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

1.绪论

1.1 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。

1.2 CPU 高速缓存

1.2.1 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

$$\mathrm{C} = \mathrm{B} * \mathrm{N} * E_n$$

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），E_n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。

1.2.2 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S_p一般表示为

$$S_p = \frac{1}{H_c \frac{T_c}{T_m} + (1 - H_c)}$$

其中，H_c为 cache 的命中率，T_c为 cache 的访问周期，T_m为主存储器的访问周期。可推断出，当$H_c \to 1$时，$S_p \to \frac{T_m}{T_c}$。研究表明[1]，H_c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。

1.2.3 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。

1.3 Cache 数据预取

1.3.1 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[2]。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。

1.3.2 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。

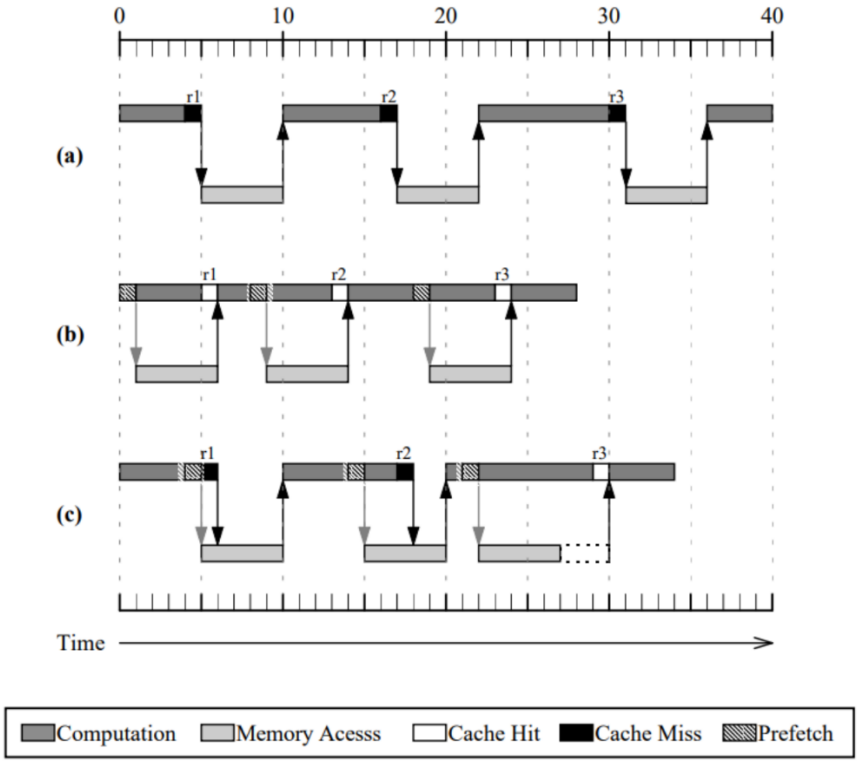


图 1 程序运行消耗的时间(a)不进行预取(b)完美预取(c)退化的预取

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[3]。

1.3.3 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

$$\mathrm{Cov} = \frac{M_p}{M_c}$$

其中，Cov 为覆盖率，M_p为因预取所减少的缺失数，M_c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

$$\mathrm{Acc} = \frac{M_p}{P_{useless} + M_p}$$

其中，Acc 为准确率，$P_{useless}$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。

1.4 深度学习

1.4.1 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。

1.4.2 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。[4]

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。

1.5 FPGA

1.5.1 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[5]。

1.5.2 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。

1.6 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。

1.6.1 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。

1.6.2 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[6]。

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[7]。

1.6.3 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，设计一个比较器，输入 2 个宽度为 8 的无符号整型数据，输出较大的结果，则可编写 Chisel 程序为

```
class Max2 extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(8.W))
    val in1 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })
  io.out := Mux(io.in0 > io.in1, io.in0, io.in1)
}
```

编译后在指定文件夹中输出的自动生成 Verilog 代码

```
circuit Max2 : @[:@2.0]
  module Max2 : @[:@3.2]
    input clock : Clock @[:@4.4]
    input reset : UInt<1> @[:@5.4]
    input io_in0 : UInt<8> @[:@6.4]
    input io_in1 : UInt<8> @[:@6.4]
    output io_out : UInt<8> @[:@6.4]

    node _T_11 = gt(io_in0, io_in1) @[Max2.scala 17:24:@8.4]
    node _T_12 = mux(_T_11, io_in0, io_in1) @[Max2.scala 17:16:@9.4]
    io_out <= _T_12
```

显然，Chisel 程序代码更加符合使用者的设计和阅读习惯，且直观易懂。另

外 Chisel 提供的预定义类和函数也简化了编写过程，一些 Verilog 固定输入信号如 clock，reset 等不必再手动定义。

2.系统分析

2.1 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 等人[8]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 等人[9]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 等人[10]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 1.6.2 节中已经讨论过。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。

2.2 系统模块设计

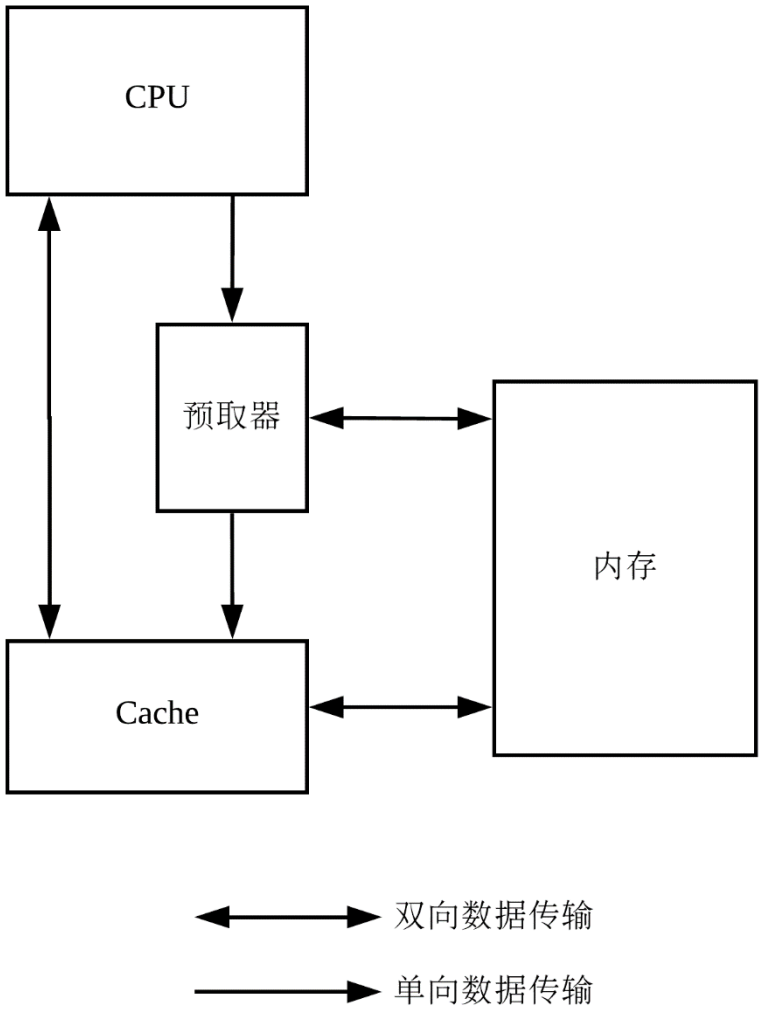


图 2 系统模块设计

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。

2.3 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，Chisel 版本 3.1.0，Scala 版本 2.11.12，sbt 版本 1.1.1。FPGA 综合及布线工具 Vivado。

实验平台的 FPGA 为 Xilinx 生产的 Zynq-7000 SoC。SoC 的特点是同时具有可编程的硬件 FPGA 和可编程软件的处理器系统两种功能。Zynq-7000 配备有 ARM Cortex-A9 双核处理器，并集成了基于 28nm Artix-7 或 Kintex®-7 的 FPGA 功能，有着极佳的性能/功耗以及灵活性优势。

2.4 测试程序

3.系统详细设计

3.1 参数设计

系统中定义了 Params 接口负责指定所有相关参数，具体数据结构如表 1 所示

表 1 Params 接口设计

数据名称	数据类型	描述
addrWidth	Uint	内存地址宽度
memSize	Int（单位：MB）	内存容量
cacheSize	Int（单位：kB）	Cache 容量
blockSize	Uint	缓存块大小
assoc	Uint	Cache 的相联度
numSets	Uint	Cache 组数

3.2 CPU 模块设计

CPU 模块的主要功能是负责模拟真实 CPU 运行程序时向 cache 发出数据请求的行为。如图 3 所示

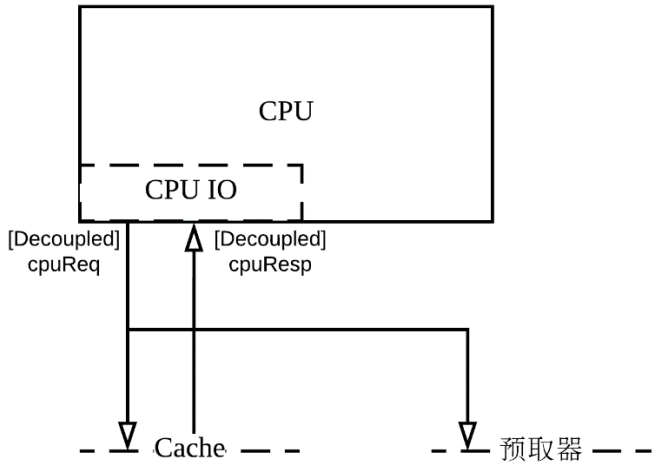


图 3 CPU 模块详细设计图

CPU 模块通过 CPU IO 向 cache 发出数据请求，同时由预取器接收到请求并进行预取判断。CPU IO 接口的具体设计列在表 2、3、4 中

接口名称	方向	数据类型	描述
cpuReq	输出	Decoupled	CPU 发送给 cache 的数据请求
cpuResp	输入	Decoupled	CPU 由 cache 接收的数据

表 2 CPU IO 接口设计

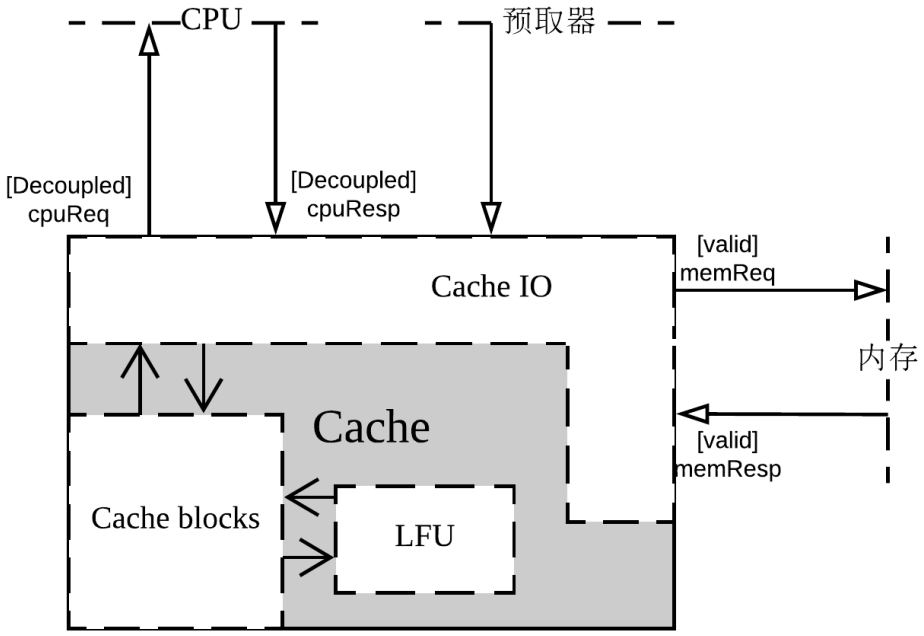
表 3 cpuReq 具体结构

数据名称	方向	数据类型	描述
valid	输出	Bool	CPU 请求有效信号
ready	输入	Bool	Cache 接收有效信号
read	输出	Bool	请求是否为读数据
addr	输出	UInt[addrWidth]	请求数据的地址

表 4 cpuResp 具体结构

数据名称	方向	数据类型	描述
valid	输入	Bool	Cache 应答有效信号
ready	输出	Bool	CPU 接收有效信号
data	输入	UInt[blockSize]	Cache 返回的数据值

3.3 Cache 模块设计



附图 Cache 模块设计图

Cache 模块由三个主要部分构成：cache blocks、cache IO、LFU 替换算法。

3.3.1 Cache blocks 结构设计

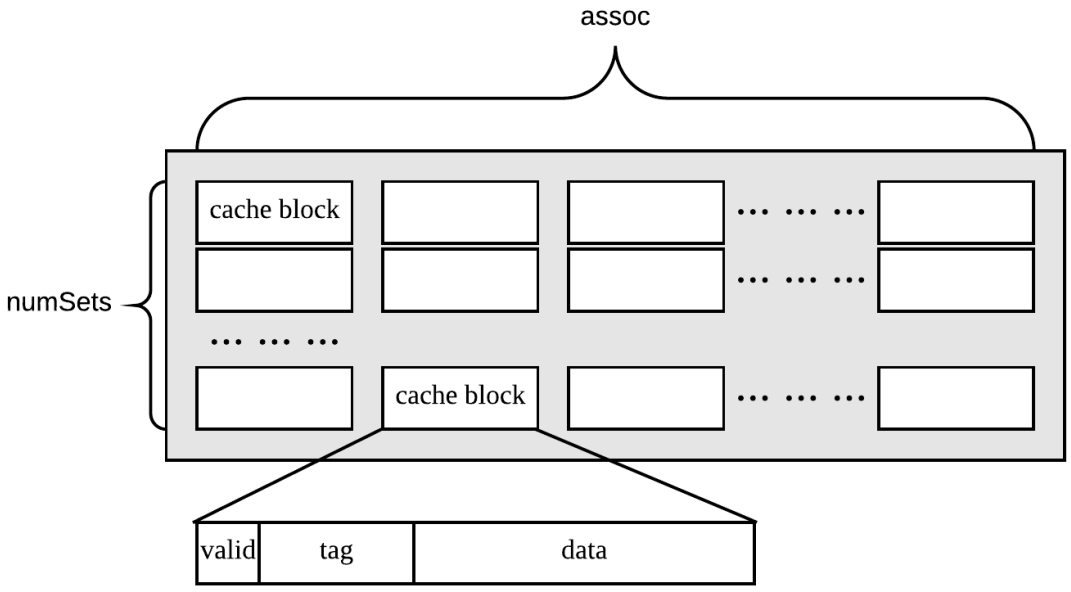


图 4 cache blocks 结构设计

本文中 cache 的存储结构采用组相联结构，每个缓存块包含有效位、tag 和数据三个部分。具体数据结构如表 5 所示

表 5 cache 存储结构设计

数据名称	数据类型	描述
cacheBlock ├ valid ├ tag └ data	Bundle Bool UInt[tagWidth] UInt[blockSize]	缓存块 缓存块的有效位 缓存块的查找标签 缓存块存放的数据
blocks	Mem[numSets]*[assoc]	按组为单位定义的二维

		数组 cache blocks，组员数据类型为 CacheBlock

3.3.2 cache IO 结构设计

Cache IO 负责与外部模块进行数据传输，具体数据结构设计如表 6、7、8、9、10 所示

表 6 Cache IO 接口设计

接口名称	方向	数据类型	描述
memReq	输出	Valid	Cache 发给内存的读取请求
memResp	输入	Valid	内存应答数据
cpuReq	输入	Decoupled	CPU 发送给 cache 的数据请求
cpuResp	输出	Decoupled	Cache 的应答数据

表 7 memReq 具体结构

数据名称	方向	数据类型	描述
valid	输出	Bool	Cache 请求有效信号
read	输出	Bool	读有效信号
Addr	输出	UInt[addrWidth]	请求的数据地址

表 8 memResp 具体结构

数据名称	方向	数据类型	描述
valid	输入	Bool	内存应答有效信号

data	输入	UInt[blockSize]	内存返回的数据

表 9 cpuReq 具体结构

数据名称	方向	数据类型	描述
valid	输入	Bool	CPU 请求有效信号
ready	输出	Bool	Cache 接收准备信号
read	输入	Bool	读有效信号
addr	输入	UInt[addrWidth]	请求数据的地址

表 10 cpuResp 具体结构

数据名称	方向	数据类型	描述
valid	输出	Bool	Cache 应答有效信号
ready	输入	Bool	CPU 接收准备信号
data	输出	UInt[blockSize]	Cache 返回的数据值

3.3.3 替换算法设计

本文使用与 LRU 算法法技术思想相同的最久没有使用法（LFU），其实现方法是记录近期使用次数的多少。记录方法是为每个缓存块设置一个计数器，具体执行流程如下

1) 被调入或者被替换的块，其计数器清零，其他计数器加 1；

2) 当访问命中时所有块的计数值与命中块的计数值进行比较，如果计数值

小于命中块的技术值，则该块的计数值加 1。如果块的计数值大于命中块的计数值，则不进行变动。命中块的计数器清零；

3) 当出现缓存缺失时，选择计数值最大的缓存块进行替换。

Cache 的缓存块替换模块的执行流程可用伪代码表示为

```
when（块命中）{
读取命中块
调用 LFU 命中函数，更新计数器值
将命中块数据发送给 CPU
}.otherwise{
调用 LFU 缺失函数，更新计数器值，返回被替换块的 id
向内存发送读数据请求
    →等待内存回应
接收内存的返回数据，赋值给被替换块
将被替换块的数据发送给 CPU
}
```

3.4 内存模块设计

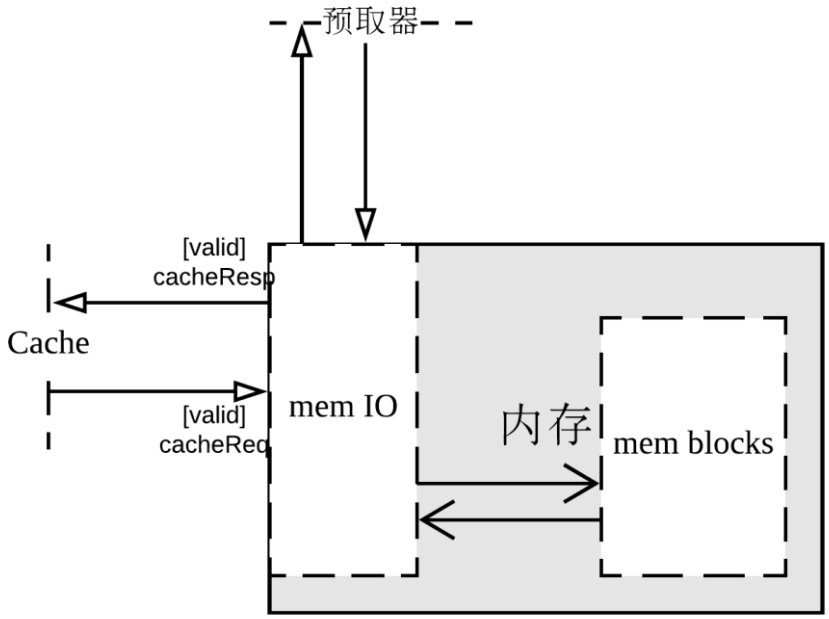


图 5 内存模块设计

内存模块由两个部分组成：mem IO 负责与外部模块进行数据传输，mem blocks 为内存的存储结构。

3.4.1 内存 IO 结构设计

内存 IO 负责与外部模块进行数据传输，具体数据结构设计如表 11、12、13 所示

表 11 内存 IO 接口设计

接口名称	方向	数据类型	描述
cacheReq	输入	Valid	Cache 发给内存的读取请求
cacheResp	输出	Valid	内存应答数据

表 12 cacheReq 具体结构

数据名称	方向	数据类型	描述
valid	输入	Bool	Cache 请求有效信号
read	输入	Bool	读有效信号
addr	输入	UInt[addrWidth]	请求的数据地址
data	输入	UInt[blockSize]	写请求时的写入数据

表 13 cacheResp 具体结构

数据名称	方向	数据类型	描述
valid	输出	Bool	内存应答有效信号
data	输出	UInt[blockSize]	内存返回的数据

3.5

3.6

4. FPGA 综合实验过程

4.1 FPGA 实验流程概述

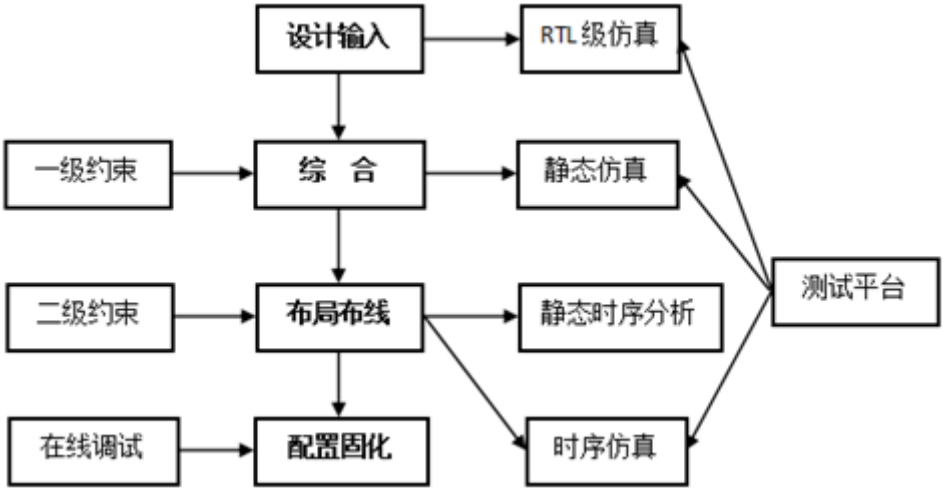


图 6 FPGA 实验流程图

如图 6 所示，FPGA 的开发流程可分为如下四个主要步骤：

1) 设计输入。设计输入方式有 3 种：原理图、HDL 和 IP 核。其中 HDL 语言具有不同层次上的抽象，这些抽象层包含开关级、逻辑门级、RTL 级（Register Transfer Level，寄存器传输级）、行为级和系统级。因为 Verilog 拥有较广泛的设计群体和其与 C 语言的相似性，且考虑到 Verilog 虽然在仿真方面具有高层建模能力不足的缺陷，但 SystemVerilog 可以在系统级和行为级上为 Verilog 做补充，所以本次设计选用的 HDL 语言 SystemVerilog。其中，RTL 级仿真（或功能仿真）属于第一道测试，对工程在寄存器描述时进行测试，查看其在 RTL 级描述功能的正确性。

2) 综合。对设计输入进行综合，得到一个可以和 FPGA 硬件资源相匹配的描述。假设 FPGA 是基于 LUT 结构的，那么就得到一个基于 LUT 结构的门级网表。其中，一级约束即综合约束，用来指导综合过程，

是小范围内实现运行速度和资源消耗平衡的一种方式。不同的约束，将会产生性能不同的电路。另外，静态仿真（或门级仿真）：是综合后 LUT 门级网表的仿真，目的是当工程用 LUT 门级描述时，从功能上验证工程的正确性。

3) 布局布线。布局考虑的问题是如何将这些逻辑上已连接的 LUT 及其他元素合理地放到现有的 FPGA 里，并且达到功能要求的同时保证质量。布线考虑的问题就是线路最优问题，具体来说就是如何让各部分连接起来，如何让输入输出信号到达相应的位置，且保证电路连接后的整体性能。其中，二级约束：即布局布线约束，可分为位置约束和时序约束。位置约束指布局策略，根据所选择的 FPGA 平台现有的硬件资源分布来决定布局。时序约束在很大程度上和布线有关，但是是先用软件默认的原则布线，然后对其结果进行静态时序分析，不满足时序要求的，再对具体的问题路径做一些指导约束。另外，布线时时延问题的截获就可以通过时序仿真完成。将工程下载到 FPGA 芯片上，可通过在线调试（或板级调试）分析代码运行的情况。

4) 配置及固化。在配置模式和初始化模式下，FPGA 的用户 I/O 处于高阻态（或内部弱上拉状态），这两个模式相继结束后，进入用户模式，此时用户 I/O 就能够按照用户设计的功能工作。固化既是将程序固化到存储器中。

5. 实验结果及分析

6. 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。

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

参考文献

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》)。

[1] Solihin, Yan. Fundamentals of parallel multicore architecture. Boca Raton, FL: CRC Press, Taylor & Francis Group. 2016, p. 163.

[2] Mowry, T.C., Lam, S. and Gupta, A., “Design and Evaluation of a Compiler Algorithm for Prefetching,” Proc. Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA, Sept. 1992, p. 62-73.

[3] Steven, VanderWiel; David J, Lilja: A Survey of Data Prefetching Techniques. 1996.

[4] Schmidhuber, J. (2015). "Deep Learning in Neural Networks: An Overview". Neural Networks. 61, 2015, p.85–117.

[5] https://en.wikipedia.org/wiki/Field-programmable_gate_array

[6] 钟文枫. SystemVerilog 与功能验证. 机械工业出版社. 2010. ISBN 978-7-111-31373-1.

[7] https://en.wikipedia.org/wiki/SystemVerilog

[8] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O’Boyle, and O. Temam, “Rapidly selecting good compiler optimizations using performance counters,” in Code Generation and Optimization, 2007.

[9] C. McCurdy, G. Marin, and J. Vetter, “Characterizing the impact of prefetching on scientific application performance,” in International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS13), 2013.

[10] S. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, “Machine learning-based prefetch optimization for data center applications,” in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, 2009, p. 56.