第34卷 第5期
2011年5月

计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS

Vol. 34 No. 5
May 2011

一种面向非规则引用的Cell多核处理器自适应Cache行策略

曹 倩 胡长军 张云星 朱于畋

(北京科技大学信息工程学院 北京 100083)

摘 要 非规则问题是大规模并行应用中普遍存在和影响程序效率的关键问题,软件Cache是Cell处理器上解决该问题的一种普遍手段.鉴于通常的软件Cache忽略了非规则引用的内存访问模式,将Cache行设定为一个固定的长度,而加重内存带宽负荷及制约Cache利用率的问题,文中提出了一种自适应的Cache行算法,它根据非规则内存访问的特点,在程序执行过程中不断地调整Cache行的大小,因此减少了传输的数据量.同时,针对不同的Cache行大小,设计了一种相应的软件Cache结构——混合行大小的Cache.它包含多种Tag项数组,每种Tag项数组对应于一种Cache行大小.该Cache设计是一种分级的结构,因为当长Cache行的Tag项数组缺失的时候直接进行缺失处理,而当短Cache行的Tag项数组发生缺失的时候启动缺失处理,同时检查长Cache行的Tag项数组是否命中,若命中,则终止缺失处理.通过对Tag项数组的分级查找,Cache的命中率有了显著的提高.除此之外,文中提出了一种新的行索引对齐的Cache替换策略,它能够在多种不同的Cache行大小并存的情况下实现LRU替换策略.实验表明该文提出的自适应的软件Cache行策略极大地减少了冗余的数据传输,提高了Cache的命中率.同时,与固定的1024B,512B,256B,128B的Cache行的性能相比,自适应的Cache行策略的执行速度分别提高了28.9%,29.7%,32.1%和33.5%.

关键词 非规则;混合;软件Cache;多核;编译优化

中图法分类号 TP311 **DOI号**: 10.3724/SP.J.1016.2011.00899

Adaptive Cache Line Strategy for Irregular References on Cell Architecture

CAO Qian HU Chang-Jun ZHANG Yun-Xing ZHU Yu-Tian

(*School of Information Engineering, University of Science and Technology Beijing, Beijing* 100083)

Abstract Software cache is a commonly used method which solves the irregular applications on Cell processor. Considering that software cache usually ignores the irregular reference memory access pattern and thus sets the cache line to a specific length, which elevates memory bandwidth overhead and limits cache utilization, this paper proposes an adaptive cache line strategy, which continuously adjusts cache line size during applications execution, therefore, the transferred data size is decreased significantly. Moreover, this paper presents a corresponding software cache—hybrid line size cache (HLSC). It introduces a hybrid Tag Entry Array, with each mapping to a different line size. It's a hierarchical design in that when a miss is occurred in the long line Tag Entry Array, misshandler is invoked at once. But if there is a miss in the short line Tag Entry Array, misshandler is invoked immediately as well the long line Tag Entry Array is checked. If it's a hit in the long line Tag Entry Array, misshandler is abandoned. The hit rate is efficiently increased because hierarchical lookups. Additionally, an original replacement policy — index

收稿日期:2009-11-07;最终修改稿收到日期:2011-03-10.本课题得到国家科技重大专项基金(2009ZX03004-004,2009ZX01045-005-002)、国家"八六三"高技术研究发展计划项目基金(2008AA01Z109,2006AA01Z105)、国家自然科学基金(60373008)和教育部科学技术研究重点项目(108008,106019)资助.**曹 倩**,女,1983年生,博士研究生,研究方向为并行计算与并行编译技术.E-mail: caoqian125@126.com. **胡长军**,男,1963年生,博士,教授,博士生导师,研究领域为并行计算与并行编译技术、并行软件工程、网格计算. **张云星**,男,1985年生,硕士,研究方向为并行计算与并行编译技术. **朱于畋**,男,1985年生,硕士,研究方向为并行计算与并行编译技术.

aligned strategy (IndAlign_LRU) is proposed to implement least recently unused replacement policy for multiple cache line sizes. Performance evaluation indicates that the adaptive cache line scheme greatly decreases the reduction of data transfer and improves hit rate. Additionally, average execution speed of the HLSC is faster than that of the cache line design with 1024B, 512B, 256B and 128B by 28.9%,29.7%,32.1% and 33.5%, respectively.

Keywords irregular; hybrid; software cache; multicore; compiling optimization

1 引 言

Cell 处理器是一款异构多核处理器,结构如图 1 所示. 它包含一个基于 Power 架构的主处理器 PPE (PowerPC Processor Element)和 8 个协同处理器 SPE(Synergistic Processor Element). PPE 是一个通用的双线程的处理器,包含两级硬件 Cache 结构. 每个 SPE 包含一个软件可控的 256KB 的本地存储,但不具有硬件 Cache. SPE 依赖直接内存访问(Direct Memory Access,DMA)操作来完成主存和本地存储之间的数据传输. PPE 和 8 个 SPE 通过互联总线 EIB 和主存储器以及 I/O 连接在一起. 内存接口控制器 MIC 提供了 EIB 和主存之间的接口.

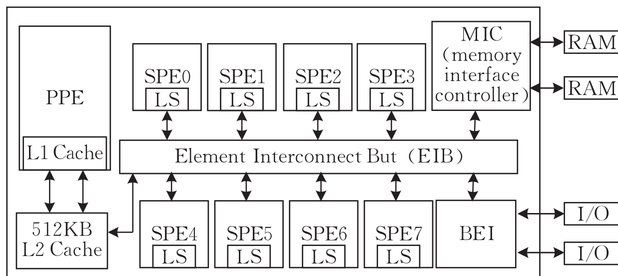

图 1 Cell 的体系结构图

非规则问题是指在编译时无法确定内存访问情况,且循环界限和数组引用下标不再是循环控制变量的线性表达式. 该类问题在程序中的一个主要体现是对间接数组的访问,如 $A[B[i]]$,它的最大特点就是内存访问的非连续性. 非规则引用问题在实际应用中普遍存在,如大规模油藏数值模拟生产系统(见图 2)、分子动力学(见图 3)和流体力学等. 非规则问题作为影响大规模科学计算效率的关键因素之一,其优化技术对于充分发挥异构多核系统的计算能力具有重要的理论和现实意义.

```
DO I=IBNONB, NBC
    DENBIO (I)=DNOILC (IKCB(I)) * 2.309
END DO
```

图 2 油藏数值模拟程序中的非规则计算程序片段

```
for (i=0; i<N; i++) {
    neighstart=neighindex[i];
    neighend=neighstart + neighlen[i];
    for (k=neighstart; k<neighend; k++) {
        j=neighlist[k];
        ...
        rho[i]+=dfn_t1;
        rho[j]+=dfn_t2;
    }
}
```

图 3 分子动力学中计算电子密度的程序片段

SPE 不能直接访问系统内存,而需要通过显式的 DMA 操作来完成主存和本地存储之间的数据传输. 而每次 SPE 对系统存储器中变量的访问,如果都简单地通过插入 DMA 传输指令进行,那么 SPE 将在等待总线数据时空转,数据总线也将在 SPE 处理数据时空等,SPE 的利用效率将非常低. 如果将这些数据分块传输并存放在本地存储中进行重复利用,将有利于提高程序性能. 所以,对于非规则的内存访问的情况,运行时系统采用有效的软件 Cache 技术来模拟硬件 Cache,便可以重用本地存储中临时的镜像数据,减少不必要的 DMA 操作. 因此软件实现的 Cache 成为 Cell 处理器上解决非规则问题时的通用手段. 但现有的软件 Cache 方案通常忽略了非规则内存访问的特点而将 Cache 行设定为一固定长度,从而加重了内存带宽负荷,制约了 Cache 的利用率.

本文将提出一种自适应的 Cache 行策略,它根据非规则引用的特点在程序运行过程中连续地、自适应地调整 Cache 行的长度. 该方案首先收集非规则引用访问到的内存地址,如果有任意两个或多个地址映射到同一条长 Cache 行的不同短 Cache 行中,则这样的地址被记作长 Cache 行地址,其它地址则被记作短 Cache 行的地址.

当地址被划分之后,需要查找 Tag 项数组(Tag Entry Array)以判断命中或缺失. 因此,我们设计了一种在 Cell 多核处理器上实现的软件 Cache 结构——混合行大小的 Cache (Hybrid Line Size

Cache，HLSC）. 该设计基于传统的四路组相联 Cache，它包含了多种 Tag 项数组（方便起见，本文中给的图仅包含了两种 Cache 行），每一种 Tag 项数组对应一种 Cache 行大小. 该 Cache 设计是分级的，表现为当长 Cache 行的 Tag 项数组缺失的时候直接进行缺失处理，而当短 Cache 行的 Tag 项数组发生缺失的时候启动缺失处理，同时检查长 Cache 行的 Tag 项数组是否命中，若命中，则立即终止缺失处理. 经过分级地查找 Tag 项数组，Cache 的命中率有了明显的提高.

为了在多种 Cache 行下解决 Cache 替换问题，本文对最近最久未使用（Least Recently Unused，LRU）Cache 替换策略进行扩展，提出一种新的行索引对齐的 LRU 策略——IndAlign_LRU 策略. 它引入一个链表数组，其中的每个链表同时对应于长 Cache 行的 Tag 项数组的一个组和短 Cache 行的 Tag 项数组的多个组. 链表结点的数据域存储了 Cache 行的索引，通过移动结点来实现多种不同 Cache 行的 LRU 替换策略.

为了评估自适应的 Cache 行策略的性能，我们在 Cell 处理器上实现了 HLSC 方案，并与传统的固定 Cache 行方案以及扩展的组索引 Cache（Extended Set-Index Cache，ESC）策略[1] 进行了比较. 实验结果表明 HLSC 策略明显地提高了 Cache 命中率，极大地减少了传输的字节总数. 与扩展的组索引 Cache 中自适应的 Cache 行技术相比，加速比可达到 1.52.

本文第 2 节介绍相关工作；第 3 节详细提出自适应的 Cache 行算法；HLSC 的结构在第 4 节作具体介绍；第 5 节对 HLSC 的操作模型进行详细描述；第 6 节应用多种测试用例评估自适应的 Cache 行策略的性能；最后为文章的结论及下一步工作.

2 相关工作及问题分析

Cell 多核处理器问世之前，关于软件 Cache 的研究已经有了一些成果[2-7].

Miller 等人[2] 提出了一种软件 Cache 设计，该方案自动地维护部分或全部的 scratchpad 内存作为 Cache. 但其只对指令进行缓存，而本文提到的应用软件 Cache 来解决非规则问题主要是对数据进行缓存. Moritz 等人[3-4] 设计的 Hotpage/FlexCache 需要相当精确的编译器指针分析以标记出可以重用某些页的内存引用，实现起来相当复杂.

Udayakumaran 等人[5] 提出的软件 Cache 方案通过代码 profiling 技术以及内存访问的频率进行代码区域分析，根据获得的信息将每个基本块赋予一个时间戳，并在程序的特定位置插入控制代码以实现系统内存与 Cache 之间进行数据传输. 这种方法对静态的内存访问很有效，但是当涉及到基于指针的内存访问时该方案就失效了.

文献[6-7]中软件 Cache 的设计主要是为了在嵌入式系统中减少能耗. Witchel 等人[6] 提出了 Direct Addressed Caches，它依赖硬件记住 Cache 行的准确位置，因此消除了 Tag 查找的开销. 但是该方案需要定义新的寄存器将 load/store 操作跟具体的 Cache 行联系起来，同时还需要提供新的 load/store 命令才能充分发挥寄存器的作用. Fryman 等人[7] 提出了软件 Cache——SoftCache，它基于对二进位的重写，需要对二进制图进行完整的数据流和控制流分析，因此实现起来很复杂.

具体到 Cell 多核处理器，由于 SPE 本身没有硬件实现的 Cache，软件 Cache[8-12] 已经成为近几年的一个研究热点.

Eichenberger 等人[8] 提出了一个在 Cell 处理器上用软件实现的传统的四路组相联的 Cache 设计. 该设计采用 LRU 的 Cache 替换策略，以单指令多数据（SIMD）的方式在组内查找是否有匹配.

Balart 等人[9] 设计了一种面向 Cell 处理器的 Cache，它使得用户可以确定一个代码区域，从而避免了 Cache 冲突，用户还可以在该区域中对多次访问 lookup 操作以及 misshandler 操作进行重排，从而有效地将计算与通信重叠起来. 该方案对于 Cache 访问时间局部性很高的特定循环效率很好，但是对一般的循环来讲，由于 lookup 过程需要对地址执行模操作，从而确定对多个链表中的哪一个进行遍历，因此实现的开销很大.

文献[10-12]中提到了在 Cell 处理器上实现一种混合的 Cache 结构. 该方案首先将内存访问模式划分为两类——高局部性的内存访问和非规则的内存访问，然后提出一种包含两种 Cache 结构的软件 Cache 设计，不同的 Cache 结构采用不同的 Cache 替换策略. 该方案需要精确的内存访问的分类，不但引入了额外的内存开销，实现起来比较复杂. 此外，它侧重于区分规则和非规则的内存访问，而忽视了非规则内存访问本身的数据局部性.

以上软件 Cache 实现的共同点是忽略了非规则内存访问自身所表现的数据局部性，而将 Cache 行

设置成一个固定的长度，因此增加了冗余的数据传输，加重了内存访问带宽的负荷，从而影响了非规则应用的整体性能. 显然，采用自适应的调节 Cache 行大小的策略，会极大地提高 SPE 上本地存储的利用率，减少冗余数据的传输.

目前已经有一些基于硬件实现的自适应的 Cache 行策略[13-15]，但由于 SPE 本身不具备硬件 Cache，因此需要研究 Cell 上的软件管理的自适应 Cache 行算法. 然而这方面的研究成果却很少，典型的工作是 Seo 等人[1]提出的扩展的组索引 Cache (Extended Set-Index Cache, ESC) 中的自适应的执行算法，它能够在 Cell 处理器上自适应地调整软件 Cache 行长度. ESC 的自适应调整 Cache 行的策略作用对象为一个循环，其基本思想是假设该循环在程序运行过程中被多次调用，在前几次调用中分别选用不同的 Cache 行执行该循环并测得执行时间，从而选出一个最优的 Cache 行长度. 而当该循环被再次调用时，便选择该最优的 Cache 行进行执行. 该自适应的算法存在以下不足：

(1) 该算法作用的对象为一个循环，而对于循环内的单个迭代区域不起作用. 因此，当某个循环其迭代次数很多，并且不同迭代区间表现出不同的内存访问的局部性时，该策略就不够准确. 而本文的自适应 Cache 行策略作用对象为循环内的每个迭代区间，自适应的选择过程更为准确；

(2) 为了得到某一循环最优的 Cache 行，ESC 的自适应的 Cache 行算法需要计算该循环被调用的前几次的性能，特别是当候选的 Cache 行种类比较多时，性能比较可能会引入更大的开销；

(3) 该算法只对循环在应用中被多次调用的情况起作用，而当某个循环在程序执行过程中只被调用了一次或很少的几次时，该自适应的算法就失效了. 而本文提出的自适应 Cache 行策略对循环被调用的次数没有要求.

3 自适应的 Cache 行策略

本节给出自适应的 Cache 行算法，根据非规则引用的特点连续地、自适应地调整 Cache 行的长度. 它首先将非规则引用访问的内存地址收集起来，并存放在一个临时数组 ea 中. 之后将这些地址对齐到长 Cache 行的边界，并将对齐后的地址存放在另一个数组 work_ea 中. 如果 work_ea 中有相同的元素，则说明它们的内存地址是能够在长 Cache 行的边界对齐的，并且如果该内存地址映射到不同的短 Cache 行地址中，则这样的地址就可以合并为一个长 Cache 行地址.

为了便于描述，本文从 NAS benchmarks[16] 的 CG 中提取出了一个循环，将其边界标准化，如图 4(a) 所示. 该自适应的算法针对非规则引用 p[colidx[k]] 进行最优的 Cache 行选择，为了便于描述，该算法提供了两种可选的 Cache 行，其过程如图 4(b) 所示.

```
for(k=0; k<ub; k++){
    sum+=a[k]*p[colidx[k]];
}
```

(a) 简化的原始代码

```
/*第 1 步:初始化*/
lb_tmp=0;
ub_tmp=0;
Longline=256;      //长 Cache 行
Shortline=128;     //短 Cache 行
do{
   /*第 2 步:动态地确定迭代区间*/
   ub_tmp=collect_dynamic(lb_tmp, ub_tmp);
   /*第 3 步:自适应的 Cache 行选择过程*/
   /*将收集到的地址存储到数组 ea 中*/
   for(k=lb_tmp; k<ub_tmp; k++)
       ea[k]=&p[colidx[k]];
   /*将收集到的地址对齐到 256B 的边界并存储到数组
     work_ea 中*/
   for(k=lb_tmp; k<ub_tmp; k++)
       work_ea[k]=ea[k]&(~(Longline-1));
   /*在数组 work_ea 中查找相同的值,比如,work_ea[i]=
     work_ea[j]*/
   search_same_data(work_ea, i, j);
   if((ea[i]&(~(Shortline-1)))!=
     (ea[j]&(~(Shortline-1))))
       i, j 被记作长 Cache 行地址的索引;
   其它下标被记作短 Cache 行地址的索引;
   /*第 4 步:迭代内的计算循环*/
   for(k=lb_tmp; k <ub_tmp; k++)
       if (k 为长 Cache 行地址的索引)
          sum+=a[k]*长 Cache 行对应的操作(ea[k]);
       else
          sum+=a[k]*短 Cache 行对应的操作(ea[k]);
   /*返回值 ub_tmp 作为下次循环的 lb_tmp*/
   lb_tmp=ub_tmp;
}while(lb_tmp <ub)
```

(b) 自适应的算法实现

图 4 自适应的 Cache 行算法的实例

该过程大致分为以下步：

第 1 步为初始化. 简单起见，假设有两种可选的 Cache 行，128B 和 256B.

第 2 步为确定迭代区间. 迭代区间指连续两次组冲突之间的迭代上下界. 本文提出的自适应算法作用的对象是循环内一个个迭代区间，即对每个迭代区间内的非规则访存地址进行长、短 Cache 行地址划分，因此迭代区间是一个非常重要的因素. 一方面，迭代区间不能太小，因为这些迭代要分摊后面地址收集、地址划分及计算的开销. 另一方面，迭代区

间也不能太大，因为有可能导致物理地址上很临近但访问时间差距很大的两个短 Cache 行的地址错误地合并为一个长 Cache 行的地址. 因此，该算法提出了一个动态迭代区间，它规定顺序地收集非规则引用访问的内存地址直到发生组冲突的时候停止.

为了确定何时发生组冲突，对短 Cache 行的 Tag 项数组的每个组设定一个计数器，初始化为 0. 当一个地址映射到某个组时，该组的计数器加 1，并判断该计数器是否到达 4(因为是四路组相联)，若没有到达，则继续判断下一次迭代的非规则访存的地址映射情况. 直到到达第 ub_tmp 次迭代时，非规则访存地址所映射到的组其计数器达到了 4，即发生了第一次组冲突. 此时，各个组的计数器均置为 0，该次迭代区间的上界为 ub_tmp，并将其作为下次迭代区间的下界. 为了方便起见，动态确定组冲突的过程在代码中用函数 collect_dynamic(lb_tmp，ub_tmp)表示，其返回值为停止收集地址时即发生组冲突时的迭代，该返回值将作为下次地址收集时的lb_tmp，即下次地址收集时的起始迭代.

第 3 步对本次迭代区间内的地址进行长、短 Cache 行地址的划分. 首先，将本次收集到的非规则引用的地址存放在一个临时数组 ea 中. 经过 ea[k]&(～(Longline－1))操作，ea 中的每个元素即每个地址都对齐到 256B 的边界上. 如果 ea 中的两个元素 ea[i]和 ea[j]经过 256B 边界对齐后的值相等，并且经过 128B 边界对齐后的值不等，那么这两个地址就可以合并为一个长 Cache 行对应的地址，下标 i，j 被记作长 Cache 行地址的索引. 相应地，ea 数组中不满足这样条件的地址则被记作短 Cache 行地址.

第 4 步为本地迭代内的计算. 根据第 3 步确定的下标是否属于长 Cache 行索引地址，在相应的 Cache 的 Tag 项数组中进行查找并进行相应的命中或缺失处理操作，具体的操作见第 5 节.

为了更清楚地说明自适应的 Cache 行策略，图 5 给出了一个实例. 作如下假设：

(1) 在某个迭代区间内，有 5 个有效地址 a_1 到 a_5 位于内存区间[256N，256(N＋2))内，其中，N 为一正整数；

(2) 位于内存区间[256N，256(N＋1))的数据被记作 L_i，前 128 字节的数据表示为 L_{i-1}，后 128 字节的数据表示为 L_{i-2}；L_{i-1} 和 L_{i-2} 互为“邻接行”；

(3) 位于内存区间[256(N＋1)，256(N＋2))的数据被记作 L_j，前 128 字节的数据表示为 L_{j-1}，后 128 字节的数据表示为 L_{j-2}；L_{j-1} 和 L_{j-2} 互为“邻接行”.

初始状态下，该迭代区间内所有的地址都记作短 Cache 行地址，如图 5(a). 地址 a_1 和 a_2 分别映射到两个邻接行 L_{i-1} 和 L_{i-2} 中，在响应其中率先到达的一个内存请求时，如果从系统内存中取到一个短 Cache 行，则在响应晚到达的内存请求时仍然需要再次访问系统内存取得下一个短的 Cache 行，即需要两次 DMA 操作分别从系统内存将 L_{i-1} 和 L_{i-2} 取到 Cache 中. 为了降低多次内存访问的开销，本文的自适应 Cache 行策略将两个短 Cache 行地址合并为一个长 Cache 行地址，即将两个短 Cache 行合并为一个长 Cache 行，显然响应 a_1 和 a_2 的内存请求时，只需要进行一次 DMA 操作，将长 Cache 行 L_i 一次取到 Cache 中. 同理，地址 a_3 也被记作长 Cache 行地址，见图 5(b).

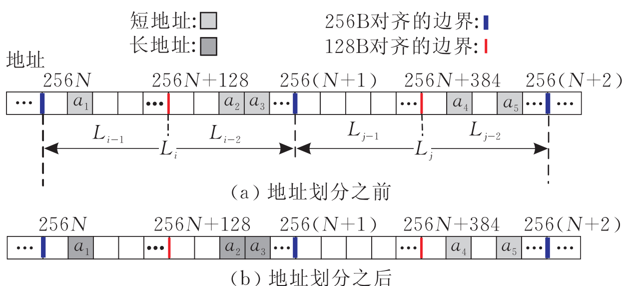

图 5　自适应 Cache 行算法图示

地址 a_4 和 a_5 均被映射到 L_{j-2} 中，而没有地址映射到其邻接行 L_{j-1} 中，所以地址 a_4 和 a_5 均为记作短 Cache 行地址.

总之，该自适应的 Cache 行策略可以作用于循环内部的迭代区间，对于不同迭代区间中最优 Cache 行发生变化的循环优势更为明显. 同时，与 ESC 中自适应方案不同，该策略对循环被调用的次数没有要求.

4　混合行大小的 Cache 结构设计

经过自适应的 Cache 行算法之后，收集到的非规则引用的地址被划分为长 Cache 行地址和短 Cache 行地址. 因此，本文设计了一种新型的软件 Cache 结构，混合行大小的 Cache(Hybrid Line Size Cache，HLSC). 它基于传统的四路组相联 Cache，但是它引入了多个 Tag 项数组(简单起见，这里以两个的为例). 其具体结构如图 6 所示.

(1) *Cache_Storage*(Cache 存储)用来存储应用数据；

(2) $Cache_Parameter_1$ 用来记录长 Cache 行相

应的参数；

(3) $Tag_Entry_Array_1$ 为长 Cache 行的 Tag 项数组；

(4) $Cache_Parameter_2$ 用来记录短 Cache 行相应的参数；

(5) $Tag_Entry_Array_2$ 为短 Cache 行的 Tag 项数组；

(6) $Index_Link_Array$ 用来存储 Cache 行的索引以实现多种 Cache 行的 LRU 替换策略；

(7) V_1 和 V_2 数组用来记录有效位；

(8) D_1 和 D_2 数组用以记录脏字节.

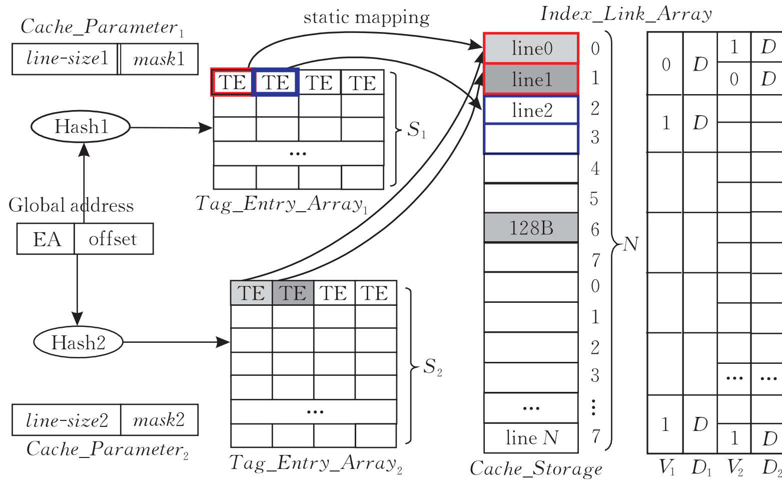

图 6　混合行大小的 Cache 的结构

$Cache_Storage$(Cache 存储)用来存储应用数据，本文将其设置为 64KB.

$Cache_Parameter_1$ 用来记录长 Cache 行相应的参数，它包含长 Cache 行的大小以及掩码 $mask_1$. 本节将长 Cache 行设置为 256B，所以长 Cache 行的数目为 $n_L=256(64KB/256B)$.

$Tag_Entry_Array_1$ 为长 Cache 行的 lookup 表，它包含了 S_1($S_1=n_L/4=64$)个组，每一个 TE 映射到一个 256B 的长 Cache 行.

$Cache_Parameter_2$ 用来记录短 Cache 行相应的参数，短 Cache 行设置为 128B，则短 Cache 行的数目 n_S 等于 512(64KB/128B).

$Tag_Entry_Array_2$ 为短 Cache 行的 lookup 表，每一个 TE 映射到一个 128B 的短 Cache 行.

$Cache_Storage$ 中的每一个基本的 Cache 行有个索引(index)，由于长 Cache 行的 Tag 项数组的一个组对应 Cache 存储中的 8 个基本的 Cache 行，所以将 Cache 存储中的基本的 Cache 行的索引值依次设为 0，1，…，7，0，1，…，7. 为了在多种 Cache 行中实现 LRU 的 Cache 替换策略，本文对传统的 LRU 替换策略进行扩展，提出了行索引对齐的 LRU 策略——IndAlign_LRU 策略. 它引入了索引链表数组($Index_Link_Array$)，图 7 给出了它的初始信息以及它与 Cache 行的映射关系.

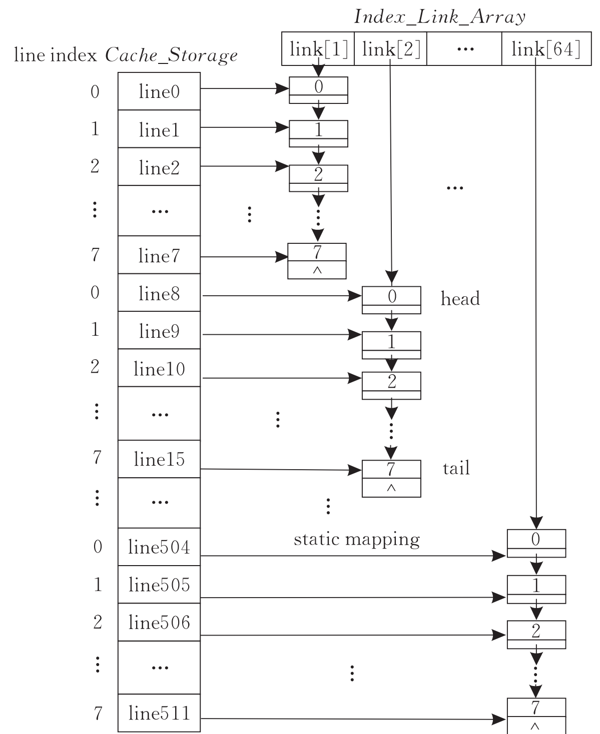

图 7　初始的索引链表数组及其与 Cache 行的映射关系

$Index_Link_Array$(索引链表数组)是一个由 Cache 行的索引组成的链表数组，该数组包含了 S_1 个链表，每个链表同时映射到 $Tag_Entry_Array_1$ 的一个组和 $Tag_Entry_Array_2$ 的两个组. 每个链表包含了 8 个结点，每个结点的数据域存储行索引

值.每个链表从头结点到尾结点其数据域依次存储了从最早访问到最新访问的 Cache 行的索引值，通过将对应链表中的结点移动到链头(head)或者链尾(tail)即可以记录基本 Cache 行的活跃信息.显然行索引对齐的 LRU 替换策略既可以在单一的 Cache 行又可以在多种 Cache 行下实现 LRU 替换策略.

V_1 和 V_2 数组用来记录 Cache 行的数据是否有效.当赋值为 1 和 0 时，分别表示有效和无效.

D_1 和 D_2 数组用来记录 Cache 行中的脏字节，以减少 Cache 行被剔出去时传输的数据量.

该 Cache 设计是一种分级的结构.为了说明该问题，图 8 给出了长、短 Cache 行地址的组掩码(*SetMask*).因为长、短 Cache 行的 Tag 项数组的组数均为 2 的幂，所以在计算组号(*SetID*)时可以用简单的位操作代替复杂的模运算.即

$$SetID=(ea\ \&\ SetMask)\gg N_bit,$$

其中，2^{N_bit} 等于相应 Cache 行的字节数.因为短 Cache 行的 Tag 项数组包含了 128 个组，所以需要 7 位(图 8 所示的第 7～13 位，用灰色表示)来确定组号.同理，长 Cache 行的 Tag 项数组需要 6 位(第 8～13 位)来确定组号.当 Cache 接收到一个有效地址，将其与短 Cache 行的组掩码进行按位与操作，如果在确定的组内发生缺失的话，可以将它与长 Cache 行的组掩码进行按位与操作，因为确定组号的位少一位，则该地址有可能在长 Cache 行的 Tag 项数组中命中.

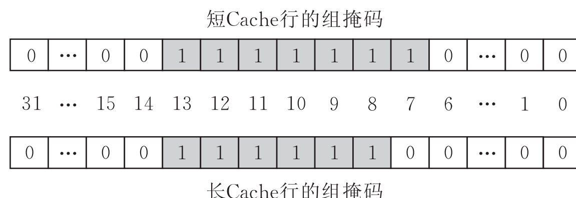

图 8　长、短 Cache 行的组掩码

5　混合行大小 Cache 的操作模型

该部分介绍 HLSC 的操作模型，主要包含 *Lookup_long*，*Lookup_short*，*IndAlign_LRU_long*，*IndAlign_LRU_short*，*Misshandler_long* 以及 *Misshandler_short*.图 9 给出了一个简单的操作流程图，下面来分别对这些操作加以介绍.

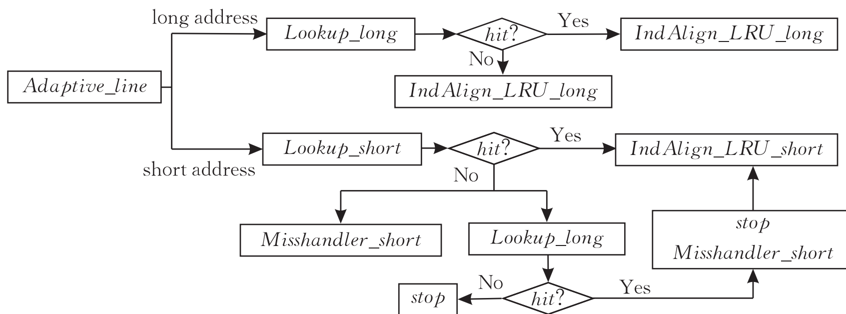

图 9　简单的 HLSC 的操作流程图

5.1　*Lookup_long* 和 *Lookup_short*

当 Cache 接收到一个有效地址为 ea 的内存请求时，根据自适应的 Cache 行算法，查询相应的 Tag 项数组.对长 Cache 行和短 Cache 行的查询过程跟传统的四路组相联的过程相似，分别叫做 *Lookup_long* 和 *Lookup_short*.

5.2　*IndAlign_LRU_long* 和 *IndAlign_LRU_short*

当长 Cache 行地址命中时，*IndAlign_LRU_long* 被调用；而当短 Cache 行地址命中时，*IndAlign_LRU_short* 被调用.

假设 SPE 接收到一个长 Cache 行地址，经过 *Lookup_long* 之后该地址命中，匹配的组和路分别为 *set_L* 和 *hit_index_L*，*IndAlign_LRU_long* 操作就是将第 *set_L* 个链表中结点数据域的值为 $(2*hit_index_L)$ 和 $(2*hit_index_L+1)$ 的结点移到链尾，即标记为最新访问的 Cache 行.图 10(a) 给出当匹配的路 *hit_index_L* 等于 0 时的一个实例.

假定随后 Cache 接收到一个短 Cache 行的地址，经过 *Lookup_short* 之后该地址命中，命中的组和路分别为 *set_S* 和 *hit_index_S*，当满足 $set_L*2=set_S$ 或者 $set_L*2+1=set_S$ 的时候，就表示 $Tag_Entry_Array_1$ 中的第 *set_L* 组和 $Tag_Entry_Array_2$ 中的第 *set_S* 组同时映射到第 *set_L* 个索引链表.

当前者满足的时候，表示短 Cache 地址命中的是 $Tag_Entry_Array_2$ 中的第偶数个组，根据 TE 与 Cache 行的映射关系，第 *set_L* 个索引链表中数据域的值为 *hit_index_S* 的结点移到链尾，记为最新被访问到的 Cache 行.图 10(b) 给出了当 *hit_index_S* 等于 3 的时候该条件被满足时索引链表的操作

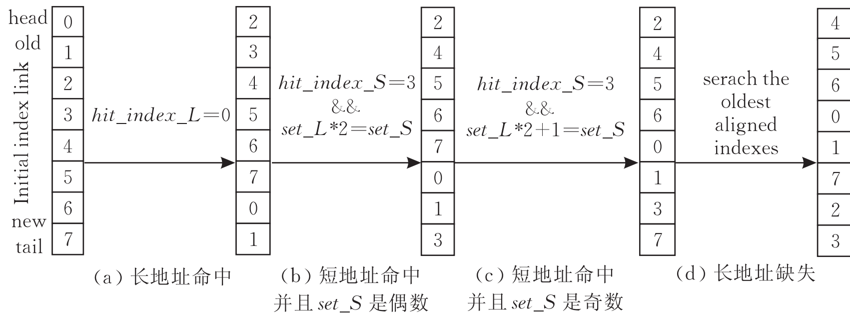

图 10 HLSC 操作的具体实例

情况.

当后者满足的时候,表示短 Cache 地址命中的是 $Tag_Entry_Array_2$ 中的第奇数个组,第 set_L 个索引链表中数据域的值为(hit_index_S+4)的结点移到链尾,记作最新被访问的 Cache 行. 图 10(c)给出了当 hit_index_S 等于 3 的时候该条件被满足时索引链表的操作情况. 总之,在短 Cache 行地址命中的情况下,对链表的操作以实现 LRU 的替换策略都称为 *IndAlign_LRU_short*.

5.3 *Misshandler_long*

当一个长 Cache 行地址缺失的时候,调用 *Misshandler_long*. 用 *IndAlign*(*index_L*)表示当链表的链头结点的数据域的值为 *index_L* 的时候,最早访问的长 Cache 行的对齐的行索引值. 为了便于描述,简单地定义 *IndAlign*(*index_L*)如下:

当 *index_L* 为偶数的时候,

$$IndAlign\ (index_L) = index_L \text{ 和 } (index_L+1) \quad (1)$$

当 *index_L* 为奇数的时候,

$$IndAlign\ (index_L) = (index_L-1) \text{ 和 } index_L \quad (2)$$

假定 Cache 接收到一个长 Cache 行地址,该地址映射到 $Tag_Entry_Array_1$ 中的第 *set_L* 组,并且该地址缺失,则 *Misshandler_long* 被调用,该过程包含以下几步:

① 选择最早访问过的长 Cache 行作为被换出的行. 因为链表索引数组规定了链头结点记录最早访问的 Cache 行的行索引,所以从第 *set_L* 个索引链表中选择链头的数据域的值,记作 *index_L*,根据 *index_L* 的奇偶性,选择定义(1)或定义(2),从而确定最早访问的长 Cache 行的行索引值. 同时将该长 Cache 行包含的两个短 Cache 行的行索引从小到大依次记作 *index_low* 和 *index_high*. 图 10(d)中 *index_low* 和 *index_high* 的值分别为 2 和 3.

② 检查对应的 V 数组并将脏字节写回主存. 如果数组 V_1 的对应元素为 1,即整个长 Cache 行的有效位为 1,则作为 victim 的长 Cache 行中的脏字节被写回主存. 如果整个长 Cache 行对应的有效位为 0,可是长 Cache 行中的两个短 Cache 行无论哪个对应的有效位为 1,则相应的短 Cache 行的脏字节写回主存,并将其有效位置为 0,同时将长 Cache 行对应的有效位置为 1.

③ 从主存中取得需要的长 Cache 行的数据,同时将作为 victim 的长 Cache 行对应的结点移至第 *set_L* 个索引链表的链尾.

5.4 *Misshandler_short*

用 *IndAlign*(*index_S*)表示当链表的链头结点的数据域的值为 *index_S* 的时候,最早访问的短 Cache 行的行索引值. 对于一个映射到 $Tag_Entry_Array_2$ 中第 *set_S* 组的短 Cache 行的地址来说,*IndAlign*(*index_S*)定义如下:

当 *set_S* 是偶数的时候,

$$IndAlign\ (index_S) = earlist\ (0,1,2,3) \quad (3)$$

当 *set_S* 是奇数的时候,

$$IndAlign\ (index_S) = earlist\ (4,5,6,7) \quad (4)$$

这里 $earlist\ (para_1, para_2, para_3, para_4)$ 中的 4 个参数表示短 Cache 行的行索引,$earlist\ (para_1, para_2, para_3, para_4)$ 表示括号中 4 个短 Cache 行中最早访问的 Cache 行的索引.

假设 Cache 接收到一个短 Cache 行的地址,该地址映射到 $Tag_Entry_Array_2$ 中的第 *set_S* 组,并且满足 $set_S / 2 = set_L$,如果该地址在 $Tag_Entry_Array_2$ 中缺失,它跟长 Cache 行的缺失不一样,在调用缺失处理的同时检查 $Tag_Entry_Array_1$. 如果在 $Tag_Entry_Array_1$ 中查到一个有效匹配,则立即终止缺失处理,之后的操作类似于短 Cache 行的命中. 如果在 $Tag_Entry_Array_1$ 中仍然缺失,则返回 *Misshandler_short* 的值.

为了清楚地描述该过程，图 11 给出了一个局部的 HLSC 的结构图. 假定短 Cache 行的地址映射到第 *set_L* 个链表，该链表对应的短 Cache 行标记为 $L[0]$到 $L[7]$. *Misshandler_short* 的具体实现如下：

① 选择最早访问的短 Cache 行作为 victim. 判断 *set_S* 的奇偶性，从而根据定义(3)或定义(4)确定最早访问的短 Cache 行的索引 *index_old*. 图 11 中假定 *index_old* 等于 5.

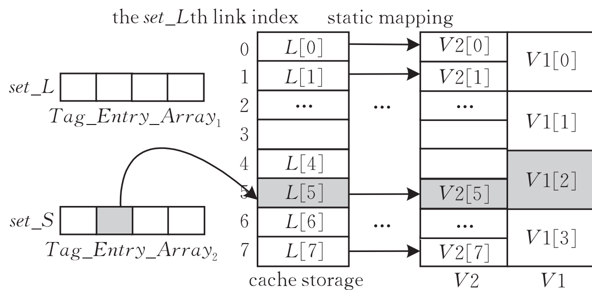

图 11　局部的 HLSC 的结构图

② 检查对应的 V 数组并写回脏字节. 先检查 *index_old* 对应的长 Cache 行的有效位 $V_1[2]$是否有效.

如果有效，则 $L[4]$ 和 $L[5]$组成的长 Cache 行的脏字节写回主存，同时将 $V_1[2]$设置为无效.

如果无效，则检查 $V_2[5]$是否有效，如果有效，则需将 $L[5]$中的脏字节写回主存.

③ 从主存中取得需要的短 Cache 行的数据，填写相应的 TE，将 $V_2[5]$设置为有效. 最后将链表中数据域的值为 5 的结点移到链尾，记作最新访问的 Cache 行索引.

6　实　验

6.1　实验环境

该实验在一个 3.2GHz 的 Cell 处理器上进行. Cell 处理器上的 PPE 包含一个 32KB 的一级指令 Cache，一个 32KB 的一级数据 Cache 以及 512KB 的二级 Cache. 每个 SPE 包含一个 256KB 的本地存储. 测试使用的系统为 Fedora9（Linux Kernel 2.6.25-14)，编译程序使用的是 Cell SDK3.1.

为了评估本文提出的方案，我们在 Cell 处理器上实现了 HLSC 方案，并与传统的固定 Cache 行方案以及 ESC 方案进行了比较. 实验中应用稀疏矩阵向量乘(SpMV)以及 NAS benchmarks[16] 中的 FT、IS、CG、MG 进行性能评估. 稀疏矩阵来源于佛罗里达大学的稀疏矩阵向量集[17]，这些矩阵涵盖了实际应用的很多领域，如计算流体力学、电路模拟、分子动力学、线性规划等. 实验中自适应的 Cache 行算法动态地从 128B，256B，512B 和 1024B 中选择最优的 Cache 行.

6.2　加速比

首先将 HLSC 的自适应策略与固定的 Cache 行设计作比较. 将 256B 的 Cache 行设计的执行速度设置为标准的速度，那么各种 Cache 设计的加速比如图 12 所示.

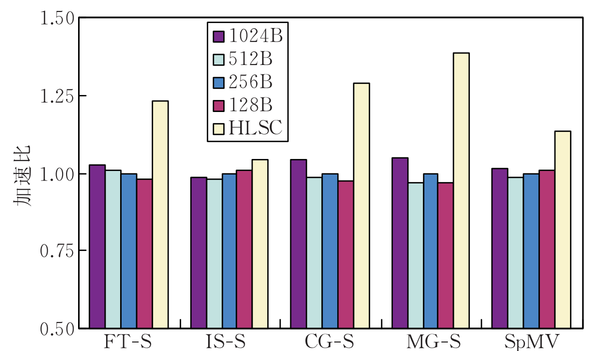

图 12　标准化的加速比

整体上来讲，HLSC 的执行速度要明显优于固定的 Cache 行设计的执行速度. 跟固定的 1024B，512B，256B 和 128B 的 Cache 行设计相比，HLSC 的平均执行速度分别提高了 28.9%，29.7%，32.1% 和 33.5%. 特别是 MG，它的性能随 Cache 行长度的改变而发生明显的变化，因此当采用自适应的 Cache 行策略之后，它的性能提升得最多. IS 对 Cache 行大小不敏感，因此跟固定的 Cache 行设计相比，HLSC 的性能改善不太明显.

然后，将 HLSC 的自适应策略跟现有的 Cell 处理器上实现的 ESC 的策略作比较. 这两种自适应的算法均从 128B，256B，512B，1024B 4 种 Cache行大小中自适应地选择最优的 Cache 行. 以 ESC 中的自适应算法的执行速度作为基准，标准化了的加速比见图 13.

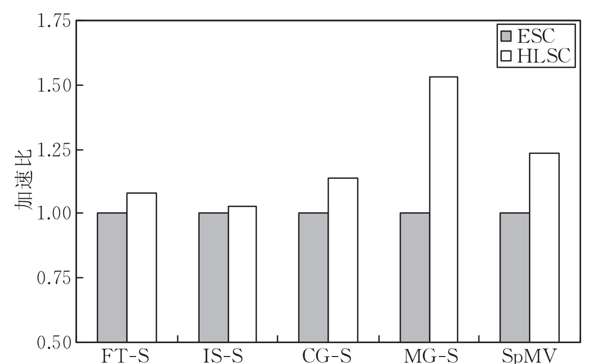

图 13　ESC 和 HLSC 的自适应的算法性能比较

整体上来讲，HLSC 的自适应算法提高了程序的执行速度，特别是对于 MG，性能提升最为明显，加速比可达到 1.52. 主要在于以下两点：

(1) HLSC 的自适应算法可以作用到循环内部每个迭代区间，可是 ESC 的自适应算法只对某个循环起作用，而对内部迭代不敏感. MG 对循环内的迭代区间 Cache 行的改变很敏感(如图 12 所示)，因此性能提升较为明显.

(2) 在某些情况下，ESC 的自适应算法选出的 Cache 行长度可能不是最优的. 为了说明问题，作如下假设：

① 有 5 种候选的 Cache 行大小，LS_0 到 LS_4，Cache 行长度依次增长；

② 对某个循环来说，这 5 种 Cache 行的性能级别分别为 1,3,2,4,0(数字越大，表示对应的 Cache 行的性能越好)，

则 ESC 中自适应的选择 Cache 行的执行过程如下：当循环第一次被调用的时候，选择中间长度的 Cache 行，即 LS_2，测得其每次迭代的执行时间 TPI (Time Per Iteration)记作 TPI_1；

当该循环被再次调用时，选择一个稍短的 Cache 行(即 LS_1)执行该循环，此时计算得到每次迭代的执行时间为 TPI_2. 根据假定的性能级别，TPI_1 肯定会比 TPI_2 大，ESC 的自适应算法会认为短的 Cache 行性能会好一些；

所以在下次该循环被调用时，ESC 自适应的算法选择一个更短一些的 Cache 行(即 LS_0)来执行该循环，并测得此时的 TPI 为 TPI_3，根据假定的 Cache 行的性能级别，TPI_3 比 TPI_2 大. 因此，ESC 自适应的算法选择最小的 TPI 对应的 Cache 行(即 LS_1)作为其最优的 Cache 行. 但是，从假设的性能级别可以看到 LS_3 才是针对此循环的最优的 Cache 行. 产生这样的问题主要因为：当 TPI_1 比 TPI_2 大时，ESC 的自适应算法会认为短的 Cache 行性能会好一些，这个推理不一定成立，因为在长度上连续的两种 Cache 行对某个循环来说，其执行性能不一定连续，即可能出现随着 Cache 长度依次递变，执行性能却发生跳变的情况. 从图 12 可以看到 MG 是一个随 Cache 行长度递变性能会出现跳变的应用，因此 MG 的性能提升受到了 ESC 中自适应算法的限制.

SpMV 中稀疏矩阵向量乘的循环只被调用了一次，为了测试两种不同自适应策略下的性能，本文将选择中间长度的 256B 和 512B 中性能较好的 256B 作为 ESC 中自适应策略确定的最优 Cache 行. ESC 算法需要多次调用某个循环以确定最优的 Cache 行，而 SpMV 中稀疏矩阵向量乘的循环只运行了一次，所以该应用从 ESC 的自适应策略中获得的性能提升并不明显. 而本文提出的 HLSC 的自适应算法对循环被调用次数没有限制，因此对于 SpMV 应用来说，HLSC 的自适应策略明显优于 ESC 的自适的算法.

IS 是一个对 Cache 行变化不敏感的应用，所以对于固定的 Cache 行、ESC 的自适应算法和 HLSC 的自适应策略性能变化都不明显.

6.3　传输的字节数

如果将 128B 的 Cache 行设计传输的字节数作为基准，则各种 Cache 行设计传输的字节数如图 14 所示. 很明显，HLSC 传输的字节数要少于固定的 1024B,512B,256B 和 128B 的 Cache 行设计所传输的字节数. 特别是跟最长的 1024B 的 Cache 行设计相比，HLSC 设计传输的字节数约占 1024B 的 Cache 行设计的 31%. 主要在于假如计算中只需要一个字节，1024B 的 Cache 行设计需要传输全部的 1024B 的数据，而 HLSC 设计可能只需要传输 128B 的数据. HLSC 设计传输的字节数之所以也会少于 128B,256B 等短的固定的 Cache 行设计是因为当短的固定的 Cache 行设计遇到缺失处理时立即传输数据(不管是直接从内存中取数还是组冲突之后先剔出 Cache 中的数据再从内存中取数据)，而 HLSC 方案有可能在较长的 Tag 项数组中找到需要的数据，而无需进行冗余的数据传输.

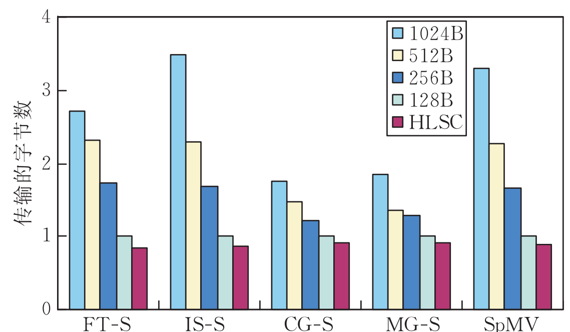

图 14　标准化了的传输字节数

6.4　命中率

图 15 给出了以 128B 的 Cache 行设计的命中率为基准的情况下，各种 Cache 行设计的命中率. 与固定的 1024B,512B,256B 和 128B 的 Cache 行设计相比，HLSC 的命中率分别提高了 20.5%,21.1%,23.4%,24.7%. 主要因为 HLSC 设计是分级的，当

短 Cache 行的 Tag 项数组发生缺失时，还要检查较长 Cache 行的 Tag 项数组是否命中.

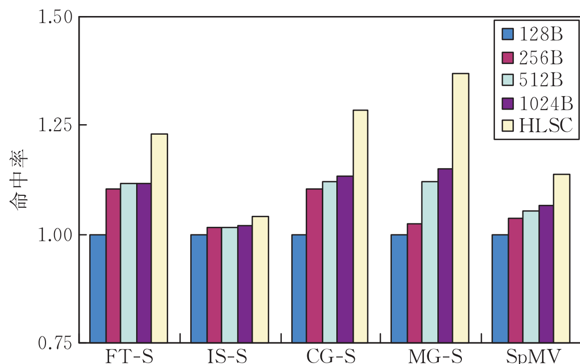

图 15 标准化了的命中率

在大多数情况下，固定的长 Cache 行的命中率要高于固定的短 Cache 行的命中率. 因为一次长 Cache 行的 DMA 操作取到的数据可能相当于多次短的 Cache 行的 DMA 操作取到的数据. 以固定的 1024B 和 512B 为例，如果 Cache 接收到两个连续的内存请求，并且请求的数据均未取到 Cache 中，这两个地址分别映射到 1024B 的 Cache 行中的前、后 512B，那么采用固定的 512B 的 Cache 行的两次请求一定都是缺失，而采用固定的 1024B 的 Cache 行设计的第一次请求的地址缺失，但第二次请求的数据已经被第一次 DMA 操作取到 Cache 中，因此命中.

从图 14 和图 15 中可以看出，HLSC 性能提高不仅来源于减少了冗余的数据传输，还在于提高了 Cache 的命中率.

6.5 存储开销比较

为了形式化表示存储的开销，作如下假定：

(1) Cache 存储中包含 N 个 128B 的 Cache 行，则四路组相联的 Cache 中包含 $S(S=N/4)$ 个组；

(2) ESC 中的组数表示为 S'，依据 ESC 的设计，S' 是跟 S 最接近的 2 的幂；

(3) *Tag* 是 4 个字节的整数，脏位和有效位均为一个字节；

(4) 每个 TE 需要一个四字节的行索引(*line_index*)；

(5) “LTE”(Line Table Entry)代表行表项，在 FAC 和 ESC 结构的行表项中均包含了一个跟 TE 中一致的 Tag 项(Tag Entry，TE)，用以记录全局地址.

表 1 除 Cache 外的额外存储开销 (单位：字节)

	FAC	4WC	ESC	HLSC
TE	$Tag+line_index=8$	$Tag+line_index=8$	$Tag+line_index=8$	$Tag+line_index=8$
TE Array	$TE*N=8N$	$(4*TE)*S=8N$	$(4*TE)*S'=32S'$	$(4*TE)*(S+S/2+S/4+S/8)=15*N$
LTE	$V+D+Tag=6$	$V+D=2$	$V+D+Tag=6$	$(V+D)*(1+1/2+1/4+1/8)=15/4$
Cache Line Table (CLT)	$LTE*N=6N$	$LTE*N=2N$	$LTE*N=6N$	$LTE*N=15*N/4$
Total Size (TE Array+CLT)	$14N$	$10N$	$32S'+6N$	$75*N/4$

表 1 给出了除了 Cache 存储以外各种 Cache 结构的额外存储开销. 其中：“FAC”和“4WC”分别代表全相联 Cache(fully-associative cache)和四路组相联 Cache(4-way set-associative cache). HLSC 采用的是 128B，256B，512B 和 1024B 4 种 Cache 行结构.

HLSC 的额外存储负荷比 FAC 和 4WC 的多一些，但是在一个数量级上差别不明显. ESC 结构中的 S' 的值是最接近 S 的 2 的幂，所以 S' 的值跟 S 在同一个数量级上. 因此，4 种 Cache 结构的额外存储开销相当.

7 结论和下一步工作

本文提出了一种在 Cell 异构多核处理器上解决非规则问题的自适应的 Cache 行策略，它根据非规则访问的特点自适应地调整 Cache 行大小，极大地减少了冗余的数据传输. 同时，设计了一种混合的软件 Cache 结构——HLSC，它包含了多种 Tag 项数组，提高了 Cache 的命中率. 除此之外，文中还提出一个面向多种 Cache 行设计的 LRU 的替换策略. 实验结果证明自适应 Cache 行技术极大地提高了 Cache 命中率，跟固定的 1024B，512B，256B，128B 的 Cache 行的性能相比，自适应的 Cache 行技术的执行速度分别提高了 28.9%，29.7%，32.1% 和 33.5%. 并且与 ESC 相比，HLSC 不仅具有明显的性能提升而且应用范围更为广泛.

下一步工作中，我们将同时考虑规则和非规则两种内存引用的特点[18]，把该自适应的 Cache 行的策略扩展到混合的内存引用中[19-20]，从而使该策略更具通用性. 同时通过观察发现，预取技术能有效地将通信(DMA 操作)与计算重叠起来，因此，设计一种面向非规则引用的自适应软件 Cache 的预取技术将是我们下一步的研究重点.

参 考 文 献

[1] Seo Sangmin, Lee Jaejin, Sura Zehra. Design and implementation of software-managed caches for multicores with local memory//Proceedings of the High Performance Computer Architecture Conference (HPCA'09). Shanghai, China, 2009: 55-66

[2] Miller Jason E, Agarwal Anant. Software-based instruction caching for embedded processors//Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating (ASPLOS-XII). San Jose, CA, 2006: 293-302

[3] Moritz Csaba Andras, Frank Moritz Matthew, Lee Walter et al. Hot Pages: Software Caching for Raw Microprocessors. USA: MIT, MIT-LCS Technical Memo LCSTM-599, 1999

[4] Moritz Csaba Andras, Frank Matthew I, Amarasinghe Saman. FlexCache: A framework for flexible compiler generated data caching//Proceedings of the 2nd Workshop on Intelligent Memory Systems. Cambridge, MA, 2001: 135-146

[5] Udayakumaran Sumesh, Dominguez Angel, Barua Rajeev. Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Transactions on Embedded Computing Systems (TECS), 2006, 5(2): 472-511

[6] Witchel Emmett, Larsen Sam, Ananian C Scott et al. Direct addressed caches for reduced power consumption//Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'01). Austin, Texas, 2001: 124-133

[7] Fryman Joshua B, Lee Hsien-Hsin S, Huneycutt Chad M. SoftCache: A technique for power and area reduction in embedded systems. USA: Georgia Institute of Technology, CERCS: GIT-CERCS-03-06, 2003

[8] Eichenberger Alexandre E, O'Brien J K, O'Brien K M et al. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. IBM Systems Journal, 2006, 45(1): 59-84

[9] Balart Jairo, Gonzalez Marc, Martorell Xavier et al. A novel asynchronous software cache implementation for the cell-BE processor//Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07). Urbana, IL, USA, 2007: 125-140

[10] González Marc, Vujic Nikola, Martorell Xavier et al. Hybrid access-specific software cache techniques for the cell BE architecture//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08). Toronto, Canada, 2008: 292-302

[11] Vujić Nikola, Gonzàlez Marc, Martorell Xavier et al. Automatic pre-fetch and modulo scheduling transformations for the cell BE architecture. IEEE Transactions on Parallel and Distributed Systems, 2010, 21(4): 494-505

[12] Vujić Nikola, Gonzàlez Marc, Martorell Xavier et al. Automatic pre-fetch and modulo scheduling transformations for the cell BE architecture//Proceedings of the Languages and Compilers for Parallel Computing (LCPC'08). Edmonton, Canada, 2008: 31-46

[13] Veidenbaum Alexander V, Tang Weiyu et al. Adapting cache line size to application behavior//Proceedings of the 13th International Conference on Supercomputing (ICS'99). Rhodes, Greece, 1999: 145-154

[14] Zhang Chuanjun, Vahid Frank, Najjar Walid. Energy benefits of a configurable line size cache for embedded systems//Proceedings of the International Symposium on VLSI design (ISVLSI'03). Tampa, Florida, 2003: 87-91

[15] Zhang Chuanjun. Balanced-cache: Reducing conflict misses of direct-mapped caches through programmable decoders//Proceedings of the International Symposium on Computer Architecture(ISCA'06). Boston, MA, 2006: 155-166

[16] Bailey D H, Barszcz E, Barton J T et al. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, 1991

[17] Davis Tim, Hu Yifan. University of florida sparse matrix collection [online]. Available from: http://www.cise.ufl.edu/research/sparse/matrices, 2009

[18] Liu Tao, Lin Haibo, Chen Tong et al. DBDB: Optimizing DMA transfer for the cell BE architecture//Proceedings of the 23rd International Conference on Supercomputing(ICS'09). Yorktown Heights, USA, 2009: 36-45

[19] Wang Shao-Gang, Wu Dan, Pang Zheng-Bin, Yang Xiao-Dong. HybridTCache: Tightly coupled hybrid transactional memory system to support efficient unbounded transactions with strong isolation. Chinese Journal of Computers, 2008, 31(11): 1907-1917(in Chinese)
(王绍刚,吴丹,庞征斌,杨晓东. HybridTCache:一种基于专用事务Cache的软硬件协同事务内存系统. 计算机学报, 2008, 31(11): 1907-1917)

[20] Zhang Pan-Yong, Meng Dan, Huo Zhi-Gang. Research of collectives optimization on modern multicore clusters. Chinese Journal of Computers, 2010, 33(2): 317-325(in Chinese)
(张攀勇,孟丹,霍志刚. 多核环境下高效集合通信关键技术研究. 计算机学报, 2010, 33(2): 317-325)

CAO Qian, born in 1983, Ph. D. candidate. Her research interests include high performance computing, parallel computing and parallel compilation technology.

HU Chang-Jun, born in 1963, Ph.D., professor, Ph. D. supervisor. His main research interests include parallel computing, parallel compilation technology, parallel software engineering, network storage system, data engineering and software engineering.

ZHANG Yun-Xing, born in 1985, M. S.. His research interests include parallel computing and parallel compilation technology.

ZHU Yu-Tian, born in 1985, M. S.. His research interests include parallel computing and parallel compilation technology.

Background

Irregular application is widely used in scientific computing, which exposes unclear aliasing and data dependence information. Such applications are frequently seen in reservoir numerical simulation, molecular dynamics, etc. The work in this paper belongs to optimization of irregular applications on Cell heterogeneous multicore. Software cache promises to achieve programmability on Cell processor for irregular applications. Up to now, lots of researches have focused on the software cache designs. However, irregular references couldn't achieve a considerable performance improvement since the cache line is always set to a specific size, which ignores the irregular reference memory access pattern.

The contributions of this work include: An adaptive cache line strategy which continuously adjusts cache line size during application execution is proposed. Therefore, the transferred data is decreased significantly. Moreover, a corresponding software cache-hybrid line size cache is designed. It introduces a hybrid Tag Entry Arrays, with each mapping to a different line size. It's a hierarchical design which significantly increases the hit rate. Additionally, an original replacement policy—index aligned strategy is proposed to implement least recently unused replacement policy for multiple cache line sizes.

The research is partially supported by the National High Technology Research and Development Program (863 Program) of China under grant No. 2006AA01Z105 and No. 2008AA01Z109, Natural Science Foundation of China under grant No. 60373008, the Key Project of Chinese Ministry of Education under grant No. 106019 and No. 108008, and by Important National Science & Technology Specific Projects under grant Nos. 2009ZX03004-004 and 2009ZX01045-005-002.