第 36 卷 第 9 期 计 算 机 工 程 2010 年 5 月
Vol.36 No.9 Computer Engineering May 2010
·软件技术与数据库· 文章编号：1000—3428(2010)09—0035—03 文献标识码：A 中图分类号：TP311

选择性循环的并行方法

吴 悦，雷超付，杨洪斌
(上海大学计算机工程与科学学院，上海 200072)

摘 要：针对含有大量循环的串行程序存在的问题，提出一种基于线程级前瞻技术的循环选择方案。该方案对循环进行最优选择后建立一个可并行运行的循环集。对于该集合中的循环，选择并行效率高的代码段作并行处理，以加快串行程序运行速度。实验表明，相对于一般的简单内部循环或外部循环并行方法，该方案使 9 种基准代码的加速比平均上升 23.8%，从而提高串行程序并行运行的效率。

关键词：线程级前瞻；循环选择；并行运行；单片多核处理器

Parallel Method for Selective Loop

WU Yue, LEI Chao-fu, YANG Hong-bin
(School of Computer Engineering and Science, Shanghai University, Shanghai 200072)

【Abstract】 Aiming at the sequential codes containing a large number of loops, a new method for parallelism based on Thread-Level Speculation (TLS) by loop selection is proposed. After optimal selection of loops, a loop set is built for parallel operation. The codes of loops which are included in the loop set are selected for parallel operation because of high parallelizing efficiency to improve the speed of serial program operation. Compared with simple inner or outer loop parallelization, simulation results demonstrate the average speedup of 9 benchmark applications raise 23.8%. Therefore, higher parallelizing efficiency can be achieved.

【Key words】 Thread-Level Speculation(TLS); loop selection; parallel operation; Chip Multi-core Processors(CMP)

1 概述

采用提高主频和复杂的指令流水线技术已经很难进一步提高处理器的性能，提升处理器主频也会导致高功耗、散热等棘手问题。在此背景下，单片多核处理器(Chip Multi-core Processors, CMP)技术成为最受关注的新技术之一。虽然并行编译已经研究多年，但是大部分已有的代码经过并行编译后依然是顺序执行，很少有代码能够自动并行运行。若在保证程序语义不变的条件下，将串行程序划分为若干存在相关性的并行线程，那么能够充分利用 CMP 技术并行的优点。线程级前瞻(Thread-Level Speculation, TLS)[1]技术是当前解决串行划分技术的主要手段之一。

由于程序的执行时间 80%都花在循环上[2]，而且循环语句能够很好地体现程序的并行性[3-7]，因此使用 TLS 技术研究含有大量循环的串行程序已越来越重要。但并不是所有的循环都适合并行运行[8-9]，相当一部分循环程序存在动态指令数目较少[3]、循环结束不确定[4]等问题，这些循环在并行执行时造成线程之间通信开销过大或核资源的浪费等不利影响，以至于其运行时间反而比直接串行还长。若这些并行性差的循环体采用 TLS 技术并行执行，则会降低整个程序运行的性能。对于含有多级嵌套循环的程序，一般的方法是展开最内层或最外层循环[5]，但按照这样的方法很难保证整个循环代码运行性能最优。因此，选择并行效率高的循环作并行处理是提高串行程序并行高速执行的关键技术之一。在循环直接展开时，经常会遇到循环次数不确定[5]、循环迭代之间存在较强依赖关系[10]等问题。这些含有不确定数据相关性的循环代码不能直接作并行处理，否则，需花费大量开销代价。若对这些循环进行相关分割[10]，选择循环中并行效率高的代码段作并行处理，可以降低程序的整体运行时间，从而进一步地提高串行程序并行性。

2 循环选择

2.1 循环选择准则

含有大量循环的串行程序中含有少量的并行效率低的循环[8]，这些循环并行运行时会降低整个程序运行的性能。在并行队列中有效地去除这些循环可提高整个程序运行的效率。本文基于以下较通用的循环选择准则[9]:

(1)循环体中的动态平均指令数不得少于 100 条，这类循环更适合指令级运行。

(2)循环体的平均迭代数不得少于处理器内核数，否则会导致核资源不必要的浪费。

上述循环选择准则只考虑了循环的指令数和迭代数，没有考虑循环迭代之间的依赖性，因此，在此基础上增加一条新的选择准则，即循环体中的 break, continue 等中断语句占循环体总语句比例不得超过 10%，若超过 10%会导致推测循环结束[4]异常困难，产生大量不必要的线程和额外开销。

2.2 循环图

本文选择性循环方案的第 1 步是把源程序转换成循环图，再转换成相应的循环树。以下含有循环的代码(例 1)用于说明源程序转换循环图和循环树的方法。

例 1

```
main(){
for1:        for(j=0;j<20;j++){
               loo();
```

基金项目：上海市重点学科建设基金资助项目(J50103)
作者简介：吴 悦(1960－)，女，教授，主研方向：软件测试；雷超付，硕士研究生；杨洪斌，副教授、硕士
收稿日期：2009-10-08 **E-mail：**ywu@shu.edu.cn

```
                doo();
                work();
                  …
                }          }
        loo(){
for2:           for(i=0;i<20;i++){
                doo();
                work();
                }          }
        doo(){
for3:           for(j=0;j<10;j++)
                        …          }
        work(){
for4:           for(j=0;j<10;j++)
                        …          }
```

例 1 中的每个循环作为一个节点，循环之间的直接嵌套关系用一条有向边表示，形成如图 1 所示的循环图。其中的循环 for3 被 2 个循环(for1, for2)直接嵌套，将 for3 复制成 2 个副本(for3_A , for3_B)，分别被循环 for1, for2 直接嵌套。将该方法应用到图中所有被多个直接嵌套关系的循环节点，若循环节点有子树，其子树也递归地执行副本复制操作，最终形成如图 2 所示的循环树。

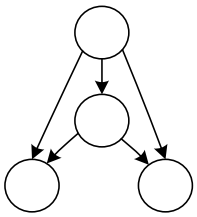

图 1　循环图

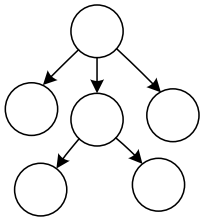

图 2　循环树

2.3　循环节点可并行度计算

覆盖率、加速度、迭代数是衡量循环并行性程度和并行效率的 3 个重要因素。其中，覆盖率是指循环代码与整个程序代码的比值；加速度表示循环理想状态下并行运行相对串行运行的加速比；迭代数是指重复执行循环体的次数。给每个循环设定一个可并行度(w)，w 越大，循环并行效率越高。其公式定义如下：

$$w = cov \times (1 - 1/p) \times \sqrt{speedup^2 + 1} \tag{1}$$

其中，cov 表示覆盖率；$speedup$ 表示循环加速度；p 表示循环迭代的次数。cov 和 p 在通过硬件检测循环得到的循环 profile 信息中获得，$speedup$ 由通过检测循环的并行时间和串行时间确定。由式(1)推导可得到整个程序加速度(*total speedup*)，直接可由 w 和 p 表示，其公式如下：

$$total\ speedup = \sqrt{\frac{w^2 \times p^2}{(1-p)^2} - 1} \tag{2}$$

2.4　选择方案

遍历 2.2 节生成的循环树，根据 2.1 节循环选择准则对循环树节点进行判定，不符合条件的循环节点直接放入串行运行队列中，遍历所有的节点后，生成多个独立的循环子树或单独的循环节点，组成集合 T，定义如下：

$$T = \{T_i \mid T_i \text{为符合条件的独立循环子树}\} \tag{3}$$

定义一个并行运行循环集 P，集合 P 中的每个元素都是一个循环。利用贪心算法逐个选取集合 T 中的元素 T_i 并移到循环集 P 中，其规则如下：

$$P_i = \begin{cases} T_i & T_i\text{是单个循环节点} \\ (T_i)_{\max} & T_i\text{不是单个循环节点} \end{cases} \tag{4}$$

其中，$(T_i)_{\max}$ 是对循环树 T_i 用深度优先算法搜索后得到的并行度最大的循环节点，若最大可并行度不唯一，按照循环节点深度升序优先原则进行选择。取出循环节点后集合 T 更新为 T'，更新规则如下：

$$T_i' = \begin{cases} \phi & T_i\text{是单个循环节点} \\ (T_i)_{\text{rest}} & T_i\text{不是单个循环节点} \end{cases} \tag{5}$$

其中，$(Ti)_{\text{rest}}$ 是对 T_i 循环树去掉可并行度最大循环节点后的部分，可能是多个子循环树或循环节点。

利用式(4)对集合 T 中所有元素进行提取，得到可并行度总值最大集合 P。且 P 中任意 2 个循环之间不存在任何嵌套关系，确保了 P 中任意 2 个循环不存在相关性，可以并行运行。集合 T 根据式(5)更新为集合 T'，对集合 T' 中的循环节点串行执行。通过这种选择方案，可使 w 值较小的循环在串行运行的同时保证 w 值较大的一组循环直接并行运行，使程序的整体性能得以提高，从而缩短整个程序的执行时间。

3　循环代码并行化方案

通过 2.4 节确定的 P 可能含有结构不好的循环，大部分结构不好的循环存在不确定性的数据相关性[3]，严重阻碍了线程的直接并行，又因为这些循环的 w 值较大，所以直接串行运行必然导致核资源的利用不充分。本文将循环的代码划分为两部分：一部分代码进行串行运行，另一部分代码并行运行，并作相关的前瞻失效处理，从而进一步提高串行程序运行的效率。

3.1　相关参数

设定程序按照指令执行可从 profile 中得到以下参数：

IS: 循环开始的位置；

CS: 循环运行的指令数；

LS: 循环运行的迭代位置；

XBEGIN: 循环开始的指令标志；

XCOMMIT: 循环提交完成的指令标志。

3.2　循环代码选择示例

本节以一个简单的循环示例说明循环代码选择的方案。源代码如例 2 所示。

例 2

```
1: while(node){
2: function(node);
3: hide(node);
4: node=node->next;        }
```

例 2 中不能确定语句 2 与语句 4 及语句 3 与语句 4 是否存在数据相关，因此，不能作简单的并行处理。例 2 中的程序代码可转换的对应循环数据依赖图见图 3。

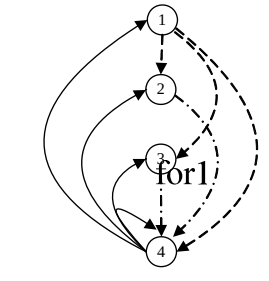

图 3　循环数据依赖图

3.3 并行化方案

对循环中的代码进行分割，把代码划分为两部分：一部分代码串行运行，另一部分并行运行。划分后的程序如下：

代码 1

```
//Sequential
    int count =0;
 1: while(node){
 5:     node_array[count++]=node;
 4:     node=node->next;
}
```

代码 2

```
//Parallel
 XBEGIN
    node = node_array[IS];
    int i=0 ;
 1': while(node && i++<CS){
 2:         fuction(node);
 3:         hide(node);
 4':        node=node->next;
          }
    if(node !=node_array[IS+CS])
    end_higher_iter_Treads();
    XCOMMIT
```

其中，代码 1 为串行运行代码，代码 2 为并行运行代码。与代码对应的循环数据依赖图如图 4 所示。代码 1 是对并行划分的代码作一些相关的外部代码优化，使其变成能够并行运行的代码。由代码 1 用一个节点数组 node_array 保存 node 的值，在代码 2 中对并行程序部分作了相应前瞻失效的检测：当节点 node 的值与节点数组值不相同时，终止后面运行的前瞻线程，从前瞻失效的位置 LS 用正确值赋值给前瞻推测值。这样既保证了程序执行的语义不变，又提高了并行运行的效率。

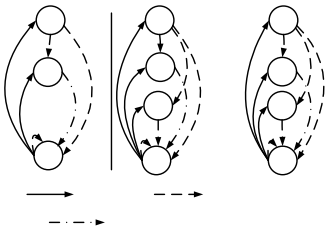

图 4　分割后的循环数据依赖图

4　实验及结果分析

4.1　实验基准代码

本文在 Linux 平台上采用 SimpleScalar 模拟工具和 Intel C++ 9.1 编译器，基于双核实验平台，选择 SPEC2000[11]的 9 个标准源代码进行测试。每个测试基准代码的特征参数如表 1 所示。

表 1　实验测试基准代码的特征参数

基准代码	循环数	平均迭代数	最大循环嵌套数
crafty	420	59 775	10
twofty	899	12 437	7
gzip	178	206 755	6
vortex	212	45 179	7
vpr	401	1 500	5
paser	532	8 820	10
gap	1 655	53 721	10
gcc	2 619	5 394	10
perlbmk	729	2 826	10

4.2　实验方法

本文所有模拟实验用 ref 输入集[12]进行测试，每个循环并行地模拟一次，通过本选择方案后，直接用模拟结果计算整个程序集，这样就避免了一个循环被不同选择方案选中后多次运行。

4.3　实验结果及分析

通过对不同的测试基准代码分别采取选择性循环的方案与一般的 TLS 方案[9]进行测试比较。选择性循环方案的实验数据如图 5 所示。对于多级嵌套循环，一般的 TLS 方案采用展开最内层或最外层的循环。例如基准代码 gzip 最外层循环的加速比只有 60%，但经过循环选择的方案后，其加速比可达到 105.8%。实验结果表明，在进行 TLS 调度时，采用选择性循环方案后 9 个标准源代码的平均加速比提高了 23.8%。

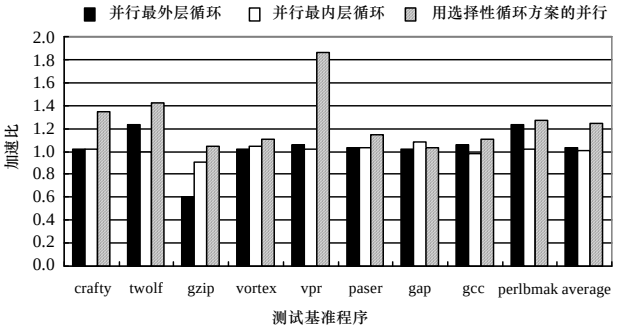

图 5　3 种方案的加速比

5　结束语

本文对含有大量循环的基准代码采用循环选择方案进行测试。选择适合并行运行的循环进行并行运行，选中的循环代码中数据依赖性不确定的循环分成两部分：一部分直接串行运行，另一部分经过适当的外部优化后并行运行。本文进行了 2 次选择，第 1 次是对循环的选择，第 2 次对循环中的代码进行选择，2 次选择后并行与串行有机结合运行程序。实验结果表明，该方案能够更好地提高串行程序并行运行的加速比，缩短串行程序的运行时间。

参考文献

[1] Johnson T A, Eigenmann R, Vijaykumar T N. Speculative Thread Decomposition Through Empirical Optimization[C]//Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, USA: ACM Press, 2007: 205-214.

[2] 邓之刚, 曾国荪, 周　静. 一种非可规约循环的投机并行方法[J]. 计算机工程与科学, 2007, 29(10): 135-138.

[3] de Alba M R, Kaeli D R. Runtime Predictability of Loops[C]//Proc. of IEEE International Workshop on Workload Characterization. Washington D. C., USA: IEEE Press, 2001: 91-98.

[4] Mafijul I M. Predicting Loop Termination to Boost Speculative Thread-level Parallelism in Embedded Applications[C]//Proc.of the 19th IEEE Int'l Workshop on Computer Architecture and High Performance Computing. Aizu, Japan: IEEE Press, 2007: 54-61.

[5] Mafijul I M, Busck A, Engbom M, et al. Limits on Thread-level Parallelism in Embedded Applications[C]//Proc. of the 11th IEEE Int'l Workshop on Interaction Between Compliers and Computer Architectures. Phoenix, Arizona, USA: IEEE Press, 2007: 40-49.

（下转第 40 页）