摘 要（小二号黑体）

对于非规则访存的应用程序, 在运行时会发生严重的 Cache 访问缺失。采用帮助线程将数据预取到离 CPU 更近的 Cache, 可以有效提高多核系统的性能。当某个应用程序的访存开销大于计算开销时, 传统的帮助线程的访存开销会高于主线程的计算开销, 从而导致帮助线程落后于主线程。 我们提出了一种改进的基于参数控制的帮助线程预取模型, 该模型采用梯度下降算法对控制参数求解最优值，从而有效地控制帮助线程与主线程的访存任务量，使帮助线程领先于主线程。实验表明， 基于参数选择的线程预取模型能获得 1.2～1.5倍的系统性能加速比。（小四宋体）  
**关键词**：**数据预取 帮助线程 多核系统 访存延迟 梯度下降**（四号宋体加粗，词组间用空格分开）

**ABSTRACT**

The applications with irregular accessing memory would causes serious Cache in the run-time. Helper thread is an effective technology to improve performance of multicore systems. Helper thread pre-fetches data from Memory to the Cache which is the closest one to CPU. If the overhead of memory access for a given application is far greater than that of computation, it would make helper thread lag behind the main thread. Hereby, we propose an improved helper thread pre-fetching model by control parameters. Furthermore, gradient descent algorithm is one of the machine learning algorithm which is adopted here to determine the optimal control parameters. The amount of the memory access tasks are controlled by the control parameters effectively, which makes helper thread be implemented ahead of main thread. The experiment results show that the speedup of the application are achieved by 1.2 times to 1.5 times.

**Key Word**: **data pre-fetch, helper thread, multi-core system, memory latency, gradient descent**

# 目 录（小二黑体）

中文摘要

**ABSTRACT**

目 录（小二黑体） 3

第一章 绪 论（小二黑体） 4

1.1 课题的研究背景和意义 4

1.2 国内外研究现状 5

1.3 课题研究的内容和主要工作 6

1.4 论文结构 7

第二章 相关研究工作 7

2.1 帮助线程技术理论基础 7

2.2 预取要考虑的要素 7

2.2 预取技术分类 8

2.2.1 硬件预取 8

2.2.2 软件预取 9

2.2.3 软硬件结合预取 9

2.3 总结 9

第三章 非规则数据预取技术研究 10

3.1 非规则应用程序的特点 10

3.2 帮助线程预取分析 10

3.3 帮助线程预取算法 11

第四章 预取算法实现 11

4.1 预取参数的定义 11

4.2 预取算法-梯度下降 12

4.2.1 梯度下降算法 12

4.2.2 预取算法流程图 12

第五章 实验验证 14

5.1 实验环境 14

5.2 实验benchmark 14

5.2.1 benchmark介绍 14

5.2.2 benchmark实验结果分析 15

第六章 总结与展望 16

参考文献 16

# 第一章 绪 论（小二黑体）

## 1.1 课题的研究背景和意义

在微处理器的发展进入多核时代之后，虽然存储器的访问速度也在逐渐提高，但是处理器的运算速度远远超过了存储器的访问速度，计算和访存的速度差距进一步拉大，例如，在运行SPECINT2000的测试集时，Alpha系统有超过50%的时间用于等待Cache失效[2]。因此，存储墙[1]仍然是制约微处理器性能提升的一个重要瓶颈。

[2] Beyls K, Hollander E H. Reuse Distance-based Cache Hint Selection[C]. Proceedings of the 8th International Euro-Par Conference on Parallel Processing, 2002.

通常计算机的层次存储结构由主存、寄存器和多层Cache,离处理器越近,容量越小，但访问速度越快,离处理器越远,容量也越大，但访问速度越慢,如图1.1所示，克服cache数据缺失问题已经成为微处理器设计的关键。到目前为止,人们提出了各种方案以减少或者隐藏由于cache缺失带来的访问存储器的延时：一是使用更大更快的存储器,缩短访问时间，但是不能从根本上解决数据缺失的问题；二是采用多级Cache存储结构，在处理器和主存储器之间加入多个层次的 Cache，利用程序运行时对数据的访问具有局部性的特征，将程序局部用到的数据提前缓存到Cache，减少数据缺失。虽然利用程序的局部性特点可以改善程序的访存性能,却并不能改变命中时间和失效损失,本质上只是减少Cache缺失次数；三是利用数据预取技术来隐藏访存延迟，在存储器访存的同时让处理器进行计算等其他有用的工作，开发计算访存时间并行性。数据预取和程序局部性优化的技术不同,数据预取的目标是隐藏访存延迟，而利用局部性进行预取仅仅减少了cache缺失的次数以及访存的次数。因此特点，数据预取技术成为缓解存储墙问题的重要手段之一。

数据预取[2]技术充分挖掘访存和计算的并行性，利用访存和计算的重叠，在处理器访问数据之前预测其访存地址并提前发出访存请求，将数据预取到离处理器更近的cache中，隐藏因Cache缺失而引起的访存延迟。数据预取可以由很多种技术实现，从硬件设计的角度,目前存在的技术有增加指令Cache块大小、顺序预取、基于步长的预取、基于分支预测的预取以及Trace Cache。从软件设计的角度，通过对编译器和处理器通信的分析,人们又提出由编译器控制的预取技术,在编译时加入特殊的预取指令,在指令被用到之前发出预取请求。它们都将访存和计算重叠起来,隐藏了访存延迟,提升了访存性能。预取是建立在利用存储器的空闲频带的基础上的,数据预取操作的目的是提前将数据从远离处理器的存储层次上移动到靠近处理器的存储层次上,甚至是寄存器中,以减少数据使用时的访存延迟等待。预取操作和处理器的计算过程并发进行,存储系统提前一定的时钟周期将数据从主存读取到Cache中。理想情况下,数据预取能够及时将数据预取到缓存中，从而避免处理器阻塞。同时预取的负面效应也是必须考虑的因素，例如Cache污染和带宽的浪费。

随着应用程序的结构越来越复杂，访存集越来越大，访存延迟对访存密集型的程序的影响也越来越大。多核处理器的出现和发展，给数据预取技术带来新的机遇和挑战。基于帮助线程的数据预取技术，在研究领域已经收到了广泛的关注和研究。文献[]利用多核处理器的空闲资源，利用帮助线程技术将主线程所需的数据预取到末级cache，并取得了一定的效果。相对于基于历史访问数据预测的预取技术，帮助线程技术利用空闲的硬件资源，仅执行长延迟的访存代码，并在主程序使用数据前将数据预取到离处理器更近的存储层次，达到预取主线程所需数据的目的，进而减少了主线程访问内存的次数。尤其重要的是，预取的线程先于主线程执行，并在主线程需要数据前触发缓存缺失，从而隐藏主线程的访存延迟。

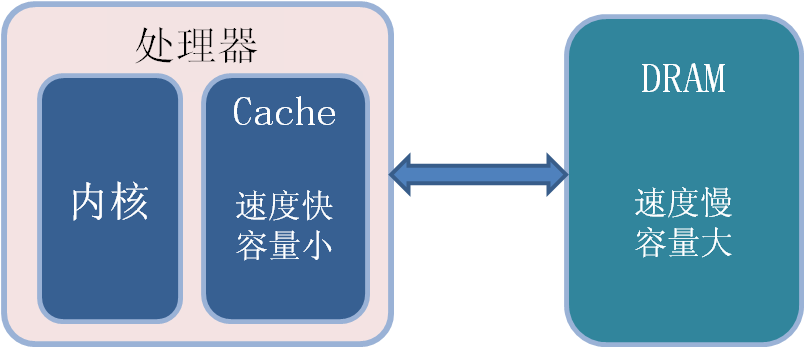


图1.1 计算机存储层次

## 1.2 国内外研究现状

随着计算机体系结构的发展，目前，基于帮助线程数据预取技术研究已经非常普遍。研究者从不同平台、不同角度等各方面都进行了大量的研究工作。帮助线程的数据预取技术，目的是利用空闲的硬件上下文执行数据预取线程，提高主线程的执行效率。帮助线程与主线程并行执行，并提前将主线程需要的数据从内存预取到 Cache 中，减少主线程的访存时间，提高系统性能。

帮助线程技术可以提高访存密集型应用程序的性能，可以通过硬件与软件的方法实现[7]。硬件方法需要在处理器中增加相关的硬件，通过指令窗口动态生成帮助线程，这种方法产生的帮助线程不需要软件上的同步，但是硬件复杂度比较高。软件方法是对程序的源代码进行分析，切片，利用语义信息，由编译器在源代码级别显式插入预取线程代码，然后利用空闲的硬件上下文去执行一个或者多个帮助线程，并优先于主线程将数据预取到cache。相对于硬件预取，软件预取的优势在于不需要复杂的硬件支持，并且易于实现。

目前，对于帮助线程的研究主要集中在帮助线程的构造，帮助线程与主线程的同步，以及帮助线程预取及时性的优化。 Kim等人[8]利用Unravel切片工具和斯坦福大学SUIF编译框架在源代码级完成了帮助线程的自动构造。他们利用PV操作来进行主线程和帮助线程之间的同步，利用全局计数器与局部计数器控制控制帮助线程的速度，只有当两个计数器之间的差异大于一个特定的阈值PD（预取距离）时，帮助线程才继续运行。Yong Hong Song等人[9]在SUNSPARC平台上基于编译实现了帮助线程的构造方法，他通过判断帮助线程的收益来进行构造,采用粗粒度和细粒度相结合的同步预取算法，但是由于热点循环体内采用了细粒度的同步机制，导致同步开销过大，反而使帮助线程数据预取的性能变低。Ou Guo dong提出的基于线程的数据预取方法[10]，在处理器上添加动态预取线程构造逻辑和预取线程控制逻辑来分析应用程序中访存行为的特点，从主线程的执行行踪中提取数据预取线程，使用空闲的现场和主线程并行执行。Yu Ji yang提出了一种线程感知的自适应的数据预取方法[11]，根据线程动态反馈信息将线程进行分类，计算个线程的访存特性和预取特性指标，从硬件层面控制线程的竞争，但是需要物理模块的支持。Lu Ji wei提出了基于多核粗粒度的同步控制算法，然而对于计算量很小的情况下，导致帮助线程落后于主线程，不能及时预取主线程所需数据。

[14] Lu Jiwei,Das A,Hsu W C,et al.Dynamic helper threaded prefetching on the Sun UltraSparc CMP processor[C]//Proc of the 38th Annual Int Symp Microarchitecture.New York:ACM.2005.93-105.

以上对帮助线程预取的研究大多集中于主线程和帮助线程的构造与同步机制。但是，对于实际的应用程序，在应用程序计算开销很小的情况下，帮助线程承担全部的访存任务，主线程只承担小部分的计算任务，帮助线程不一定总快于主线程，从而导致帮助线程落后于主线程。此时就会频繁产生同步操作，导致系统性能下降。因此，在特定的情况下，主线程如果能承担一部分访存任务，从而使得帮助线程与主线程负载均衡，提高系统访存和计算性能。

## 1.3 课题研究的内容和主要工作

传统的基于时空局部性的数据预取技术通过减少cache数据缺失的方法，仅仅对于访存规律的应用程序带来一定的性能提升。但是对于非规则数据密集性的应用，由于此类应用程序往往包含树和图等复杂的数据结构，上述方法不能发挥很好的作用。

本文是面向非规则计算的应用程序，通过对帮助线程技术研究，提出一种改进的基于梯度学习的帮助线程预取模型。该模型主要针对帮助线程的预取时机，预取工作量，以及和主线程的同步频次这三个问题进行分析和研究。根据应用程序访存的访存任务和计算任务的比例，帮助线程的开销可分为两种情况，第一，对于计算密集型的应用程序，帮助线程承担全部的访存任务。第二，对于访存密集型的应用程序，帮助线程承担部分的访存任务。如果使帮助线程取得较好的性能，我们必须根据各个程序不同的访存开销与计算开销来调整帮助线程预取数据量的大小。帮助线程从热点程序入口处开始跳过个数据之后才开始推送个数据，从而提高帮助线程预取数据的有效性。在进行预取的时候，一方面要保证帮助线程能够及时地预取到主线程所需要的数据，另一方面又要保证帮助线程不能落后或超前于主线程太长的距离，从而导致有用数据被替换掉，造成多核平台的最后一级缓存污染。

本文针对上述问题，进行深入研究，主要研究的内容如下：

1. 对现存的预取技术进行分析和研究，找出各自的优缺点。
2. 对帮助线程预取技术的理论基础进行介绍和分析；
3. 分析帮助线程进行预取应该考虑哪些因素；
4. 对前期相关的研究工作进行总结，并在此理论基础上进行分析和优化，提出自己的研究思路；
5. 通过分析非规则应用程序的特点，提出一种改进的面向非规则计算的帮助线程预取算法，并在实际机器上进行分析和验证自己算法的准确性。

## 1.4 论文结构

本文的组织结构如下：

第一章，主要介绍了课题的研究背景和意义，综述国内外的研究现状并确定本课题的主要研究内容。

第二章，主要介绍了相关的研究工作，包括预取要考虑哪些因素以及现有的预取方法的优缺点。

第三章，主要介绍了非规则数据预取的特点，分析了如何使用帮助线程对非规则应用程序进行数据预取。

第四章，提出了一种面向非规则计算的帮助线程预取算法，对现存的预取算法的缺点进行分析，并进行改进，从而提出一种改进的基于梯度学习的预取算法。

第五章，对本文提出的算法进行实验验证，选取具有代表性的基准测试程序进行了实验分析和研究。

第六章，总结与展望，该章节总结了本课题的主要工作的和贡献，同时也对存在的问题进行了反思，最后对后续工作进行了展望。

# 第二章 相关研究工作

本章首先讲述了帮助线程技术的理论基础以及帮助线程预取要考虑的三个关键因素，然后介绍了国内外的各种预取技术，包括软件预取，硬件预取，以及混合预取，最后对这几种技术的优缺点进行总结。

## 2.1 帮助线程技术理论基础

在多核系统中，利用帮助线程进行数据预取一种常用的隐藏访存延迟的方法。多核处理器通过在一个芯片上集成多个计算核心，每个核心都是一个相对简单的微处理器，多个计算核心可以并行的执行程序代码，其结构如图2所示。由于计算核心的增多，帮助线程可以利用硬件空闲资源与主线程并行执行，在主线程需要数据之前把数据预取到离处理器更近的片内共享缓存，以减少主线程的访存延时，从而提高主线程的执行效率。

帮助线程

Core1

CPU

L1 Cache

L2 cache and control

System bus

Core2

CPU

L1 Cache

图2 多核系统体系结构

## 2.2 预取要考虑的要素

对于非规则数据访存引起的cache缺失，通过帮助线程进行数据预取是个很好的选择。帮助线程预取是在cache失效之前提前访问存储器，将数据读取到cache, 在CPU真正需要这些数据时候能够直接从缓存得到，从而减少内存延时和阻塞的时间。帮助线程和主线程的计算过程并发进行,存储系统提前一定的时钟周期将数据从主存读取到Cache中。但是帮助线程的时效性如何得到保证是一个一直研究的问题。理想情况下，如图1所示，帮助线程能够为处理器及时提供所需的数据,从而避免访存延迟。

综上所述，如何利用帮助线程进行预取，从而保证预取的时效性，我们要考虑以下三个因素：

1. 帮助线程什么时候进行预取；如果帮助线程预取时间控制不当，就会造成大量无用数据的预取，从而造成带宽的浪费以及cache的污染。预取时机过早或过晚，都会导致数据在没有使用前就被替换出去或者是要使用时还未取回。
2. 帮助线程一次预取多少数据；控制帮助线程每次的预取工作量可以减少帮助线程的访存开销，缓解处理器核存储器之间的通信压力。
3. 帮助线程与主线程的同步频次；控制好帮助线程与主线程的同步频次，使帮助线程不能过于超前或者落后于主线程，从而避免大量的无效预取。

Compute

Load

Compute

Load

Compute

Load

源程序：

Compute

Compute

Compute

主线程： Compute：计算操作

Load：访存操作

帮助线程：

Load

Load

Compute

图1：理想情况下帮助线程预取

## 2.2 预取技术分类

### 2.2.1 硬件预取

基于硬件的预取技术主要针对访存规则的数据进行预取。硬件预取[3,4]在预取引擎的控制下对程序访存的模式进行识别和预测，通过专用的硬件机制来预测可能会发生的失效，自动地进行预取。因此，这种预取方法要求应用程序的数据相对集中，并且访存具有一定的规律性，便于通过硬件进行预测未来可能要访问的数据。

从硬件设计角度来看，微处理器的访存延迟主要是由总线传输延迟，处理请求延时，以及访存延时组成。为了隐藏访存延迟，常见的硬件预取方法有基于步长的预取。基于步长的预取针对访存地址序列有等距离的步长，首先动态识别步长序列中的步长，然后提前发出预取请求。由于步长序列出现在循环中，如果一定的步长的访问来自相同的load指令，只要跟踪这条load指令的访存序列，步长就可以被计算出来。通常我们采用访问预测表的方法来确定步长，将每次cache缺失的指令地址和数据地址插入到访问预测表中。如果有相同的load指令再次发生失效，其在访问预测表中命中，此时步长就等于两次访存的地址之差。

由于硬件预取仅仅依赖于访存的历史信息，预取的准确率不能得到保证，会导致大量无用数据的取回，占用访存带宽。

### 2.2.2 软件预取

现在大多数微处理器都提供了预取指令来支持软件预取。软件预取[5]是通过分析程序代码，由程序员或编译器在代码中插入预取指令，提前将数据取入 Cache，从而避免在计算的时候由于数据缺失而导致的执行暂停。

目前大多软件预取方法都是基于编译器的框架下实现的，利用编译器实现源代码的分析，切片以及包含预取代码的生成。程序员可以不必关心帮助线程的构造，而致力于程序代码的实现。

### 2.2.3 软硬件结合预取

由于软件预取和硬件预取的缺点，随后提出了软硬混合预取方法,主要应用在支持多线程的处理器上, 利用硬件支持单独的线程预取数据并用软件的方法实现预取线程与计算线程的同步。由于软件给了硬件一些程序的信息，克服了单纯的硬件预取的盲目性，通过编译提供的预取时机，预取步长，以及预取的元素等信息，指导硬件完成预取动作。Won W.Ro等人实现了一种软硬件结合的数据预取模型[18]。Won W.Ro, Jean-Luc Gaudiot. Compiler Support for Dynamic Speculative Pre-Execution[J]. Interaction Between Compilers and Computer Architectures, 2003:14-23.

模型主要由两部分构成：帮助线程的构造和触发。他们首先利用软件技术，通过编译器静态的实现帮助线程的构造，并对全局的控制和数据信息进行分析，提高预取效率。然后利用硬件技术来实现帮助线程的触发，有效的降低了线程同步的开销。但是此种方式依赖于指令队列的大小，以及对分支预测的效率较低。

## 2.3 总结

综上所述，各种预取方法的优缺点如表1 所示，硬件预取技术往往具有一定的盲目性，并且硬件开销大。软件预取依赖于编译器，会带来一定的预取指令开销。软硬件相结合的数据预取方式，不仅最大限度的降低了硬件实现的复杂度，而且能对全局的数据流和控制流进行掌控，使得数据预取工作更加高效。但是此种方法需要特定的软硬件的支持，其移植性较差,无法保证在所有体系结构的处理器上都适用。

表1：各种预取方案的优缺点

|  |  |  |
| --- | --- | --- |
| **方案** | **优点** | **缺点** |
| 硬件预取 | 没有指令开销，不需要编译和程序员的支持 | 基于历史信息，不能结合程序的特点，硬件开销大 |
| 软件预取 | 结合程序的特点，预取准确度高 | 增加额外指令开销， |
| 软件硬件结合 | 结合软件硬件预取优点 | 需要软硬件同时给予支持 |

本文将重点介绍基于软件的帮助线程的预取方法，通过分析各种预取方法的优缺点，针对于非规则访存的应用程序，克服传统预取方法的缺点，提出一种基于梯度学习的帮助线程的预取模型。

# 第三章 非规则数据预取技术研究

## 3.1 非规则应用程序的特点

非规则计算出现在许多重要的科学领域，例如计算流体力学，分子动力学以及机器学习。这类应用往往具有以下特点【35】：

（1）存在大量的非规则计算，具体表现为算法和程序的计算依赖关系和访存模式的非规律性，从而使其局部性和并行性受到限制。此类应用程序通过直接实现或依赖编译器的自动优化一般很难获得高性能。

（2）数据以不可预知的、无规律的方式进行访问。不能根据历史访问地址表来预测将要访问的数据。

（3）一般采用非线性的数据结构，常包含有树和图等复杂的数据结构，增加了数据预测的难度；

此类应用程序的数据存取呈现非规则性，数组存取的下标是循环索引变量的非封闭表达式或者数组存取下标包含处理处理循环索引变量的函数或数组【6】，如图1所示的程序片段，编译器的静态分析和优化方法都无法预知数据的存取模式，因此无法对这种非规则引用模式进行优化。

DO I= IB NON B , N BC

DEN B I O( I)= DNOI LC( IKCB ( I ) )\* 2. 309

END DO

图 1 某生产用油藏数值模拟程序中的非规则计算程序片段

Fig. 1 Code seg ments fo r ir regular co mputing in petr oleum reserv oir simulation

（[35] J.Borkowski, D.Kopanski, M.Tudruj. Global predicate monitoring applied for control of parallel

irregular computations[J]. Parallel, Distributed and Network-Based Processing, 2007:105-112.

（[ 6 ] Yuan Lin. Com piler analysis of s parse and irregular computations [D]. Illinois : Univ ersity of Illinois at Urbana-Ch am paign,M ay 2000.））

随着大数据的兴起，人们已经从编程模型、语言和算法等方面，对一些经典的非规则应用程序的访存模式进行了研究，但是由于其具有上述特点，对其进行有效的预取，只能通过执行代码本身来预测。本文将主要研究非规则的数据访问，并针对非规则计算的两个特点，通过基于帮助线程的数据预取技术来解决非规则数据访问带来的访存延迟等方面的问题。

## 3.2 帮助线程预取分析

帮助线程预取技术，在多核研究领域已经受到了广泛的关注。相对于其他预取技术，帮助线程只需承担主线程的部分访存操作，并在主程序使用数据前将数据预取到离处理器更近的存储层次，达到预取主线程所需数据的目的，进而减少了主线程访问内存的次数。帮助线程与主线程并行执行，且领先于先于主线程，在主线程需要数据前触发数据去取，从而隐藏主线程的访存延迟。

帮助线程预取技术实质上是一种Leader/Follower结构，帮助线程是去除了了原有线程计算任务的“精简版本”，它往往比原有线程运行得快，因此帮助线程可提前于主线程发出长延迟访存请求，从而达到加速程序执行速度的目的[12]。帮助线程仅仅起到预取的作用，不修改主线程的体系结构状态，因此不会引起程序的错误执行。在理想的情况下，主线程需要某个数据的时候，帮助线程恰好能将需要的数据预取到LLC。但是，如果访存开销和计算开销差别较大的时候，帮助线程并不能每次领先主线程，从而导致预取的数据不能及时到达LLC，还可能造成Cache污染。根据不同程序中访存开销和计算开销的规模，可将程序划分为以下三种类别。由于非规则访存带来大量的访存开销，使得程序的访存开销远大于计算开销，因此本文着重针对第二种模型提出参数控制的帮助线程预取方法。设访存时间为，计算时间为。

1. 计算开销与访存开销大小相当,即。此时帮助线程能够很好地发挥作用。
2. 计算开销大于访存开销，即。此时要控制好帮助线程的预取时机，防止过早预取，从而导致真正使用的时候数据已被替换出去。
3. 计算开销小于访存开销，即。此时主线程计算开销小，帮助线程访存开销大，主线程需要数据时候帮助线程还没有预取到，主线程可能要进行多次同步操作，等待数据的到达。

对于非规则数据的计算密集型应用程序，通常使用图、树或者链表等数据结构，其访存行为呈现非规则性，访存模式难以在静态编译阶段进行准确预测[6]。由于其可利用的局部性受到限制，使得传统的软硬件预取方法失效，其访存模式只能通过执行代码本身来进行预测[6]。本文采用帮助线程预取技术能够有效处理非规则访存程序。帮助线程负责访存任务，主线程负责计算任务。帮助线程提前将主线程所需的数据预取到LLC（Last-level cache）末级缓存，从而达到隐藏访存延迟的目的。我们采用梯度下降算法确定帮助线程与主线程的访存开销，通过参数控制的方法使得多线程预取程序的性能达到最优。

## 3.3 帮助线程预取算法

（1）纯遍历推送

纯遍历推送方法的核心思想是：帮助线程和主线程以同样的控制流逻辑执行程序，但是帮助线程中剔除了那些与计算相关的语句，只执行访存操作。因此，帮助线程总能领先于主线程。此方法在主线程设置一个计数器m，表示主线程当前循环遍历的迭代次数；在帮助线程中也维护一个计数器p,表示帮助线程的循环迭代次数。为了保证主线程和帮助线程的以恰当的速度运行，设置同步块block\_size，表示帮助线程的领先于主线程的幅度要在block\_size范围内。在初始情况下，m，p均为0，帮助线程领先于主线程开始执行，帮助线程每经过一个节点，计数器p加1，主线程每经过一个节点，计数器m加1。在主线程和帮助线程的并行运行过程中，p和m的值都在不断变化，当p-m≥block\_size时，表示帮助线程过于领先主线程，此时，帮助线程进入等待状态，循环检测p和m的值，知道p-m<block\_size,帮助线程才继续运行。整个过程，帮助线程运行，等待，在运行的状态如图1所示。

主线程

帮助线程

Block\_Size= 4

1. m=0,p=2

主线程

帮助线程(block\_size=4,等待)

Block\_Size= 4

1. m=2,p=6

主线程

帮助线程(block\_size<4,继续运行)

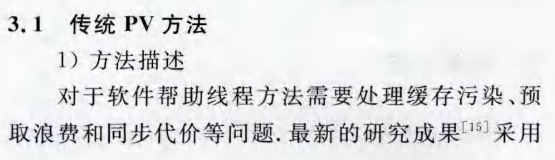
Block\_Size= 4

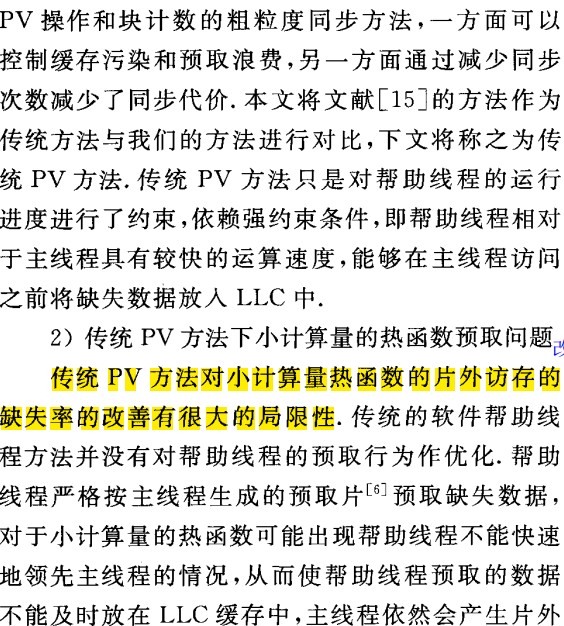
1. m=3,p=6

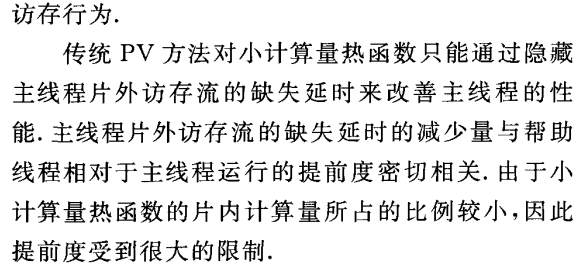
图1 纯遍历推送

但是此算法需要设置主线程和帮助线程计数器，并且帮助线程要时刻检测是否超过了主线程，增加了帮助线程的负担。

（2）传统的PV方法







本文提出的帮助线程预取模型，基于多核平台，直接通过分析程序的热点模块来构造帮助线程，不需要对主线程的计算任务进行分解，在访存量远远大于计算量的时候，主线程要承担部分的访存任务，并通过适当的控制策略来保证帮助线程预取的时效性和准确性。

# 第四章 预取算法实现

## 4.1 预取参数的定义

传统的帮助线程预取技术往往没有考虑计算量和访存量的大小，无法构造一个相对于主线程有效的帮助线程。因此我们采用GU ZhiMing等人提出的KPB[13]（Skip-Push-Block）参数,也即所有预取算法中必须考虑的三个参数，在分析原程序计算访存工作量的前提下，通过动态调整K、P、B三个参数值，使得帮助线程的性能达到最优。预取模型如代码1所示，帮助线程跳过k个数据，然后预取P个数据。

1. K即skip，表示跳过多少个数据，即主线程负责skip个数据的访存，其他的访存任务由帮助线程来完成，此参数主要用于控制帮助线程预取的触发时机。若程序的计算开销远远大于仿存量，此时，与传统的帮助线程预取机制一样，帮助线程承担全部的访存工作。
2. P即push，表示帮助线程给主线程推送多少个数据，即帮助线程预取的数据量。此参数用于控制帮助线程预取工作量的大小。
3. B即block，表示帮助线程与主线程多长时间同步一次。一般情况下。此参数用于控制帮助线程与主线程的同步频次，本文采用文献[9]所述的线程间同步机制。

While（i<n）

{ //帮助线程跳过skip个数据，主线程负责skip个数据的访存与计算

For（j=0；j<skip; j++）

{

Load（data[i]）；

Computer（data[i]）；

}

//帮助线程预取prefetchSize个数据

For（k=skip；k<prefetchSize+skip; k++）

{

Prefetch(data[k]);

}

}

代码1：KPB帮助线程模型伪代码

### 4.1.1 K值范围分析

K值表示主线程承担的访存任务量，K的值取决于程序的不同的工作量，我们根据访存任务与计算任务的比值来确定K的大小。

**定义1**  在源程序的执行过程中，其热点区域的计算任务和仿存任务的比值，我们记做R，，，即热点区域计算/访存延迟。

如果R大于1，此程序属于计算密集型程序，参数K的值为0；如果R小于1且接近于0，此程序属于访存密集型程序，此时K的值大于0。K的最小值可以用公式（1）表示：

 (1)

其中,Miss Latency失效延迟是指热点循环中平均的内存延迟时间，Iteration Time迭代时间是指热点循环中每次循环执行的时间。K的最大值可用公式2表示：

 (2)

我们将cache容量和每次迭代的访存任务量的比例记做Rc，为了防止cache污染和平衡帮助线程和主线程的访存任务量，K<=Rc/2。在没有计算任务量的情况下，主线程和帮助线程平分所有的访存任务。

### 4.1.1 P值范围分析

**定义2** 帮助线程的预取比例，其定义如公式（3）：

 (3)

的取值范围为(0,1),如果，帮助线程能够对主线程起到一定的性能改善。P代表帮助线程的预取工作量，由公式（3）可以推出

 (4)

结合公式（1）（2），我们可以得出P的取值范围为：

 (5)

### 4.1.1 B值范围分析

B代表同步块的大小，代表主线程与帮助线程的同步频次，可以由K，P的值获得。一般情况下。

## 4.2 预取算法-梯度下降

### 4.2.1 梯度下降算法

由于梯度下降算法使用简单、收敛速度快、效果可靠等优点得到了普遍的应用。作为一种最优化的算法，它利用负梯度方向来决定每次迭代的新的搜索方向，使得每次迭代能使待优化的目标函数逐步减小。在梯度下降算法中，求函数极大值是沿着正梯度的方向走，可以最快达到极大值，沿着负梯度方向走，可以最快的达到极小值。

梯度下降法的步骤如下：

1. 确定代价函数；
2. 对参数θ赋值，这个值可以是随机的，也可以让θ是一个全零的向量；
3. 改变θ的值，使得按梯度下降的方向进行减少。

我们常用此算法来调整参数寻取最优值。相对于其他最优化算法来讲，梯度下降算法程序设计简单、计算量小、存储量小并且计算效率在最初几步迭代较高，常与其他方法一起使用；除此之外，对初始点没有特别要求，有着很好的全局收敛性。

目前帮助线程预取模型中参数的选取大多都是通过枚举实验来获取，根据帮助线程的预取率，，选取符合范围的K,P进行多次试验，最后选取实验中CPI最小的K,P参数组合即为最优的。本文使用的梯度下将算法主要代码如下：



plot\_linerRegression\_Graident<-function()

{

print(require(ggplot2));

data<-read.table('ex1data1.txt',sep=","); #read data from txt file

X<-as.matrix(data[,1]);#get X matrix

y<-as.matrix(data[,2]);# get y matrix

X<-formulization(X);

# initial\_plot<-point\_plot(data.frame(x=X,y=y));

m<-nrow(X);

#formalizate matrix X

#add 1 to matrix X

X<-as.matrix(cbind(rep(1,m),X))

#initialize theta

theta<-as.matrix(rep(0,2));

#get the theta

theta<-gradient(X,y,m,theta,0.01);

theta\_1<-theta[1,1];

theta\_2<-theta[2,1];

# plot contour

x\_axis<-seq(-1,1.5,length=50);

y\_axis<-seq(-1,1.5,length=50);

z\_axis<-matrix(rep(0,50\*50),ncol=50,nrow=50);

xy\_axis<-matrix(nrow=50\*50,ncol=3);

sum<-1;

for(i in 1:50){

for(j in 1:50){

xy\_axis[sum,1]<-x\_axis[i];

xy\_axis[sum,2]<-y\_axis[j];

xy\_axis[sum,3]<-computeCost(X,y,m,as.matrix(c(x\_axis[i],y\_axis[j])));

z\_axis[i,j]<-computeCost(X,y,m,as.matrix(c(x\_axis[i],y\_axis[j])));

sum<-sum+1;

}

}

theta\_jval<-data.frame(axis\_x=xy\_axis[,1],axis\_y=xy\_axis[,2],axis\_z=log(xy\_axis[,3]));

contour<-ggplot(theta\_jval,aes(axis\_x,axis\_y,z=axis\_z))+

geom\_contour()+

scale\_x\_continuous(limits=c(-10,10))+

scale\_y\_continuous(limits=c(-1,4));

#create persp

plot2<-persp(x\_axis,y\_axis,z=z\_axis ,theta = 135, phi = 30, col = "green3", scale = FALSE,ltheta = -120, shade = 0.75, border = NA, box = TRUE,xlab="theta\_0",ylab="theta\_1",zlab="J(theta)");

}

### 衡量KPB参数最优的标准

本文选取CPI作为参数最优的的衡量标准，CPI表示某个程序的平均每条指令执行所需的时钟周期。大部分的cache研究都使用cache的失效率和CPI作为衡量性能的标准，但是对于带有预取算法的程序来讲，cache失效率只能作为算法性能的一个参考，不能作为一个合理的衡量尺度。比如，对于一个处理器来说，完全的失效延迟为4个cpu周期，如果一个算法提前3个周期发起预取请求，而另外一个算法提前1个周期发起预取请求。此时，cache的失效率是一样的，但是最终的性能是有差别的，第一种算法减少了2个周期的延迟，而第二种算法仅仅减少了一个周期的延迟。因此，从性能方面考虑，第一种预取算法优于第二种预取算法。

综上分析，用CPI作为梯度下降参数选取最优的衡量标准更加合理，，其中T表示程序执行的总的时钟周期，IC表示总共的指令条数。总的时钟周期等于指令的正常执行周期数加上cache失效的时钟周期数。有了时钟周期数，就能计算出CPI的值，当CPI最小时参数的值，代表梯度下降算法的最优解，此时帮助线程的性能达到最优。

### 4.2.2 预取算法流程图

本文采用梯度下降算法来进行帮助线程参数的构造。基于梯度学习的参数控制预取模型主要由两部分组成，热点分析和代价函数的构造。预取算法的基本流程如图1所示。具体步骤如下：



图1 参数控制的预取模型流程图

（1）确定程序的热点部分

程序的热点部分，也即程序的性能瓶颈，往往是由于长延迟访存指令引起。我们采用Intel 的性能分析工具Vtune[14]对程序进行离线Profiling，收集CPU的时钟周期和共享Cache的缺失信息，找出引起Cache缺失的长延迟访存指令，从而确定要进行预取的热点循环部分。

（2）构造代价函数

由于KPB值的选取与源程序的访存任务量（访存所消耗的时间）和计算任务量（计算所消耗的时间）的大小密切相关。我们使用Vtune性能分析工具，分析热点程序的访存任务量与计算任务量的大小。为确保程序性能最优，我们采用梯度下降算法来寻取最优值，我们将性能最优是所满足的条件设为代价函数,记作。



根据预取模型，主线程负责的任务是：（I）个数据的访存与计算 (II)个数据的计算，可记做,其中，为单次循环的访存时间，为单次循环的计算时间。帮助线程负责的任务为：（I）个数据的预取，可记做。理想情况下，帮助线程与主线程完全并行，此时程序性能达到最优，即



 (1)



的绝对值最小。假设给主线程分配的访存任务为，根据经验可知，随着的增加，也增加，，的关系可设为公式(2):

(2)

假设帮助线程分配的访存任务为，同样，根据经验可知，随着的增加，也增加，，关系可设为公式(3):

 (3)



我们将(2) (3)代入(1)得, 根据上述推断，可知代函数:



 (4)



(3)计算最优的，



梯度学习作为一种求解最优参数的迭代算法，广泛应用于机器学习各式model参数的求解中。梯度下降算法是一种迭代方法，先随意选取初始，然后不断的以梯度的方向修正，最终使收敛到最小。因此，本文选择梯度下降算法进行最优值的求解，通过选择不同的, 的样本值，我们可以训练出满足代价函数最小的,。



我们通过，即

 (5)



我们又有

 (6)



通过公式(5)(6)，我们可以得出近似最优解，的值。由于求解的，可能为小数，因此我们可以在近似最优解附近,选三组整数数据进行测试，通过运行程序，使用Vtune分析实验结果，得到**CPI（Clocks per Instruction）**最小的，即为最优解。



# 第五章 实验验证

## 5.1 实验分析工具以及环境

### 5.1.1 Vtune分析工具

VTune是Intel一个比较强大的性能分析软件，可以对并行程序的运行过程进行分析。主要包括一下三个小工具：

（1）Performance Analyzer：性能分析，能够找到软件性能比较热的部分，一般也就是性能瓶颈的关键点，可以对软件操作进行时间上的总结和统计，用户可以根据自己需要对数据进行分析。

（2）Intel Threading Checker：用于查找线程错误, 能够检测资源竞争、线程死锁等问题，将程序运行中发生的所有读写操作和死锁现象体现给用户，可以通过此功能检测是否有多线程相关的错误。

（3）Intel Threading Profiler：线程性能检测工具,此功能可以对多线程执行进行一个统计和图形展示。多线程可能会存在负载比平衡, 同步开销过大等线程相关的性能问题，该工具可以发现每一个线程每一时刻的状态，帮助我们提高线程应用的性能。

本文利用Vtune来分析程序的热点循环部分以及帮助线程与主线程的负载情况，为实验操作提供支持。

### 5.1.2 实验环境

本文选取的硬件实验平台为X86，处理器是 Intel® Core™ 2 Q6600 四核处理器，该处理器共有8MB的二级高速缓存，每对核共享4MB的二级高速缓存。在linux环境下进行实验。

表3：硬件配置

|  |  |
| --- | --- |
| 处理器 | Intel® Core™ 2 Q6600 |
| Memory | 2 GB (DDR 667, non-ECC) |
| L1 Data-Cache | 32 KB × 4 8 set-association cache line 64 bytes |
| L1 I-Cache | 32 KB × 4 8 set-association cache line 64 bytes |
| L2 Cache | 4,096 KB × 2 16 set-association cache line 64 bytes |

## 5.2 实验验证

### 5.2.1 benchmark介绍

本文实验选取的测试程序为Olden Benchmark中用于科学计算的测试程序 EM3D (Electromagnetic wave propagation in a 3D object)、MST(Minimum spanning tree of a graph)，SPEC CPU 2006中的MCF进行帮助线程预取性能的评估。通过Vtune性能分析工具的分析，本文选取的热点模块以及输入集如表1所示，分别为EM3D中的fill\_from\_field，MST中的hashLookup，MCF中的refresh\_potential模块。

MST是一个求解最小生成树的Olden Benchmark程序，此程序主要数据结构为链表，其中链表的每个节点包含一个hash结构，程序在运行的过程中，cache的缺失主要发生在对hash表的节点的查找上。即图 5.1 中 Hash table 所指向的链表(b)。

EM3D是Olden Benchmark中模拟电磁波在一个三维对象内传递的过程的程序，该程序的主要数据结构是链表，其中链表的每一个节点都包含一个指向数组的指针。此程序中cache的缺失主要发生在访问数组的某个元素上。图 5.2 显示了 em3d 的数据结构图。

MCF是 SPEC CPU 2006 中的基准测试程序，主要用于在大众交通中的进行车辆调度，改程序的主要数据结构包括链式数据结构和数组。图 5.3 显示了 refresh\_potential 模块的数据结构图示。

表1 Benchmark 参数配置表

Tab.1 Benchmark parameter setting

|  |  |  |  |
| --- | --- | --- | --- |
| Benchmark | 热点函数(Hot function) | 输入(Input) | Suite |
| EM3D | Fill\_from\_fields | 400000nodes ,arity 128 | Olden |
| MCF | Refresh\_potential | Ref | SPEC 2006 |
| MST | Hashlookup | 10000 nodes | Olden |

### benchmark实验结果分析

#### 5.2.3.1 预取算法性能

如图2所示，EM3D，MST和MCF测试程序采用传统帮助线程（帮助线程负责全部的访存任务）、参数枚举法和本文的基于梯度学习的参数控制方法（参数学习法）相对于串行执行（不使用帮助线程的源程序）时的性能加速比，其中参数学习法获得了1.2-1.5倍的最高加速比。MST的Hashlookup模块属于访存密集型程序，使用传统的帮助线程方法与原串行程序相比加速比反而降低了4.8%；使用参数学习方法，性能提升了近50%。EM3D 的Fill\_from\_fields模块属于计算密集型的程序，使用参数学习方法与传统帮助线程方法获得的加速比相当，仅提高了4.9%。由于参数枚举法取决于经验与启发式实验，枚举粒度的大小直接影响到结果的准确性。粒度过小，需要进行大量的重复试验；粒度过大，可能错过最优值。因此，参数枚举法并不总能得到最优解。参数学习法不依赖与经验，而是通过梯度下降算法来获取最优值，比参数枚举法效率更高。

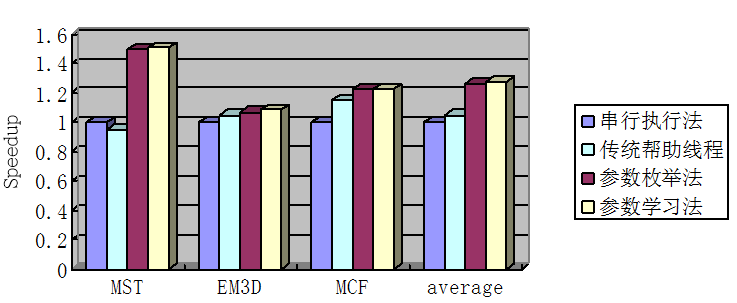


图2 性能加速比

Fig.2 Performance speedup of different methods

图3给出了测试程序在传统帮助线程、参数枚举法和参数学习法情况下的Cache缺失率的相对值。其中，MST，EM3D，MCF的热点程序的Cache缺失率相对于采用传统帮助线程的情况下分别减少了12%，10%，27%。MST，EM3D相对于参数枚举法分别减少了2.5%，1.7%。因此，通过机器学习的梯度下降算法取得K、P的最优值比参数枚举法效率更高。

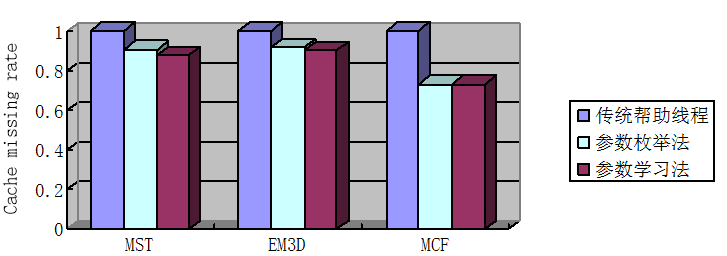


图3 Cache缺失率

Fig.3 Cache missing rate

#### 5.2.3.2 能耗

由于活动计算核(Active Cores)数量的增加以及对资源的竞争，采用帮助线程的程序执行相比于串行程序执行，将会在一定程度上增加功耗。如果帮助线程的收益大于额外增加的功耗，则体现了帮助线程的有效性。帮助线程额外增加的功耗表示相对于串行程序执行功耗[15]增加的比例，帮助线程收益表示相对于串行程序执行时间所减少的比例。如图4所示，帮助线程的平均收益大于平均功耗。其中，MST、MCF的收益均大于功耗，因此帮助线程能有效提升MST、MCF的执行性能。因为EM3D 属于计算密集型程序，访存量很小，帮助线程反而带来了很大的同步开销，不足以弥补帮助线程带来的收益，所以，帮助线程对提高EM3D执行性能的效果有限。

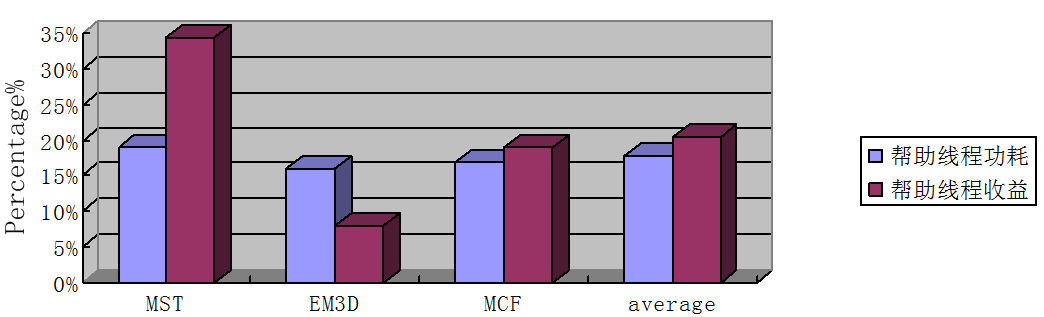


图4 帮助线程功耗和收益比

Fig.4 Comparison between the energy overhead ratio and the performance gain of the helper thread

# 第六章 总结与展望

通过分析传统的帮助线程不能有效的控制预取实时性和覆盖率缺陷，以及对访存密集型的程序预取效率低的缺点，本文提出了一种基于梯度学习的参数控制帮助线程预取模型。通过采用机器学习的梯度下降算法确定K、P的值，根据K、P的值选择性地预取部分数据，使得帮助线程与主线程的工作量相对均衡，从而使程序的执行性能达到最优。

由于帮助线程与主线程同时访存，可能引起带宽的竞争。因此，下一步的工作将考虑帮助线程对带宽的影响，具体分析程序的访存地址，适当增加预取步长，将预取相邻地址的预取指令进行合并，减少预取次数，从而可以降低带宽的竞争，提高执行性能，降低额外功耗。

# 参考文献

[1] Wilkes M V．The memory wall and the CMOS end point[J]. ACM SIGARCH Computer Architecture News，1995,23 (4):4- 6.

[2] Byna S,Chen Yong,Sun Xianhe.A Taxonomy of Data Pre-fetching Mechanisms[J].Journal of Computer Science and Technology,2009,24(3):405-417.

[3] Vnderwiel S,Lilja D. Data prefetch mechanisms[J]. ACM Computer Surveys(CSUR),2000,32(2):174-199.

[4] Mouneshwar K. An Energy Efficient Analysis of Hardware Prefetching Techniques:A Review[J].International Journal of Advanced Research in Computer and Communication Engineering,2012,1(8):557-562.

[5] Chi-Keung Luk. Tolerating memory latency through software controlled pre-execution in simultaneous multithreading processors[C].//Proc of the 28th Annual International Symposiumon Computer Architecture(ISCA),Goteborg,Sweden,2001:40-51．

[6] J.Saira Banu. Study on Data Prefetching techniques for Linked Data Structure Applications .IRACST,2(1), 2012:12-15.

[7] 张建勋,古志民.帮助线程预取技术研究综述[J].计算机科学,2013,40(7):19-23.

[8] Kim D,Yeung D. Design and evaluation of compiler algorithms for pre-execution[C]. //Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems ( SanJose, Calif). New York: ACM, 2002:159-170.

[9] Song Yong hong, Kalogeropulos S, Tirumalai P. Design and Implementation of A Compiler Framework for Helper Threading on Multi –Core Processors[C]. //Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques( PACT):Saint Louis,MO,USA , 2005: 99- 109 .

[10] Ou Guodong, Zhang Minxuan. Thread-Based Data Prefetching[J]. Computer Engineering & Science, 2008, 30(1):119-122.

[11] Yu Jiyang , Liu Peng . A Thread-Aware Adaptive Data Prefetcher[J] . IEEE ,2014:278-285

[12] Lee J, Jung C,Lim D, Solihin Y. Prefetching with helper threads for loosely coupled multiprocessor systems[J].IEEE Transactions on Parallel and Distributed Systems,2009, 20(9): 1309-1324.

[13] Zhang Jianxun, Gu Zhimin, Huang Yan,et al.Helper Thread Prefetching Control Framework on Chip Multi-processor[J]. Int J Parallel Prog ,2013,43:180–202.

[14] Intel Vtune Performance Analyzer for linux [EB/OL].2015.https://software.intel.com/en-us/ intel-vtune-amplifier-xe.

[15] Singh, K., Bhadauria, M., McKee, S. A Prediction-based power estimation and scheduling for CMPs[C].//

Proceedings of the 23rd International Conference on Supercomputing (ICS)2009: 501–502.