**摘要**

**目录**

**图目录**

**表目录**

**第 1 章 研究背景**

全球的数据量出现爆炸式增长，数据成了当今社会增长最快的资源之一。根据国际数据公司 IDC 的监测统计，即使在遭遇金融危机的2009 年，全球信息量也比 2008 年增长了 62% ，达到 80 万 PB ，到 2011 年全球数据总量已经达到 1. 8ZB ，并且以每两年翻一番的速度飞速增长，预计到 2020 年全球数据量总量将达到 35 ZB，10年间增长 20 倍以上（如**图 1** 所示）。同时，数据类型也越来越复杂，85%以上的数据以非结构化或半结构化的形式存在，这样的数据无法用传统数据库工具对其内容进行抓取、管理和处理。众所周知，大数据有四个基本特征: 数据规模大(Volume)，数据种类多(Variety)，流动速度快(Velocity)，数据价值密度低(Value)，即所谓的四 V 特性。

**图 1 IDC 全球数据使用量预测**

传统的数据处理手段很难去处理这样的数据，所以，与大数据相关的大数据技术、大数据工程、大数据科学和大数据应用等迅速成为信息科学领域的热点问题。大数据研究领域吸引了产业界、政府和学术界的广泛关注，并且已经开始影响社会的发展与人们的日常生活。

传统的计算机体系结构已经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。

本文通过对现有比较热门的大数据相关应用领域的 Benchmark 进行分析比对，发现字符串匹配算法是大数据应用的基本算法，大部分其他算法也都会用到字符串匹配算法，因此设计一种针对字符串匹配算法的加速结构，这能大大提高大数据的处理能力。

**第 2 章 相关研究**

大数据领域自发展以来，一直是人们研究的热门领域，无论是研究相应的 Benchmark 以对系统结构进行评估，还是设计新型的体系结构对热门算法进行加速，都有了一些相当不错的成果。

**2.1 Benchmark 研究现状：**

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。此外，Benchmark 的测试结果也能促使大数据系统的不断改进。因此，Benchmark 作为量化设计方法的基础，已经被广泛用于评估新设计和新系统的优劣。

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。

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 |

**表 1 不同 Workload 的应用领域以及应用场景**

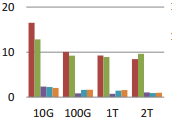
**表 1** 列出了常用 Workload 的应用领域以及应用场景。

**2.2 字符串匹配算法的重要性**

由表 1 可以看出字符串匹配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。字符串匹配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。Rajaraman 认为，对于大数据应用来说，简单的算法比复杂的算法算法更好，因为复杂的算法计算开销大。并且现在几乎所有比较成功的 Benchmark 都提取了字符串匹配算法 (Grep)。所以本文选择 Grep 作为研究对象。

如**图 2** 所示，在数据量不断增大的情况下， Cache 和 TLB 的失效率是不同的，逐渐降低，但是数据量足够大后，基本趋于稳定。这种现象对于体系结构研究者尤为重要，因为模拟过程是非常耗费时间的，那么过大的数据量是不利于模拟的。如果可以使上述失效率降低甚至不受数据量的影响，这对于新的体系结构的设计有相当大的益处。由此可见，体系结构设计领域还存在巨大的挑战。

据了解，现在还未曾有针对 Grep 的加速器，这为本文的研究方向指明了道路。

C:\Users\xmy72\AppData\Roaming\Tencent\Users\1285074029\QQ\WinTemp\RichOle\RTUU0B}Q35XAEM1FLD]B_05.png

C:\Users\xmy72\AppData\Roaming\Tencent\Users\1285074029\QQ\WinTemp\RichOle\_4QK0XX}G298$`6)5S$RT%7.png

C:\Users\xmy72\AppData\Roaming\Tencent\Users\1285074029\QQ\WinTemp\RichOle\OG$~NQO`BE[XZJ_89V2O%JX.png

C:\Users\xmy72\AppData\Roaming\Tencent\Users\1285074029\QQ\WinTemp\RichOle\{U(NVFOLB5AA1~DAIX)3$0K.pngC:\Users\xmy72\AppData\Roaming\Tencent\Users\1285074029\QQ\WinTemp\RichOle\]CIWW4}HS9VW0J8%U09(SF7.png

**图 2 Cache and TLB Behaviors of Grep**

**2.3 硬件加速现状：**

提高性能大体有两种方法，开发并行性和编译器优化。计算机系统结构设计一般会通过开发并行性来提高性能。有三种层次的并行性。

1) 指令级并行

在过去的几十年里，体系结构设计者主要通过指令级并行来提升体系结构性能，也取得了巨大的成功。指令级并行可以保持程序二进制兼容的前提下提高处理器性能。例如指令流水线、多发射、乱序执行概念的引进，并在此基础上不断的优化，使得系统性能大大的提升。但是，指令级并行的开发已经达到了顶峰，现在进一步挖掘指令级并行的空间已经不大。

2) 数据级并行

现在有很多应用程序拥有大量的数据级并行，例如科学计算中的矩阵运算、媒体图像和声音的处理。

数据级并行主要指单指令多数据流（SIMD）的向量结构。 20 世纪 70 年代，以 Cray 为代表的向量机十分流行，从 Cray-1、Cray-2，到后来的 Cray X-MP、Cray Y-MP。直到 Cray-4 后， SIMD 沉寂了一段时间，现在又开始越用越多。例如 X86 中的 SSE 多媒体指令可以用 128 位通路做 2 个 64 位运算或 4 个 32 位运算。 SIMD作为指令级并行的有效补充，在流媒体领域发挥着重要的作用。尤其在专用处理器中应用的比较多，一个典型代表就是 GPU (实际上就是多线程 SIMD 处理器)。通用处理器中 80% 的面积都被用做 Cache 及转移猜测等为运算提供指令和数据的逻辑，真正用于计算的部分不到 10%；而在 GPU 中，可能其中 50% 到 60% 的面积都被用来做运算，对特定算法效率很高。

类似 GPU 的专用加速器还有 FPGA、 CryptoManiac (一款针对Cryptographic Services的快速灵活的协处理器，主要解决加密和解密问题)、 DianNao 和 DaDianNao (针对机器学习领域的CNN和DNN算法) 等。他们都针对特定的算法或特定的应用场景，有非常好的加速效果。

3) 线程级并行

多线程处理器，支持多个线程以交错形式在单个多发射处理器上运行，并行粒度比较大，例如一个线程中包含几十条或几百条，甚至更多的指令。

由于单处理器性能的增长速度已达最高峰，而云计算与大数据也发展迅速，多处理器的重要性一直在不断的增加。这使我们进入计算机系统体系结构的一个新时代，多处理器在从低端到高端的各个领域都扮演了重要角色。多处理器指通过由单一处理器控制其他多个处理器的协调与使用，通过共享地址来共享存储器。很多多处理器也包括对多线程的支持。

**2.4 本章小结**

本章首先介绍了 Benchmark 的作用，并对国内 Benchmark 领域的相关研究做了简单的罗列。通过分析发现， Grep 是最基础的算法也是最典型的算法，大数据几乎所有领域都有 Grep 的应用场景，而且几乎所有的 Benchmark 都提取了 Grep 作为 Workload。除此之外，现在还未曾有针对 Grep 的加速结构，并且现有结构还有很大的优化空间。因而，面向大数据应用的字符串匹配算法加速结构研究势在必行。

计算机系统结构并行性设计有三个层次。指令级并行技术为计算机系统性能的提升做出了很大的贡献，但是已经到达了顶峰。数据级并行技术促使了很多专用加速器的产生，例如 GPU 等，在计算机系统中发挥重要的作用。线程级并行当前正快速发展，是计算机系统结构设计的焦点。由于进一步挖掘指令级并行的空间不大，所以我们将主要采用数据级并行以及线程级并行设计面向大数据应用的 Grep 的加速结构。

**第 3 章 GREP算法分析**

**3.1 GREP 算法简介**

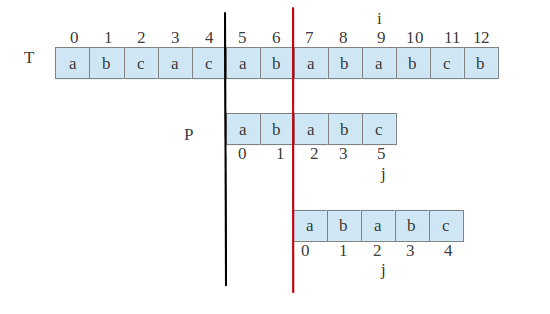
Grep是一个常用的文本处理工具，在大数据处理中也是必不可少的，用于正则表达式的字符串查找。与常用的Linux中的grep命令功能相同。Grep中核心的工作就是字符串匹配。比较经典的字符串匹配算法有KMP和Boyer-Moore算法。算法本身比较复杂，但是具体到底层操作，就是字符的读取和比较操作。

Boyer-Moore 算法比 KMP 算法快数倍，是更为优化的算法。传统字符串比较算法时间复杂度为 O(M \* N)（其中 M 为模式串长度， N 为目标串的长度），KMP算法的时间复杂度为 O(M + N)，而 Boyer-Moore 算法的时间复杂度为 O(N / (M + 1))。但是，KMP 算法与 Boyer-Moore 算法的核心操作是一样的，都是两个字符的比较。所以本文选用 KMP 算法作为代表进行论述。

**3.2 KMP 算法**

KMP 算法用于字符串模式匹配，目标串 T[0]......T[n - 1]，模式串 P[0]......P[m - 1]，这里 n>=m，i 代表 T 的索引指针，j 代表 P 的索引指针。传统字符串匹配算法，在 T[i] != P[j] 的时候，i 指针需要回退到 (i - j + 1) 的位置，同时 j 回退到 0，也就是模式串 P 开始的位置，所以传统算法的匹配过程的复杂度就是 O(m \* n)。其实在 T[i] != P[j] 的时候，i 指针不需要回退，T[i - j + 1]......T[i - 1] 和 P[0]......P[j - 1] 是相等的，可能只需要让 j 回退到中间的某个位置 k，使得T[0]......T[k - 1] 和 T[j - k]......T[j - 1]相等。这个过程相当于把已经匹配的P[0]......P[j - 1]找出最长的相同的前缀和后缀。

从图 3 可以看出，模式串 P，目标串 T，在 i = 9，j = 5 的位置字符不匹配，这时传统的做法是将 i 回退到这轮匹配开始的字符 i = 5的下一个字符，即 i = 6，同时将 j 回退到 0 的位置。但是我们发现已经匹配的部分 P[0]......P[3] “对称性”很强，可行的一种做法就是， i 不需要回退，将模式串 P 向前移动两位，将 j 回退到 j = 2 再尝试T[i]和P[j]的匹配。即在字符串匹配过程中，已匹配的子串是已知的，利用这个已知信息将搜索位置尽量向后移动，这样就提高了效率。为了实现这个做法，根据模式串可算出一个前缀数组，即 next 数组。



**图 3 KMP 算法示例图**

KMP算法的精髓就是 next 数组，每一个模式串有一个固定的next数组，它记录着字符串匹配过程中失配情况下可以向前多跳几个字符，当然它描述的也是子串的对称程度，程度越高，值越大，当然之前可能出现再匹配的机会就更大。Next[j] 表示 P[0]......P[j] 的"前缀"和"后缀"的最长的共有元素的长度。例如针对图 2 中的模式串计算 next[3]，因为 ”abab” 的前缀为 ”a”,”ab”,”aba”，后缀为”bab”,”ab”,”b”，前缀和后缀的最长共有元素的长度为2，所以 next[3] = 2。



**图 4 KMP 算法流程图**

图 4 为KMP 算法的流程图，首先需要对模式串进行提取，然后计算其跳转数组next[]。得到跳转数组之后，模式串与目标串逐个字符进行比较，当匹配失败时，根据 next 数组将模式串向后移动的相应的位数，继续进行匹配，直到匹配成功或者搜索结束为止。

**3.3 KMP 算法特征提取**

3.3.1 建立搜索字符的 next 数组：

对于模式串，比较其子串的内容，若有相同的前缀和后缀，则建立 next 数组，基本操作为 ldrb 指令、str 指令、cmp 指令。

3.3.2 目标串匹配：

在未完成目标串的遍历的前提下将模式串与目标串进行逐字符比较，若字符失配，则根据 next 数组向后跳转并进行下次比较。主要操作为 ldrb 指令，cmp 指令。



**图 5 KMP 算法操作tree示意图**

KMP算法操作tree示意图如图 5所示，由于一般情况下模式串长度 m 比较小，大数据领域目标串长度 n 非常大，求解 next 数组所占的比重很小，所以本文优化的重点在于字符的匹配过程，即主要针对 ldrb 操作以及 cmp 操作。

由以上分析内容，可总结出 KMP 算法的特征以及对体系结构的需求，如下表 2 和表 3 所示。

|  |  |
| --- | --- |
| **编号项** | **特征** |
| **计算特征** | KMP 算法主要是大数据量的字符比较，时间复杂度是O(m+n)。 |
| **访存特征** | 由于 cmp 操作都是针对字符的，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** | 目标串长度 m 较大时，可以切割目标串，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** | KMP 算法属存储密集型应用，有大量的存储器读操作。 |

**表 2 KMP 算法特征分析**

|  |  |
| --- | --- |
| **结构设计** | **具体需求** |
| **指令级** | KMP 算法为字符串匹配算法，主要包含访存 (ldrb) 指令、比较 (cmp) 指令以及简单的加减 (add/sub) 操作。ldrb 指令和 cmp 指令可以作为固定计算模式合并为复合指令进行执行。 |
| **协处理部件** | 对固定的“取数和对比”操作可以利用协处理部件对其进行大规模并行执行，加快处理速率。 |
| **内部高速总线** | KMP 算法可对目标数据进行并行搜索，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。核间通信并不多，内部总线主要需求为保证核的访存效率。 |
| **外部高速总线** | KMP 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。 |

**表 3 KMP 算法对体系结构的需求**

**3.4 本章小结**

本章首先简单地介绍了 GREP 算法，比较经典的字符串匹配算法有 KMP 和 Boyer-Moore 算法。Boyer-Moore 算法虽然比 KMP 算法快好几倍，但是他们的核心操作是相同的，都是单个字符的比较，本文选取 KMP 算法进行论述。简要介绍了 KMP 算法，并对其进行特征提取，通过分析，得到了 KMP 算法的特征以及对体系结构的需求。

由此，本文拟采用如下方式进行加速：

1. 并行处理，即采用多核多线程对目标串分任务进行处理。
2. 核心操作为字符比较，则会有固定模式”ldrb r0, addr”,”ldrb r1, addr1”,”cmp r0, r1”，考虑复合指令以及固定的运算部件。
3. 由于在匹配目标串时，数据的局部性很强，考虑增大 cache 的大小等方式对 cache 进行设计。
4. 由于数据量庞大，考虑增大 memory 存储空间。

**第 4 章 KMP 算法热点评估热点评估**

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。

**4.1 工具介绍**

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。gprof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。

1. 使用检测的方法，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；
2. 使用采样的方法，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。

注：由于统计各个函数所占用的时间采用的是采样的方法，所以当一个函数所占用的时间接近 100% 的时候，由 gprof 给出的此函数所占用的时间可能会超过 100%。

表 4 列出了 gprof 工具所输出的各项信息所代表的含义。

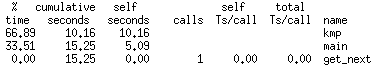
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| %time | Cumulative seconds | Self Seconds | Calls | Self TS/call | Total TS/call | name |
|
| 该函数消耗时间占程序所有时间百分比 | 程序的累积执行时间（只是包括 gprof 能够监控到的函数） | 该函数本身执行时间（所有被调用次数的合共时间） | 函数被调用次数 | 函数平均执行时间（不包括被调用时间，函数的单次执行时间） | 函数平均执行时间（包括被调用时间，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

**表 4 Gprof 产生的信息解释**

**4.2 评估方法**

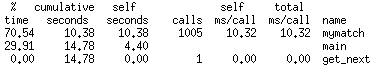
1. 测试集选择 1G，比 1G 还小的测试集运行时间太短，不利于函数时间统计，比 1G 大的测试集运行时间太长，不利于实验的展开。
2. 先评估源程序，找出比较耗时的函数，再将此函数中疑似为热点区域的代码抽出，以函数形式调用，然后再用 gprof 工具找出最耗时的函数，如果为刚刚抽出的函数，则验证成功，否则继续以此方法分析程序其他处。
3. 当找出热点区域，可增大测试集进一步验证。

**4.3 实验结果**



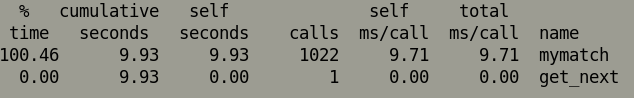
**图 6 1G 测试集，抽取热点前**

如图 6 所示，由抽取热点前的评估结果可以看出程序主要的运行时间集中在函数 kmp 上，即字符匹配过程，这不难理解。所以进一步缩小范围，将 kmp 函数中疑似最占时间的部分分离为 mymatch 函数，继续进行验证。



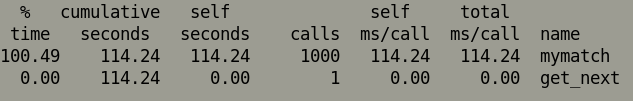
**图 7 1G 测试集，抽取热点后**

由图 7 可以看出，程序中最耗时的函数明显由 kmp 函数转移到 mymatch 函数，证明提取的热点区域是正确的，优化的重点区域就是mymatch 函数中的内容。同时也发现一个问题，main 函数占用的时间也不容忽视，测试集增大其占用的时间可能会进一步增大。为了排除这个隐患，在仔细分析源码后，发现目标串读入的方式不够优化，是在循环中用 fgetc() 的方式，将其修改为 fread() 后，即没有循环的情况下，再次验证，结果如图 8 所示，main 函数基本不占用时间，此时程序耗时基本全在 mymatch 函数上，这对我们的优化是十分有利的。



**图 8 1G 测试集，优化输入方式后**

增大测试集为 10G 进一步验证抽取的热点区域是否正确，结果如图 9 所示，依然是mymatch 函数占据了接近 100% 的时间。



**图 9 10G 测试集**

**4.4 本章小结**

本章内容先介绍了 x86 平台下的 gprof 工具，简略介绍了抽取热点的步骤，最后实验评估 kmp 算法的热点区域。结果和预想的一模一样，kmp 算法的热点区域就是字符逐个比较的过程，并且在用 1G 和 10G 的测试集验证时，此部分占用的时间十分接近 100%。这对本文的优化设计工作十分有利，这部分的字符比较是比较固定的计算模式，将这部分的计算大幅速度提升，则程序的整体性能也会有十分大的优化效果。

**第 5 章 加速结构设计**

**5.1 已有技术基础**

实验室已研发出千核万线程模拟器，又称千线程并行模拟器，已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。



**图 10 模块关系例图**

各个组件的功能、性能等已十分稳定，本文的加速结构设计的平台已十分完善，只需设计搭建一个能对 kmp 算法计算性能大幅提升的模拟器即可。通过统计信息能十分方便地得知程序运行总的 cycle 数，加速效果十分容易量化。

**5.2 加速方案设计**

**5.2.1 多线程**

由于目标串数据量巨大，所以采用并行处理，即采用多核多线程对目标串分任务进行处理。据了解，现在一些常用的处理器例如 intel 的 i3 4150、i7 4770K 都采用多核结构，每一个核支持双线程。所以本文采用拓扑 4 x 4 结构，即 16 核结构，每个核也支持双线程。

**5.2.2 复合指令以及新功能部件**

由第三章可知，kmp 算法核心操作为字符比较，且核心操作是固定的计算模式”ldrb r0, addr”,”ldrb r1, addr1”,”cmp r0, r1”，设计复合指令 LLCMP 完成上述计算，输入为两个地址 addr 和addr1，输出为两个字符的比较结果。并设计相应的功能部件完成这一操作。

......

**5.2.3 cache**

一般 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。

**5.2.4**

......还需要设计其他方案进一步加速。

**5.3 加速方案实现**

**5.3.1 多线程**

采用拓扑 4 x 4 结构，即16核结构，并在 core 组件中 ./core/src/conf.h 中将宏定义 MAX\_THREADS 的值设为 2 ，满载为 32 线程。

**5.3.2 cache**

增大 cache 的带宽，将 ./obj\_conf/cache\_conf.c 中将 cache 初始化矩阵 cache\_matrix[] 中倒数第四列数 nports 由 1 改为 2，这样即可使。

**5.3.2 复合指令以及新功能部件**

**5.3.2.1 指令定义方式**

指令定义在 ./core/src/machine.def 中，并将指令统一拆分出若干信息，如指令类型、opcode、源操作数、目的操作数、需要用到的功能部件种类等，然后所有指令用一个固定的格式来表示。在具体实现时，通过宏定义的方式来定义每条指令包含的信息以及具体的执行操作。

指令定义时将不同类型的指令转换成统一的格式，由宏 DEFINST 来定义。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：

|  |  |
| --- | --- |
| 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： |
| <fu\_req> | 用于表示指令所需用到的功能部件。可选枚举值包括： |
| <iflags> | 指令标志位，用于标识指令类型。可选值包括： |
| <output eps…> | 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： |
| <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 |

**表4 指令集定义模块参数说明表**

DEFINST 结构定义了指令的各种信息，同时，我们再为每条指令定义一个宏 （OP\_IMPL） 用于表示指令的具体功能。以定点寄存器加法操作为例，在这种方式下可定义为：

DEFINST(

ADD, 0x0, “add”, “d,s,t”, ALU, F\_ICOMP,

DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA

)

#define ADD\_IMPL

{

SET\_GPR(RD, GPR(RS)+GPR(RT))

}

指令定义之后，在程序中，通过以下方式即可方便地实现的实现指令的功能模拟：

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");

}

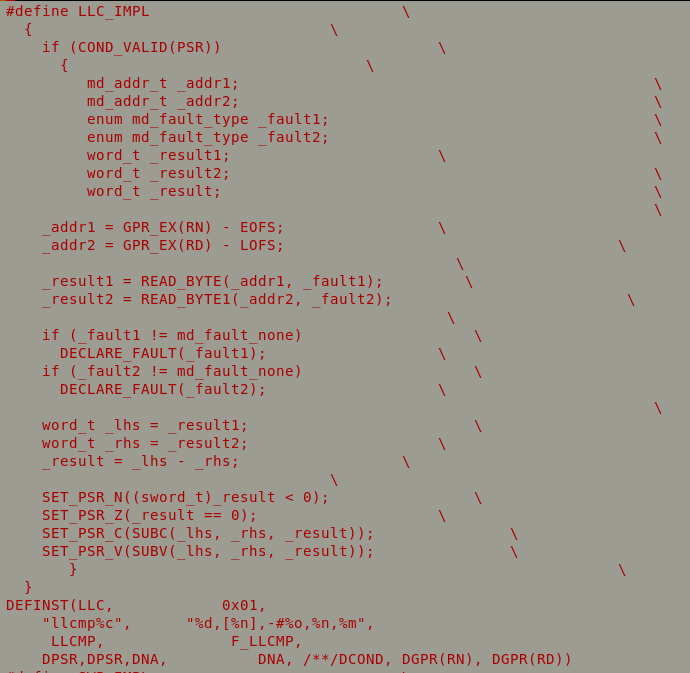
**5.3.2.2 复合指令定义**

复合指令的十六进制格式为：0xe1001010，此指令完成 ldrb、ldrb、cmp 的功能。二进制表示如下图所示：



当执行此指令时，会将 r0 和 r1 寄存器中的值取出作为地址，减去 OFFSET，然后ldrb 两个字符作比较，依据比较结果对相应标志位进行置位。

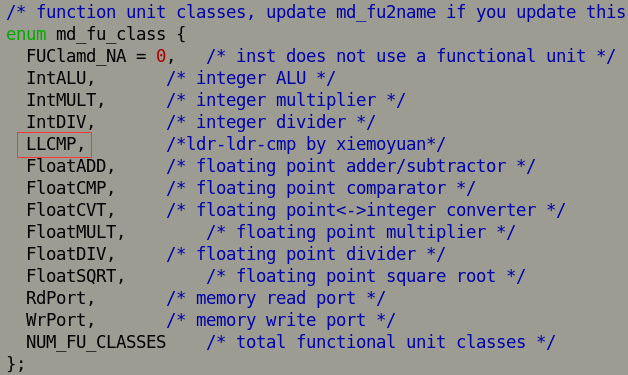
在 ./core/src/machine.def 中已定义好的译码树中根据译码规则，找到译码树一个空的子节点，插入新定义的复合指令。



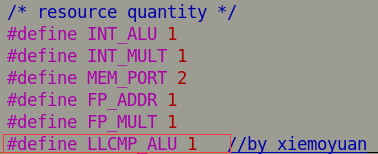
定义的指令名称为 LLC，但是在编写程序时不能直接用 LLC 指令，因为编译器无法识别此指令。所以我们使用嵌入式汇编的方式将指令以数据的形式嵌入到代码中，当程序执行到嵌入的复合指令处，取指后在译码阶段就能顺利进入译码树并执行相应的操作，所以我们只需将源代码中 “ldrb r0, [r0]\n”、”ldrb r1, [r1]\n”、”cmp r0, r1\n” 这三条指令换成 “.word 0xe1001010\n” 即可。

**5.3.2.3 新功能部件定义**

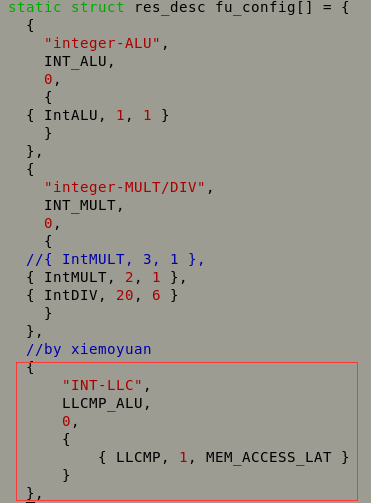
定义新功能部件 LLCMP 以执行 LLC 指令，在 ./core/src/machine.h 中定义如下：



在 ./core/src/conf.c 中定义功能部件 LLCMP 可使用的个数为 1。



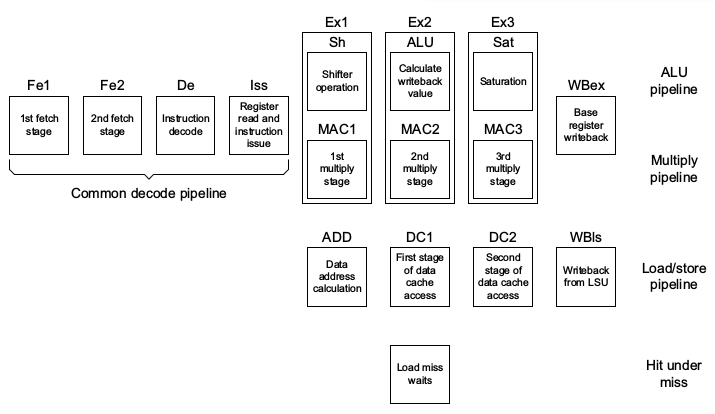
在 ./core/src/core.c 中定义功能部件 LLCMP 的初始化参数如下：



则新功能部件 LLCMP 基本定义完毕，再在相应的地方加上打印信息相关的内容，对新功能部件的使用情况进行统计。

**5.3.2.4 流水线修改**

流水线在模拟器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

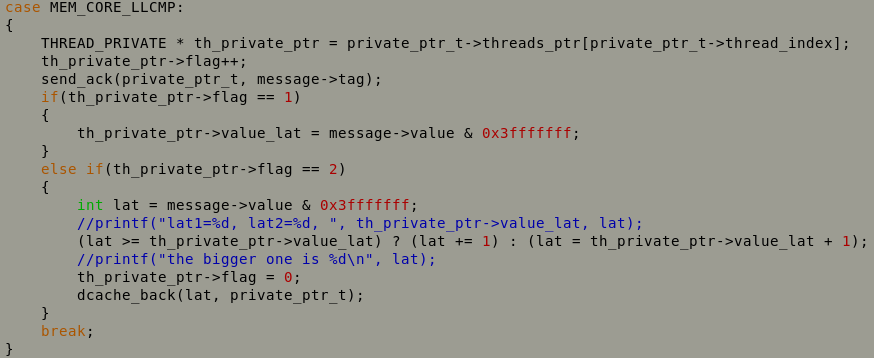
}

}

整个指令的功能在 ss\_decode 函数即译码阶段已经完成，接下来的流水线是为了模拟时序，得到程序运行的 cycle 数以及功耗等统计信息。

由于复合指令涉及访存操作，所以添加消息事件 CORE\_MEM\_LLCMP 和 MEM\_CORE\_LLCMP 分别表示 core 发给 memory 的 llcmp 消息和 memory 返回给 core 的 llcmp 消息。

在ss\_\_execute\_stage2里发出两条消息，消息类型是 CORE\_MEM\_LLCMP，这两条消息将 LLC 指令的两个源操作数即两个地址发给 memory 模拟ldrb 操作，在这之前要先判断 cache 是否命中，即消息经过 cache，若命中则返回消息给 core，得到访存延时；若 cache 不命中，则发消息给 memory 模拟 ldrb 操作，返回消息给 core，得到访存延时。 在 ./core/src/core.c 中对返回消息进行处理，得到复合指令的最终延时，代码如下：



**5.2.4**

......

**5.4 本章小结**

本章内容首先介绍了现阶段已有的技术基础，提出了加速方案以及加速方案的实现方法。下一章将介绍此加速结构的优化效果。

**第 6 章 加速结构实验评估**

**6.1 评估方法**

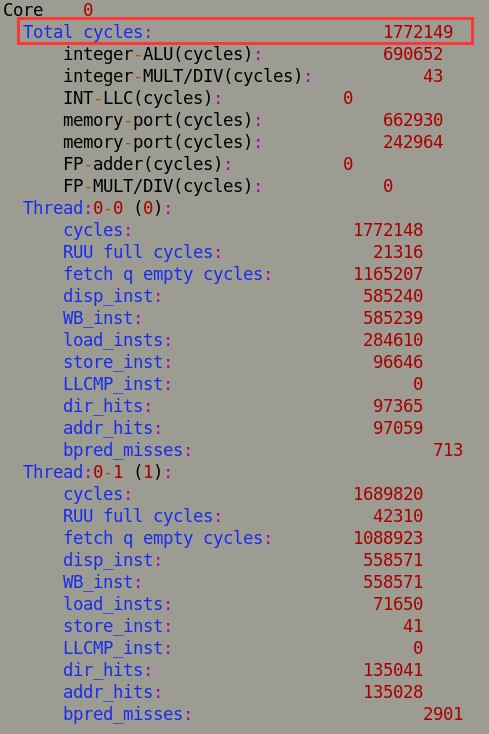
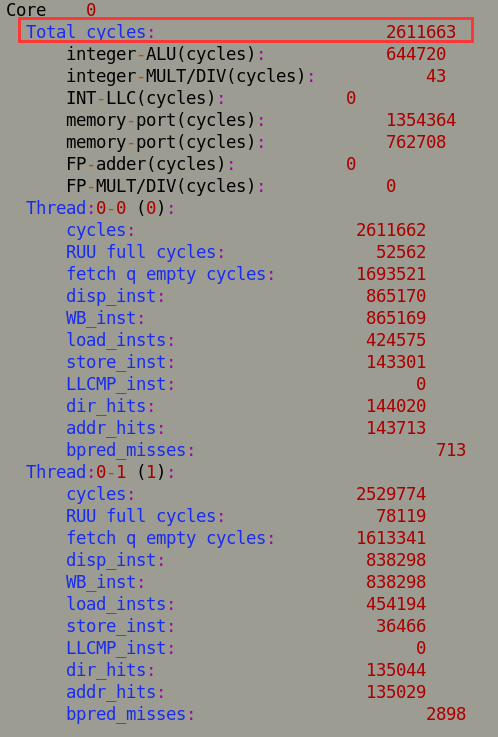
如果只嵌入一小段汇编代码，即只在字符比较的地方加入新指令，通过实验发现并没有加速效果，对照反汇编代码发现，编译器会在编译环节对嵌入的汇编代码前后作相应的处理，例如压栈出栈等操作，而此段代码又是循环比较字符的过程，对程序整体产生了很多多余的指令，所以程序没有加速效果反而变慢了。

为了避免这个问题，本文先将热点代码全部由汇编编写，先软优化，然后在此基础上再加上新指令进行评估。

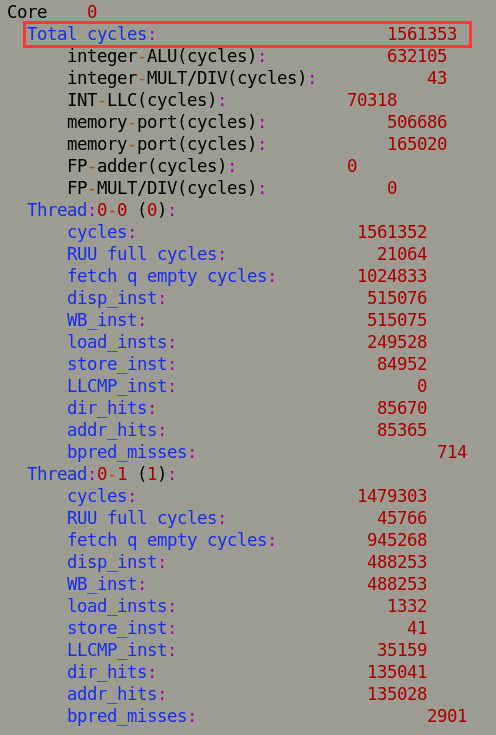
在选择测试集时，先以 1M 作为测试对象，再逐渐加大测试集到 10M、100M 进行测试。

**6.2 评估结果**

**6.2.1 1M 测试集测试结果**

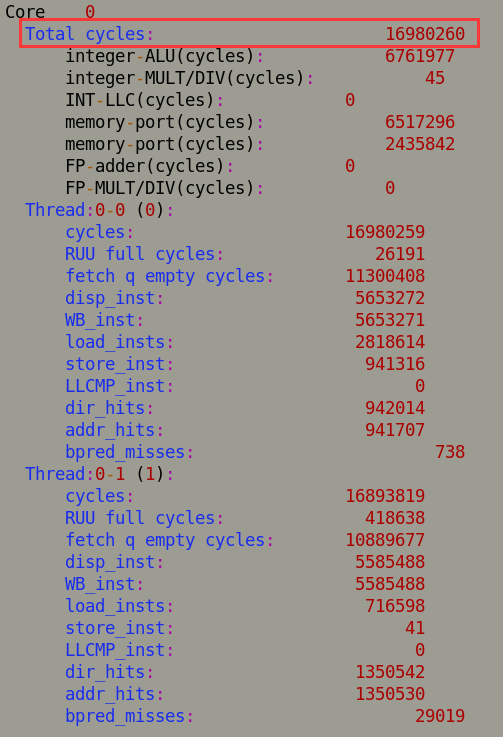
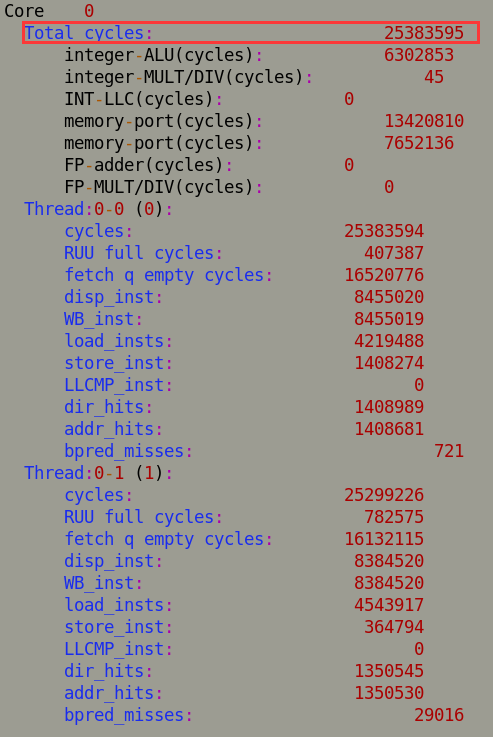


**图 11 C 语言版部分统计信息 图 12 汇编版部分统计信息**

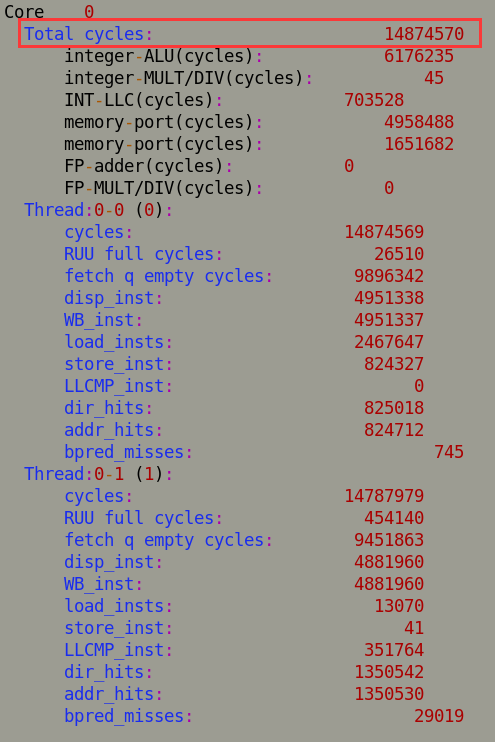


**图 13 加入加速结构后部分统计信息**

**6.2.2 10M 测试集测试结果**

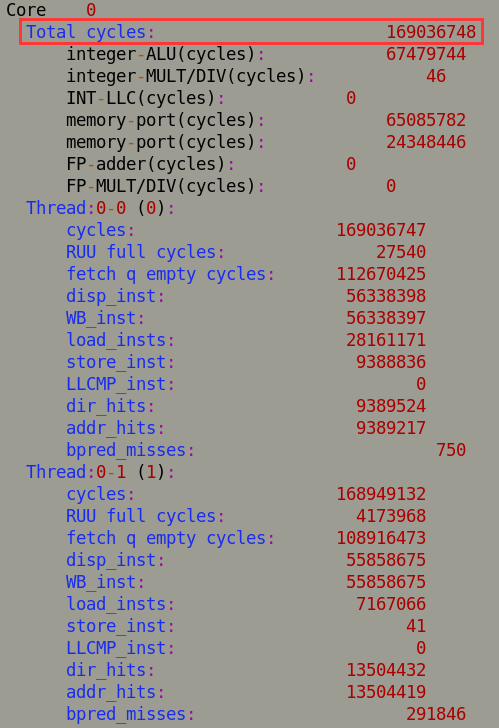
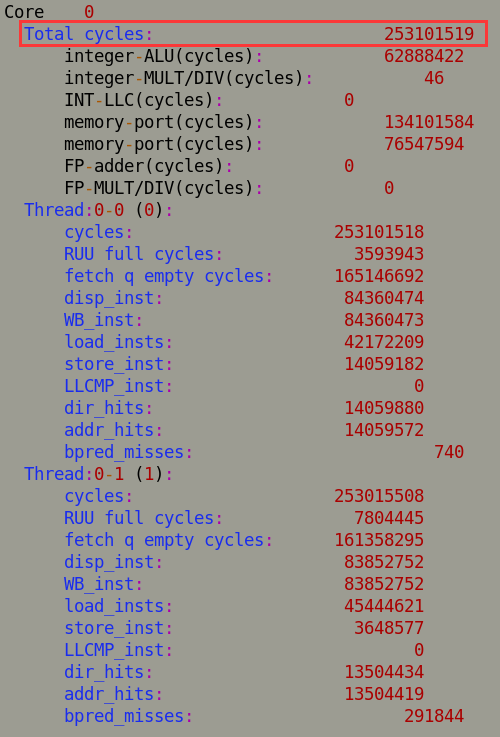
****

**图 14 C 语言版部分统计信息 图 15 汇编版部分统计信息**

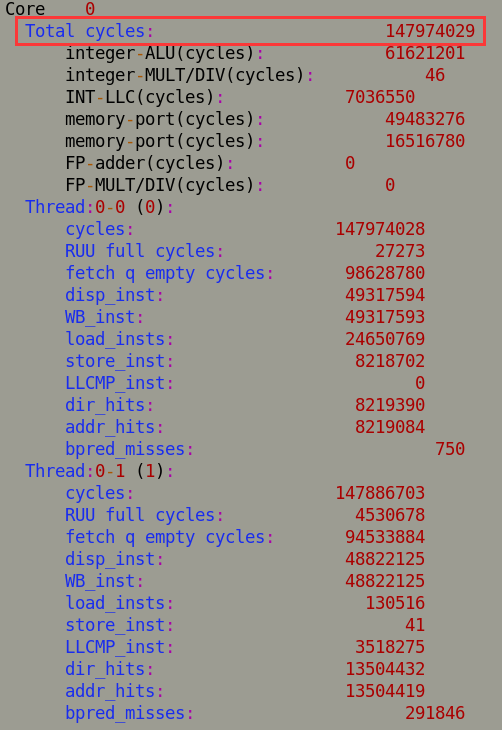
****

**图 16 加入加速结构后部分统计信息**

**6.2.3 100M 测试集测试结果**

****

**图 17 C 语言版部分统计信息 图 18 汇编版部分统计信息**

****

**图 19 加入加速结构后部分统计信息**

**6.2.4 实验结果分析**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1M测试集 | 10M测试集 | 100M测试集 |
| c语言版cycle数 | 2611663 | 25383595 | 253101519 |
| 汇编版cycle数 | 1772149 | 16980260 | 169036748 |
| 新加速结构cycle数 | 1561353 | 14874570 | 147974029 |
| 软优化结果 | 0.321448058 | 0.331053777 | 0.332138548 |
| 加速结构优化结果 | 0.118949366 | 0.124008113 | 0.124604379 |

**表 5 cycle 数统计以及优化效果**

软优化指汇编版比 C 语言版的优化，加速结构优化指新加速结构比汇编版的优化。现阶段软优化的幅度较大，但我们关注的重点为加速结构的优化，只有 10%+ 的优化效果，远远不够。

并且在实验能接受的数据集大小来看，增大数据集加速结果只有微幅的增长，虽然增长缓慢，但由此可做合理地推测，更大数据集的加速效果不会比小数据集差。

**6.3 本章小结**

本章内容首先介绍了测试方案，又将实验所得结果进行罗列，并将加速结果进行量化。现阶段优化效果仅 10%+，还需进一步优化。

**第 7 章 总结与展望**

**参考文献**