**面向大数据应用的词频统计算法加速结构研究**

摘要

Abstract

1. 前言

1.1研究背景及意义

人类已经步入大数据时代，根据国际数据咨讯（IDC）公司监测，全球数据量大约每两年翻一番，预计到2020年，全球将拥有35ZB的数据量。(如图1所示)大数据应用具有数据量巨大且相对独立、数据类型多样而依赖性低、流动速度快和浮点运算少的特点，通常由大量松耦合的小规模作业组成，而非一个大规模作业。传统的高性能计算追求的是峰值性能下的浮点运算速度，因此不能很好的适应大数据应用场景。传统的计算机体系结构也远远达不到大数据分析所需要的性能需求。因此，对大数据的处理不仅需要从软件层面进行优化和改变，更需要从底层硬件对大数据的特殊需求进行变革性的适应和支持。

大数据分析算法分类

|  |  |  |  |
| --- | --- | --- | --- |
| **领域分类** | **Workload/算法** | | **打分** |
| **Basic operation** | Sort | Terasort | 5 |
| Shuffle | 5 |
| Grep | KMP | 5 |
| Boyer-Moore | 5 |
| Wordcount | | 5 |
| **Classification** | Naïve Bayes | | 5 |
| SVM | | 5 |
| **Cluster** | K-means | | 5 |
| Fuzzy c-means | | 3 |
| **Segmentation** | HMM | | 4 |
| **regression analysis** | regression analysis | | 4 |

Basic operation是在大数据领域经常会用到的几个算法，几乎所有应用的后台工作都要涉及到，在大数据应用分析和Benchmark设计过程中是必须要有的，最常用如Sort、Grep、WordCount几个算法。多数互联网服务都会用到词频统计来提高服务质量，特别是在网络服务相关的应用中用于日志分析，可以用于挖掘大量有价值的信息。而目前尚无针对词频统计算法的专用加速器，因此本文对此展开研究。

1.2国内外研究现状

1.2.1GPU加速

大规模数据并行应用对可扩展性、计算能力和存储带宽的迫切需求促使高性能微处理器正在向众核体系结构演变。作为一种新型的众核体系结构,图形处理器（GPU）采用大量晶体管用于计算单元,控制逻辑简单,具有非常高效的存储带宽层次，实现更为广泛的数据并行计算。

通用加速器GPU虽然吞吐量高，浮点运算能力强但浮点运算量小且任务并行不规则，并不完全适应于高通量，实时性强的大数据应用。

1.2.2FPGA加速

当今的FPGA处理性能潜力非常大，不仅能够支持深度可变的流水线结构，而且提供丰富的并行计算资源，使得FPGA能够在一个时钟周期内完成非常复杂的处理功能。而且FPGA具有可再编程的能力，不仅能够灵活改变协处理器的处理能力，而且可以对协处理器的功能进行灵活改变。FPGA有着很强大的内置存储器，器件引脚的数量将不再会对协处理器的逻辑存储器访问带宽产生限制影响，由于存储器能够与运算逻辑进行紧密结合，就不再需要借助于外部的高速存储器进行数据缓冲，从而保证了数据的流通便利，节省了数据访问的时间，避免了数据不一致的问题。

虽然FPGA具有上述优点，但不可忽略的是功耗和不同时钟域的转换问题，高速电路设计和信号完整性也是FPGA当前的设计难点。在面向大数据应用的硬件加速结构设计时，还有待改进和提高。

1.2.3专用硬件加速器

研究表明，专用加速器可以得到主流通用处理器1000~10000倍的效率。因此，面向特定领域或特定算法设计专用硬件加速器是非常有必要的。例如，针对机器学习领域的CNN和DNN算法，陈云霁等人设计的DianNao和DaDianNao（一种小尺度的高吞吐率机器学习加速器）与通用芯片和GPU相比，计算速度提高几十倍，功耗只有十分之一，整体能效提高450倍。

专用加速器但大都针对特定算法或特定问题，而目前尚无针对词频统计算法的专用加速器，

1.2.4面向大数据应用的硬件加速结构研究

在当今大数据火热发展的趋势下，学术界对大数据相关应用领域的 Benchmark也已经有了很多的研究成果。例如HiBench (Intel 开放的一个 Hadoop Benchmark Suit)、 YCSB (2010 年 Yahoo 研究院针对 NoSQL 系统开发的开源基准测试框架)、 DCBench (一个对于数据中心负载的 Benchmark 集合)、 LinkBench (Facebook 开发的一套用于对社交网络数据库进行性能测评的工具集)、 CloudSuite (针对当前最为热门的云计算而开发的一套测试程序集)、 BigDataBench (中科院计算所开发的一套互联网大数据应用相关的 Benchmark 集) 等等。

1.3本文主要内容

本文共有以下五大章节。各章主要内容简介如下：

第一章:主要介绍了本课题的研究背景及意义，现阶段国内外对Benchmark的研究现状，以及对本文章节进行全面阐述的主要内容介绍。

第二章:首先介绍了算法的重要性以及算法的功能和实现思路，并对算法进行特征分析，从计算特征，访存特征，并行性和线程间的依赖关系入手，根据其特征考虑算法对体系结构的需求，来设计加速方案。为了找出最适合优化的点，在x86平台下用gprof工具测试1G的数据量，找到程序最耗时的函数即为热点，将热点精确化，发现此部分的字符比较是固定的指令模式，如果将这种固定模式的速度大幅提升，则程序的整体性能也会有明显的优化效果

第三章：

第四章：本章先介绍了现有的模拟器平台，接着对指令的定义和功能部件的模拟实现做了详细的说明，最后是对模拟器core，cache和memoey组件的主要修改描述。

第五章：本章首先介绍了对测试用例所做的修改，接着描述出测试环境，并对比不同测试集大小时的优化效果。

第六章 总结与展望：总结回顾了本文的研究内容和研究成果，并简单介绍了今后研究工作的大体方向，对今后的研究工作提出了展望。

1.4本章小结

本章介绍了本文的研究背景及意义，并从国内外研究现状对本文进行分析，最后对本文要阐述的研究内容进行了主要内容分析。

2.wordcount算法分析

2.1wordcount算法简介

2.1.1算法重要性

Benchmark 提供了一种比较不同大数据系统的基准。同时，Benchmark 也表征了大数据应用的典型需求。在设计 Benchmark 的工作中，分析并提取核心 Workload (核心 Workload 是指能够反映出整个应用程序特点和硬件需求的核心算法或者核心执行模块) 是非常重要的一步。

|  |  |  |
| --- | --- | --- |
| Workload | 应用领域 | 应用场景 |
| Grep | 搜索引擎  社交网络  电子商务 | 记录分析 |
| 网络信息提取 |
| 模糊搜索 |
| Bayes | 社交网络  电子商务 | 垃圾邮件识别 |
| 网页分类 |
| SVN | 社交网络  电子商务 | 图像处理 |
| 数据挖掘 |
| 文本分类 |
| PageRank | 搜索引擎 | 网页排序 |
| Fuzzy K-means  K-means | 搜索引擎  社交网络  电子商务 | 图像处理 |
| 高分辨率地形分类 |
| HMM | 搜索引擎  社交网络 | 语音识别 |
| 语句分割 |
| 手写识别 |
| WordCount | 搜索引擎  社交网络  电子商务 | 词频统计 |
| TFIDF值计算 |
| 获取统计用户操作 |
| Sort | 搜索引擎  社交网络  电子商务 | 文件分类 |
| 页面排序 |

**不同 Workload 的应用领域以及应用场景**

由表可以看出wordcount算法是当下热门领域如搜索引擎、社交网络以及电子商务的交叉点，应用面广泛。词频统计算法也是大数据领域最常用的算法之一，它很简单但是它是一种非常典型的操作。因此，本文选择wordcount算法作为研究对象。

2.2.2算法的功能及实现思路

本文描述的词频统计算法能够准确统计文本文件中所有英文单词出现的次数，并按单词频次由小到大的顺序输出为文档。Wordcount算法典型的实现方式是基于Map/Reduce模型来实现，Map/Reduce就是任务的分解与结果的汇总，即“Map”是将一个大的任务分解成为多个小任务，”Reduce”是将分解后的小任务处理结果进行汇总，从而得出最后的分析结果。

具体流程如下：(1)本文为了提高程序的并行性，采用多线程编程的手段，将文件根据线程数均分成等长数据段，可根据文件的数据量和统计环境来更改线程数。每个线程对自己分到的数据段进行处理，将单词依次拆分出来，生成<key,value>对，key值即单词在内存中存放的ascii值，value即线程的id。

(2)得到<key,value>对后，根据key值来完成哈希散列，找到单词在哈希表中的位置，并将它们按照key值大小进行排序链接，将key值相同的个数累加合并，得到线程内部链接结果。

(3)在哈希表的同一列上，对应着不同线程相同或相似单词的散列值，按列把不同线程上的单词链表根据key值进行排序链接，将key值相同的个数累加，并作为WordCount的最终输出结果。

根据上述分析可以得出Wordcount在执行过程中主要操作的流程图如图16所示。



Wordcount算法流程图

2.2wordcount算法特征提取

2.2.1算法特征分解步骤

步骤1：在算法的输入初期，需要将文件切割为固定的文件输入流，便于并行执行，每个文件为一个split，将分割好的split进行处理，生成<key,value>键值对，主要包括load指令，比较指令。

步骤2：根据key进行哈希散列，并加入哈希表，主要操作为把此单词和哈希表中当前位置已有此单词比对，如果此单词已存在，直接将个数加1，否则说明这是一个当前位置中不存在的新单词，按顺序加入链表。主要为load-load-compare的指令模式。

步骤3：将不同线程同一哈希列的链表进行combine合并，主要包括LOAD指令、比较指令、交换指令、ADD累加操作。

Wordcount算法特征分解

|  |  |  |
| --- | --- | --- |
| 算法关键步骤 | 代表函数 | 对应指令操作 |
| 提取键值对 | If(TEXT[i]!=’ ’)  word[m][n++]= TEXT[i++];  Else  m++; | 主要为比较操作 |
| 根据key进行哈希散列，并加入哈希表 | While（p）  {if(strcmp(p->word,word)==0)  {  p->num++;  return;  }  p=p->next;  } | 主要为LOAD-LOAD-COMPARE  指令模式 |
| 对不同线程同一哈希列的单词排序合并 | 不同的排序算法具有不同的代表函数 | 主要为比较操作，交换操作，ADD累加操作 |

2.2.2wordcount算法特征分析

（再补充）

|  |  |
| --- | --- |
| 编号项 | 特征 |
| 计算特征 | Wordcount算法主要为字符比较及字符频次统计，涉及Hash函数及少量排序算法，算法复杂度为O(NlogN)。 |
| 访存特征 | 由于对比操作都是对字符的，访存粒度多为8 bits。顺序访存，具有很好的局部性。但是在匹配失败时候，需随机访问数组表，所以局部性较差。访存和计算比大致为2:1，属存储密集型应用。 |
| 并行性 | 样本数量巨大时，可以切割样本，多任务并行处理。每个任务负责一部分数据的比较。 |
| 线程间依赖性 | 并行之后不需要合并处理，只需处理好切割部分的匹配即可。所以线程间没有通信需求，依赖性低，可以完全并行执行。 |
| I/O | 属存储密集型应用，有大量的存储器读写操作。 |

2.2.3wordcount算法对体系结构的需求

（再补充）

|  |  |
| --- | --- |
| 结构设计 | 具体需求 |
| 指令级 | Wordcount算法主要是大规模字符串样本的对比和哈希及计数操作，涉及LOAD/STORE指令、CMP指令以及简单的ADD/SUB指令。“取数-取数-对比”是其主要的操作模式。可以用一条复合指令实现固定模式的操作。 |
| 协处理部件 | 在此算法里主要是CMP操作和LOAD/STORE操作，没有复杂的计算操作，只有简单的ADD/SUB操作。协处理可以涉及计算矩阵，处理“取数-对比-计算-写会”模式，加快处理速率。 |
| 内部高速总线 | Wordcount算法可对样本数据进行并行搜索，加快执行效率。因其属访存密集型应用，可以增大片上存储空间。核间并不多，内部总线主要需求为保证核的访存效率。 |
| 外部高速总线 | Wordcount算法数据量巨大，需要频繁的与外存进行数据交换。无论使用加速卡、多处理器或多机进行并行加速时，都需要外部能提供类似PCIe的连接通道，以及类似DMA这样的块传输机制，达到快速高效的数据迁移。 |

2.3算法的热点分析

2.3.1热点分析平台及gprof工具简介

KMP 算法热点的评估是在 x86 平台上完成。首先，无论在什么平台上完成评估，KMP 算法的热点不会改变。其次，在 x86 平台上有很多非常方便但十分好用的性能分析工具，例如 gprof 工具。这在研究前期抓取算法重点区域非常方便，利于工作的展开。

默认情况下linux系统当中都带有这个工具。

GNU编译器工具包提供了一种剖析工具GNU profiler (gprof)。gprof可以为 Linux 平台上的程序精确分析性能瓶颈，能精确地打印出程序运行中各个函数消耗的时间，给出函数被调用次数和函数调用关系。gprof采用混合方法来收集程序的统计信息，它使用检测方法，在编译过程中在函数入口处插入计数器用于收集每个函数的被调用情况和被调用次数；也使用采样方法，在运行时按一定间隔去检查程序计数器并在分析时找出程序计数器对应的函数来统计函数占用的时间。

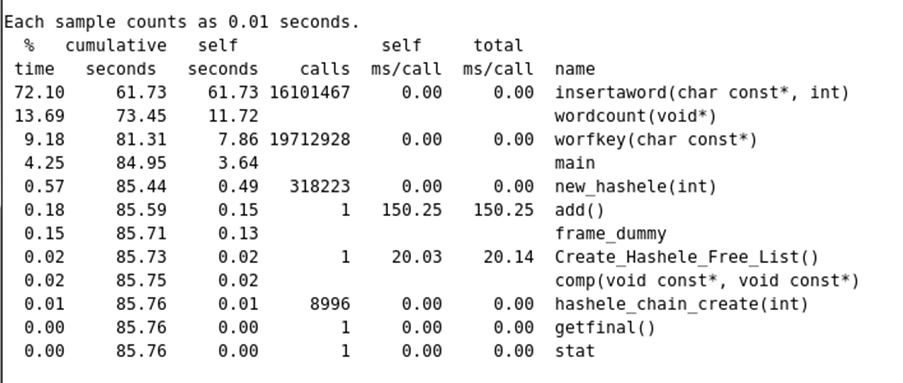
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| %time | Cumulative seconds | Self Seconds | Calls | Self TS/call | Total TS/call | name |
|
| 该函数消耗时间占程序所有时间百分比 | 程序的累积执行时间（只是包括 gprof 能够监控到的函数） | 该函数本身执行时间（所有被调用次数的合共时间） | 函数被调用次数 | 函数平均执行时间（不包括被调用时间，函数的单次执行时间） | 函数平均执行时间（包括被调用时间，函数的单次执行时间） | 函数名 |
|
|

表 Gprof统计信息解释

2.3.2测试集选择及热点的抓取

从小规模测试集逐渐增大，发现当测试集较小的时候，并不能体现大数据的特征，而且程序运行时间过短，不利于热点的抓取。测试集选择1G大小时，程序运行时间在实验可接受范围内，统计信息也比较方便。当测试集过大，运行时间过长，不利于实验的展开。因此，选择1G大小的测试集进行热点抓取。

实验结果：

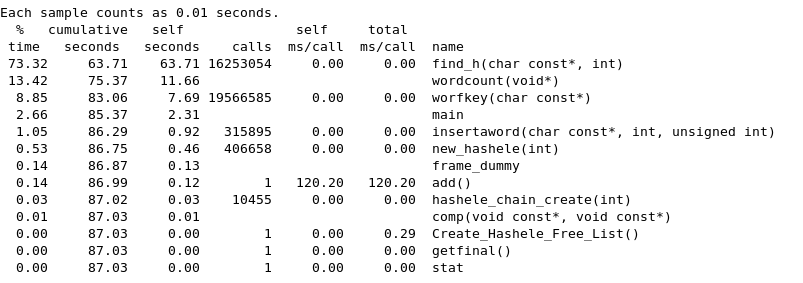


显而易见，程序的主要运行时间集中在函数insertaword上，占了函数总体运行时间的72%，也就是说，在大范围上，insertaword函数即为算法的热点。这不难理解，因为此函数完成的功能是线程内部的词频统计过程。为了便于加速结构的设计，本文进行热点的精确化查找，继续缩小范围。

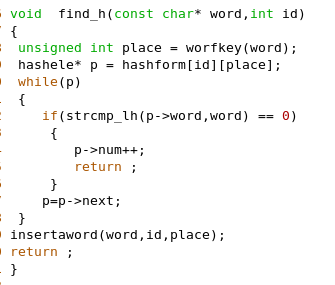
2.3.3热点的精确化

本文将insertaword函数中疑似最占用时间的部分分离出来，并构造为函数的形式，再次进行热点抓取。

实验结果：



由图 可以看出，程序中最耗时的函数由insertaword转移到 find\_h函数，证明提取的热点区域是正确的，优化的重点区域就是fiind\_h 函数中的内容。



find\_h()中的内容非常简单，主要就是while循环里的字符串比较，通过反汇编可以看出，字符串比较的指令模式就是ldrb-ldrb-cmp模式。

2.4本章小结

本章首先介绍了算法的重要性以及算法的功能和实现思路，并对算法进行特征分析，从计算特征，访存特征，并行性和线程间的依赖关系入手，根据其特征考虑算法对体系结构的需求，来设计加速方案。为了找出最适合优化的点，在x86平台下用gprof工具测试1G的数据量，找到程序最耗时的函数即为热点，将热点精确化，发现此部分的字符比较是固定的指令模式，如果将这种固定模式的速度大幅提升，则程序的整体性能也会有明显的优化效果。

3.硬件加速结构设计

3.1load-load-compare加速部件的设计

3.1.1分析热点

3.1.2针对load-load-compare模式的加速结构设计

3.1.3功能部件的设计

3.1.4指令的设计

3.2load-load-compare指令的模拟实现

3.2.1模拟器平台介绍

实验室已具备千核万线程模拟器平台，也称千线程并行模拟器，实现了千线程并行模拟器框架结构及各个功能模块，包括支持组件模式、第三方组件接口，支持ARMv6指令集并支持SMT、分支预测的Core组件、流水线结构（线程数、转移预测机制、取指和发射宽度、流水线级数）、片上网络结构等关键级数特征的配置（包括Mesh，双环，多级），支持存储层次cache机制（cache/SPM、策略、容量、层次、宽度和端口数等）、支持并行加速，支持低功耗研究所需的各类计数器，并允许开关，支持各种性能数据统计和功耗数据统计及结构化输出等。

|  |  |  |
| --- | --- | --- |
| 序号 | 模块名称 | 模块说明 |
| 1 | Core模块  (ARMv6指令级、SMT、分支预测) | ARM V6指令集定义、译码，实现相关通用及系统寄存器定义，实现中断、异常处理功能  单个ARM Core组件支持核内8线程SMT  ARM1176单发射8级流水，实现分支预测功能，实现并行流水线等功能 |
| 2 | Pthread库支持模块 | 核内，核间Pthread机制支持 |
| 3 | Mesh模块 | 支持大规模片上互联网络的组件，实现互联结构可配置，路由、仲裁、死锁避免等功能 |
| 4 | 双环模块 | 支持大规模片上互联网络的组件，实现互联结构可配置，路由、仲裁、死锁避免等功能 |
| 5 | Cache模块 | 能够配置为多级属性的Cache功能组件，支持LRU等替换策略，支持容量、宽度、端口数可配，支持时序精确级模拟 |
| 6 | SPM模块 | 支持大小、映射范围可配置 |
| 7 | Memory模块(MCU) | 实现内存管理单元组件，实现存储器功能组件，支持时序级模拟 |
| 8 | 时序信息统计模块 | 统计性能模拟时各个功能部件的运行时状态 |
| 9 | 并行加速模块 | 利用多线程加速，实现8线程3倍的加速比 |
| 10 | 低功耗统计支持模块 | 提供低功耗研究所需的各类计数器，并允许开关配置 |
| 11 | 统计信息格式化模块 | 使用YAML数据结构格式化输出信息 |
| 12 | 框架模块 | 模拟器底层框架 |

表 模拟器主要模块

用户可以使用Core、片上网络、Cache、Memory、SPM等组件任意搭建目标系统。其中Core组件支持ARMv6指令集，支持ARM1176流水线，SMT，分支预测，性能统计输出等内容。Cache组件支持多级配置，支持大小、类型、替换算法等参数的配置，支持性能统计输出等功能。片上网络包括Mesh组件和双环组件，支持虚通道、数据包分片等技术，支持性能统计输出及自动化分析等功能。Memory组件支持虚实地址转换，支持万级任务加载和管理等功能。通过以上自由独立的功能组件可以搭建符合实际需求的任意拓扑结构，满足模拟需求。

模拟器的正确性和健壮性已得到保障，本文的加速结构设计的平台已经具备，只需设计搭建一个能对wordcount算法计算性能大幅提升的模拟器即可。通过性能统计输出能直观获取程序运行总的 cycle 数及子线程的cycle数，加速效果易于量化。

3.2.2指令的定义和功能部件的模拟实现

ARMv6指令集主体定义在machine.def文件中，定义的方式与SimpleScalar的定义相同。由不同的宏定义

• DEFINST :定义一条指令，包含其名字、依赖的部件和寄存器等

• DEFLINK :在译码树中定义一条新的连接，指向一棵子树

• CONNECT :定义一棵新的译码子树

此外，每条指令在执行时要进行的具体操作也直接定义在指令集中，由“指令名+ \_IMPL后缀”的格式定义。执行指令的操作在译码时一并完成。

指令集定义模块中将不同类型的指令转换成统一的格式，DEFINST宏结构如下所示：

DEFINST (

<enum>, <opcode>, <opname>, <operands>, <fu\_req>, <iflags>,

<output deps...>, <input deps...>, <expr>

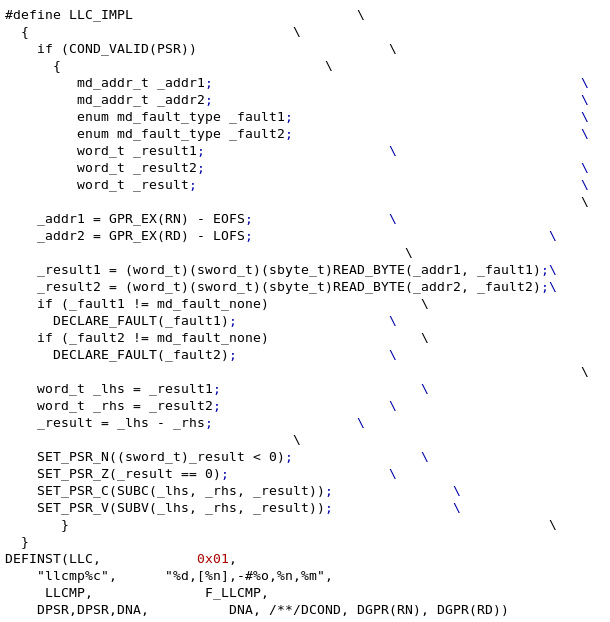
)

该定义中的每部分的具体含义解释如下表所示：

表 指令集定义模块参数说明表

|  |  |
| --- | --- |
| **参数** | **功能描述** |
| <enum> | 当指令被译码后返回的枚举值、用于标识每条指令 |
| <opcode> | 指令的操作码 |
| <opname> | 指令名称，对应于汇编码中的表示 |
| <operands> | 用于表示指令所有操作数的类型，具体的种类包括：   |  |  |  |  | | --- | --- | --- | --- | | J | 绝对跳转的目标域 | j | 相对跳转的目标（偏移量+PC） | | s | rs1寄存器 | t | rs2寄存器 | | d | rd寄存器 |  | disp30偏移量 | | S | 浮点rs1寄存器 | T | 浮点rs2寄存器 | | D | 浮点rd寄存器 | A | asi地址空间 | | i | simm13立即数 | I | imm22立即数 | | c | 条件跳转域 |  |  | |
| <fu\_req> | 用于表示指令所需用到的功能部件。可选枚举值包括：   |  |  |  |  | | --- | --- | --- | --- | | FUNA | 未使用单元 | ALU | 整数ALU | | IMULT | 整数乘法器 | IDIV | 整数除法器 | | FADD | 浮点加/减法器 | FCMP | 浮点比较器 | | FCVT | 浮点/整型转换器 | FMULT | 浮点数乘法器 | | FDIV | 浮点数除法器 | FSQRT | 浮点数平方根运算器 | |
| <iflags> | 指令标志位，用于标识指令类型。可选值包括：   |  |  |  |  | | --- | --- | --- | --- | | F\_ICOMP | 整型计算 | F\_FCOMP | 浮点计算 | | F\_CTR | 控制指令 | F\_UNCOND | 无条件转移 | | F\_COND | 条件转移 | F\_MEM | 访存指令 | | F\_LOAD | Load指令 | F\_STORE | Store指令 | | F\_DIRECT | 直接寻址 | F\_DIRJMP | 直接跳转指令 | | F\_INDIRJMP | 间接跳转指令 | \_COND | 条件跳转指令 | | F\_FPCON | 浮点条件跳转指令 | F\_TRAP | Trap指令 | |
| <output deps…> | 用于表示目的操作数的依赖关系，对应每个目的操作数有一项。可选的依赖关系包括：   |  |  |  |  | | --- | --- | --- | --- | | DGPR(N) | 定点寄存器N | DGPR\_D(N) | 双字定点寄存器N | | DFPR\_F(N) | 单精度浮点寄存器N | DFPR\_D(N) | 双精度浮点寄存器N | | DCC | 定点条件码 | DFCC | 浮点条件码 | | DNA | 无依赖选择器 |  |  | |
| <input deps…> | 用于表示源操作数的依赖关系，对应每个源操作数有一项，可选的类型与上面的<output deps>相同 |
| <expr> | 可用于实现其他辅助信息，比如实现指令功能的一段C程序，指令定义时可选 |

在已定义好的译码树中根据译码规则，找到译码树一个空的子节点，插入新定义的加速指令，指令的十六进制格式为：0xe1001010,一条指令完成ldrb-ldrb-cmp的功能。指令定义如下：



指令的十六进制格式为：0xe1001010,二进制表示如下图所示：



指令定义之后，在程序中加入相应的功能部件，通过以下方式即可方便地实现指令的功能模拟：

#define SYSCAT(X,Y) X##Y

switch (op)

{

#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3) \

case OP: \

SYMCAT(OP,\_IMPL); \

break;

#include "sparcv8.h"

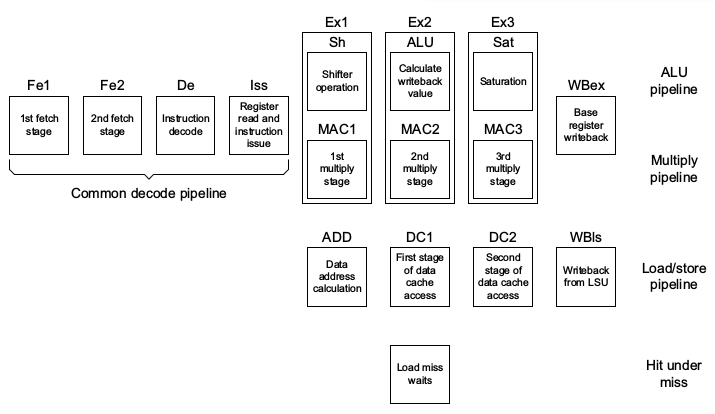
default:

panic("attempted to execute a bogus opcode");

}

3.2.3模拟core组件流水线处理机制

流水线在代码中被封装为数个函数，在每个时钟周期内按照从后向前的顺序调用。流水线的结构如下图所示。



ARMv6 8级流水线结构

流水线的函数调用顺序为：

start\_a\_new\_cycle(){

for(;;)

{

ss\_writeback(); //写回

ss\_execute\_stage2(); //执行

ss\_execute\_stage1();

ss\_issue(); //发射

ss\_decode(); //译码

ss\_fetch\_with\_callback(); //取指

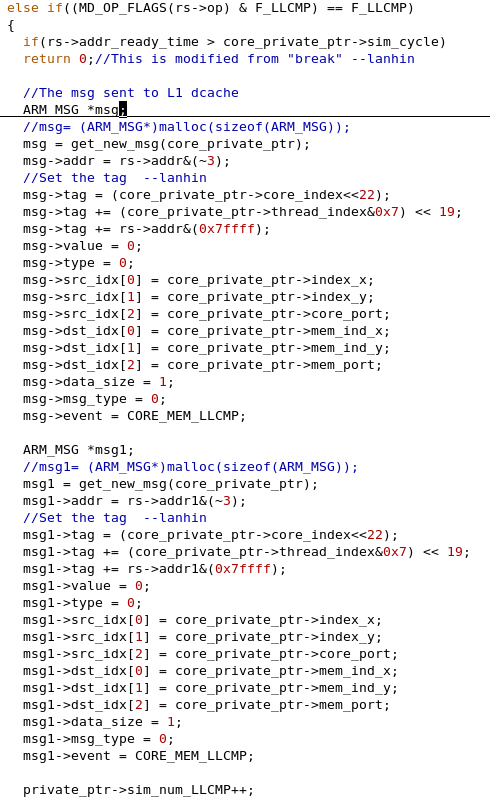
}

}

1. 功能模拟在ss\_decode函数即译码阶段已经完成，接下来的步骤是为了模拟时序，也就是进行性能模拟。添加消息事件CORE\_MEM\_LLCMP和MEM\_CORE\_LLCMP分别表示core发给memory的llcmp消息和memory发给core的llcmp消息

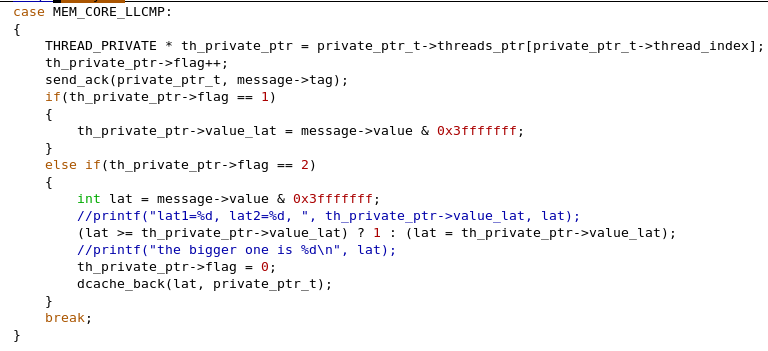
在ss\_\_execute\_stage2里定义发出的两条消息,两条消息的区别是地址不同，消息类型是CORE\_MEM\_LLCMP,并将消息发给框架

，代码如下：



另外在core的消息处理函数Port\_In中加入对llcmp消息的处理，接受两条访存消息的返回时，记时延较大大的为此次访存的latency。

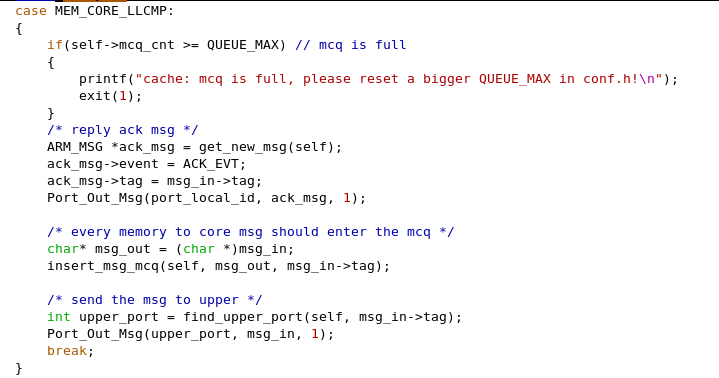
代码如下：



再加上组件的定义，性能统计信息的打印输出，对core组件的修改就完成了。

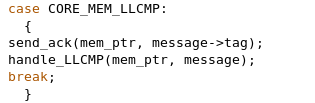
3.2.4模拟cache组件对新指令处理机制

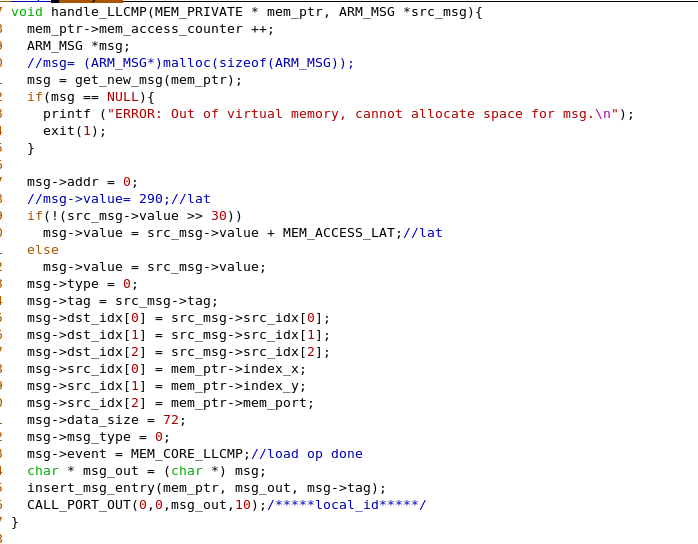
访存时cache 一次处理一个地址，而 LLCMP 指令一次传输两个地址，所以增大 cache 带宽，使之一次可处理两个地址，这样能大大减少访存时间。在cache消息处理函数中增加处理MEM\_CORE\_LLCMP消息的部分，先发送ack消息给core表示收到此类型消息，如果在cache中命中，则发消息给core；如果在cache中查找失败，则发消息给memory，代码如下：



3.2.5模拟memory组件对新指令处理机制

在memory消息处理函数里增加对LLCMP消息的处理，先发送ack消息给cache，再进行消息的处理，并返回message给cache，代码如下：





3.3继续加速方案(后续试验补充)………

3.4本章小结

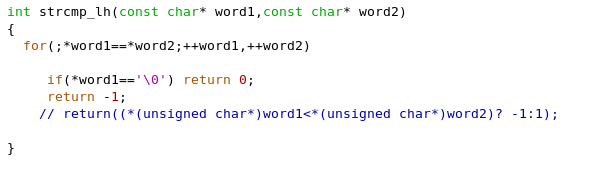
本章先介绍了现有的模拟器平台，接着对指令的定义和功能部件的模拟实现做了详细的说明，最后是对模拟器core，cache和memoey组件的主要修改描述。

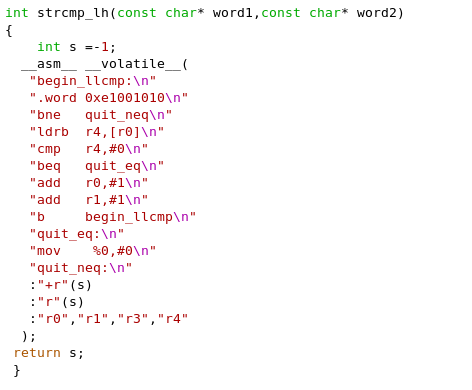
本章首先介绍了

4.加速效果评测与分析

4.1测试用例设计

因为加速指令的设计是不能通过编译器编译，所以只能用嵌入式汇编的形式把指令的十六进制码嵌进去，这样可以通过编译，也可以在修改过的模拟器上运行测试。测试用例C语言和嵌入式代码如下所示：





4.2实验环境及不同级别测试集的优化效果比对

因为模拟器是用软件模拟硬件，所以它的运行速度很慢，所以测试用例选择不能太大。

实验环境：

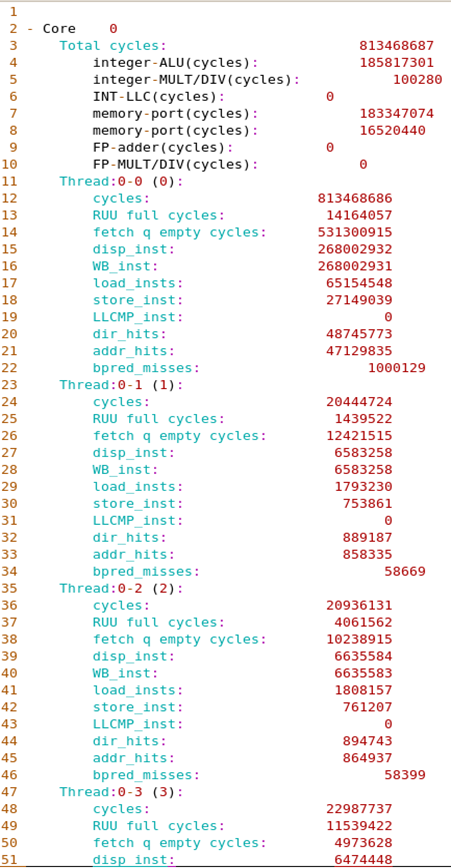
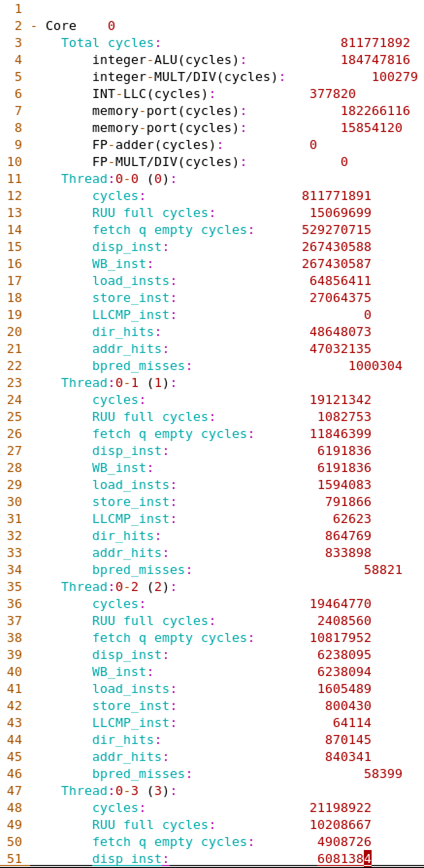
1.mesh4x4 每个核支持的最大线程数是4（在core的conf.h里定义）

2.cache的物理端口（nports=2）

3.程序的线程数是40

当测试数据集大小是1w行（约1.5M）

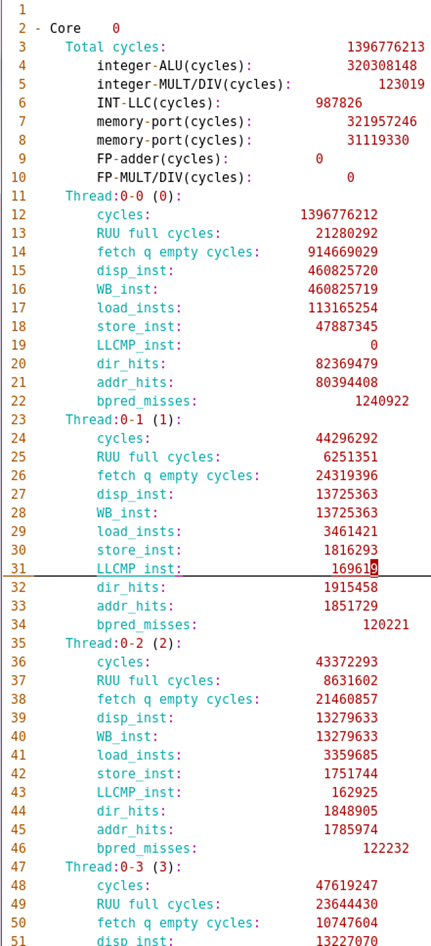
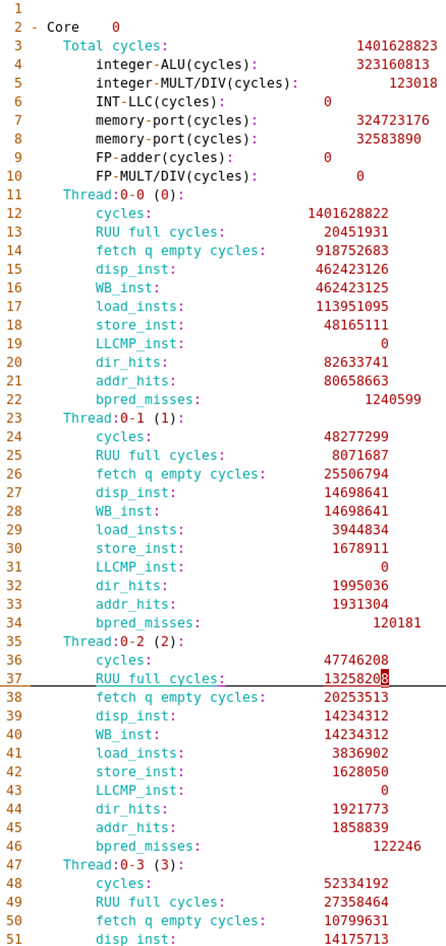
使用新指令： c语言版：



效果：子线程cycle数是原来cycle数的92%，加速了8%

当测试集增大到2w行（约3M）

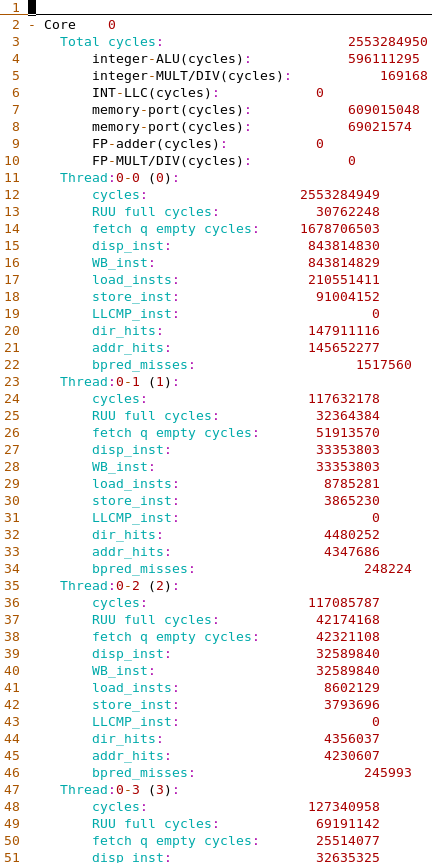
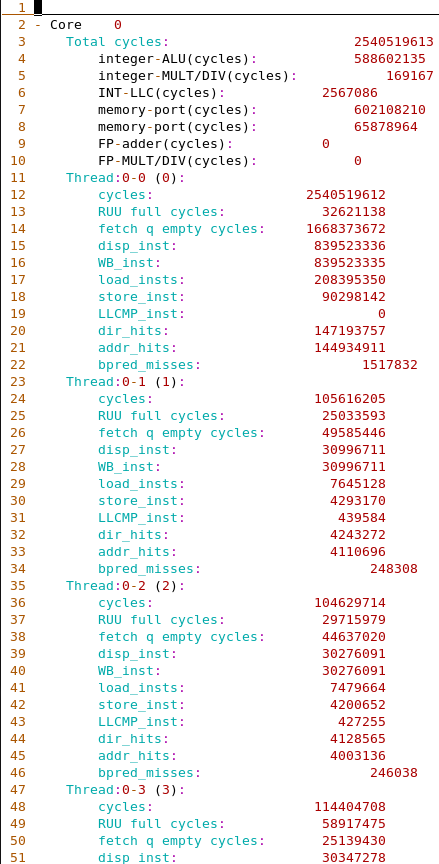
C语言版： 使用新指令：



效果：子线程cycle数是原来cycle数的90%，加速了10%

当测试集增大到4w行（约6M）

C语言版： 使用新指令：



效果：子线程cycle数是原来cycle数的88%，加速了12%

虽然现在加速效果并不明显，但是可以看出，随着测试集的增大，加速效果越来越明显。使用加速指令的优势也更能显现出来。

4.3本章小结

本章首先介绍了对测试用例所做的修改，接着描述出测试环境，并对比不同测试集大小时的优化效果。

结论

参考文献

致谢