****

**毕 业 论 文**

题 目 基于深度学习的数据预取 质量优化算法设计与实现

姓 名 李芳达

学 号 13070038

指导教师 蔡旻

日 期 2017.5.15

**北京工业大学**

毕业设计（论文）任务书

题目 基于深度学习的数据预取质量优化算法设计与实现

专业 计算机科学与技术 学号 13070038 姓名 李芳达

主要内容、基本要求、主要参考资料等：

**主要内容：**

（1）了解数据预取的基本原理、算法实现和相关工作。

（2）了解深度学习的基本原理、算法实现和相关工作。

（3）熟悉多核体系结构模拟器的使用与扩展编程。

（4）在多核体系结构模拟器中实现基于深度学习的数据预取质量优化算法，并对其性能与功耗进行分析比较。

**基本要求：**

1．参与本课题的同学将根据用户需求，进行系统分析、系统设计、系统实现。

2．系统分析、设计、实现过程应遵循系统开发规范。

3．课题进行期间，每周保证不少于40学时从事课题研究工作；每周至少一次到校汇报课题进度及接受指导。

4．课题结束应整理出系统相应文档。

**参考文献：**

[1] Babak Falsafi and Thomas F. Wenisch, “A Primer on Hardware Prefetching,” in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2014.

[2] Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao, “Customizable Computing,” in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2015.

完成期限：2017年06月10日

指导教师签章：

专业负责人签章：

2017年 02 月 01 日

**独 创 性 声 明**

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签名： 日期：

**关于论文使用授权的说明**

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签名： 导师签名： 日期：

**摘要**

针对数据预取的优化旨在减少存储访问延迟产生的时间消耗。通过把即将被处理器访问的数据提前从主存移动到cache，预取可以有效降低存储访问的延迟。现代处理器配有多个硬件预取器，每个预取器针对特定的存储层次，并且使用各自独立的预取算法。但是，为了使不同程序的运行性能达到最大，需要采用不同的预取器子集。启用所有预取器很难产生最佳的性能结果，并且在某种情况下，预取甚至会降低性能。

在本篇文章中，我们讨论了单线程代码的预取效果，并展示了一种利用机器学习预测出给定程序的最佳预取器组合的实现方法。我们的实验使用gem5模拟器对x86结构中L2的四个预取器进行测试。通过对程序进行特征化并结合决策树算法来获得简明可表达的特征集。本实验结果能使预取达到一定的加速比。

关键词 数据预取 机器学习 多预取器控制 特征提取与分类

**Abstract**

Optimizations for data prefetching are designed to reduce the time consumed by storage access delays. By pre-fetching the data to be accessed by the processor from main memory to cache, prefetching can effectively reduce the latency of memory access. Modern processors are equipped with multiple hardware prefetchers, each prefetcher for a specific storage hierarchy, and use separate prefetching algorithms. However, in order to maximize the performance of different programs, different prefetcher subsets need to be used. Enabling all prefetchers is difficult to produce the best performance results, and in some cases, prefetching can even degrade performance.

In this article, we discuss the prefetching effects of single-threaded code, and show a way to use machine learning to predict the best prefetcher combination for a given program. Our experiment used the gem5 simulator to test the four prefetchers of L2 in the x86 architecture. Through the characterization of the program and combined with the decision tree algorithm to obtain a concise and expressive feature set. The results of this experiment enable pre-fetching to achieve a certain speedup.

**Keywords：**Data Prefetch, Machine Learning, Multiple Prefetcher Control, Feature Extraction and Classification

**目录**

[**摘要** I](#_Toc514247921)

[**Abstract** II](#_Toc514247922)

[1. 绪论 1](#_Toc514247923)

[1.1 课题背景及意义 1](#_Toc514247924)

[1.2 CPU高速缓存 1](#_Toc514247925)

[1.2.1 结构和参数 1](#_Toc514247926)

[1.2.2 性能指标 2](#_Toc514247927)

[1.2.3 替换算法 2](#_Toc514247928)

[1.3 Cache数据预取 2](#_Toc514247929)

[1.3.1 背景及原理 2](#_Toc514247930)

[1.3.2 主要分类 3](#_Toc514247931)

[1.3.3 优化目标 4](#_Toc514247932)

[1.4 深度学习 5](#_Toc514247933)

[1.4.1 基本原理 5](#_Toc514247934)

[1.4.2 深度神经网络 5](#_Toc514247935)

[1.5 FPGA 6](#_Toc514247936)

[1.5.1 FPGA简述 6](#_Toc514247937)

[1.5.2 主要厂商 6](#_Toc514247938)

[1.6 硬件描述语言 7](#_Toc514247939)

[1.6.1 Verilog 7](#_Toc514247940)

[1.6.2 SystemVerilog 7](#_Toc514247941)

[1.6.3 Chisel3 8](#_Toc514247942)

[2. 系统分析 8](#_Toc514247943)

[2.1 需求分析 8](#_Toc514247944)

[2.2 系统模块设计 8](#_Toc514247945)

[2.3 开发环境 9](#_Toc514247946)

[2.4 测试程序 9](#_Toc514247947)

[3. 系统详细设计 9](#_Toc514247948)

[**4.** **实验步骤** 10](#_Toc514247949)

[**5.** **实验结果及分析** 10](#_Toc514247950)

[**6.** **结论** 10](#_Toc514247951)

[**致谢** 10](#_Toc514247952)

[**参考文献** 27](#_Toc514247953)

# 绪论

课题背景及意义

为了降低处理器和内存之间因访问速度差距过大造成的延迟，在计算机系统中加入CPU高速缓存（CPU cache，下文简称cache），使处理器访问数据的速度接近处理器本身的频率。同时，现代处理器还会配有多个硬件预取器，每个预取器针对特定的存储层次，并且使用各自独立的预取算法。预取通过监视并推断流访问模式，将数据超前预取到更高层次的缓存中来降低内存延迟。预取在现在的体系结构中是一种关键的技术转型，决定优化预取的参数存在多个挑战。第一，预取器必须精确地预测存取模式。如果预测错误，就会增加存储访问负担，并且更重要的是，会在容量小且昂贵的缓存中造成冲突。第二，预取指令必须及时。如果预取造成数据早于需要之前被放置到更高层缓存中，可能会被那些更紧迫需要的数据覆盖掉。这些挑战在多线程程序中被更进一步地放大。L2等更低层次的缓存可以被多个线程共享，每个线程可能需要不同位置的数据，准确地决定出读取顺序是一件困难的事情。

在本文中，我们设计并实现了一种基于深度学习技术的有效的预取策略。

CPU高速缓存

### 结构和参数

在计算机系统中主要采用组相联结构。组相联缓存把缓存空间分为多个组，每组包含若干缓存块。通过建立内存数据和组索引的对应关系，一个内存块可以被载入到对应组内的任意缓存块上。本文中所使用的组相联缓存均表述为

其中，C为缓存容量，B为每个数据块的大小，N为相联度（每组中有N个数据块），为组数。当使用组相联时，在通过索引定位到对应组之后，必须进一步地与所有缓存块的标签值进行匹配，以确定查找是否命中。

### 性能指标

本文中对cache的主要性能评价指标有加速比和cache的命中率。加速比一般表示为

其中，为cache的命中率，为cache的访问周期，为主存储器的访问周期。可推断出，当→1时，→。研究表明，的大小受到cache的预取算法影响，本文即着重于通过深度学习优化预取算法来提高cache性能。

### 替换算法

对于组相联缓存，当一个组的全部缓存块都被占满后，如果再次发生缓存失效，就必须选择一个缓存块来替换掉。存在多种算法决定哪个块被替换。

最简单的替换算法是随机法（Rand法），即随机决定被替换的缓存块。而先进先出（FIFO）法替换掉进入组内时间最长的缓存块。这种方法虽然考虑了程序运行的历史状况，但无法正确地反映程序的局部性。最近最少使用法（LRU算法）则跟踪各个缓存块的使用状况，并根据统计比较出哪个块已经最长时间未被访问。这种方法反映程序局部性规律，因为最近最少使用的块，很可能在将来的近期也很少使用，因此LRU算法的命中率比较高。但是这种方法比较复杂，硬件实现比较困难，对于2路以上相联，这个算法的时间代价会非常高。

本文使用与LRU法技术思想相同的最久没有使用法（LFU），其实现方法为记录近期使用次数的多少，然后替换最少的那一个。

Cache数据预取

### 背景及原理

尽管cache层级技术的应用有效地减少了那些最常用数据的访问延迟，在科学计算程序中花费超过一半的时间用于内存请求仍不少见。大型且密集的矩阵操作是许多科学计算程序的基础，而这些操作往往使得cache的利用效率低下。处理器在发现cache缺失后必须等待cache访问内存获取数据，然后继续进行运算。这种数据获取策略使每一个首次访问的数据块都会成为一次缓存缺失（即强制失效）。如果被访问的数据是一个大型数组操作的一小部分，它很有可能在之后被替换出cache，为数组后续的数据成员进入cache腾出空间。当同样的数据块再次被需要时，处理器必须重新将其从内存中提取出来，产生更高的访问延迟（即容量失效）。

因此，如果能在处理器还未用到某个数据块之前就提前将其放入cache中，便能进一步提高cache的命中率。这种操作与处理器运算同时进行，使得数据在处理器需要时刚好到达了cache中。这样既利用了空间局部性，又能覆盖传输延迟，这种技术即为cache的预取（Prefetch）。本文将讨论如何通过优化预取算法达到提高程序运行效率的目的。

### 预取技术类型

现代计算机使用的预取主要分为两类，一是软件预取，二是硬件预取。软件预取多由编译器进行。指令集会提供预取指令供编译器优化时使用。编译器则负责分析代码，并把预取指令适当地插入其中。这类指令直接把目标预取数据载入缓存。本文使用硬件预取进行优化，因此将着重讨论硬件预取技术的特点和可优化空间。

硬件预取在cache旁添加支持元件，可以实时动态地进行预取，并且不需要编译器介入。典型的硬件指令预取会在缓存因失效从内存载入一个块的同时，把该块之后紧邻的一个块也传输过来。第二个块不会直接进入缓存，而是被排入指令流缓冲器（Instruction Stream Buffer）中。之后，当第二个内存访问指令到来时，会并行尝试从缓存和流缓冲器中读取。如果该数据恰好在流缓冲器中，则取消缓存访问指令，并将返回流缓冲器中的数据。同时，发起起一次新的预取。如果数据并不在流缓冲器中，则需要将缓冲器清空。

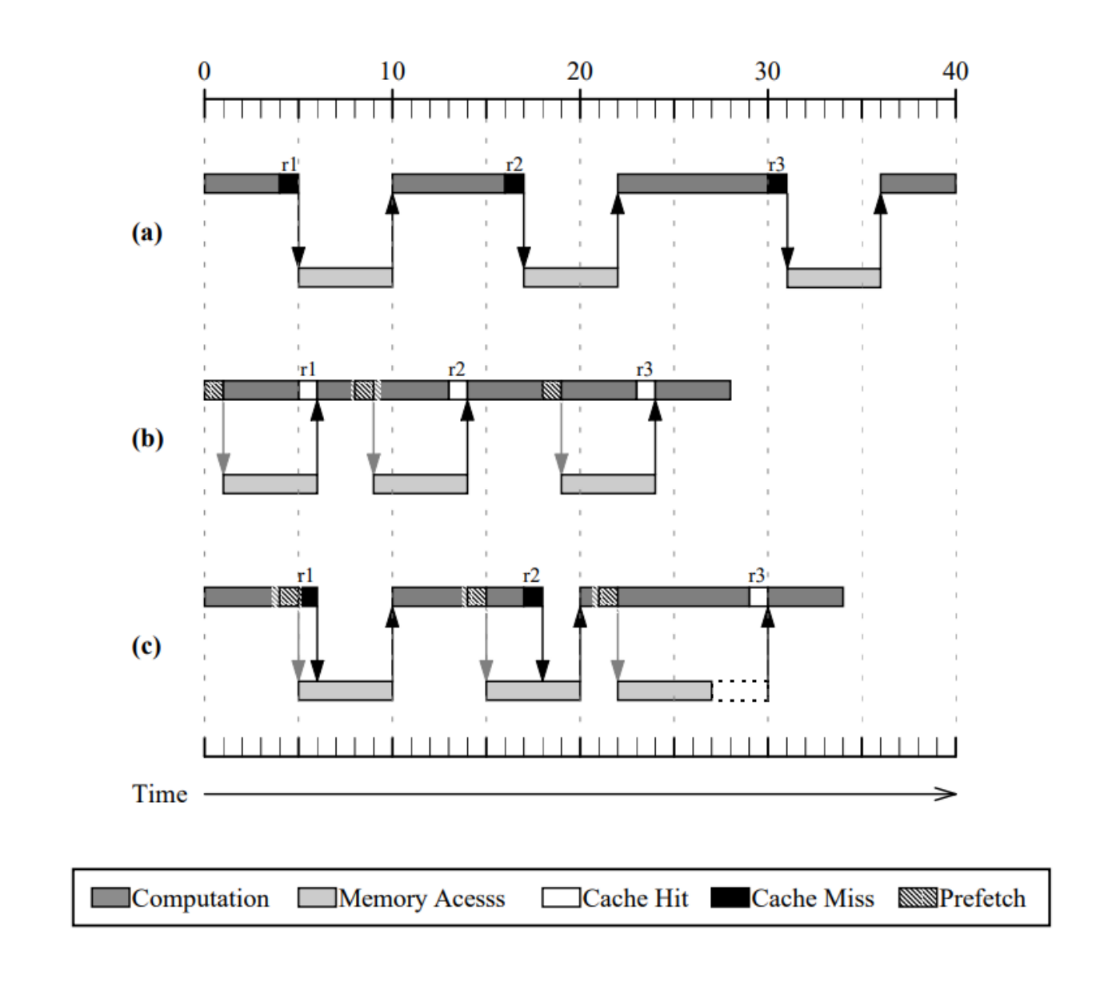


图 1程序运行消耗的时间(a)不进行预取(b)完美预取(c)退化的预取

在数据预取方面，图1展示了不同情况下典型的程序运行情况。当不使用预取时（图1(a)），处理器在访问r1, r2, r3时发生了强制失效，并需要停止运算以等到相应的cache将数据从内存中获取之后才能继续执行运算。为了解决这个问题，一种越来越普遍的数据预取启动方式是由处理器明确发出fetch指令。一个fetch指令会最低限度指定被放入cache的数据块的地址。当指令执行时，此地址会直接传递给内存系统，而不会让处理器停滞并等待应答。Cache以类似应答load的方式回应fetch，但并不会在数据块进入cache后传入处理器。图1(b)展示了这种方法如何通过并行执行访存和处理器运算从而隐藏内了存访问延迟。这种理想情况下，数据刚好在其被处理器需要之前被预取到cache中。图1(c)则展示了一种不乐观的情况，在本图中对r1和r2的预取执行过晚，使处理器仍需要停止运算以等待数据到来，但预取操作的确减少了处理器等待的时间。而r3的预取则执行过早，这块数据将暴露在cache的替换候选之中，如果r3在被处理器引用前被替换掉，则需要重新进行访存操作。

其他几种硬件预取技术不需要使用fetch指令，这些技术使用特殊硬件对处理器进行监测来做出预取选择。尽管硬件预取不会造成指令上的额外负担，但它们往往都产生比软件预取更多的无效预取，从而产生更多缓存污染消耗内存带宽。

### 优化目标

在使用预取技术时，必须妥善考虑进行时机和实施强度，并且仅产生少量的负担。如果过早地进行预取，则有可能在预取数据被用到之前就已经因为冲突置换被清除。如果预取得太多或太频繁，则预取数据有可能将那些更加确实地会被用到的数据取代出cache。

cache预取技术的优劣可以由三个指标评价，一是覆盖率，表示因预取所减少的缺失数占总cache缺失数的比例，可用公式表示为

其中，Cov为覆盖率，为因预取所减少的缺失数，为总cache缺失数。

二是准确率，表示有效预取所占比例，可用公式表示为

其中，Acc为准确率，为无效的预取。

三是及时性，及时性的定义为块预取的时间相对于块被使用的时间提早了多少。

深度学习

### 基本原理

深度学习是机器学习中一种基于对数据进行表征学习的算法，其基础是机器学习中的分散表示（distributed representation）。分散表示意为假定观测值是由不同因子相互作用而生成。在此基础上，深度学习进一步假定这一相互作用的过程可分为多个层次，代表对观测值的多层抽象。不同的层数和层次的规模可用于不同程度的抽象。

在深度学习方法中，更高层次的概念从低层次的概念学习得到。这一分层结构常常使用贪婪算法逐层构建而成，并从中选取有助于机器学习的更有效的特征。深度学习多使用非监督式，或半监督式的特征学习和分层特征提取高效算法来替代手工获取特征，成为其优于其他算法的一大特点。

### 深度神经网络

本文使用的深度神经网络（deep neural network，DNN）是一种在输入与输出之间有多个隐藏层次的人工神经网络（artificial neural network，ADD）。深度神经网络能够为复杂的非线性关系提供建模，当对象表达为多层次基本数据类型时DNN构架生成复合模型，多出的层次可以从更低层次组合特征，因此与相似的浅层网络相比可以使用更少的单元达成对复杂数据的建模。

深度结构在几种基础方法上有许多变种，每种结构在不同特定领域都有着显著的成效。不同的结构之间很难直接比较性能优劣，除非使用相同的数据集进行评估。深度神经网络一般是前馈网络，数据流从输入层传向输出层而不进行传回。主要的类型有递归神经网络（Recurrent neural network，RNN），数据可以流向任何方向，常使用于语言建模中。卷积神经网络（Convolutional Neural Network，CNN）用于计算机图像处理，目前也应用于声学模型以进行自动语音识别。

FPGA

### FPGA简述

FPGA为Field Programmable Gate Array的缩写，即现场可编程逻辑阵列。是在原有的可编程逻辑器件的基础上发展而来的。以硬件描述语言描述的逻辑电路，可以利用逻辑综合和布局、布线工具软件，快速烧录到FPGA上进行测试，这一过程是现代集成电路设计验证的技术主流。它是作为专用集成电路（ASIC）领域中的一种半定制电路而出现的，可以实现任何ASIC上的逻辑功能，并且一次性工程费用很低（但元件费用更高）。尽管FPGA的速度要慢，无法完成更复杂的设计并且耗电量更大，但其具有高度的灵活性，内部逻辑可以被反复修改从而大幅降低了除错成本，既解决了全定制电路的不足，又克服了原有可编程逻辑器件门电路数有限的缺点。

### 主要厂商

目前世界上的两大厂商Altera和Xilinx占有将近90%的市场，它们均成立于上个世纪80年代。Altera在1984年推出了业界第一款可重复编程逻辑硬件EP300。Xilinx的联合创始Ross Freeman和Bernard Vonderschmitt在1985年发明了首个可商业化FPGA——XC2064。目前Xilinx和Altera都提供windows和Linux平台的设计软件（ISE/Vivado和Quartus），设计者可以利用这些软件设计、分析、仿真和综合（编译）他们的应用。

硬件描述语言

本文在使用FPGA作为实验平台的基础上，选择硬件描述语言进行编程。在对目前多种语言进行比较后决定使用由加州大学伯克利分校开发的开源语言Chisel3，并在下文阐述原因及其特点。

### Verilog

Verilog是电气电子工程师学会（IEEE）的1364号标准，它主要用于设计和验证抽象化的寄存器传输级的数字电路，也用于模拟电路和混合信号电路，以及生物合成电路。Verilog的基本语法和C语言相近，因此对熟悉C语言的设计人员来说可以很快掌握。

使用Verilog进行程序设计的基本思路是将复杂的电路划分为多个模块（module），模块作为提供简单功能的基本结构。工程师可采用自顶向下的思路进行模块分层、划分。

### SystemVerilog

SystemVerilog是一种由Verilog发展而来的硬件描述、硬件验证统一语言，前一部分基本上是2005年版Verilog的扩展，而后一部分功能验证特性则是一门面向对象程序设计语言。面向对象特性很好地弥补了传统Verilog在芯片验证领域的缺陷，改善了代码可重用性，同时可以让验证工程师在比寄存器传输级更高的抽象级别，以事务而非单个信号作为监测对象，这些都大大提高了验证平台搭建的效率。

相较于Verilog，SystemVerilog定义了两种数据生存周期：静态和自动，添加了几种新的数据类型，添加了三种新的程序块类型，新增interface类型改善了Verilog原有port类型在多层次电路中大型模块间连接过于复杂时难以管理的问题。在硬件验证方面，SystemVerilog拥有的多种功能一般用于协助创建扩展、灵活的test bench而非综合。它包含一些新的数据类型、支持面向对象程序模型等等。

### Chisel3

Chisel是Constructing Hardware In a Scala Embedded Language的简称，它是嵌入在高级编程语言Scala中的硬件构建语言。换言之，Chisel是遵循Scala使用规则的一系列特殊类定义、以及预定义的对象，设计者相当于使用Scala语言进行硬件图构建。Chisel的目前已经更新到了第3个版本号，其主要具备的特征有：

* 抽象数据类型和接口
* 面向对象编程和函数构建
* 使用高度参数化的元编程（metaprogramming）
* 自动生成可以在标准ASIC或FPGA上使用的Verilog程序

例如，设计一个多路比较器，输入n个宽度为w的无符号整型数据，输出结果为输入中最大的输入数据，则可编写Chisel程序为

|  |
| --- |
| MaxN |

编译后在指定文件夹中输出的自动生成Verilog代码

|  |
| --- |
|  |

*注：绪论是综合评述前人工作，说明论文工作的选题目的和意义，国内外文献综述，以及论文所要研究的内容。*

# 系统分析

需求分析

在前文中已经探讨过，缓存预取有可能导致性能降低，并造成许多不良后果。这说明预取仍存在很多优化空间，有许多研究已经在相关方面做出了进展。Cavazos et al【参考文献】开发了一种机器学习模式，能够找出SPEC CPU2006标准程序组的最佳优化配置，使程序得到性能提升。这证明了使用人工智能进行性能优化的可行性。但他们的工作目的是找出一组最优化编译，而本文将更注重于硬件优化。McCurdy et al【参考文献】使用性能事件描述了预取对系统应用程序的影响。他们试验了AMD处理器上的多种标准程序的组合。Liao et al【参考文献】展示了一种基于机器学习的方法，用于选择优化预取配置。这两项研究都为基于人工智能的硬件预取优化提供了思路。

受此启发，本文使用了深度神经网络的方法进行优化（具体补充）。本实验将构建一个精简的处理器-缓存-内存系统并展开实验。实验平台选择FPGA以利用其可以随时修改，重新烧录的高度灵活性特征。本实验中使用的Zynq-7000还集成有双核AMD Cortex-A9处理器，可以在进行FPGA实验的同时直接输出处理结果。由于本系统采用多层次多模块设计，使用传统Verilog语言可能产生大量重复性工作，并因大量数据定义使得查错工作难以进行，这在上文【文献链接】中已经讨论过。因此实验决定采用高级程序设计语言Scala的硬件构建工具Chisel3进行编程，这样既减轻了设计难度，也提升了程序可读性。

系统模块设计

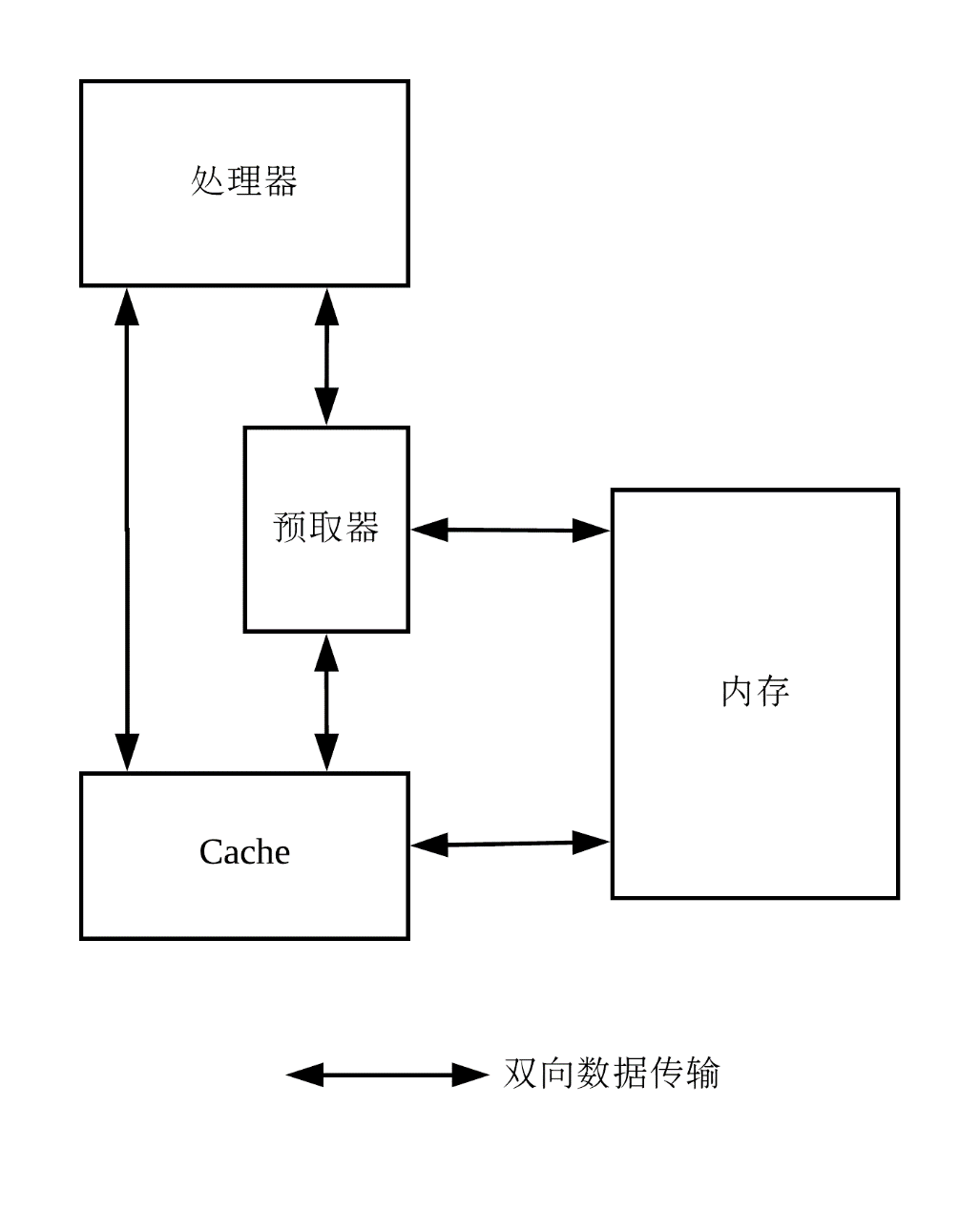


图 2系统模块设计

本实验的目的旨在通过优化预取器的预取行为达到提升程序性能，缩短运行时间的目的，可以看做提升cache的各项性能指标，因此系统设计集中于处理器-缓存-内存部分，不考虑其他因素和元件的影响。如图1所示，本系统主要分为五个大模块：处理器模块、cache模块、预取器模块和内存模块。

开发环境

本实验开发时使用的操作系统为8GB内存64位Ubuntu16.04，软件开发工具为IntelliJ IDEA，（补充sbt、Scala、chisel版本）FPGA综合及布线工具Vivado。

测试程序

# 系统详细设计

模块设计

接口设计

**FPGA综合实验过程**

**实验结果及分析**

**结论**

*注：结论（或结束语）作为单独一章排列，但标题前不加“第XXX章”字样。结论是整个论文的总结，应以简练的文字说明论文所做的工作，一般不超过两页。*

**致谢**

*注：对导师和给予指导或协助完成毕业设计（论文）工作的组织和个人表示感谢。文字要简捷、实事求是，切忌浮夸和庸俗之词。*

预取监视器和外推流获取应用程序的模式然后将数据超前预取到更高层次的缓存中来降低内存延迟。由于CPU和内存之间的速度差，预取在代码优化中一直是一个重要的技术。在现在的多核结构中预取变得更加具有重要意义。随着被附加在同一个存储系统上的核心数量的增加，通常来自同一程序的不同线程中的预取候补流数量也会增加。有效的多预取流协调不仅能降低存储延迟，还能直接影响并行效率。

虽然预取在现在的体系结构中是一种关键的转型，决定优化预取参数造成了多个挑战。第一，预取器必须精确地预测存取模式。如果预测错误，就会增加存储访问负担，并且更重要的是，会在空间很小且贵重的缓存中造成冲突。第二，预取指令必须及时。如果预取造成数据早于需要之前被放置到更高层缓存中，可能会被那些更紧迫需要的数据覆盖掉。这些挑战在多线程程序中被更进一步地放大。L2等更低层次的内存可以被多个线程共享，每个线程可能需要不同位置的数据，准确地决定出读取顺序是一件困难的事情。

在这篇论文中，我们会处理这些挑战，并制定出了一种基于机器学习技术的有效的预取策略。我们探讨在PARSEC程序和两个理想化的顺序和随机存取模式程序上的预取效果。我们在授权给独立硬件预取器，然后将它组合到一个内置4个预取器的英特尔处理器上，以检测性能。为了得到所给程序需要使用的预取器的建议，我们尝试了数个机器学习算法。

一个成功使用基于机器学习的方法的关键步骤是将程序以定量的方式特性化，以获取程序的基本区别。我们使用硬件性能事件来达到这个目的。由于有数百种可用的时间，我们设计了一种简单但是有效的程序来减少特性化一个程序需要的事件的数量。我们通过测试决策树上的不同特性集来展示剪枝算法的效果。最后，我们开发了一个为未知程序推荐硬件预取配置的框架。这个框架可以使用户获得硬件预取器提供的平均为96%的加速比。

使用这种推荐系统调整硬件配置有几个优点。第一，可以使我们更有效地利用已有硬件。第二，无需更改被优化程序的源代码。即可直接适用于可执行程序。最后，框架依赖于开源技术，因此易于扩展和移植到其他体系结构。

1. 相关工作

预取是被广泛开发过的领域，预取的益处被广泛地记录和研究过。Lee et al.是第一批提出数据预取概念的研究者之一，通过硬件的数据预取来减少存储访问延迟。Chen et al.在他的早期工作中，研究了硬件和软件数据预取的效果。之后人们在数据预取上进行了许多研究工作。

随着多核处理器的提出，预取的效果变得更加有趣。预取会降低性能，这种不良后果在[4][5]中已经探讨过。Puzak et al论证了预取在何处降低了性能并指出了一种基于时间线的特性化预取，这种方法覆盖范围广且准确度高。Lee et al引导了一种深入的关于何时以及为何预取工作的调查。他们对软件和硬件预取在SPEC CPU2006标准程序上的串行工作负载性能进行了广泛地分析。

Jayasena et al论证了一种决策树的效果，减少了检测false sharing使用的性能事件数量。他们的方法激发了我们将决策树纳入我们的工作之中。Cavazos et al开发了一种机器学习模式，能够找到SPEC CPU2006标准程序组的最佳优化配置。他们工作的目标是找出一组最优化编译，使程序得到性能提升。他们报告了使用性能事件作为学习算法特征时的高分类精准度。Milepost GCC是一个针对更进一步的多源优化学习的大项目。Milepost GCC使用了静态特征来描述程序的特征。Demme et al做了类似的工作使用基于程序数据和控制流的图表聚类，来描述程序的行为。然而，由于他们修改了编译器或程序代码的中间表示，这种描述很难移植到其他平台。

在我们讨论的这些工作中，以下两个与我们的工作最为接近。McCurdy et al使用性能事件描述了预取对系统应用程序的影响。他们试验了AMD处理器上的多种标准程序的组合。他们的工作主要集中在串行工作负载，但他们也讨论了同时运行串行程序的效果。他们的工作取决于成功地使性能事件孤立，这足以表现预取效果的获取。他们精选了试验中AMD体系结构的性能事件，并会在任何新体系结构上重复研究所有可用性能事件的步骤。Liao et al展示了一种基于机器学习的方法，用于选择优化预取配置，这种方法与我们的方法相似。然而，他们的工作也取决于制定了特定的体系结构性能事件，并且是手动制作的。另外，他们的工作集中在串行工作负载，而我们对并行程序更感兴趣。

1. 机器学习框架

图片1展示了我们机器学习框架要执行的主要任务。共有两个关阶段：训练和测试。在训练阶段，运行样本程序用来生成特征。此外，程序在所有可能的预取器设置下运行，以确定性能并确定优化硬件配置。这个信息用来训练学习模块。测试阶段很简单。第一步，将未使用过的程序运行一次以收集其特征。第二步，将这些特征送入训练模块，得出推荐的预取配置。

1. 预取器配置

许多现代处理器使用硬件预取器。例如我们使用英特尔酷睿2处理器，它带有如下四个预取器：

1. Data cache unit (DCU) prefetcher：此预取器会识别流算法并预取下一行到L1 data cache。
2. Instruction pointer (IP) based stride prefetcher：此预取器只会跟踪load指令，并检测跨步访问。它可以检测最高2kB的步长并且预取到L1 cache。
3. Spatial prefetcher (CL)：此预取器通过读取成对的数据行将一个cache line放入L2，以组成一个128Byte对齐的区块。
4. Stream prefetcher (HW)：此预取器检测L1 chache的流请求并将预期的chache lines放入L2和LLC。

需要注意的是最近更多的英特尔架构，例如SandyBridge 和 Haswell，都包含了同样的预取器。有趣的是，缺省条件下这四个预取器并没有被定义开启或关闭。在本工作中，我们定义了一种配置，用四位掩码的形式表示预取器的开启/关闭状态。掩码的每一位代表一个预取器。1值意味着对应的预取器开启，0值意味着关闭。掩码代表的预取器按照他们的重要程度的降序排列：HW，CL，DCU，和IP。我们使用开源工具Likwid控制处理器的预取器配置。

如果训练所有的预取器，例如配置1111，似乎能得到最佳的性能结果。然而，其他配置能够表现的更加优秀。事实上，图片2中说明了在某种情况下，配置1111反而降低性能。报告中的加速比值都是从我们的基准配置0000实时改进的。对每个配置，我们运行三次然后使用最佳运行时从操作系统的抖动中保护结果。因为运行产生的最佳性能不仅有最少的终端，还能在最短时间内运行完成了任务，我们在努力提高结果以接近最佳值。

串行的观察可以通过图片2完成。第一，在同一程序中，改变预取器的配置对性能有着重要的影响。例如，freqmine和streamcluster在调整配置后性能上有着很大差异。第二，配置1111并不总是最佳选项，可参考random, stream, streamcluster, 和 vips的情况。第三，配置0000可能成为最佳配置，也就是说使用预取器反而会降低性能，参考canneal, fluidanimate以及串行配置的random的情况。最后，在串行情况下，有多种“好”的配置，也就是说串行配置的表现与最佳表现相差无几。

1. 特征提取

我们测量所有受我们使用的处理器支持的性能事件，并使用它们作为描述程序的初始特征集。计数事件例如L2 cache丢失率和停滞CPU周期证明一个程序行为的可数灵视，并且被抢先使用在描述程序中。然而，使用所有的可用事件是有问题的，因为这要消耗很长的测量时间，而且事件会交叉使用处理器。我们使用下面描述的剪枝算法来解决这些问题。

特征一旦生成，会正规划为事件数/每百万执行指令。这样的缩放有必要用来比较不同程序和同一程序的不同运行情况。例如，对一个执行十亿条指令的程序来说一百个页错误也许并不严重，但对只执行一万条指令的程序来说非常严重。对于执行一万条指令的程序来说，正规化后的数值会比十亿条指令的程序得到的值更高，因此能够正确地获取严重程度的差别。

图片3展示了我们使用的14个程序的所有事件的计数。X轴代表事件，轴被移除意味着这张图只是用于形象化表示。两个关键观察点可以从图中找到。第一，许多事件有相近的值，因此他们对程序行为并没有造成差别的信息。第二，许多事件有着很高的值，并使其他有更小域值范围的事件消失。我们通过特征缩放来解决这个问题。

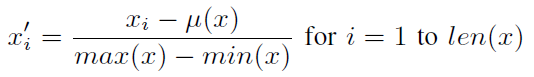
1. 选择特征

我们使用算法1来减少特征集。算法1由两个子程序组成：FindEvents 和 EventsUnion。FindEvents获取两个程序的事件计数作为输入。对于每个特定的事件，此子程序检查两个数值是否至少相差Φ，Φ是以范围在（0.0,1.0）的数字表示的相对差。如果是，事件包含在集合中。在子程序的最后，集合包含了所有充分获取两个程序区别性的事件。Diff函数返回它的参数除以他们的最大值后的绝对差值。我们试验发现Φ=0.95非常合适，并且只使用这一阈值。

EventsUnion子程序将一列程序作为输入，然后对所有可能的程序对调用一次FindEvents子程序。它包含了多个返回集，最终包含了所有程序对的表现的事件。接着，EventsUnion子程序使用一个map计算每个事件在这些返回集中出现的次数的数值。最后，map中存储了所有数值。

这个算法的思想是，如果一个事件在两个程序中有着很高的特定数值，它获取了两个程序差别的重要形势，因此我们要考虑选择这个事件。然而，很多事件对任何两个程序来说都能明显显出差别，我们最后会得到大量的事件，因此我们使用map给事件排优先级，map中表明了事件在区别两个程序时起到作用的频数。排序后的map我们可以简单地选取最重要的事件来建立我们的最终特征集。

学习阶段前的最后一步，我们称为特征缩放。学习模块用于当特征的数值范围有很大分歧之时。例如，如果一个特征的范围是1~10而同一个特征集里的另一个特征的范围是-10，000~+10,000，学习模块可能画出精确的决定边界。作为补救，我们将所有特征通过下列公式缩放为向量x：



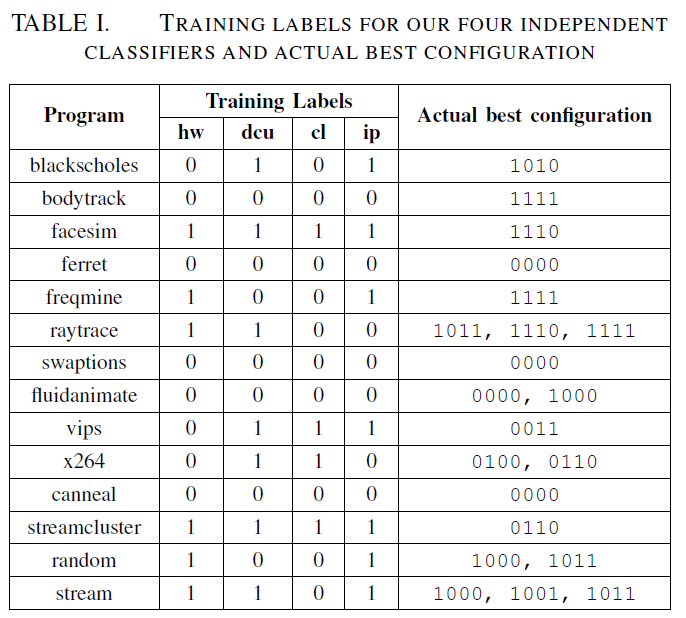
μ(x)是特征向量x中值得平均值，进行缩放后，所有特征的范围会在-1~+1之间。

图片4展示了特征缩放的效果。绘图与图片3有着明显的不同。并且，程序行为的差别也更加明显。例如，缩放前blackscholes和bodytrack的图表比缩放后的图表更难辨别。canneal缩放后大体上表现得和blackscholes、bodytrack有了区别。这是因为许多事件数量在canneal中并非0，与图片3中的表现相反。类似地，swaptions, vips, 和 x264在没有缩放时看起来十分相似，但在特征进行缩放后已经有明显的差别。对其他程序也是同样的结果。未缩放的特征的图表稀疏，相反缩放后不同程序的特征图表大体上差别更大，而且密集。

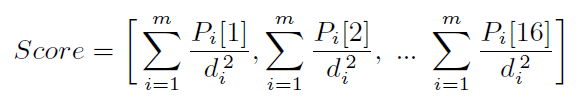
1. 学习模块

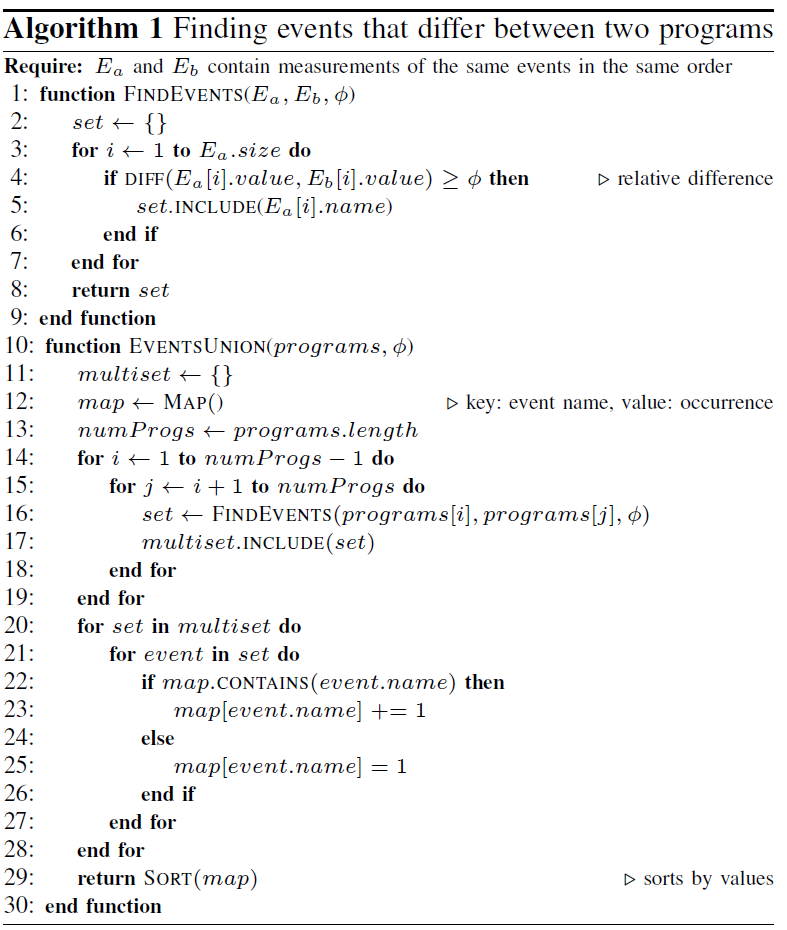
我们的框架的最后一部分是学习模块。我们评估两个不同的传统学习模块，逻辑回归和决策树，并设计了一种基于欧几里得几何的分类器，剪裁到我们需要的并且获取有用的信息。

1. 学习标签：为了达成两种目的，学习目标将以两种方法公式化。在第一个公式中，我们使用一个二进制分类器来制定程序是否从预取中获益。如果最佳预取配置相对配置0000的加速比大于等于10%，则认为程序从预取中获益。根据这个矩阵，程序ferret, swaptions, fluidanimate, 和 canneal并不获益。我们用这个公式决定要使用哪些算法1中得出的顶层事件。第二个公式单独检查每个预取器的效果。如果为给定程序开启一个特定预取器，使得其相对于配置0000的加速比大于等于2%，我们将这个预取器归类为对此程序有用。这里执行四个学习模块的实例。它同时列出了每个程序的最佳配置。需要注意学习模块对每个类型标签，都应该被多个样本训练。因此要使用四个实例而非一个包含所有预取器组合的实例，等等。16个特定类别标签，需要比我们的14个程序更大量的训练程序。



1. 基于欧几里得距离的模块：我们使用欧几里得距离作为预测未知程序的好预取器配置时的相似矩阵。这个方法背后的动机是四个独立的分类器不会获取预取器之间的相互作用。将分类器与表1中的最佳配置比较说明了这个问题。分类器的组合通常不是最大加速比的配置。例如，程序raytrace的个别推荐预取器是hw和dcu，对应的配置是1100。然而，这两个预取器的组合并不表现出最佳性能。为了抵消这个问题，我们的欧几里得模块从训练集中计算未知程序和所有已知程序的距离。然后用这个举例作为权重，计算出每个16种可能的配置的得分。具体来说，模块计算下列16元向量：





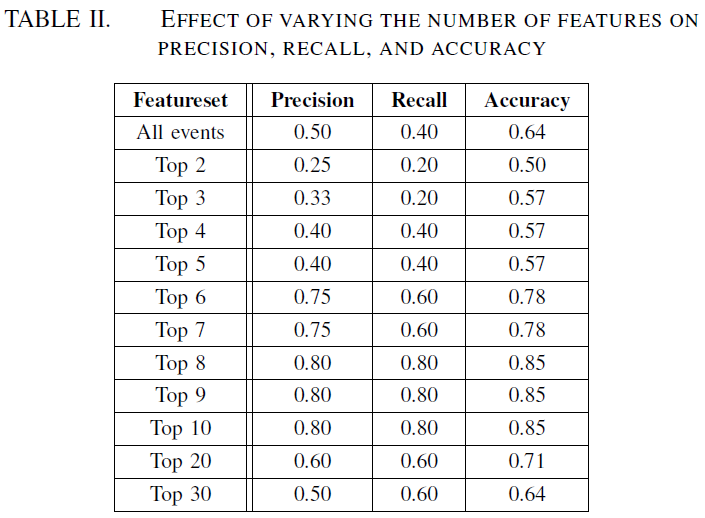
m是训练集中程序的数量。Pi是一个与程序i关联的16元向量。此向量中的每一元对应一个配置，元的脚标的值对应相同二进制值得配置。元的值代表了在程序i上使用对应配置后与可达的加速比的分数。例如，P2[3]代表第二个训练程序使用配置0011获得的加速比分数，3(10)=0011(2)。最后，di指示了未知程序和第i个训练程序的欧几里得距离。

结果向量中包含每个预取器配置的权重分数。推荐配置是归属于最大元的索引的二进制表示。这个方法结合了每个预取器配置对每个训练程序的效果，并使权重得分使用距离的平方。因此，相似程序的预取性能比不同程序带有更大的权重。此外，如果未知程序和数个程序相近，我们的方法可避免最相似的程序成为唯一的推荐基础。

4．结果与分析

A．实验环境

我们在2.4GHz英特尔酷睿2 Quad Q6600处理器上执行测量，处理器带有8个32KB L1 caches和两个4MB L2 caches。所有程序使用GCC 4.8.2下-O2优化级别编译，操作系统为Ubuntu 14.04。PARSEC程序由parsecmgmt脚本调用，装入程序集中并用八个线程的native输入。我们在不同的线程下重复试验，但结果并没有显示出预取器的效率有明显的改变。



B．评估方法

我们使用(k-1)交叉验证评价我们的学习模块。起初，我们集中在预测准确度。然而并没有获得推荐的质量，因为多预取器配置会产生近似最佳性能。例如，streamcluster的最佳配置是0110有着1.37的加速比，但0111的结果是1.34。因此我们发现哪个可达加速比的分数可以通过学习模块的推荐获得时更加有用。

1. 选择特征最佳数量

为了测试算法1中选择特征程序的功效，我们用最高的两个事件训练并测试了决策树，然后和数量增加的事件一起送至模块，以此来确定保和点。参考表2此点在我们使用最高的八个事件时达到最大。大于八个事件会降低学习性能。这个剪枝算法的示例可以指定真正获取程序区别的事件。此后，我们使用最高的八个事件作为特征集，唯一的例外是欧几里得模块，我们发现只在最高六个事件下效果最佳。

1. 推荐好的预取配置

图5展示了三种学习模型的性能下，推荐配置和最佳配置加速比的接近程度。

逻辑回归模型达到了平均92.4%的准确度。这种模型在freqmine上的表现很差，原因是训练标签的派生方式。分别接受cl和dcu预取器对此程序的没有获得加速比，既模型被训练关闭它们的原因。然而，对freqmine程序的好的配置中这两个预取器都启动了。因此，这是两个预取器在单独情况下不提升加速比，但组合的加速比要比他们单独的和更高的一种情况。

决策树模型比逻辑回归模型有着更好的表现，达到了平均95.3%的准确度。对于freqmine程序，可以得出最佳配置的结果。然而，由于预取器之间的相互作用没有被加入到独立模型中，这种情况只会偶然发生。

欧几里得距离模型达到了平均96.1%的准确度。这个模型可以为freqmine程序推荐出好的配置，因为模型考虑到测试程序和训练程序组之间的相似性。此外，模型利用了所有训练程序中每一个可能的预取器配置的性能信息。相比之下，其他分类器基于独立的预取器影响来预测最佳配置。

欧几里得距离模型在facesim程序的表现上大致低于其他模型。因为它认为random是最为相似的程序。Random的高效配置极大地影响了模型的决定。当我们检查模型内部，我们发现第二顺位的推荐是0110，达到了98%的可达加速比。很明显。相似矩阵在这种情况下显示出了不完美性。

所有模型在对streamcluster的运行结果都很差。逻辑回归和决策树分类器由于受到上述问题的影响，虽然训练标志是1111但实际的最佳配置是0110。欧几里得距离模型结果不佳是因为streamcluster与其他任何程序相比都有明显差异。因此在训练集中没有相似的程序。此程序和stream最接近，使用stream的配置可以使加速比达到1.07，相当于只能达到最佳配置0110的加速比1.37的78%。

5．总结

本论文展示了一种帮助使用者将系统硬件预取器的效率提升至最大的框架。我们在试验中使用的处理器带有4个预取器，因此共有16种可能的预取配置。我们的框架对这16种组合进行了详细的研究，并且记录了每种配置在训练程序集中的性能。框架使用硬件性能事件来描述代码，以达到找出新输入程序和训练程序之间相似性的目的。由于现代处理器已经支持数百种性能事件，我们的框架使用了剪枝算法来构造简明的特征集。特征集中组合了多种预取器配置的性能，被用来训练三种机器学习模型。其中基于欧几里得距离的模型是我们为推出预取器配置设计的特殊模型。此模型能够在PARSEC标准程序组和两个额外程序上，推出平均96%的可达的预取加速比。

缓存的存储结构

结构上，一个直接映射（Direct Mapped）缓存由若干缓存块（Cache Block，或Cache Line）构成。每个缓存块存储具有连续内存地址的若干个存储单元。在32位计算机上这通常是一个双字（dword），即四个字节。因此，每个双字具有唯一的块内偏移量。

每个缓存块有一个索引（Index），它一般是内存地址的低端部分，但不含块内偏移和字节偏移所占的最低若干位。一个数据总量为4KB、缓存块大小为16B的直接映射缓存一共有256个缓存块，其索引范围为0到255。使用一个简单的移位函数，就可以求得任意内存地址对应的缓存块的索引。由于这是一种多对一映射，必须在存储一段数据的同时标示出这些数据在内存中的确切位置。所以每个缓存块都配有一个标签（Tag）。拼接标签值和此缓存块的索引，即可求得缓存块的内存地址。如果再加上块内偏移，就能得出任意一块数据的对应内存地址。

因此，在缓存大小不变的情况下，缓存块大小和缓存块总数成反比关系。下图中所示的缓存块来自一个数据总量为4KB、每个缓存块大小为16B的直接映射缓存，其标签长度为20bits（ {\displaystyle 32-\log \_{2}(4096\div 16)-\log \_{2}16=20} 32-\log \_{2}(4096\div 16)-\log \_{2}16=20）。

一个大小为16字节的缓存块。从属于一个数据总量为4KB的直接映射缓存。

此外，每个缓存块还可对应若干标志位，包括有效位（valid bit）、脏位（dirty bit）、使用位（use bit）等。这些位在保证正确性、排除冲突、优化性能等方面起着重要作用。

运作流程

下面简要描述一个假想的直接映射缓存的工作流程。这个缓存共有四个缓存块，每个块16字节，即4个字，因此共有64字节存储空间。使用写回（Write back）策略以保证数据一致性。

CPU缓存的运作流程（注意内存左侧给出的地址是字地址而不是字节地址）

系统启动时，缓存内没有任何数据。之后，数据逐渐被载入或换出缓存。假设在此后某一时间点，缓存和内存布局如右图所示。此时，若处理器执行数据读取指令，控制逻辑依如下流程：

(将地址由高至低划分为四个部分：标签、索引、块内偏移、字节偏移。其中块内偏移和字节偏移各占两位，后者在以下操作中不使用。)

用索引定位到相应的缓存块。

用标签尝试匹配该缓存块的对应标签值。如果存在这样的匹配，称为命中（Hit）；否则称为未命中（Miss）。

如命中，用块内偏移将已定位缓存块内的特定数据段取出，送回处理器。

如未命中，先用此块地址（标签+索引）从内存读取数据并载入到当前缓存块，再用块内偏移将位于此块内的特定数据单元取出，送回处理器。这里要注意的是，（1）读入的数据会冲掉之前的内容。为保证数据一致性，必须先将数据块内的现有内容写回内存。（2）尽管处理器请求的只是一个字，缓存仍必须在读取的时候把整个数据块都填充满。（3）缓存的读取是按缓存块大小为边界对齐的。对于大小为16字节的缓存块，任何因为0x0000、或0x0001、或0x0002、或0x0003造成的未命中，都会导致位于内存0x0000—0x0003的全部四个字被读入块中。

在右图中，如此时处理器请求的地址在0x0020到0x0023之间，或在0x0004到0x0007之间，或在0x0528到0x052B之间，或在0x05EC到0x05EF之间，均会命中。其余地址则全部未命中。

而处理器执行数据写入指令时，控制逻辑依如下流程：

用索引定位到相应的缓存块。

用标签尝试匹配该缓存块的对应标签值。其结果为命中或未命中。

如命中，用块内偏移定位此块内的目标字。然后直接改写这个字。

如未命中，依系统设计不同可有两种处理策略，分别称为按写分配（Write allocate）和不按写分配（No-write allocate）。如果是按写分配，则先如处理读未命中一样，将未命中数据读入缓存，然后再将数据写到被读入的字单元。如果是不按写分配，则直接将数据写回内存。

*关于插图：*

*插图要精选。图序可以连续编序（如 图52），也可以逐章单独编序（如 图6.8），编序方式应与表格、公式的编序方式统一，图序必须连续，不得重复或跳跃。仅有一图时，在图题前加‘附图’字样。*

*由若干个分图组成的插图，分图用a,b,c,……标出。*

*图序和图题置于图下方中间位置。*

*关于引用文献：*

*正文中引用文献的标示应置于所引内容最后一个字的右上角。当提及的参考文献为文中直接说明时，则用小4号字与正文排齐，如“由文献[8,10～14]可知”。*

*不得将引用文献标示置于各级标题处。*

表2.1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| \*\*\* | \*\* | \*\* | \*\*\*\* | \*\* | \*\*\* | \*\*\* |
| \*\* | \*\*\*\* | \*\*\*\* | \*\* | \*\*\*\*\* | \*\*\*\*\* | \*\* |

*关于插表：论文的表格可以统一编序（如：表15），也可以逐章单独编序（如：表2.5），编序方式应和插图及公式的编序方式统一。表序必须连续，不得重复或跳跃。*

*表格中各栏都应标注量和相应的单位。表格内数字须上下对齐，相邻栏内的数值相同时，不能用‘同上’、‘同左’和其它类似用词，应一一重新标注。*

*表格的结构应简洁。*

*表序和表题置于表格上方中间位置，无表题的表序置于表格的左上方或右上方（同一篇论文位置应一致）。*

*数字用法：公历世纪、年代、年、月、日、时间和各种计数、计量，均用阿拉伯数字。年份不能简写，如1999年不能写成99年。*

*数值的有效数字应全部写出，如：0.50:2.00不能写作0.5:2。*

*软件：软件流程图和原程序清单要按软件文档格式附在论文后面。特殊情况可在答辩时展示，不附在论文内。*

*工程图按国标规定装订：图幅小于或等于3#图幅时应装订在论文内，大于3#图幅时按国标规定单独装订作为附图。*

*计量单位的定义和使用方法按国家计量局规定执行。*

**参考文献**

1. \*\*\*\*.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
2. \*\*\*.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*
3. \*\*\*.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
4. \*\*.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*注：为了反映论文的科学依据和作者尊重他人研究成果的严肃态度，同时向读者提供有关信息的出处，正文之后一般应刊出主要参考文献。列出的只限于那些作者亲自阅读过的，最重要的且发表在公开出版物上的文献或网上下载的资料。参考文献表上的著作按论文中引用顺序排列，著作按如下格式著录：序号 著者. 书名(期刊). 出版地: 出版社，出版年顺序列出(据GB 7714-87《文后参考文献著录规则》)。*

**【特别提醒】本模版仅供参考，如因版本等原因发生变化以及其他未尽事宜，请以《北京工业大学本科生毕业设计（论文）撰写规范》为准。**