第44卷 第12期 西 安 交 通 大 学 学 报 Vol.44 №12
2010年12月 JOURNAL OF XI′AN JIAOTONG UNIVERSITY Dec. 2010

面向片上多核处理器的推测多线程机制下的独立栈模型

韦远科，赵银亮，宋少龙，王旭昊，阴培培，李挺
（西安交通大学计算机科学与技术系，710049，西安）

摘要：在推测多线程并行执行时，各线程中借助共享栈来实现函数调用的过程存在互斥访问问题，为此提出了一种采用片上多处理器(CMP)的推测多线程机制下的独立栈函数调用模型，核栈采用一对一方式，栈之间的有机配合支持推测多线程中的函数调用.通过在模拟器端添加两条指令实现栈空间的管理，从而消除了共享栈模型中的加锁问题.为了保证程序正常运行，编译器对生成的代码作相应的调整，模拟器方面则增添了 get 和 update 两条指令，以便管理相应的栈空间.因为独立栈函数调用方法消除了共享栈模型中的栈加锁问题，使得成功线程发起的数目有不同程度的提高，从而提高了程序的并行加速比.Olden 基准程序的测试表明，独立栈模型相对于共享栈模型使程序的平均并行加速比提高了 3.85%.但是，由于影响程序推测并行加速比的因素复杂，某些测试程序也出现了独立栈的加速比低于共享栈的情况.

关键词：推测多线程；函数调用栈；代码生成；存储管理
中图分类号：TP302 **文献标志码**：A **文章编号**：0253-987X(2010)12-0010-06

A Separate Stack Model in Speculative Multithreading Based on Chip Multi-Processor

WEI Yuanke, ZHAO Yinliang, SONG Shaolong, WANG Xuhao, YIN Peipei, LI Ting
(Department of Computer Science and Technology, Xi′an Jiaotong University, Xi′an 710049, China)

Abstract: Function calls in threads adopt the shared stack model in speculative multithreading based on Chip Multi-Processor. A separate stack model is presented to eliminate the locking problem in the shared stack model, in which one core is associated with one stack on a one-to-one basis and the cooperation of those stacks supports the function calls in speculative multithreading. The stack memory management is achieved by means of adding two instructions to the speculative multithreading simulator so that the locking problem can be effectively eliminated. The compiler is modified to generate appropriate code to ensure the semantic correctness of the program, and the two instructions, get and update, are added to the simulator that supports speculative multithreading to manage the corresponding stack memory. Since the locking problem in the separate stack model is eliminated, the number of determinate threads spawned increases in various degrees and the parallel speedup of the program is increased. Experimental results show that the average speedup of the Olden benchmark suite is increased by 3.85%. However, since factors that affect the parallel speedup are complex, there are some cases that the parallel speedup decreases in the separate stack model.

Keywords: speculative multithreading; function call stack; code generation; memory management

发掘程序中的并行性是提高程序执行性能的有效方法.传统的处理器主要通过超标量、超长指令字

收稿日期：2010-06-23. 作者简介：韦远科(1985—)，男，硕士生；赵银亮(联系人)，男，教授，博士生导师. 基金项目：国家高技术研究发展计划资助项目(2008AA01Z136).

和流水线来发掘程序指令级的并行性，以提高程序的执行性能，但是受访存延迟、指令间依赖关系和集成电路制造工艺等方面的影响，通过指令级并行来提高程序执行效率变得愈发困难，而且这种通过硬件的改进来改善程序执行效率的办法可扩展性差，没有从根本上解决现实中人们对计算资源的需求.

为了充分利用目前处理器中的多核资源，研究人员用更加激进的方式在线程级并行的基础上提出了推测多线程(Speculative Multithreading，SpMT)[1-2]这一概念. 当前，面向 CMP(Chip Multi-Processor)的支持推测多线程机制的模拟器采用了基于共享栈的函数调用栈模型. 在这种栈模型中，所有的处理器单元共享同一个栈空间，函数调用活动记录的申请和释放都在同一个栈上进行. 为了保证多个处理器单元上函数调用的正常进行，活动记录空间的申请必须对栈空间进行加锁，这在一定程度上限制了线程发起的数量，从而影响了程序最终的并行加速比.

本文提出了一种面向独立栈空间的函数调用栈模型. 通过对推测多线程各个执行阶段的分析，本文给出了独立栈模型的详细设计，并在已有的 Prophet 系统上进行了实现. 独立栈模型消除了共享栈模型中的栈加锁问题，提高了程序的并行加速比.

1　推测多线程概述

推测多线程下，串行程序首先被编译器划分为一些可以并行运行的线程，各个线程可以在不同的处理器单元上并行地运行. 与传统线程级并行不同的是，这些划分出来的线程之间可以存在数据依赖和控制依赖，而程序串行语义的正确性则交由底层的硬件根据相应的机制进行保证. 正是由于推测多线程技术在线程划分自由度上的放开，那些原本因为数据依赖、控制依赖无法被划分为多线程执行的程序现在也可以并行执行，程序潜在的并行性得到了发掘.

推测多线程中，线程由线程发起点和准控制无关点进行标识. 如图 1 所示，SP 是线程的发起点，而 CQIP 点是线程的分隔点，它既是前一个线程的结束点，也是后一个线程的开始点. 当程序执行到 SP 点的时候，会发起一个新的线程并行推测执行 CQIP 点之后的线程. 在推测执行的过程中，执行模型会检测推测线程的运行情况，如果线程推测正确，没有发生控制和数据依赖，那么推测线程将一直执行直到自身的 CQIP 点. 如果推测失败，该线程在当前的状态下会被重新执行，以获取正确的执行结果. 推测执行的多个线程其实是串行代码的不同部分，因此在逻辑上有着一定的顺序关系. 在程序中，逻辑顺序最早的线程是以确定方式执行的，它在执行的过程中会发起其他的线程，而新发起的推测线程也还会发起其他的推测线程. 在程序执行的整个过程中，确定执行的线程只有一个，它的执行不会受到其他线程的影响，它的执行结果是直接提交到内存里的. 当确定执行的线程结束时，它会去检测相应的推测线程. 如果该推测线程推测正确，那么它将转为确定执行，成为程序中新的确定线程. 如果该推测线程推测失败，那么它将会被撤销，相关的资源也会被释放，其后的代码将由确定线程继续往下执行.

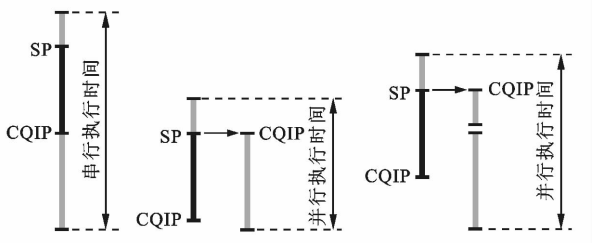


图 1　推测多线程模型

2　推测多线程下的函数调用关系

传统的程序运行方式中，程序在某一时刻只有一个函数处于运行状态，因此函数的调用返回关系可以通过一个先进后出的栈结构进行维护. 但是在推测多线程执行方式下，系统中同时运行着多个线程，而每个线程中都有可能出现函数调用，因此同一时刻就会有多个函数调用的活动记录，这里传统的栈模型不再适用. 在推测多线程执行方式下，多个处理器单元将分别运行输入程序代码的不同部分，各个处理器单元上的函数活动记录的关系在逻辑上将呈现复杂的树形链接关系. 如图 2 所示，假设线程单元 TU0 正在执行函数 func 的代码，当其碰到线程发起指令“spawn cqip1”时，如果推测线程成功发起，且新发起的线程单元为 TU1. 那么 TU1 将推测地执行 cqip1 指令之后的代码，此后 TU0 和 TU1 将并行执行. 当它们执行到函数调用的时候，将分别为 bar 函数和 foo 函数建立相应的活动记录，此时 func 函数、bar 函数和 foo 函数的活动记录将呈现如图 3 所示的关系，也就是由于 TU1 的推测执行，导致函数活动记录的链接关系增加了一个新的由 foo 到 func 的推测分支. 其中，推测执行的线程单元 TU1 仅对自己的活动记录拥有读写权限，而对于其他的活动记录，如 func 函数的活动记录，为了保证程序执行的正确性，它只有读权限. 类似地，bar 函数和 foo 函数的执行中也有可能碰到函数

调用并发起到空闲的处理器单元上运行,产生新的函数活动记录,从而向图中添加新的分支.

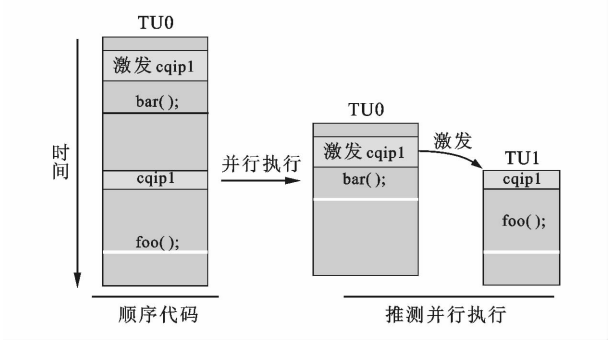


图 2 推测多线程代码的执行

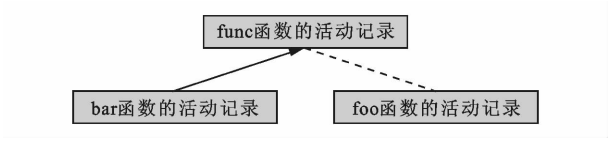


图 3 图 2 代码运行对应的活动记录关系

从全局看,在推测多线程方式下,整个程序运行过程中函数活动记录的关系将是一个树形链接关系,并且这个树形结构具有如下的特点.

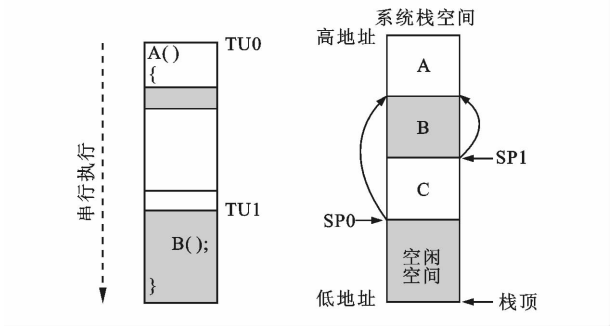
(1)树的根节点为程序入口函数的活动记录,如在 C 语言中,根节点为 main 函数的活动记录.

(2)树的根节点到叶节点的路径,代表了当前程序的执行情况. 树的主干为确定执行的线程所代表的函数活动记录关系,只有一个,而各个分支则都是以推测方式执行的函数活动记录的反映. 当推测线程被验证为正确时,推测分支转变为新的树的主干,代表着当前程序的执行流向;当推测线程被验证为失败时,整个推测分支将被剪掉,从而忽略推测失败线程产生的结果.

(3)同一时刻,叶节点或者分支的最大数目为系统中处理器单元的数目.

3 共享栈模型的缺陷

共享栈模型[3]中,各个处理器单元共享同一个栈空间,所有处理器单元所需的函数活动记录的建立和释放都在同一栈空间上进行. 如图 4a 所示的一段代码,函数 A 刚开始运行时由 TU0 在栈空间上分配 A 函数活动记录所需的空间. 紧接着,TU0 碰到线程发起指令 spawn,假设此时线程成功发起,那么 TU0 将发起 TU1 执行 A 函数的后一部分代码,此后 TU0 和 TU1 将并行执行. TU1 在执行的过程中将碰到函数调用 B,它将会在栈空间中为 B 函数分配活动记录. TU0 在执行的过程中也会碰到函数调用 C,同样 C 函数的活动记录也会在栈空间上分配. 这种情况形成了如图 4b 所示的栈空间布局.



(a)代码示意图 (b)堆栈示意图

图 4 共享栈的函数活动记录关系

本质上,共享栈是在逻辑上对树状关系活动记录的线性化. 在共享栈模型中,同一处理器单元上的活动记录不一定相邻,各个处理器单元上的活动记录相互交错存放. 但是,由于共享栈下所有的处理器单元共享同一个栈空间,在分配活动记录空间的时候需对栈空间加锁,这样就影响了线程发起的数目,并最终影响程序的并行加速比.

4 独立栈模型的设计和实现

为解决共享栈中存在的栈加锁问题,本文结合推测多线程模型,提出实现函数调用的独立栈模型. 独立栈模型将消除栈加锁对线程发起的影响,从而提高程序的运行效率和系统的整体加速比. 不同于共享栈,独立栈模型中每个处理器单元拥有自己的栈空间. 处理器单元可以访问其他处理器单元的栈空间,但是只能在自己的栈空间上分配函数活动记录. 模拟器在初始化的时候根据处理器单元的数目,在共享的内存空间中为每个处理器单元分配固定大小的栈空间. 当某个处理器单元发生函数调用时,就在自己的栈空间上分配相应大小的空间,当函数调用返回时,再对此空间进行释放.

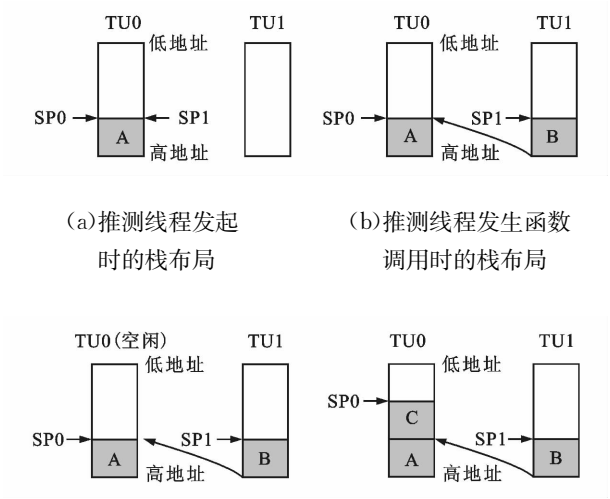
下面从推出多线程执行模型的角度,分为推测线程发起、执行函数调用、退出函数调用、推测线程被验证、发起新的推测线程五个方面来说明独立栈函数调用方法.

(1)假设处理器单元 TU0 在运行的时候碰到线程发起指令,并且线程成功发起到处理器单元 TU1 上,此时 TU1 和 TU0 共享同一个活动记录,TU1 自己的栈空间为空,TU1 对 TU0 的活动记录仅拥有读取权限. 此刻的栈空间布局如图 5a 所示.

(2)当 TU1 执行中碰到函数调用,它在自己的栈空间中分配相应大小的空间. 此时的栈空间布局如图 5b 所示.

(3)当 TU1 从函数调用中返回的时候,它将相应的活动记录释放,此刻它的栈空间为空. 此时的栈空间布局如图 5a 所示.

(4)当推测线程被验证时,如果推测线程推测失败,那么推测线程被撤销,推测线程所在的处理器单元被释放,栈空间中的相应的活动记录也被释放. 如果推测线程推测正确,那么确定线程所在的处理器单元转为空闲状态,但是其栈空间中的相关活动记录还不能释放. 推测线程转为以确定方式执行,它同时在 TU0 和 TU1 上拥有函数的活动记录. 此刻的栈空间布局如图 5c 所示.



(a)推测线程发起时的栈布局　(b)推测线程发生函数调用时的栈布局

(c)推测线程被验证正确时的栈布局　(d)推测线程再次被发起时的栈布局

图 5　独立栈不同情况下的栈布局

(5)当 TU1 被验证正确后,TU1 将以确定的方式执行程序的代码,此时如果 TU1 在执行的过程中碰到线程发起指令,并且线程成功发起到 TU0 所在的处理器单元上,当 TU0 以推测的方式执行函数调用时,就会在自己的栈空间中建立相应的函数活动记录. 此时的栈空间如图 5d 所示.

通过以上的分析,结合推测多线程执行机制,独立栈函数调用方法必须解决如下几个问题.

(1)维护函数之间的调用返回关系,即保证函数调用结束后能正确地返回到调用函数中. 如图 5c 中 TU1 在结束函数调用后,必须能正确地返回到活动记录 A 上.

(2)获取当前栈空间的栈顶. 由于当前处理器单元的栈空间有可能存在没有释放的历史活动记录,因此在建立新的活动记录的时候,必须获取当前栈空间的栈顶,以便确定新活动记录的开始位置.

(3)释放函数的活动记录. 如果函数的活动记录是在当前处理器单元的栈空间上,可以通过简单的栈指针的移动来进行释放. 如果要释放的活动记录在其他的处理器单元上,则必须设计相应的栈空间释放方法.

对于问题(1),可在函数的活动记录中增加相应的调用者的 SP 和 FP 寄存器值,以便在调用结束后顺利返回. 在这种情况下,函数活动记录之间的关系通过 SP 和 FP 寄存器构成了一个链状关系.

对于问题(2)和(3),模拟器端提供了一个 DYNAMIC_SP的内部寄存器保存每个处理器单元当前的栈顶. 为此,在独立栈函数调用方法下,模拟器需增加额外的两条指令,指令的格式和语义如表 1 所示. 这里需要特别注意的是,由于独立栈模型下函数调用方法中涉及到活动记录的释放,因此模拟器端 update 指令在实现时,须根据不同的情况进行处理,具体来说分为如下两种情况.

表 1　独立栈模型引入的指令

指令	汇编格式	指令语义
get	get reg, immed	获取当前处理器单元的栈顶. 该指令将 DYNAMIC_SP 寄存器的值拷贝到指定的 reg 寄存器中,立即数 immediate 供将来扩展之用.
update	update reg, immed	更新当前处理器单元的栈顶. 该指令用给定 reg 寄存器的值更新 DYNAMIC_SP 的值,立即数 immediate 供将来扩展使用.

(1)update 指令执行时 reg 寄存器中存放的地址在当前处理器单元的栈空间内. 对于这种情况,处理器单元首先将 DYNAMIC_SP 寄存器的值更新为 reg 寄存器的值,完成当前函数活动记录的释放,紧接着查看当前栈顶内存单元是否被标记为无用地址,如果是,则继续移动栈顶指针,直到碰到一个有效的内存单元,从而完成函数活动记录的释放.

(2)update 指令执行中,如果 reg 寄存器中存放的地址不在当前处理器单元的栈空间内,说明当前函数返回到了另外一个处理器单元的栈空间上. 在这种情况下,当前处理器单元仅仅将相应的活动记录空间标记为无用,不进行活动记录的释放,真正活动记录的释放将在后续通过其他的处理器单元进行.

独立栈模型下,函数调用各部分的代码序列如下.

(1)函数序言部分

```
move $t0, $SP  #父过程SP的保存
la $SP,-128($SP)  #分配活动记录空间
update $SP, 12  #更新处理器单元栈顶
sw $t0, 0($SP) #保存FP寄存器
sw $FP, 4($SP) #保存SP寄存器
…  #其它代码序列
(2)调用前部分
move $a0, $t0 #通过寄存器传递参数
#其他参数的传递
move $FP, $SP  #保存当前SP指针
get $SP, 12  #获取新活动记录开始位置
sw $t5,-16($SP) #其他参数保存到栈上
…  #其他代码序列
(3)函数结语部分
lw $t0, 0($SP)
update $t0, 13  #更新处理器单元栈顶
lw $SP, 4($SP)  #恢复SP寄存器
lw $ra, 8($SP)  #恢复其他相关寄存器
…  #其它代码序列
```

5 实验结果和分析

独立栈方案的具体实现包括两个部分:一是编译器的修改,根据独立栈模型代码序列的特点,生成符合要求的代码;二是模拟器端的修改,增加get和update两条指令以支持独立栈模型.本文在Prophet推测多线程系统[4]上实现了独立栈模型.Prophet是一个支持推测多线程的原型系统,它包括编译器部分和模拟器部分.Prophet编译器[5]根据数据流和控制流分析的结果,通过插入相关的指令,将串行程序合理地划分为可并行运行的多个线程.Prophet模拟器[6-7]则是一个支持推测多线程机制的多核模拟器,它加载Prophet编译器编译产生的rom文件,给出程序运行的结果以及运行过程中的运行数据,如程序的并行加速比、线程发起的数目等,从而来评价程序并行化的效果.

本文采用Olden基准测试程序[8]对独立栈模型进行了测试,并和共享栈模型进行了对比.测试结果显示,对于大多数程序,线程发起的数目和程序的平均加速比都有较大的提高.表2给出了Prophet系统在模拟器8核处理器及共享栈模型中运行各测试程序所需的时钟周期数以及因栈加锁失败浪费的时钟周期数.由表2可见,由于共享栈模型中存在的栈加锁问题,当多个线程同时申请栈空间时,就会出现申请失败的情况,因此影响了线程总的发起数量,从而影响了程序并行加速比的提升.独立栈函数调用模型的设计初衷就是消除栈加锁对线程发起的影响,尽可能多地发起线程,从而提高程序的并行加速比.

表2 共享栈下各测试程序运行的时钟周期数

测试程序	总时钟周期数	栈加锁失败的时钟周期数
mst	27 302	97
bh	3 435 183	566
power	5 250 591	3 570
tsp	6 297 522	3 022
Em3d	90 321	124
bisort	518 133	18 868
voronoi	3 307 544	8 248
health	119 241	1 096
perimeter	464 164	8 290
treeadd	1 330 417	12 289

表3给出了Prophet模拟器在8核处理器及独立栈和共享栈模型中线程发起数目方面的对比.从表3中可以看出,由于独立栈消除了共享栈模型中的栈加锁问题,除了health测试程序,其他程序线程发起的数目都有很大程度的增加,达到了独立栈设计的目标.表4给出了模拟器在8核处理器及独立栈和共享栈模型中成功线程发起数目方面的对比.结合表3可以看出,尽管独立栈模型在很大程度上增加了总的线程发起数目,但是那些对程序最终并行加速比起积极作用的成功线程的数目则根据测试程序特性的不同,分别有不同程度的增加.结合推测多线程执行的特点,这主要有以下两点原因:一是程序可开发的并行性总是有限的,因此当达到某一个特定的点后,所有发起的后续线程都将因为程序的语义而验证为失败线程;二是对于特定的目标机器,可利用的处理器核总是有限的,由于消除了栈加锁的影响,那些被验证为失败的线程发起导致了其他具有潜在正确性的线程发起的延迟,并且在此过程中还涉及到失败线程的撤销等操作,因此影响了成功线程的数目.

图6给出了Prophet模拟器在8核处理器及独立栈和共享栈模型中程序并行加速比方面的对比.相对于共享栈函数调用方法,独立栈模型下Olden基准测试程序的平均并行加速比有3.85%的提高.从图6可以看出,大多数程序的加速比都有所提高,这主要是因为独立栈函数调用方法消除了共享栈模型中的栈加锁问题,使得成功线程发起的数目有不同程度的提

高，从而提高了程序的并行加速比. 但是，由于程序的并行加速比不仅与成功发起的线程数目相关，而且与所发起的线程大小以及线程预计算片段相关，另外程序可开发的并行性也受制于具体程序的结构，因此对于有些测试程序，也出现了独立栈加速比低于共享栈的情况.

表 3　独立栈和共享栈下总的线程发起数目

测试程序	共享栈下总的线程发起数目	独立栈下总的线程发起数目
mst	780	911
bh	122 570	123 102
power	139 583	140 311
tsp	177 745	179 048
Em3d	2 759	3 132
bisort	22 991	23 896
voronoi	117 130	117 708
health	4 482	4 414
perimeter	10 888	10 968
treeadd	43 012	82 953

表 4　独立栈和共享栈下成功线程发起的数目

测试程序	共享栈下成功线程发起数目	独立栈下成功线程发起数目
mst	485	486
bh	115 663	115 952
power	101 126	101 192
tsp	135 105	135 105
Em3d	1 829	2 435
bisort	9 149	9 629
voronoi	102 900	102 901
health	3 843	3 833
perimeter	6 607	6 757
treeadd	20 484	20 484

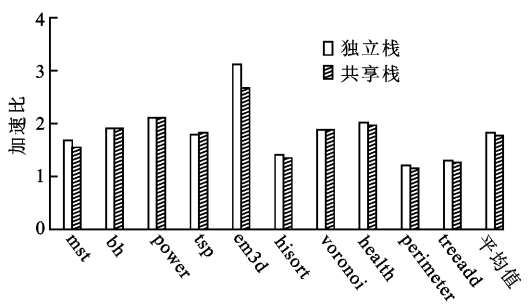


图 6　独立栈和共享栈下程序运行的加速比

6　结　论

本文给出了面向 CMP 的推测多线程下的独立栈函数调用方法. 它消除了共享栈模型中的栈加锁问题，提高了线程发起的数目，从而最终提高了程序运行的并行加速比. 独立栈模型涉及编译器和模拟器两个部分. 为了保证程序运行正常进行，编译器需对生成的代码作相应的调整，模拟器方面则增添了 get 和 update 两条指令，以便管理相应的栈空间. Olden 基准程序的测试表明，独立栈模型相对于共享栈模型使得程序的并行加速比提高了 3.85%.

参考文献：

[1] KRISHNAM V, TORRELLAS J. A chip-multiprocessor architecture with speculative multithreading[J]. IEEE Trans Comput, 1999, 48(9):866-880.

[2] WANG Shengyue. Compiler techniques for thread-level speculation[D]. Minneapolis, Minnesota, USA: University of Minnesota, 2007.

[3] HOGEN G, LOOGEN R. A new stack technique for the management of runtime structures in distributed implementations[R]. Aachen, Germany: RWTH Aachen University, 1993.

[4] CHEN Zheng, ZHAO Yinliang, PAN Xiaoyu, et al. An overview of Prophet [C]// Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing. Berlin, Germany: Springer, 2009: 396-407.

[5] PAN Xiaoyu, ZHAO Yinliang, CHEN Zheng, et al. A thread partitioning method for speculation multithreading // Proceedings of the 8th International Conference on Embedded Computing. Piscataway, NJ, USA: IEEE, 2009: 285-290.

[6] DONG Zhaoyu, ZHAO Yinliang, WEI Yuanke, et al. Prophet: speculative multithreading execution model with architectural support based on CMP[C] // International Conference on Embedded Computing. Piscataway, NJ, USA: IEEE, 2009:103-108.

[7] 宋少龙，赵银亮，韦远科，等. 支持推测多线程的多核模拟器 Prophet+[J]. 西安交通大学学报，2010，44(12)：13-17.
SONG Shaolong, ZHAO Yinliang, WEI Yuanke, et al. Prophet: an extended multicore simulator for speculative multithreading[J]. Journal of Xi′an Jiaotong University, 2010, 44(12):13-17.

[8] CARLISLE M C. Olden benchmark suite [EB/OL]. (1996-06-15)[2007-06-30]. http://www.cs.princeton.edu/~mcc/olden.html.

（编辑　武红江）