第 36 卷第 11 期 电 子 与 信 息 学 报 Vol.36No.11
2014 年 11 月 Journal of Electronics & Information Technology Nov. 2014

基于性能预测的推测多线程循环选择方法

刘 斌 赵银亮* 韩 博 李玉祥 吉 烁 冯博琴 武万杰
(西安交通大学计算机科学与技术系 西安 710049)

摘 要：线程级推测(Thread-Level Speculation, TLS)是多核上一种加速串行程序的线程级自动并行化技术。循环具有规则的结构并在运行时占有大量的执行时间，因此循环是挖掘并行性的理想对象。然而，选择哪些循环并行才能提高程序的加速比是一个很难决定的问题。为了解决该问题，该文提出一种基于性能预测的循环选择方法。基于输入训练集获取程序预执行的剖析信息，同时结合各种推测因素，构建了循环结构的性能预测模型。预测结果定量评估了循环推测并行的加速比并决定该循环在运行时是否适合并行。实验结果表明，该文提出的方法能有效地预测循环并行时所蕴含的并行性，并依据预测结果准确地选择具有并行收益的循环推测并行，最终 Olden 基准测试集加速比性能平均提升了 12.34%。

关键词：并行处理；线程级推测；循环选择；性能预测
中图分类号：TP314 **文献标识码**： A **文章编号**：1009-5896(2014)11-2768-07
DOI: 10.3724/SP.J.1146.2013.01879

A Loop Selection Approach Based on Performance Prediction for Speculative Multithreading

Liu Bin Zhao Yin-liang Han Bo Li Yu-xiang Ji Shuo Feng Bo-qin Wu Wan-jie
(*Department of Computer Science and Technology*, *Xi'an Jiaotong University*, *Xi'an* 710049, *China*)

Abstract: Thread-Level Speculation (TLS) is a thread-level automatic parallelization technique to accelerate sequential programs on multi-core. Loops are usually regular structures and programs spent significant amounts of time executing them, thus loops are ideal candidates for exploiting the parallelism of programs. However, it is difficult to decide which set of loops should be parallelized to improve overall program performance. In order to solve the problem, this paper proposes a loop selection approach based on performance prediction. Basing on the input training set, the paper gathers profiling information during program pre-execution. Combining profiling information associated with the program and various speculative execution factors, the paper establishes a performance prediction model for loops. Then, based on the result of prediction, the paper can quantitatively estimate the speedup of loops and decide which loops should be parallelized on runtime. The experimental results show that the proposed approach effectively predicts the parallelism of loops when speculative execution and accurately selects beneficial loops for speculative parallelization according to the predicted results, finally Olden benchmarks reach 12.34% speedup performance improvement on average speedup.

Key words: Parallel processing; Thread-Level Speculation (TLS); Loop selection; Performance prediction

1 引言

随着半导体按照摩尔定律的发展，处理器进入了多核时代，如何有效利用丰富的核资源成为当前研究的热点。当指令级并行在挖掘并行性方面遇到了瓶颈，线程级并行(Thread-Level Parallelism, TLP)逐渐成为更佳的选择，尤其适合于片上多处理器(Chip MultiProcessor, CMP)[1]。为了提升程序在CMP 上的性能，大量非规则程序需要重构以便适合在多核上并行执行，从而提高程序的加速比性能。线程级推测(Thread-Level Speculation, TLS)作为线程级并行的一种重要方法，允许多个线程激进地并行执行以提高程序的加速比，较为典型的代表有Multiscalar[2], Hydra[3], POSH[4], Mitosis[5], DOMORE[6], SEED[7], DOE[8]和 Prophet[9]。

循环具有规则的结构并且在程序执行时占有大量的时间，因此循环成为并行执行最理想的对象。目前，大量的研究关注循环并且从循环中提取多线程[10]。文献[11]结合数据和控制依赖提出一种误推测代价模型，并基于该模型对循环进行性能评估，然

2013-12-02 收到，2014-03-21 改回
国家自然科学基金(61173040)，国家“863”计划项目(2012AA011003)和博士学科点专项科研基金(20130201110012)资助课题
*通信作者：赵银亮 zhaoy@mail.xjtu.edu.cn

后对循环进行改造并选择合适的循环并行执行。与本文类似文献[11]也选择了具有收益的循环并行。与本文不同的是该方法仅考虑数据和控制依赖构建模型，而本文方法考虑了多种推测因素。文献[12]为 Java 语言提出一种动态循环选择框架。框架采用硬件运行系统提取依赖时间和推测状态等信息评估了循环的加速比性能。与本文相似的是对循环评估了循环的加速比并反向指导线程划分。不同之处在于该框架研究的是面向对象的 Java 程序，而本文研究的是非规则 C 语言程序，这两者程序的特征不同所引起的推测开销也就不同。文献[13]采用专家经验手工并行化了 SPEC2000 基准测试集。 文献[13]的研究工作非常耗时，容易发生错误，而本文的研究工作是一种自动并行化的技术。

本文提出一种基于性能预测的循环选择方法。基于输入训练集多次预执行程序获取反映程序行为的剖析信息。综合考虑剖析信息和影响推测并行的各种因素，构建了循环推测并行时的性能预测模型，模型的预测结果决定了该循环是否进行推测并行。本文基于 Prophet 编译器实现了性能预测模型。使用 Olden 基准测试集进行了性能测试，并与传统方法进行了比较和分析。实验结果表明本文提出的预测模型能有效地预测循环中所蕴含的并行性，并准确地选择有收益的循环推测并行，最终 Olden 基准测试集加速比性能平均提升了 12.34%。

2　执行模型

TLS 将串行程序在多核上并行执行以提高程序的加速比，执行模型如图 1 所示，串行程序被一系列的线程激发点-准控制无关点(Spawning Point-Control Quasi-Independent Points, SP-CQIPs)指令对映射成多线程程序。同时按照串行语义顺序，在每一时刻只有一个线程提交数据到内存，该线程被称为确定线程，而其他线程为推测线程。每个推测线程由串行程序代码片段及预计算片段组成。 预计算片段是由编译器根据切片技术生成的一小段代码，用来预测推测线程使用的 live-ins(推测线程使用但并非由该线程定义的变量值)。如图 1(a)所示，如果忽略 SP-CQIPs 指令对，推测多线程程序就转换成串行程序；如图 1(b)所示，当程序执行到确定线程 T_1 中的 SP 时，如果有闲置核资源，将激发一个新推测线程 T_2；当确定线程 T_1 执行到 CQIP 将验证直接后继线程 T_2 在预计算片段中使用的 live-ins。若验证正确，则确定线程 T_1 提交执行结果，将核资源释放，该核可以执行其他的线程。然后将确定执行的权限传递给直接后继线程；若验证失败则会导致推测执行失败。当出现验证失败时，则撤销推测线程 T_2，跳过预计算片段，串行执行此直接后继线程 T_2，如图 1(c)所示；当出现读后写(Read After Write, RAW)依赖违规时，如图 1(d)所示，则在当前的状态下重新启动该线程。

3　性能预测模型

3.1 基本思想

在循环级并行中，现有的方法大多都采用静态、定性的方法进行性能评估。在本文中，结合程序的剖析信息和各种推测因素，提出一种定量预测并行性能的循环选择方法。首先，基于输入训练集获取程序在预执行时的剖析信息并以注释的形式添加到对应的 SUIF 中间文件(SUIF-IR)中。然后，综合剖析信息和推测因素，构建了一种性能预测模型。该模型力图更加精确地预测程序行为和定量地评估各种推测因素对加速比性能的影响，并利用预测结果评判推测执行性能的好坏程度，最终决定是否在程序运行时并行该循环。

3.2 剖析信息

程序剖析是通过收集程序过去运行时的信息，进而动态调整程序将来的执行。理论上讲，如果编译器能预测出精确的程序行为，它能产生出任何平台下并行性能最好的代码。然而实际上，编译器仅能获取部分精确的程序行为。为了提高程序行为的精确性，本文开发了 Profiler 剖析器，分析了程序集的输入特征，根据输入特征构造了基准程序的输入训练集。通过多次预执行程序捕获了程序运行时的剖析信息，为性能预测模型的构建提供了更为精确的程序行为。

Profiler 的工作流程如图 2 所示，当一个程序指令被执行时，首先判断指令的类型，当指令为循环指令时，记录当前循环的迭代次数 D 和动态指令数 M。那么循环的平均动态指令数为 $L_D=(L_{D\text{-}1}\times(D\text{-}1)+M)/D$，其中 $L_{D\text{-}1}$ 为上一次循环执行时的平均动态指令数。通过多次预执行程序，求其平均值可获取循环部分的剖析信息并以注释的形式标注在对应的 SUIF-IR 文件中。

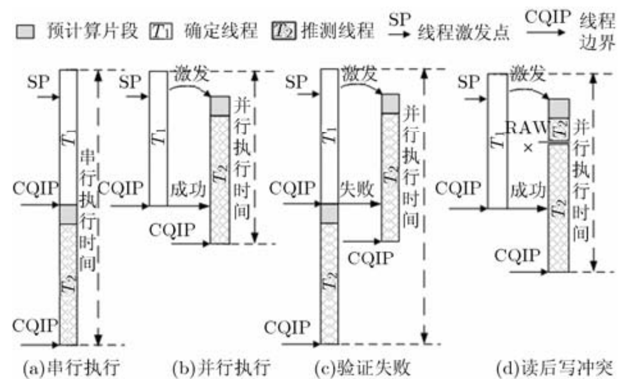

图 1　推测多线程执行模型

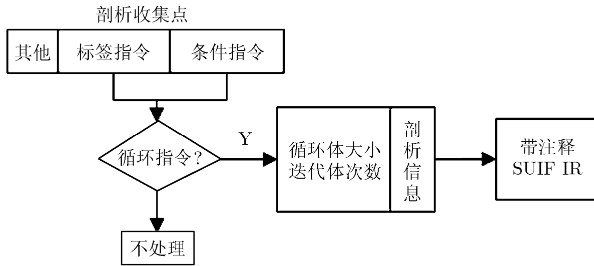

图 2 程序剖析器

3.3 推测并行影响因素

线程粒度、数据依赖、线程分发和激发距离是影响循环并行性能的关键影响因素。一般来说这些因素被综合考虑决定并行性能的好坏。

(1)线程粒度(thread size)是指一个线程的动态指令数。线程分发、线程启动与重启、线程冲突和线程间的通信会引起推测并行开销，所以线程粒度大小是影响推测性能的关键因素。线程粒度过小，推测执行中的各种开销将会抵消掉推测并行所带来的收益。线程粒度过大，线程间过多的控制和数据依赖将会导致线程冲突的概率增大，进而增加推测并行时的开销。

(2)数据依赖(data dependence)是指子线程中的指令引用了父线程中定义的变量。数据依赖距离通常用来刻画线程间依赖的程度。数据依赖距离是指线程推测并行时子线程执行时需要等待的指令数直到它所需要的数据被父线程产生。如果数据依赖距离过小子线程中需要的值还未由父线程产生，会引发额外的推测代价。

(3)线程分发(thread dispatch)是指当激发一个新的线程时，需要分发一个处理器单元执行该线程。线程分发需要复制父线程堆栈的参数和寄存器里的数值，这需要占用一定的开销。

(4)激发距离是指在父线程中定义的 SP 点到子线程开始 CQIP 点之间动态指令数。如果激发距离过长，子线程执行过程中使用的变量还未产生，将会导致数据推测失败。如果激发距离过小，推测执行所引起的各种开销将抵消掉推测并行所获得的收益。

3.4 性能预测

通过 3.3 小节推测因素对并行性能的定性分析，进一步利用 3.2 小节获取的剖析信息量化这些因素构建性能预测模型。受阿姆达尔定律的启发，程序加速比由程序的串行执行时间和并行时间来决定，加速比 Sp 为程序串行执行时间 T_{seq} 与并行执行时间 T_{par} 的比值，用式(1)表示为

$$\mathrm{Sp} = T_{\mathrm{seq}}/T_{\mathrm{par}} \tag{1}$$

3.4.1 计算串行执行时间 如图 3 所示，通常循环中的每个迭代体形成一个独立的推测线程，在程序运行时上一个迭代体激发下一个迭代体形成多线程程序。循环分支指令是并行线程间唯一的控制依赖，同时线程间存在的数据依赖可能会导致数据推测失败而重启线程。假定处理器在一个时钟周期内执行一条指令，那么一个循环在串行执行时所需时间 T_{seq} 为迭代体的平均指令数 S 和循环的平均迭代次数 N 之积，循环串行执行时间为

$$T_{\mathrm{seq}} = S \times N \tag{2}$$

3.4.2 计算并行执行时间 由于循环并行执行时间与线程粒度、数据依赖距离、线程分发及激发距离密切相关，所以并行执行时间 T_{par} 由两部分组成：理想情况下线程并行执行时间 T_{exe} 和误推测代价时间 T_{miss}。其中，T_{exe} 是假设在没有误推测情况下并行执行的时间，T_{miss} 是由于误推测而浪费的平均执行时间，表示为

$$T_{\mathrm{par}} = T_{\mathrm{exe}} \times T_{\mathrm{miss}} \tag{3}$$

如图 4 所示，理想情况下线程并行时间 T_{exe} 使用串行执行时间 T_{seq} 减去并行时提前执行时间 T_{ahead}。此外，循环并行时相邻两个迭代体提前时间 T_{iter} 为执行激发距离代码片段的时间 T_{dis} 减去执行预计算代码片段的时间 T_{cnt} 和线程分发开销所需要的时间 T_c，相邻迭代间提前完成时间为

$$T_{\mathrm{iter}} = T_{\mathrm{dis}} - T_{\mathrm{cnt}} - T_c \tag{4}$$

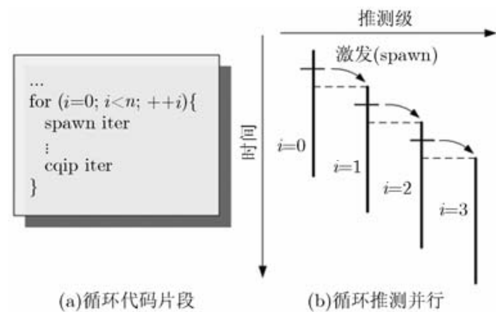

图 3 循环激发示意图

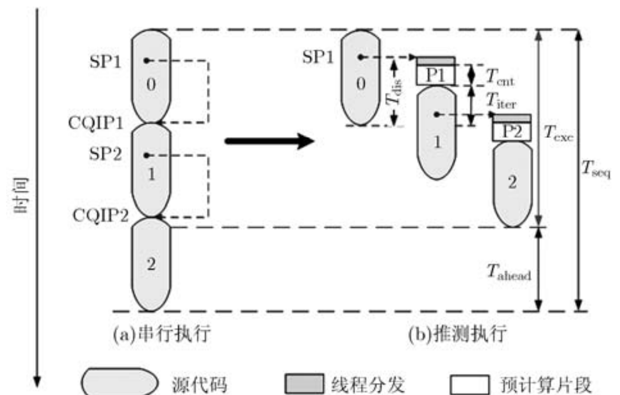

图 4 推测执行的加速效果

那么，循环提前执行时间 T_{ahead} 为

$$T_{\mathrm{ahead}} = T_{\mathrm{iter}} \times (N-1) \tag{5}$$

程序并行执行时间 T_{exe} 为循环串行执行时间 T_{seq} 与提前执行时间 T_{ahead} 之差，由式(6)计算。

$$T_{\mathrm{exe}} = T_{\mathrm{seq}} - T_{\mathrm{ahead}} \tag{6}$$

为了便于描述循环并行时线程间的数据依赖关系，引入一些基本的概念。定值点 q 是对某一变量 v 赋值时指令所处的位置编号。引用点 p 是对某一变量 v 使用时指令所处的位置编号。依赖弧 A 用来表示线程间数据依赖关系，每一个有向弧和一个定值-引用链 $< p,q >$ 相对应。图 5 描述了循环并行时产生的并行线程 T_1 和 T_2。线程 T_1 中对一个变量 a 定值，随后控制流从线程 T_1 转向线程 T_2，在线程 T_2 中引用了该变量 a，用从 T_2 中的引用点指向 T_1 的定值点的有向弧 A_1 来表示此数据依赖关系。数据依赖弧 A_1 的距离 $|A_1|$ 是指 T_1 与 T_2 同时并行时，子线程 T_2 引用变量 a 的位置编号减去父线程 T_1 定义变量 a 的位置编号，$|A_1|$ 表示两个线程并行执行时数据依赖的强弱情况。在图 5 中，当循环推测并行执行时，线程间存在的数据依赖关系可能会导致读后写冲突，冲突的线程会被撤销。因此使用由于冲突而抛弃的指令数来量化其对总体并行执行的影响。浪费的工作量取决于冲突何时被检测出来。例如，如果一个线程开始于时钟周期 m_1，在时钟周期 m_2 时检测出线程冲突，那么浪费的时钟周期为 $m_2 - m_1$。然而，线程间有多个数据依赖关系，同时程序实际执行时上下文环境复杂多变，这些依赖关系都有可能引起线程冲突而导致后继线程撤销，所以采用线程间因冲突而撤销的平均指令数与循环的撤销概率之积，量化线程冲突对并行执行的平均性能的影响，线程间浪费的平均执行时间由式(7)计算：

$$T_{\mathrm{miss}} = D \times P_{\mathrm{miss}} \tag{7}$$

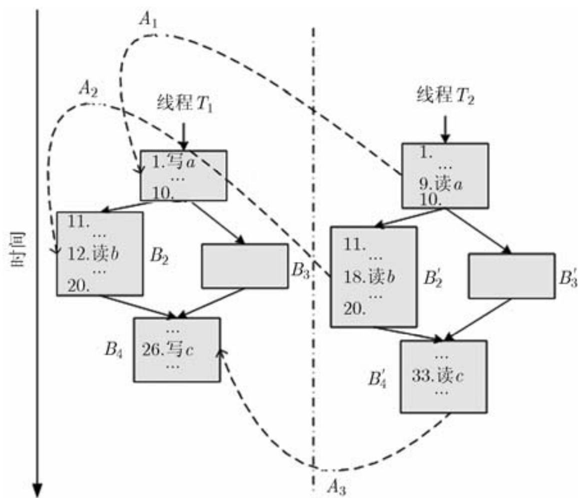

图 5 线程间定值-引用对

式中 D 为线程间撤销的平均指令数；P_{miss} 为循环的撤销概率，P_{miss} 通过前面的预执行剖析信息获取。

在构建的预测模型中，线程间撤销的平均指令数 D 是指所有具有数据依赖关系的指令对从使用点(子线程)到定值点(父线程)数据依赖距离的平均值。如图 5 所示，当子线程 T_2 与父线程 T_1 推测并行执行时，子线程 T_2 与父线程 T_1 之间有 3 对定值-引用关系(A_1 , A_2 和 A_3)线程间撤销的平均指令数由式(8)计算：

$$\begin{aligned} D &= \left(\sum_{i=1}^{n} |A_i|\right) \Big/ n \\ &= \frac{(9-1)+(18-12)+(33-26)}{3} = 7 \end{aligned} \tag{8}$$

由式(2)，式(6)和式(7)最终得到循环推测并行时预期加速比为

$$\mathrm{Sp} = T_{\mathrm{seq}} \big/ \left(T_{\mathrm{exe}} + T_{\mathrm{miss}}\right) \tag{9}$$

如果加速比 Sp 小于 1，说明该循环推测并行所引起的开销大于并行所带来的收益，推测并行效果不理想，所以此循环在程序运行时将被串行执行。如果数值大于 1，该循环在推测并行时并行所获得的收益大于并行所引起的开销，可以提高加速比性能，所以在运行时该循环将被推测并行执行。

4　实验结果和性能分析

本文基于 SUIF/MACHSUIF[14]开发了 Prophet 编译器[15]，并在编译器中实现了性能预测的循环选择方法。同时在 Prophet 模拟器[9]中验证了方法的有效性，Prophet 模拟器是基于 Tomasulo[16]算法实现的超标量流水线 4 核处理器。最后选用了 Olden 基准程序集[17]中的子集测试了本文提出的评估方法。

4.1 剖析信息统计

通过分析 Olden 程序集的输入特征，程序的输入参数个数不同但都为整型。根据均匀分布随机函数构造了程序的训练输入集，采用渐进式预执行来获取程序的剖析信息。随着程序预执行次数的增加，程序的剖析信息在程序某次执行之后趋于稳定。本文希望在稳定点之后停止预执行，最终每个程序的预执行次数如表 1 所示，剖析信息统计如图 6(a)所示。从图 6(a)可以看出程序预执行后所获取的剖析信息，其中横轴代表了循环的平均迭代次数，纵轴代表了循环平均迭代体大小。图 6(b)为图 6(a)左下角局域放大图，从中可以看出 Olden 中 89.16%的循环具有中小规模。同时，7.23%的循环是比较大的循环体。从表 1 可以看出，每个程序预执行次数各不相同，程序 mst 的执行次数最多，高达 1098 次，而程序 bh 仅执行了 106 次。经过分析，主要原因是每个程序本身的特征不同，从而导致循环剖析信息达到稳定状态时所执行的次数也就不同。

表 1 Olden 基准测试集剖析信息统计

Olden	bh	voronoi	mst	em3d	health	power	tsp
输入参数个数	2	2	2	3	3	2	2
预执行次数	106	131	1098	123	366	216	353
平均迭代次数	9.14	1.30	3.92	8.73	3.51	2.88	94.50
循环体平均大小	200.45	137.00	830.59	265.99	172.88	2625.27	1267.25

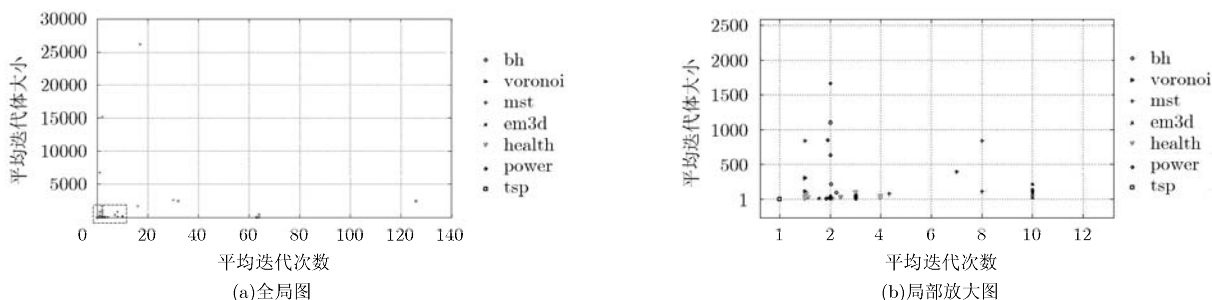

图 6 Olden 基准测试集循环剖析信息

理论上讲，预执行次数越多，所构建的预测模型也就越准确。然而，由于时间的限制，再多的预执行也无法覆盖所有的执行路径。这种情况说明剖析方法在实现过程中，剖析信息有可能不够完全精确。然而，随着预执行次数的不断增加，这些输入的剖析信息会变得更为精确，从而不断提高预测模型的精确程度。

4.2 加速比性能比较

图 7 给出了基于传统方法[18]和本文改进方法加速比性能的实验数据。实验结果显示，大多数 Olden 基准测试集的加速比性能都有一定程度的提升。特别是程序 em3d, health 和 bh 性能提升较为显著，而程序 mst 和 tsp 的加速比性能未发生改变。为了定量分析，定义性能增长率为

$$R=\frac{I-O}{O}\times 100\% \tag{10}$$

式中 I 表示改进方法程序获得的加速比；O 表示传统方法取得的加速比；R 代表性能增长率。

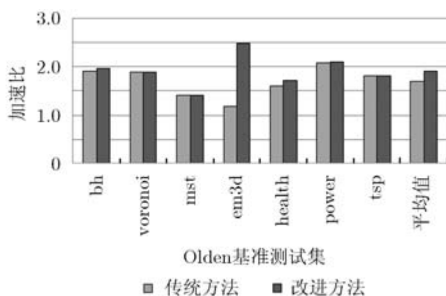

图 7 加速比性能比较

表 2 统计了 Olden 测试集基于传统的方法和改进的方法在 Prophet 模拟器中产生的动态信息及性能增长率。由表 2 中的最右列可以看出，与传统的方法相比，本文提出的方法使 Olden 基准测试集的增长率变化范围由 0 到 107.61%，其平均增长率可达 12.34%。实验数据显示，本文的改进方法能充分地挖掘程序中的并行性同时获得了更好的性能改善。此外，表 2 和表 3 分别统计了 Prophet 系统产生的动态线程统计信息。

程序 em3d 性能提升高达 107.61%，原因在于程序 em3d 中循环蕴含的并行性较大，同时本文采用性能预测方法并行了 8 个具有并行收益的循环，而传统方法仅依靠启发式并行了 4 个外层循环(表 2)。实验数据表明改进方法从程序 em3d 中激发出的线程数目是传统方法的 2.41 倍，同时改进方法保证了程序 em3d 激发成功率也有一定程度的提高，最终本文方法允许更多的推测线程参与并行从而提高了加速比性能。程序 bh 包含 83 个循环，是循环数最多的程序。传统方法仅并行了 30 个外层循环，本文方法选取具有并行收益的 69 个循环并行执行，并行的循环数量占循环总数的 83.13% 而传统方法仅占 36.14%(表 2)。改进方法虽然导致线程执行的成功率下降，但并行更多的循环使成功激发的线程数目却增加了，因此性能提升了 3.16%。由表 2 中数据可以看出，程序 health 和 power 执行行为与程序 bh 相似。本文方法为这两个程序分别选取了 9 个和 13 个具有并行收益的循环推测并行。虽然激发成功率都降低了(表 2)，但成功激发的线程数目增加了，其加速比性能各自提升了 6.39%和 0.93%。实验结果表明

本文方法能激发更多的推测线程参与并行从而提高程序的加速比性能。

程序 voronoi 有 18 个循环体，但循环较小同时蕴含的并行有限。由表 2 可见，相比传统方法，虽然本文方法并行循环数增加了 1 个，但程序 voronoi 成功激发的线程数目与传统方法几乎相近，所以性能提升了，但提升的程度较为有限。程序 mst 和 tsp 加速比性能没有发生任何变化。对于程序 mst，由表 2 可见，循环体数有 12 个，传统方法和本文的改进方法都选取了 6 个循环并行。虽然并行的循环数相同，但本文改进方法选择了具有并行收益的循环和传统方法所选择的最外层循环并不相同，最终导致成功激发的线程数目相差不大。同样，针对程序 tsp，改进方法选取了 5 个循环并行化。由表 2 中可见，虽然激发的线程数目增加了，但是相应的线程激发的成功率却降低了，结果显示成功激发的线程数目和传统方法激发的线程数基本一样，由此可见，加速比性能不仅与线程数目相关而且还与每个线程对并行性能的贡献相关。最终由于本文方法所获取的收益与传统方法所获取的收益一致导致性能未发生变化。因此程序 mst 和 tsp 的加速比性能保持了原来的性能。

表 2 Olden 基准测试集动态信息统计

测试程序	激发线程数目		激发成功率		成功激发线程数目		并行循环数		循环总数	性能提升(%)
	传统	改进	传统	改进	传统	改进	传统	改进		
bh	119085	122887	0.9684	0.9552	115322	117382	30	69	83	3.16
voronoi	110198	110242	0.9327	0.9323	102782	102779	7	8	18	0.26
mst	490	483	0.8796	0.9068	431	438	6	6	12	0
em3d	1564	3705	0.8542	0.8683	1336	3217	4	8	14	**107.61**
health	3684	3963	0.9541	0.9336	3515	3700	9	11	14	6.39
power	118717	122781	0.8478	0.8415	100648	103320	12	13	17	0.93
tsp	174319	197819	0.7750	0.6830	135097	135110	4	5	7	0
平均值	73570	77837	0.8483	0.8365	62409	65111	9	11.25	21	12.34

表 3 Olden 基准测试集线程信息统计

测试程序		bh	voronoi	mst	em3d	health	power	tsp	平均值	方差
传统方法	线程粒度	52.24	139.33	39.5	43.00	51.33	61.47	41.00	**61.12**	**1248.70**
	Live-ins	3.55	2.78	2.83	3.00	3.22	3.67	3.20	3.18	0.12
	P-slice 比例	0.087	0.027	0.097	0.093	0.082	0.076	0.103	0.081	0.001
改进方法	线程粒度	37.49	133.15	40.00	38.44	47.00	54.37	39.17	**55.66**	**1204.20**
	Live-ins	3.22	3.23	3.00	3.11	3.55	4.11	4.00	3.46	0.19
	P-slice 比例	0.113	0.032	0.100	0.107	0.095	0.094	0.128	0.096	0.001

改进方法线程粒度平均值与方差均小于传统方法(表 3)，说明 Olden 基准测试集更适合细粒度并行。同时，改进方法产生的 Live-ins 变量比传统方法要多。实验结果表明本文方法有效的减少了推测并行时的开销，提升了有效工作时间比例，从而提高了程序的加速比性能。

5 结束语

该文提出和验证了一种基于性能预测的循环选择方法。该文主要创新在于：(1)基于程序的输入训练集获取了程序预执行的剖析信息，相比静态分析方法更加精确地预测了程序执行行为。(2)研究了影响线程级推测的关键因素。(3)提出一种性能预测模型。基于性能预测 Olden 基准测试集加速比性能平均提升了 12.34%。实验表明，该文提出的性能预测方法能够有效地评估循环推测并行时所蕴含的并行性，并依据评估结果选择具有并行收益的循环推测并行从而提高程序的加速比性能。

参 考 文 献

[1] Yang L and Zhai A. Dynamically dispatching speculative threads to improve sequential execution[J]. *ACM*

Transactions on Architecture and Code Optimization, 2012, 9(3): 13:1–13:31.

[2] Vijaykumar T N and Sohi G S. Task selection for a multiscalar processor[C]. Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, Dallas, 1998: 81–92.

[3] Hammond L, Hubbert B A, Siu M, *et al.*. The stanford hydra cmp[J]. *IEEE Micro*, 2000, 20(2): 71–84.

[4] Liu W, Tuck J, Ceze L, *et al.*. POSH: a TLS compiler that exploits program structure[C]. Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, 2006: 158–167.

[5] Madriles C, García-Quiñones C, Sánchez J, *et al.*. Mitosis: a speculative multithreaded processor based on precomputation slices[J]. *IEEE Transactions on Parallel and Distributed Systems*, 2008, 19(7): 914–925.

[6] Jialu H, Jablin T B, Beard S R, *et al.*. Automatically exploiting cross-invocation parallelism using runtime information[C]. Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, Shenzhen, 2013: 1–11.

[7] Gao L, Li L, Xue J, *et al.*. SEED: a statically greedy and dynamically adaptive approach for speculative loop execution[J]. *IEEE Transactions on Computers*, 2013, 62(5): 1004–1016.

[8] Sharafeddine M, Jothi K, and Akkary H. Disjoint out-of-order execution processor[J]. *ACM Transactions on Architecture and Code Optimization*, 2012, 9(3): 19:1–19:32.

[9] 宋少龙, 赵银亮, 冯博琴, 等. 支持推测多线程的扩展多核模拟器 Prophet+[J]. 西安交通大学学报, 2010, 44(10): 13–17.
Song Shao-long, Zhao Yin-liang, Feng Bo-qin, *et al.*. Prophet+: an extended multicore simulator for speculative multithreading[J]. *Journal of Xi'an Jiaotong University*, 2010, 44(10): 13–17.

[10] Wang S Y, Yew P C, and Zhai A. Code transformations for enhancing the performance of speculatively parallel threads[J]. *Journal of Circuits, Systems and Computers*, 2012, 21(2): 1–23,

[11] Du Z H, Lim C C, Li X F, *et al.*. A cost-driven compilation framework for speculative parallelization of sequential programs[J]. *Association for Computing Machinery Special Interest Group Programming Languages Notices*, 2004, 39(6): 71–81.

[12] Chen M and Olukotun K. TEST: a tracer for extracting speculative threads[C]. Proceedings of the 2003 International Symposium on Code Generation and Optimization, San Francisco, 2003: 301–312.

[13] Prabhu M K and Olukotun K. Exposing speculative thread parallelism in SPEC2000[C]. Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Chicago, 2005: 142–152.

[14] Smith M D and Holloway G. An introduction to machine suif and its portable libraries for analysis and optimization[OL]. http://www.eecs.harvard.edu/hube/software/nci/overview.html, 2013.

[15] Chen Z, Zhao Y L, Pan X Y, *et al.*. An Overview of Prophet[M]. Berlin: German, Springer, 2009: 396–407.

[16] Hennessy J L and Patterson D A. Computer Architecture: A Quantitative Approach [M]. Amsterdam, Elsevier, 2012: 176–183.

[17] Carlisle M C, and Rogers A. Software caching and computation migration in olden[J]. *Journal of Parallel and Distributed Computing*, 1996, 38(2): 248–255.

[18] Pan X Y, Zhao Y L, Chen Z, *et al.*. A thread partitioning method for speculative multithreading[C]. Proceedings of the 8th International Conference on Scalable Computing and Communications, Piscataway, 2009: 285–290.

刘 斌： 男，1981 年生，博士生，研究方向为并行计算与机器学习.

赵银亮： 男，1960 年生，教授，博士生导师，研究方向为程序语言与编译系统、大数据并行处理及挖掘.

韩 博： 男，1975 年生，高级工程师，研究领域为软件编程方法学、信息管理学.