型中反映出来.这些因素有些可能是有利的,如数据局部性(data locality),相信 ROPE 在这方面具有天然优势;有些则是未知的,如虚拟机实现中的其他重要加速技术与 ROPE 模型的相互影响,这种影响对于加速比的提高是有利还是有害的,我们目前并不知道.

几乎所有基于硬件的方法都设计了全新的体系结构,其衡量指标也与我们的方法完全不同.而基于软件的方法除了加速比外,大多还会考虑诸如程序员需要在多大程度上介入、在哪个级别上谈论程序语意、与现存软件系统的兼容性等更多方面的因素.此外,有些系统甚至为了测评而发明了一套全新的、特别适合自己方法的基准程序[35].而最令我们感到困难的则是,我们无法得到这些系统以进行有效的评估.因为以上这些原因,按照惯例,我们并没有将本文方法与其他推测多线程方法进行对比.这样做完全是出于公道,而非缺乏信心.不过,为

了给读者以更直观的印象,本文仍与我们自己的另一个推测多线程系统,即在控制流中进行线程划分的硬件推测多线程系统 Prophet[11,36]进行了一些对比:如图 20 所示,除了基准程序 TSP 外,ROPE 在其他几个程序上均具有更高的加速比.在输入规模增加的情况下,ROPE 更具可伸缩性,但也要求更多的硬件核.相比 Prophet 对于各种程序的变化不大的加速比,ROPE 在不同程序上的表现截然不同,这或许与 ROPE 对于高层语意信息的利用有关.当然,最后还需注意,因为缺乏相同的标准,将二者的加速比直接进行对比,其真实意义远比想象中的要小.真正值得关注的是两种方法对于不同类型程序的效果,以及对于问题规模变化的适应性.

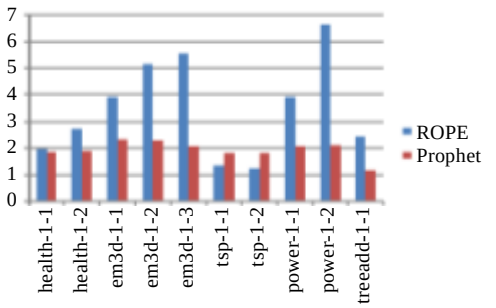

Fig.20 Speedup of ROPE and Prophet on Olden benchmarks

图 20 ROPE 与 Prophet 在 Olden 基准程序集上的加速比

6 相关工作

Actor 模型[23]是一种用于刻画并行计算的数学模型,该模型包含多个 actor 以及在 actor 之间传递的异步消息.Actor 在对消息进行响应时,可以:1) 给其他 actor 发送消息;或者 2) 创建新的 actor;或者 3) 指定如何响应下一条消息.除用作理论研究的框架之外,actor 模型还催生了若干技术实现,其中最为我们熟知的是 Active object[37],一种面向对象软件设计模式.在这种模式中,方法的调用方与执行方相互分离,运行在各自的线程中.实现 active object 模式要求程序员直接进行并行程序设计.另外,由于采用异步消息通信,程序的确定性无法得到保证.相比之下,ROPE 只要求程序员在已有的串行程序中选择一些类进行标记(某些情况下还要编写少量 RVP 方法),其工作量和难度要低得多(当然效果也要差些),而且程序员无需担心并行化会导致意想不到的错误.另外,在 ROPE 模型中,异步消息作为推测消息,最终都要经过确定的同步消息的验证,所以程序的确定性是绝对有保证的,这在某些应用环境中会是一种优势.

Galois[26,38−41]是一个线程级推测并行系统,与 ROPE 对高层语意信息的利用相比,它更进一步,它要求程序员使用一些新的语言设施来表明程序中哪些工作(一般是方法)可以并行进行.这些原本只有程序员才知道的高层信息被传递给编译器和运行时系统,用来对程序进行并行化.与 ROPE 相比,Galois 手里掌握的并行机会更多,但为了利用这些并行机会,也需要来自程序员更多的脑力劳动.而且,因为 Galois 对于是否可以并行的判断是建立在抽象数据结构的基础之上的,所以其对正确性和确定性的保证也被限制在抽象数据结构的层次上.也就说,在具体数据结构的层面上,经过 Galois 并行化后的程序,其行为既不正确也不确定.这种特点是优点还是缺点完全取决于应用环境和需求.

Prometheus[42]是一个支持多线程的 C++模板库,在所有已知的研究中,它的理念无疑是最接近 ROPE 的:将程序的数据空间划分为若干区域(domain),同一区域上的操作串行执行;不同区域上的操作并行执行.不过,二者的相似性也就到此为止了,在其他方面二者有着根本的差别:首先,Prometheus 不是一种推测多线程方法;其次,Prometheus 采用的是一种传统的主从式分派方式.因为是主从分派方式,Prometheus 要求方法的返回值必须为空(void),否则就不能继续分派操作;而且在对象 *a* 的方法中不能调用另一区域中对象 *b* 的方法,这两种限制在很大程度上扭曲了程序原有的设计,为了使用 Prometheus,程序员需要对程序进行深度的重构,而 ROPE 则完全不存在这种问题.由于 Prometheus 不属于推测并行,并行化后程序的正确性需由程序员来保证;而在 ROPE 中,正确

性是由运行时系统保证的,程序员无需担心.

Chorus 是一种全新的并行编程模型,其对象集(object assembly)[24]的概念与 ROPE 的对象组有些许相似,但要更加灵活,它是编程语言直接支持的第 1 类对象,其行为是由程序员通过编写代码来定义的.相比之下,ROPE 则要保守得多,它完全是建立在 Java 语言现有的框架之下,对象组也仅仅是对程序中已有对象的某种划分,其自身并不具备独立的行为.如果说 Chorus 的用途是进行并行程序设计的话,那么 ROPE 的用途则是对已经写好的串行程序进行并行化.

以数据为中心的思想早已有之,面向对象的程序设计便是这一思想在程序设计领域的应用.除此之外,它还在事务性内存(transactional memory)[29,43]中得到了应用.事务性内存是与推测多线程关系密切的另一大类技术,它与后者共享很多的基础技术.不同的是,事务性内存要解决的是同步问题,即如何同步并行程序中的多个线程,这些线程是程序的一部分,是程序员可以加以控制的;而推测多线程则要解决的是划分问题,即如何把一个串行程序划分为多个 SpMT 线程,这些 SpMT 线程并不是程序自身的一部分,对于程序员也是不可见的.传统的事务性内存方法是以代码为中心来同步线程的,而新的事务性内存方法如 DCS[44−46]则使用了一种以数据为中心的方式.类似地,传统的 SpMT 方法是以代码为中心来划分串行程序的,而 ROPE 则使用了一种以数据为中心的方式来做这件事.也就是说,ROPE 之于推测多线程,就像是 DCS 之于事务性内存.

Safe future[47]与 ROPE 一样实现了对象版本管理,但不同的是,在 safe future 中,对象是在线程间共享的,不同的线程使用对象的不同版本.这与 ROPE 中线程间绝不共享对象、同一对象的多个版本只属于同一个线程的做法截然相反.与 safe future 相比,ROPE 的做法应当更有利于 cache locality.另外,与 ROPE 在专门的多版本状态缓冲区中维护同一对象的多个版本不同,safe future 则在堆中以普通对象的形式维护着多个不同版本的对象,为了减少开销,safe future 采用了另一种 copy-on-write 的技术来复制对象,这一点与 ROPE 的增量修改方式很类似,不同的是,safe future 是以整个对象为单位,而 ROPE 则是以字(4 字节)为单位.这里没有优劣之分,完全是为了适合各自的需要.

Mitosis[16,48]是一个传统的 SpMT 系统,其特点在于利用预计算片段(p-slice)来提前获取发起推测线程所必须的入口数据(live-in data).预计算片段是线程激发点与被激发线程起始点之间代码的简化版本.ROPE 的返回值预测方法正是受预计算片段的启发而设计的.

Speculative Slicing[49]与 ROPE 一样,也是从串行程序中抽取出相关代码以构成线程,而不是对串行程序作顺序切割.但二者处理问题的语意层次不同:Speculative Slicing 是在较低的语意层面(内存地址、指令)上讨论并行化问题,而 ROPE 则是在较高的语意层面(对象、操作)上讨论并行化问题.另外,Speculative Slicing 线程是细粒度线程,而 ROPE 线程是粗粒度线程.在推测多线程的背景下,由于推测执行开销较大的缘故,粗粒度线程要优于细粒度线程.Speculative Slicing 在考虑划分时,是针对程序中的各个 hot region 独立进行的,而 ROPE 是将整个程序作为一个整体来考虑的.在 CMP 的架构之下,每个核都有自己的 Cache,为了高效利用这些 Cache,必须尽量让操作同一组数据的代码在同一个核上运行.然而,因为 Speculative Slicing 的各个 hot region 之间没有必然的联系,属于不同 hot region 的几个线程,即便它们是先后操作在同一组数据上,也很可能被分配到不同的核上去执行,从而导致 Cache 不命中.Speculative Slicing 依赖于运行时的 profile 信息,被划分区域必须经过至少两次才能发挥划分的效果.而 ROPE 不依赖于 profile,所以不存在这些问题.因为切片技术[50]本身固有的限制,Speculative Slicing 要对循环进行切片,必须首先将循环展开(loop unrolling),所能利用的并行性受限于编译时展开的次数.

与 ROPE 一样,Smart data structures 也意识到了关于数据结构的高层知识对于并行化的重要性.它要求程序员对数据结构之上的各种操作进行仔细的标注以提供关于它们的高层语意信息,然后利用这种语意信息来决定两个操作是否能够并行.因为没有推测执行的支持,程序员必须保证这种信息的正确性,一旦出错,会导致并行化结果错误.这对于一般的应用肯定是一个限制,所以只能用于高质量的库代码的并行化中,Smart data structures 正是这样一个精雕细琢的库,其中提供的主要数据结构多为容器类.

OOoJava[28]同样要求程序员对代码进行手工标注以暴露并行机会,即将一些可能并行执行的代码标记为 task,然后编译器进行依赖分析,运行时系统利用这些信息来各个协调 task.但与我们的做法正好相反,OooJava

是在控制流中进行标注.

Program demultiplexing[51]和国内研究者[52]在传统的沿控制流进行划分的基础上,将方法作为潜在的并行单位.这些研究显现了这样一种趋势,即为了挖掘更多的并行性,有必要更加有效地利用程序中蕴含的高层语意信息,ROPE 以对象为并行单位的做法可看作是这一趋势的自然延伸.

Prophet[11,36]是我们自己的另一个推测多线程系统,它属于传统的沿控制流进行划分的方法.其特点是通过训练输入来获取程序在运行时所走的路径的信息,然后在最有可能的路径上激发线程.Prophet 在一个 cache 中保存同一数据的多个版本的做法,深刻地影响了 ROPE 关于多版本状态缓冲区的设计.

7 结 论

在软件推测多线程环境中处理面向对象程序时,推测执行失败带来的开销是惊人的,很有可能抵销并行带来的好处,使得推测并行得不偿失.对于这类程序,为了保证推测并行的收益不会被开销所湮没,一个好的划分必须要保证两点:第一,线程之间较少发生冲突;第二,线程必须是粗粒度的.要得到不易冲突的划分,就必须将数据空间的划分作为首要问题加以考虑.然而,由于面向对象程序的复杂性,现有的程序分析技术尚无法提供我们所需的支持,于是,我们索性放弃程序分析,不再依靠程序分析技术提供的信息在较低的语意层面上来讨论数据空间的划分,而是转而依靠包含高层语意信息的数据,即数据结构,来在较高的语意层面上讨论数据空间的划分.同时,考虑到面向对象程序过程体较小、单个过程内缺少划分空间的特点,为了保证线程粒度,我们决定突破过程边界来进行划分.这两种需求综合起来,启发我们选择一条直接从数据划分出发来划分线程的道路,即并行化的第 2 种路线.目前,我们离第 2 条路线的全自动化实现尚有一定距离,还需要少量的人工介入(即需要由程序员来标注分组策略),但并行化的大部分工作都是由运行时系统自动完成的,可以说是第 2 种路线的一种半自动化实现.

7.1 创新性

我们在以下方面做出了创新:

- 高层语意层面上的数据划分与数据依赖:对象分组与消息依赖

推测多线程环境下面向对象程序对于不易冲突、粗粒度划分的迫切的需求,促使我们从整个程序数据空间的划分出发来考虑线程划分问题.考虑到目前的程序分析技术在处理非规则程序时存在的诸多限制,以及面向对象程序自身结构中所蕴含的丰富的高层语意信息,我们放弃了利用传统的程序分析技术在较低的语意层面上讨论数据划分与数据依赖从而划分线程的做法,转而利用程序中现有的数据结构(对象),辅以程序员以标注形式提供的分组策略来对程序的数据空间进行划分,然后,在运行时根据数据空间的这一划分来动态地构造线程.至于线程间不可避免的数据依赖,则被转化为线程间的消息依赖.也就是说,我们完全是在高层语意的层面上来讨论数据划分与数据依赖.这一选择为日后突破过程边界动态地构造粗粒度线程奠定了基础.

- 突破过程边界构造粗粒度线程

人们早就注意到,对于 Java 这类小方法的面向对象程序,一次只优化一个过程并没有多少优化机会,于是就有了过程间分析.但就并行化来说,目前却仍旧是一次一个过程地进行,即只在单个过程内考虑如何划分.考虑到面向对象程序的特点(即:过程体较小,对同一对象的操作可能分布在多个过程之中,而在同一个过程中又可能同时对多个对象进行操作),在过程之内进行线程划分的选择余地是较小的,这就要求我们突破过程边界的限制,在整个程序的范围内考虑划分.另外,对于推测多线程,特别是软件推测多线程来说,因为推测开销的问题,只有粗粒度线程才有利可图,本文方法因为是围绕高层语意层面的数据结构来划分线程,所以能够突破过程边界的限制构造出粗粒度线程.

- 不受程序输入变化影响的动态线程划分

之前的推测多线程划分方法都是沿着控制流进行线程划分[11].为了选择在哪条路径上进行划分,编译器一般都要依赖于采集到的 profile 信息.然而,为了获取 profile 信息必须提供给程序一定的输入,也就是说,划分其实是针对特定样本输入下的程序做出的.一旦划分好的程序在真实运行过程中获得的输入与采集 profile 信息时

所用的样本输入出现较大偏差,那么划分效果必然受到影响(输入的变化对过程内细粒度划分的影响相对较小,但对于整个程序范围内的粗粒度划分的影响却相对较大).传统的推测多线程划分方法中,线程包含哪些代码是在编译时就确定下来的,而在我们的方法中,线程包含哪些代码只有在运行的过程中才能逐步确定.具体说,在我们的方法中,线程划分包含了 3 个步骤,即:a) 编译时静态地确定数据空间的划分准则(分组策略);b) 运行时动态地划分数据空间(对象分组);c) 然后,根据数据空间的划分动态地构造线程(操作分派).输入会影响程序的控制流,但却不会影响数据结构与操作的关系,所以我们的方法不是单纯地在控制流中划分代码(那样必定会受到输入的影响),而是根据程序的数据结构来划分数据空间,然后围绕数据空间的划分根据运行时的具体情况将操作从串行的控制流中分离出来以构造不同的线程.这使得我们的方法可以不受输入变化的影响,从而对输入变化有较好的适应性.

- 利用数据局部性优化 Cache 性能

Cache 通过缓存最近使用的数据来利用时间局部性(temparal locality),通过一次性读入相邻数据来利用空间局部性(spatial locality).面向对象程序所处理的数据一般为多次动态分配得到,相互间用指针联系在一起,在空间上并不相邻,破坏了空间局部性;而面向对象程序在同一过程内同时(即短时间内)对多个对象进行操作,对同一个对象的操作分散在多个方法中的特点则又破坏了时间局部性,从而妨碍了 Cache 发挥作用(也就是说,面向对象程序本就不利于 Cache 发挥作用).就 JOlden 这套基准程序来说,约有 15%~20%的时间都耗费在 Memory Stall 上[10],为了减少 Memory Stall,就得尽量让所需的数据保持在 Cache 中.然而,Cache 的大小却是有限的,让一个大的 work set 使用一个小的 Cache,必然会导致 hit miss.在我们的方法中,因为是按照数据进行划分,每个对象的数据都由特定的处理器进行处理,极大地缩小了每个处理器的 work set,有助于提高 Cache 性能.

7.2 下一步工作

从第 5 节对实验数据的分析中我们注意到,程序在不同的阶段往往表现出不同的特点,对这样的程序进行并行化,其实相当于是在对几个完全不同的程序进行并行化,用一组不变的分组策略显然不够灵活.在下一步的研究中,我们将探索为程序的不同阶段指定并应用不同的分组策略的方式、方法,从而让划分能够更好地适应程序不同阶段的特点.

致谢 在此,我们向对本文的工作给予支持的各位老师、同学表示感谢.另外,还要特别感谢给我们提出宝贵意见的评审人.

References:

[1] Bhowmik A, Franklin M. A general compiler framework for speculative multithreading. In: Proc. of the 14th Annual ACM Symp. on Parallel Algorithms and Architectures. ACM Press, 2002. 99−108. [doi: 10.1145/564870.564885]

[2] Liu W, Tuck J, Ceze L, Ahn W, Strauss K, Renau J, Torrellas J. POSH: A TLS compiler that exploits program structure. In: Proc. of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). ACM Press, 2006. 158−167. [doi: 10.1145/1122971.1122997]

[3] Sohi GS, Breach SE, Vijaykumar TN. Multiscalar processors. In: Proc. of the 22nd Annual Int'l Symp. on Computer Architecture. ACM Press, 1995. 414−425. [doi: 10.1145/223982.224451]

[4] Hammond L, Hubbert BA, Siu M, Prabhu MK, Chen M, Olukotun K. The stanford hydra CMP. IEEE Micro, 2000,20(2):71−84. [doi: 10.1109/40.848474]

[5] Borkar S. Thousand core chips—A technology perspective. In: Proc. of the 44th Annual Design Automation Conf. ACM Press, 2007. 746−749. [doi: 10.1145/1278480.1278667]

[6] Hill MD, Marty MR. Amdahl's law in the multicore era. Computer, 2008,41(7):33−38. [doi: 10.1109/MC.2008.209]

[7] Artho C, Havelund K, Biere A. High-Level data races. Software Testing, Verification and Reliability, 2003,13(4):207−227. [doi: 10.1002/stvr.281]

[8] Du YN, Zhao YL, Han B, Li YC. A data structure centric method and execution model for partitioning sequential programs into multiple speculative threads. In: Proc. of the High Performance Computing and Communication (HPCC). IEEE, 2012. 556−563. [doi: 10.1109/HPCC.2012.81]

[9] Du YN, Zhao YL, Han B, Li YC. Optimistic parallelism based on speculative asynchronous messages passing. In: Proc. of the IEEE Int'l Symp. on Parallel and Distributed Processing with Applications (ISPA). IEEE, 2010. 382−391. [doi: 10.1109/ISPA.2010.43]

[10] Cahoon B, McKinley KS. Data flow analysis for software prefetching linked data structures in Java. In: Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT). IEEE, 2001. 280−291. [doi: 10.1109/PACT.2001.953309]

[11] Chen Z, Zhao YL, Pan XY, Dong ZY, Gao B, Zhong ZW. An overview of prophet. In: Proc. of the Int'l Conf. on Algorithms and Architectures for Parallel Processing. Berlin: Springer-Verlag, 2009. 396−407. [doi: 10.1007/978-3-642-03095-6_38]

[12] Aho AV, Lam MS, Seth R, Ullman JD. Compilers: Principles, Techniques, and Tools. 2nd ed., New York: Addison Wesley, 2007. 1−1009.

[13] Lim AW, Cheong GI, Lam MS. An affine partitioning algorithm to maximize parallelism and minimize communication. In: Proc. of the 13th Int'l Conf. on Supercomputing. New York: ACM Press, 1999. 228−237. [doi: 10.1145/305138.305197]

[14] Lim AW, Lam MS. Maximizing parallelism and minimizing synchronization with affine transforms. In: Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 1997. 201−214. [doi: 10.1145/263699.263719]

[15] Lim AW, Liao SW, Lam MS. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In: Proc. of the 8th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming. New York: ACM Press, 2001. 103−112. [doi: 10.1145/379539.379586]

[16] Madriles C, García-Quiñones C, Sánchez J, Marcuello P, González A, Tullsen DM, Wang H, Shen JP. Mitosis: A speculative multithreaded processor based on precomputation slices. IEEE Trans. on Parallel and Distributed Systems, 2008,19(7):914−925. [doi: 10.1109/TPDS.2007.70797]

[17] Pickett CJF, Verbrugge C. SableSpMT: A software framework for analysing speculative multithreading in Java. ACM SIGSOFT Software Engineering Notes, 2006,31(1):59−66. [doi: 10.1145/1108768.1108809]

[18] Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R. Parallel Programming in OpenMP. Morgan Kaufmann Publishers, 2000. 1−231.

[19] Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou YL. Cilk: An efficient multithreaded runtime system. SIGPLAN Notices, 1995,30(8):207−216. [doi: 10.1145/209937.209958]

[20] Karmani RK, Shali A, Agha G. Actor frameworks for the JVM platform: A comparative analysis. In: Proc. of the 7th Int'l Conf. on Principles and Practice of Programming in Java. New York: ACM Press, 2009. 11−20. [doi: 10.1145/1596655.1596658]

[21] Srinivasan S, Mycroft A. Kilim: Isolation-Typed actors for Java. In: Proc. of the 22nd European Conf. on Object-Oriented Programming (ECOOP). Springer-Verlag, 2008. 104−128. [doi: 10.1007/978-3-540-70592-5_6]

[22] Haller P, Odersky M. Event-Based programming without inversion of control. In: Lightfoot D, Szyperski C, eds. Proc. of the JMLC 2006. LNCS 4228, Berlin, Heidelberg: Springer-Verlag, 2006. 4−22.

[23] Hewitt C. Viewing control structures as patterns of passing messages. Artificial Intelligence, 1977,8(3):323−364. [doi: 10.1016/0004-3702(77)90033-9]

[24] Lublinerman R, Chaudhuri S, Černý P. Parallel programming with object assemblies. In: Proc. of the 24th ACM SIGPLAN Conf. on Object oriented Programming Systems Languages and Applications (OOPSLA). New York: ACM Press, 2009. 61−80. [doi: 10.1145/1640089.1640095]

[25] Upadhyaya G, Midkiff SP, Pai VS. Using data structure knowledge for efficient lock generation and strong atomicity. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). New York: ACM Press, 2010. 281−292. [doi: 10.1145/1837853.1693490]

[26] Kulkarni M, Pingali K, Walter B, Ramanarayanan G, Bala K, Chew LP. Optimistic parallelism requires abstractions. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2007. 211−222. [doi: 10.1145/1250734.1250759]

[27] Pickett CJF, Verbrugge C. Software thread level speculation for the Java language and virtual machine environment. In: Proc. of the 18th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC). Berlin, Heidelberg: Springer-Verlag, 2006. 304−318. [doi: 10.1007/978-3-540-69330-7_21]

[28] Jenista JC, Eom YH, Demsky B. OoOJava: Software out-of-order execution. In: Proc. of the 16th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP). New York: ACM Press, 2011. 57−67. [doi: 10.1145/1941553.1941563]

[29] Grahn H. Transactional memory. Journal of Parallel and Distributed Computing, 2010,70(10):993−1008. [doi: 10.1016/j.jpdc.2010.06.006]

[30] Rossbach CJ, Hofmann OS, Witchel E. Is transactional programming actually easier? In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). New York: ACM Press, 2010. 47−56. [doi: 10.1145/1837853.1693462]

[31] Zhang C, Ding C, Gu XM, Kelsey K, Bai TX, Feng XB. Continuous speculative program parallelization in software. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). New York: ACM Press, 2010. 335−336. [doi: 10.1145/1837853.1693501]

[32] Lindholm T, Yellin F, Bracha G, Buckley A. The Java™ virtual machine specification. JSR-000924, ORACLE, 2012. 1−586.

[33] Codrescu L, Wills DS. On dynamic speculative thread partitioning and the MEM-slicing algorithm. In: Proc. of the '99 Int'l Conf. on Parallel Architectures and Compilation Techniques. New York: IEEE, 1999. 40−46. [doi: 10.1109/PACT.1999.807404]

[34] Michael MM, Vechev MT, Saraswat VA. Idempotent work stealing. SIGPLAN Notices, 2009,44(4):45−54. [doi: 10.1145/1594835.1504186]

[35] Kulkarni M, Burtscher M, Caşcaval C, Pingali K. Lonestar: A suite of parallel irregular programs. In: Proc. of the 2009 IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS). New York: IEEE, 2009. 65−76.

[36] Li YC, Zhao YL, Li MR, Du YY. Thread partitioning algorithm for speculative multithreading based on path optimization. Ruan Jian Xue Bao/Journal of Software, 2012,23(8):1950−1964 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4148.htm [doi: 10.3724/SP.J.1001.2012.04148]

[37] Lavender RG, Schmidt DC. Active Object: An Object Behavioral Pattern for Concurrent Programming. Addison-Wesley Longman, 1996. 483−499.

[38] Kulkarni M, Burtscher M, Inkulu R, Pingali K. How much parallelism is there in irregular applications? In: Proc. of the 2009 ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). ACM Press, 2009. 3−14. [doi: 10.1145/1594835.1504181]

[39] Kulkarni M, Carribault P, Pingali K, Ramanarayanan G, Walter B, Bala K, Chew LP. Scheduling strategies for optimistic parallel execution of irregular programs. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA). ACM Press, 2008. 217−228. [doi: 10.1145/1378533.1378575]

[40] Kulkarni M, Pingali K. Scheduling issues in optimistic parallelization. In: Proc. of the 21st Int'l Parallel and Distributed Processing Symp. (IPDPS). New York: IEEE Computer Society, 2007. 301−307.

[41] Kulkarni M, Pingali K, Ramanarayanan G, Walter B, Bala K, Chew LP. Optimistic parallelism benefits from data partitioning. In: Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York: ACM Press, 2008. 233−243. [doi: 10.1145/1346281.1346311]

[42] Allen MD, Sridharan S, Sohi GS. Serialization sets: A dynamic dependence-based parallel execution model. In: Proc. of the 2009 ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). ACM Press, 2009. 85−95. [doi: 10.1145/1594835.1504190]

[43] Larus J, Kozyrakis C. Transactional memory. Synthesis Lectures on Computer Architecture, 2007,1(1):1−226.

[44] Ceze L, von Praun C, Cascaval C, Montesinos P, Torrellas J. Concurrency control with data coloring. In: Proc. of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC). New York: ACM Press, 2008. 6−10. [doi: 10.1145/1353522.1353525]

[45] Ceze L, Montesinos P, von Praun C, Torrellas J. Colorama: Architectural support for data-centric synchronization. In: Proc. of the 13th Int'l Symp. on High Performance Computer Architecture (HPCA). New York: IEEE Computer Society, 2007. 133−144. [doi: 10.1109/HPCA.2007.346192]

[46] Vaziri M, Tip F, Dolby J. Associating synchronization constraints with data in an object-oriented language. SIGPLAN Notices, 2006,41(1):334−345. [doi: 10.1145/1111320.1111067]

[47] Welc A, Jagannathan S, Hosking A. Safe futures for Java. In: Proc. of the 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). New York: ACM Press, 2005. 439−453. [doi: 10.1145/1094811.1094845]

[48] García-Quiñones C, Madriles C, Sánchez J, Marcuello P, González A, Tullsen DM. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2005. 269−279. [doi: 10.1145/1065010.1065043]

[49] Wang C, Wu YF, Borin E, Hu SL, Liu W, Sager D, Nagai T, Fang J. Dynamic parallelization of single-threaded binary programs using speculative slicing. In: Proc. of the 23rd Int'l Conf. on Supercomputing (ICS 2009). ACM Press, 2009. 158−168. [doi: 10.1145/1542275.1542302]

[50] Weiser M. Program slicing. IEEE Trans. on Software Engineering, 1984,SE-10(4):352−357.

[51] Balakrishnan S, Sohi GS. Program demultiplexing: Data-Flow based speculative parallelization of methods in sequential programs. In: Proc. of the 33rd Int'l Symp. on Computer Architecture (ISCA). IEEE Computer Society, 2006. 302−313. [doi: 10.1109/ISCA.2006.31]

[52] Liang B, An H, Wang L, Wang YB. Exploring of speculative thread-level parallelism from subroutine. Journal of Chinese Computer Systems, 2009,30(2):230−235 (in Chinese with English abstract).

附中文参考文献:

[36] 李远程,赵银亮,李美蓉,杜延宁.一种基于路径优化的推测多线程划分算法.软件学报,2012,23(8):1950−1964. http://www.jos.org.cn/1000-9825/4148.htm [doi: 10.3724/SP.J.1001.2012.04148]

[52] 梁博,安虹,王莉,王耀彬.针对子程序结构的线程级推测并行性分析.小型微型计算机系统,2009,30(2):230−235.

杜延宁(1978－),男,甘肃镇原人,博士生,主要研究领域为并行计算,体系结构,编程模型.
E-mail: duyanning@gmail.com

韩博(1975－),男,博士生,高级工程师,主要研究领域为软件编程方法学,信息管理学.
E-mail: bohan@xjtu.edu.cn

赵银亮(1960－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为程序语言及编译技术,并行计算,机器学习.
E-mail: zhaoy@mail.xjtu.edu.cn

李远成(1981－),男,博士,讲师,主要研究领域为并行计算,体系结构,机器学习.
E-mail: yuanch_li@163.com