

SAGA: 一种由流特性制导的微处理器 高速缓存分配策略

陈 彧 林隽民 乔 林 汤志忠

(清华大学计算机科学与技术系 北京 100084)

摘 要 传统的缓存替换策略,如广泛使用的 LRU 算法,在程序工作集大于缓存容量的情况下,不能有效开发流式数据的重用性,导致缓存性能很差.文中提出一种流特性制导的缓存分配策略(SAGA).该策略利用流检测引擎来发掘程序中的流特性信息,进而动态地在发生缓存缺失时指导是否为缺失数据分配缓存块,最终提高数据缓存的性能.实验表明,对于 SPEC2000FP 程序集,在 1MB 缓存上,比较于 LRU 策略,使用 SAGA 策略时缓存的缺失平均减少了 31%,程序平均 CPI 降低 4%.

关键词 高速缓存;替换策略;流

中图法分类号 TP302 **DOI 号:** 10.3724/SP.J.1016.2008.01929

SAGA: A Stream Attribute Guided Cache Allocation Policy for Microprocessors

CHEN Yu LIN Jun-Min QIAO Lin TANG Zhi-Zhong

(Department of Computer Science and Technology, Tsinghua University, Beijing 100084)

Abstract Traditional cache replacement schemes such as the commonly used LRU policy often fail to exploit reuse of stream data when the working set size of the application is bigger than the cache size, resulting in poor cache performance. In this paper, the data cache performance is improved by enhancing LRU policy with a novel Stream Attribute Guided Cache Allocation (SAGA) policy, which dynamically utilizes streaming information in applications detected by stream engines on microprocessors to guide whether allocate a new cache line or not when a cache miss occurs. Experiments show that SAGA outperforms LRU by 31% in terms of cache misses and 4% in terms of CPI for SPEC2000FP benchmark on a 1MB cache.

Keywords cache; replacement policy; stream

1 引 言

片上高速缓存(Cache)对于现代高性能微处理器来说是至关重要的.这是由于,如果数据访问在片上缓存上缺失,将引起数十倍于缓存访问时间的片

外内存访问延迟,从而使流水线停顿很长时间.片上缓存的命中时间通常在 2~40 个时钟周期之间,取决于命中缓存所在的层次(L1/L2/L3)、缓存的容量(容量越大延迟越大)及片上网络的延迟.现代超标量计算机普遍具有 64~128 大小的指令发射窗口,当指令的延迟比较小时,可以以乱序执行的方式,发

掘程序的指令级并行性(Instruction Level Parallelism, ILP),来维持流水线的持续装入.但是,目前片外内存模组的访问时间多在 100~400 个时钟周期之间.在这么长的访存延迟下,处理器没有能力在程序中发掘出足够的指令级并行性,从而流水线只能停顿.经验上,在 IA32 体系结构下,当片上缓存的每千条指令缺失(Miss per 1000 Instructions,或 MPKI)达到 1 以上时,程序的性能将主要受存储系统的速度影响.

然而,在传统的缓存替换算法下,片上缓存的性能存在着很大的缺陷,为缓存的设计带来极大的难题.传统缓存替换策略的代表是 LRU 算法及其近似算法,其主要的缺陷是,在某些应用程序上,为处理器显著地增加缓存容量,却不能带来命中率的提高,从而浪费了功耗和芯片面积.例如,对于 SPEC CPU2006 中的 libquantum 程序^[1],缓存容量为 128KB 时 MPKI 高达 13,但是,随着容量的增加,缓存的缺失并不下降,在容量增加至 32MB 之前,MPKI 一直维持在 13 左右.对于这一类程序,如果运行在一个小于 32MB 缓存的处理器上,不但性能很差,而且缓存没有起到作用,反而浪费功耗和芯片面积.在本文中,我们将解释这种现象的成因在于程序的流式访存行为,并提出相应的解决方案.

传统缓存替换算法的局限性将在未来的微处理器上体现得更加明显.首先,处理器—存储器时延剪刀差呈现增长趋势.微处理器进入片上多处理器(CMP)时代后,每周周期发出片外访存请求将随着核心数目的增长而增长.新型片外 DRAM 模组的访问延迟时间虽然也越来越短,但是性能提高的速率较低,仍不及处理器速度的增长.其次,最新的研究指出,新兴的应用程序有更大的工作集和更强的流式访存行为^[2].更大的工作集意味着程序的访存行为更密集,需要更大的缓存容量来容纳它的数据.更强的流式访存行为则意味着并不是简单的增加缓存容量就可以提高命中率.

本文提出一种由流特性制导的缓存分配策略(Stream Attribute Guided cache Allocation, SAGA).在 SAGA 策略中,首先,我们利用现代处理器中普遍采用的流检测引擎,收集程序流行为的信息.其次,我们对每一条流增加而外的记录信息,用以追踪流对缓存的影响.最后,在每次发生缓存缺失时,我们使用已采集的流特性信息来制导是否为缺失的缓存块在缓存中分配空间.在本文中,我们将说明 SAGA 策略只对缓存结构做了简单的修改,却能显著的提升访存密集型的应用程序的性能.

本文第 2 节介绍相关领域的研究工作;在第 3 节中我们以一个简单的例子介绍 SAGA 的动机;第 4 节则介绍 SAGA 算法的细节;我们在第 5 节中展示实验方法和实验的结果;最后在第 6 节中进行总结并探讨下一步的工作.

2 相关工作

流检测是普遍应用在处理器中的一项技术,相关研究很早就出现了. Jouppli 首次将流预取技术应用在处理器上^[3]. Palacharla 提出流缓冲(Stream Buffer)甚至可以作为普通高速缓存的替代^[4]. Mohan 提出了评估程序中流访问行为的方法,并认为程序的流行为对软/硬件优化具有指导意义^[5]. 本文的流检测框架以前述研究作为基础,实现机制与 Stream Buffer^[4]是类似的.

许多研究对各个领域的应用程序的流行为进行了评估^[1,6-8]. 如物理模拟、工程计算、服务器应用等. Chen 等对未来多核处理器上的新兴应用进行了评测^[2],认为在多核处理器的新兴应用程序上,程序的流行为要比现有的典型程序强许多,同时工作集也比现有程序大很多. 本文所提出的 SAGA 算法适合于这一类大工作集、流行为强的工作负载.

在文件和存储领域,有很多研究工作集中于替换算法的改善. Megiddo 提出了一种在 LRU 和 LFU 之间动态切换的策略^[9]. Jiang 提出的 LIRS 算法可以避免程序流行为带来的缓存抖动问题^[10]. 然而,存储领域采用的解决方案是软件的,这一类解决方案应用在硬件中开销过大.

近来有许多研究致力于改善微处理器中的缓存性能. Qureshi 提出了低开销的 DIP 策略^[11],当程序的工作集大于缓存容量时以一定概率随机地保留一部分数据在缓存中. Basu 提出了 Scavenger 作为一种结合了频度的最后一级缓存替换策略^[12]. 然而,这些方案有各自的缺陷. 应用 DIP 策略时的最佳时机是程序工作集比缓存容量稍微大一点,而当工作集比缓存容量大很多时, DIP 有可能将重用性好的非流式数据丢弃. 此外, DIP 策略需要一定数量的缓存组(Set)固定为某一种基本替换策略,当缓存容量较小时(如 L1),组的数目不足,可使 DIP 的效果受到影响. 对于 Scavenger,其缺点是复杂的硬件实现机制和开销,使其缺乏很好的实用价值.

3 原 理

本节将以简单的例子解释 SAGA 算法的工作

原理.

首先我们给出在本文中流的定义. 我们定义流为:一段内存访问序列的子序列,使得该子序列任意相邻两个元素的地址差相等. 该地址差则称为步长. 依此定义的流式访存序列也可称为常量步长流. 根据步长的大小,我们又将常量步长流分为步长为 1 个缓存块的单元步长流以及步长超过 1 个缓存块的多单元步长流.

许多程序含有大量的流式访存行为. 例如,一个实现线性代数算法的程序中,程序的主要数据结构可能是一个或多个向量(矩阵). 程序在运行过程中需要在向量(矩阵)上逐个元素进行计算和更新,然后反复这一过程. 这种情况下,程序对该向量(矩阵)的访问会形成一条或多条访存流.

对于流式访存行为比较强的程序,传统的 LRU 缓存替换策略可能会出现抖动的问题. 例如,假设程序中有如下的访存序列片段:

...ABXCXXDEXFXGYHAYXBCYDYEXF-
GYXHXABXYCYDXXEXFGYYHABXXC-
DEYFYGH...

其中,A 到 H 代表一条长度为 8 的流,X 和 Y 代表其它局部性好的数据. 假设缓存为全相联结构,那么在 LRU 策略下可以计算缓存缺失数与缓存容量的关系如图 1 所示. 设缓存容量为 C ,若希望某个流数据流数据 v 的某一次访问在缓存中命中,易知在 LRU 算法下必须满足: v 的上一次访问和本次访问之间所访问不同的数据元素个数不超过 $C-1$ 个. 可见,当缓存容量小于工作集大小(10)时,无论缓存大小如何增大,流数据总是缺失的. 这是因为,流数据在它被装入之后,在 LRU 算法逐渐将其搬移至栈底的过程中,都不能得到重用. 这种流数据完全不能得到重用的过程即称为“抖动”(thrashing).

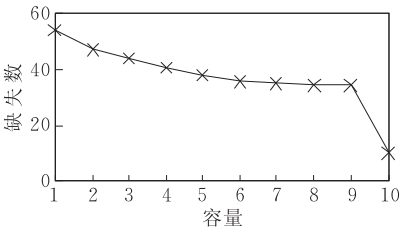


图 1 简单例子的缺失-容量曲线

为了解决在 LRU 算法下缓存对流数据产生抖动的问题,我们设法将一部分流数据留在缓存中,另一部分则不予分配缓存空间. 例如,在上面的简单例子中,当缓存容量为 8 时,我们把 A 到 F 共 6 个数据分配在缓存中,G 和 H 不予分配缓存空间,那么缺失数可以从 34 下降到 16.

4 SAGA 算法

本节介绍 SAGA 算法的具体细节. 图 2 描述了算法的框架. 由于片上最后一级缓存的性能对于减少片外高延迟访存操作很重要,所以我们把 SAGA 的应用目标定为最后一级缓存. 图中以一个 3 级缓存结构为例,对 L3 的访问同时送入流检测引擎(Stream Engine)中,识别访存流. 流引擎根据 SAGA 算法的结果,制导 L3 的分配策略,即是否为缺失的缓存块分配缓存空间. 若不分配空间,则缓存中内容不变,否则按照 LRU 算法操作,即将最久未用块踢出.

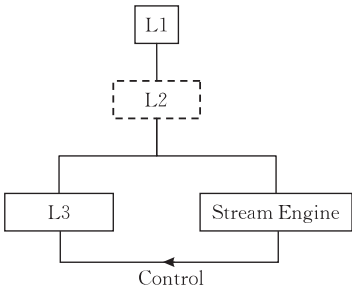


图 2 SAGA 框架

4.1 流检测算法

首先介绍我们使用的流检测算法. 为了在程序执行过程中获得程序访存流特性的信息,我们必须依赖于有效的流检测算法. 我们使用的算法与 Stream Buffer 非常相似^[4],因此本文不展开讨论. 对流检测算法的准确度及更好的改进等方面感兴趣的读者请参阅 Stream Buffer 的实现和相关文献[1-2,4-5,7-8]. 流检测在最后一级缓存之前进行. 即对最后一级缓存的访问同时也送入流检测引擎中. 因为指令地址(IP)不会在各级缓存中传输,我们不使用基于 IP 的流检测,而使用基于内存地址的方法来进行检测. 我们在流引擎中使用 3 张表. 表中每一项的元素包括步长、最后一次的访存地址等.

检测算法简述如下:

- 1. 算法的输入是一次访存操作所访问的内存地址,算法的输出是该次内存访问是否属于一条访存流以及该流的步长、起始地址等. 对于每一个访存请求,将依此访问 3 张表,直到找到对应的一项.
- 2. 第 1 张表存放已经检测出来的流,称为稳态流表. 如果访存请求与表中的某一项中的下一访存地址(等于最后访问地址+步长)相等,则检测成功结束;否则访问第 2 张表;
- 3. 第 2 张表存放尚在训练中的单元步长(即步长等于一个缓存块大小)流,称为单元步长流检测表. 如果访存地址与表中某一项的最后访问地址相邻(递增或递减),那么检测成功结束,在第 1 张表中增加一条表项. 如果没有找到匹配

项,则在该表中新建一条表项,把当前访存地址作为一条新的待训练流的起始点,并访问第 3 张表;

4. 第 3 张表存放尚在训练的多单元步长(即步长>一个缓存块)流,称为多单元步长流检测表.该表以访存地址的页地址作为索引以查找表项,并与已有的表项对比,计算新的步长,当步长稳定在同一数值上的次数满足某个阈值时,识别成功结束,在第 1 张表中分配新的表项.

4.2 SAGA 算法

在介绍了本文所使用的流检测算法后,我们接下来介绍 SAGA 算法如何利用流检测的结果来指导缓存分配策略.

设 r 表示一次访存操作,我们定义一组记号:

$A(r)=r$ 所访问的内存地址 (1)

$m(A)=\begin{cases} 0, & A \text{ 在缓存中} \\ 1, & \text{否则} \end{cases}$ (2)

$s(r)=\begin{cases} e, & r \text{ 在流引擎中命中稳态流表} \\ \epsilon, & \text{否则} \end{cases}$ (3)

$a(r)=\begin{cases} 1, & \text{为 } r \text{ 分配缓存空间} \\ 0, & \text{否则} \end{cases}$ (4)

其中,式(1)~(3)为算法的输入,式(4)为算法的输出.每一个向最后一级缓存的访问 r 同时送入缓存与流检测引擎中.缓存部件进行查找, $m(A(r))$ 表示是否命中,简记为 $m(r)$,流引擎部件则对 r 尝试归入某一条流中,记号 ϵ 表示空表项, e 表示一条有效的稳态流表项,即 r 被识别为流. SAGA 算法根据 $m(r)$ 和 $s(r)$ 来确定 $a(r)$,如图 3 所示.当数据从内存中取出时,若 $a(r)$ 为 1,则为该数据分配缓存空间;若 $a(r)$ 为 0,则数据不在缓存中分配空间,旁路后送至上一级缓存.

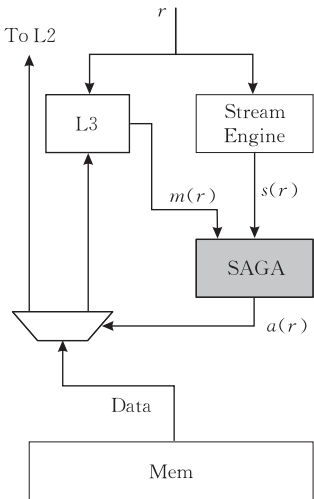


图 3 SAGA 算法示意图

下面我们将介绍 SAGA 算法中输入与输出的关系.

定义一组记号:

$e=s(r)=(b,k,l)$ (5)

其中 e 为一条流表项, b 为流的起始地址, k 为步长, l 为最后一个数据的地址,即 $A(r)$.

$\Omega=\{A|A \bmod M=0\}$ (6)

其中 A 是任意的内存地址, M 是参数. Ω 表示了所有模 M 为 0 的内存地址所构成的集合. 本文实验中设定的 M 为一个质数 11. 我们称 Ω 为采样地址空间, Ω 中的地址为采样地址.

$P(e)=\{A|b\leq A\leq l, A=b+k_i, i=0,1,2,\dots\}\cap\Omega$ (7)

即 $P(e)$ 中的每个内存地址,既是属于流 e 的元素,又是采样地址,我们称 $P(e)$ 为流 e 的采样地址空间.

$p(e)=\begin{cases} \max\{P(e)\}, & k>0 \\ \min\{P(e)\}, & k<0 \end{cases}$ (8)

即 $p(e)$ 为流 e 的采样地址空间内最接近 $A(r)$ 的内存地址. 我们称 $p(e)$ 为流 e 关于 r 的采样点.

于是我们定义 SAGA 算法的输出为

$a(r)=\begin{cases} 0, & m(r)=1 \wedge s(r)=e \wedge m(p(e))=1 \\ 1, & \text{否则} \end{cases}$ (9)

即对访存操作 r 不分配缓存空间当且仅当 r 缺失, r 属于某条流 e ,且 r 关于 e 的采样点也缺失.

通过改造流检测引擎的表结构,我们实现对采样点的追踪. 方法是,在流引擎的每一条表项 e 中,增加一个字段 t 用于记录采样点是否发生缺失,即 $e=(b,k,l,t)$. 当第 1 张表(稳态流表)被更新时,更新 t 为 t' :

$t'=\begin{cases} m(r), & l=p(e) \\ t, & \text{否则} \end{cases}$ (10)

即当流“经过”一个采样点时,依照采样点是否命中缓存来更新 t . 因此,算法的输出 $a(r)$ 也可以改写为

$a(r)=\begin{cases} 1-t, & s(r)=e\neq\epsilon \\ 1, & \text{否则} \end{cases}$ (11)

4.3 SAGA 算法的开销

SAGA 算法的空间开销是可以忽略的. 流检测引擎在许多处理器中已经实现,用于进行数据预取等操作. 如 IBM Power5 处理器支持 8 条流的检测和预取^[13]. Intel Core 微体系结构的双核处理器上共有 8 个流检测与预取单元^[14]. 除了在流检测表中增加一个 1 比特位的字段外, SAGA 算法并不使用额外的存储空间. 因此, SAGA 芯片面积的需求是可忽略的.

SAGA 算法在时间上的开销同样可以忽略. 这是由于, SAGA 算法的计算不在访存的关键路

径. 访存操作命中缓存时, SAGA 不做任何处理, 不产生时间开销. 而访存操作发生缓存缺失后, 访问片外 DRAM 芯片与 SAGA 的计算是同时进行的. 通常的片外访存都需要 100 周期甚至更长的时间, 从而 SAGA 计算而引起的延迟完全能够被片外 DRAM 访问所隐藏. 因此 SAGA 所产生的延迟可以忽略.

5 实验评估

5.1 实验方法

本文采用 SPEC2000FP 浮点测试程序集作为实验负载. 我们把同一程序但不同输入集作为不同的负载. 采用这个测试程序集的原因是, (1) SPEC2000FP 是典型的科学及工程计算程序, 不仅代表了目前处理器的典型负载, 并且这些程序当中的核心算法与未来应用的核心算法是类似的, 如矩阵计算、线性代数等. (2) SPEC2000FP 的运行时间较短, 我们进行了全程序缓存功能模拟实验, 使数据更可靠.

为了测量缓存性能, 我们采用执行驱动的方式进行缓存功能模拟. 所有 SPEC2000FP 程序使用 Intel ICC 编译以及 -fast 优化级别, 目标程序为 IA32 指令集. 我们使用 Pin 对二进制可行性文件进行插桩^[15], 截取所有的访存指令, 并送入一个缓存功能模拟器模型. 我们对目标程序进行全程序模拟, 即从原始程序开始到结束都送入模拟器中运行, 而不采用快进或截取时间片段等采样的方式.

为了测量程序总体性能的影响, 除了功能模拟外, 我们还对程序进行周期精确级的时序模拟. 在时序模拟实验中, 我们采用 SimpleScalar 3.0d 作为模拟器^[16], 目标程序为静态编译为 Alpha 指令集的 SPEC2000FP 二进制可执行文件. 模拟器参数如表 1 所示. 进行时序模拟时, 我们首先快进一段程序片段, 然后进行 2 Billion 指令片段的时序模拟. 快进的长度由 SimPoint 工具提供^[17], 如表 2 所示.

表 1 时序模拟参数	
参数	值
取指缓冲长度	4
分支预测	bimod, 2K table
发射宽度	4
功能单元	4ALU+4MUL
ITLB	64-entry
DTLB	128-entry
IL1	32KB, 64B, 4-way, 2-cycle
DL1	32KB, 64B, 4-way, 2-cycle
Memory	250-cycle

表 2 程序模拟参数	
程序	快进长度/十亿指令
wupwise.ref	58.4
applu.ref	1.8
art.ref1	6.7
art.ref2	6.8
ammp.ref	213.0
sixtrack.ref	8.2
swim.ref	0.5
mesa.ref	8.9
equake.ref	19.4
lucas.ref	3.5
apsi.ref	4.6
mgrid.ref	0.6
galgel.ref	315.0
facerec.ref	13.6
fma3d.ref	29.8
wupwise.ref	58.4
applu.ref	1.8

我们模拟两种缓存结构, 一种为两级缓存结构, L1 为 32KB, 64B 块, 4 路组相连; L2 为 512KB/1MB, 64B 块, 8 路组相连, 6-cycle 延迟. 另一种为三级缓存结构, L1 为 32KB, 64B 块, 4 路相连; L2 为 1MB, 64B 块, 8 路组相连, 6-cycle 延迟; L3 为 4MB/8MB, 64B 块, 16 路相连, 12-cycle 延迟. 流引擎放置在最后一级缓存之前, 使用基于地址的流检测, 每张流检测表存放 32 条表项. 检测单元步长流时, 达到稳态的阈值为 1, 即只要一次访问命中在某个表项中, 就成功结束检测. 而检测多单元步长流时, 设定阈值为 2, 即需要连续两次命中同一表项. 我们把 SAGA 算法应用在最后一级缓存上, 即两级缓存结构中的 L2 和三级缓存结构中的 L3. 这是因为这一级缓存的缺失延迟难以被处理器隐藏. 处理器通过发掘指令级并行度, 可以隐藏相当一部分缓存命中延迟, 而片外内存访问延迟相当高, 达到 100 到 200 个处理器周期, 处理器难以找到足够多可供执行的指令来隐藏这个延迟.

5.2 LRU 算法性能

我们首先给出 LRU 算法性能作为参考. 如图 4 所示为 LRU 算法在 SPEC2000FP 程序上的性能. L2 缓存的大小设为 512KB 及 1/4/8MB. 图中给出了各个程序在 L2 上的每千条指令缺失数 (Misses per Kilo Instructions, MPKI).

从图 4 中可见, 大部分 SPEC2000FP 程序是访存密集型的. 在 1/4/8MB 的 L2 上, 15 个程序中分别有 13/8/7 个程序的 MPKI 在 1 以上. 此外, 大部分 SPEC2000FP 程序的工作集都在 1MB 以上.

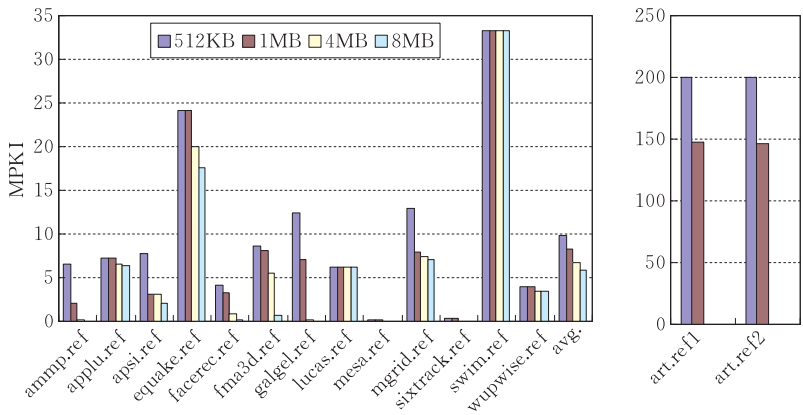


图 4 LRU 算法性能(我们将图分为两部分,因 art 的缺失较其他程序高出很多影响阅读.左图最后一列 avg 表示所有程序的平均)

5.3 流特性

在给出 LRU 算法性能之后,下面介绍 SPEC2000FP 测试程序集的流特性.图 5 描述了在我们的流引擎下检测出对流的访问的情况.在图中,我们把所有向 L2 的访问按照所形成的流的长度进行分类,分为非流、训练中、短流(该访问形成的流长度 ≤ 16)以及长流(该访问形成的流长度 > 16).我

们称某次访问为“训练中”,当这次访问没有被流引擎识别为稳态的流,但是其后存在一个或多个访问,与此次访问一同形成了流.一条单元步长的流需要 1 个元素作为训练元素,即位于流头的那一次访问.而一条多单元步长的流需要 2 个训练元素,第 3 个训练元素才能到达稳态.

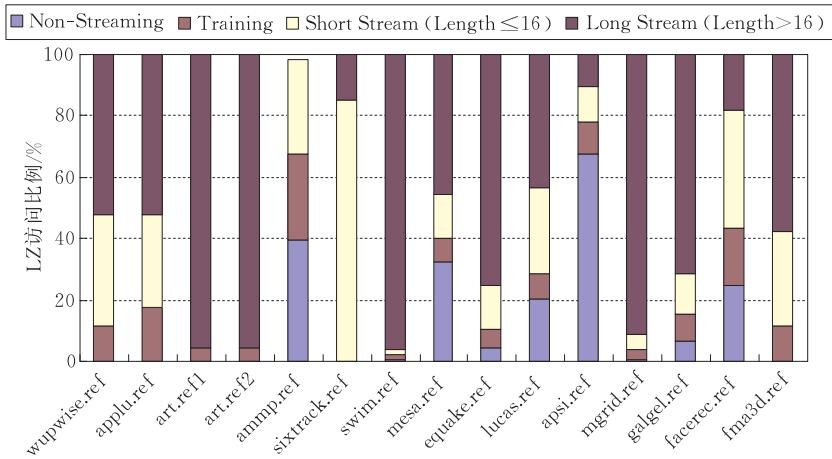


图 5 SPEC2000FP 程序的流特性

实验显示 SPEC2000FP 程序显示出很强的流访问特性,即有很高的比例访存操作是对流数据的访问.在 15 个程序中,有 9 个程序的流访问比例在 90%以上,有 14 个程序的流访问比例在 60%以上.

在 5.2 节和 5.3 节中,我们刻画了 SPEC2000FP 测试程序集对内存子系统的压力以及访存操作的流特性.我们观察到大部分测试程序对访存系统的压力较大,且流式访存行为很多.下面我们将测量通过发掘和利用程序的流访问特性,能够如何改善程序的访存压力,提高缓存命中率.

5.4 SAGA 算法的缓存性能

图 6 描述了对于 SPEC2000FP 所有程序,SAGA 算法在不同的缓存配置(最后一级缓存的容量)下的缓存性能.其中 512KB 和 1MB 容量使用的是两级配置,4MB 和 8MB 容量使用的是三级配置.比较的基准为 512KB 容量下 LRU 算法的缺失数量.最右侧一列是所有程序的算术平均.

从图 6 中可见,对比于基准 LRU 算法,SAGA 算法在 512KB 和 1MB 容量下能够明显的减少缓存缺失.例如,对于 galgel.ref 程序,SAGA 在 512KB 容量时减少了 20%的缓存缺失,在 1MB 容量时则

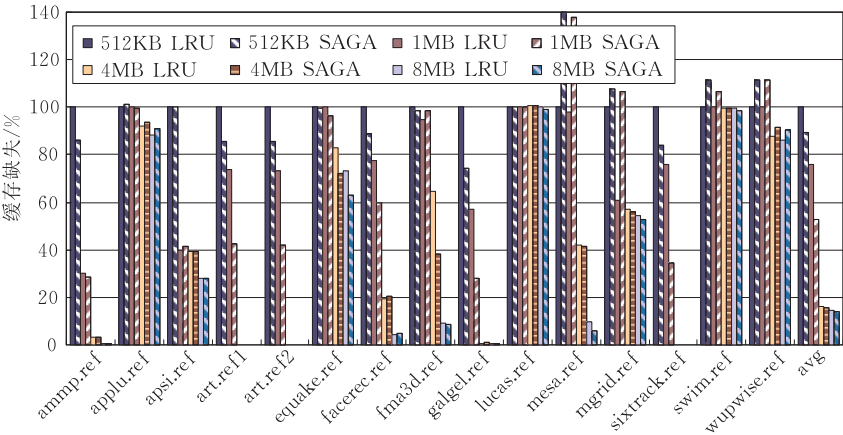


图 6 SAGA 算法性能(相对于 512KB LRU)

减少了 50%。对于所有程序,平均上,SAGA 在 512KB 容量时减少了 11% 的缓存缺失,在 1MB 容量时则减少了 31% 的缓存缺失。

对于 mesa.ref, mgrid.ref 和 swim.ref 程序, SAGA 在两级缓存结构中的性能比 LRU 差,但在三级缓存结构下,则比 LRU 性能好。这是因为,这一类程序对流数据的访问形成两级工作集。第一级工作集在 1MB 以内,这一级工作集内的流数据时间局部性非常好, SAGA 算法会使一部分流数据不分配在缓存中,失去被重用的机会,因此在 1MB 容量下 LRU 的性能比 SAGA 好。而第二级工作集则在 1MB 以上,这一级工作集中的流数据时间局部性不好,重用性较差。因此,三级缓存结构下, LRU 管理的 L2 以及 SAGA 管理的 L3 分别与对应工作集的数据访问行为相适应,总的性能比 LRU 要好。

SAGA 在 4MB 和 8MB 容量下减少的缓存缺失并不那么明显,分别为 5% 和 4%。这是由于 2MB 容量是 SPEC2000FP 程序的一个工作集大小。当

缓存容量大于 2MB 时,有的程序数据能装入缓存中。而另外一些程序则需要非常大的容量,例如 applu.ref、apsi.ref、swim.ref 和 wupwise 程序需要 128MB 以上, fma3d.ref、mgrid.ref 需要 64MB 以上, equake.ref 需要 32MB, 等等。实验使用的缓存容量远比这些程序的需求小,因此改善替换算法的潜力并不大。然而,随着工艺的进步,缓存容量逐渐增大, LRU 替换算法仍然有被改进的空间和需求。

5.5 对程序性能的影响

在以上的实验中我们基于功能模拟的实验方法,为了衡量 SAGA 对程序总体性能的影响,我们接下来使用基于时序模拟的实验方法。实验结果如图 7 所示。图中描述了在每一种缓存容量配置下, SAGA 相对于 LRU 的每指令周期数(Clock per Instruction, 或 CPI)的差异百分比。即

$$\text{差异} = \frac{CPI_{\text{LRU}} - CPI_{\text{SAGA}}}{CPI_{\text{LRU}}} \times 100\%,$$

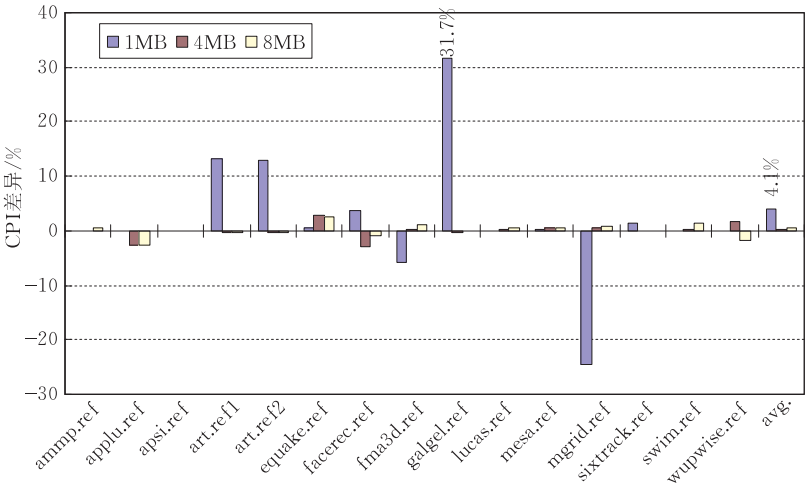


图 7 对整体性能的影响(SAGA vs. LRU)

图中最后一列是所有程序平均 CPI 的差异,即

$$\text{平均差异} = \frac{\overline{CPI}_{LRU} - \overline{CPI}_{SAGA}}{\overline{CPI}_{LRU}} \times 100\%,$$

其中 \overline{CPI} 为各个程序 CPI 的算术平均。

从图 7 中可见,总体上,在各种容量下 SAGA 算法都优于 LRU 算法。例如,在 1MB 容量下, SAGA 算法使程序平均获得 4.1% 的运行时间缩短量。在 4MB 以及 8MB 容量下,尽管差异不明显,但仍然观察到 0.3% 和 0.7% 的提高。

SAGA 对于一类流行为好而缓存容易发生抖动的程序,效果非常好。如 art.ref1、art.ref2 以及 galgel.ref,能够获得超过 10% 的运行时间缩短。其中 galgel.ref 运行时间缩短了 32%。

我们观察到当流数据的时间局部性非常好的时候, SAGA 不优于 LRU 算法。例如 mgrid.ref 程序,在 512KB 容量下 SAGA 比 LRU 性能要差许多。相关的原因我们在 5.4 节中已经进行了分析,此处不再赘述。

当程序工作集过大或过小时, SAGA 算法与 LRU 算法差异不明显。如在 8MB 缓存容量下,平均差异只有 0.7%。这是因为当工作集过小时,缓存完全能装入数据,缺失很少,而当工作集过大时,有限的缓存容量能改善的空间也非常有限。这与 5.4 节的分析是一致的。

6 结论与下一步工作

本文提出了一种新型的基于流特性的缓存分配算法 SAGA。SAGA 算法通过已有的流检测硬件发掘程序访存流信息,进一步动态调整流数据在缓存中的分配额,从而提高命中率,提高程序整体性能。实验结果表明, SAGA 算法的的总体性能优于 LRU,并且在某一类流式程序中性能提高非常大。此外, SAGA 算法的硬件开销非常低,对已有的体系结构只做了简单的修改,在延迟和面积上的开销是可以忽略的。

SAGA 算法有自身的局限性。首先, SAGA 必须依赖流检测硬件,不同的流检测硬件结构对 SAGA 的影响尚未评估。其次,虽然总体上优于 LRU,但 SAGA 对于某一类程序中时间局部性很好的流式数据处理仍不优于 LRU。因此, SAGA 算法在未来处理器体系结构中的应用要受到处理器自身流检测硬件结构以及处理器实际工作负载访存特点的限制。

我们的下一步工作包括几方面。一是继续发掘流式访存特性的更多有用信息,更有效地指导缓存替换策略。SAGA 已经为我们展示了处理器内存子系统不应孤立地看待每一个缓存块,而应当发掘缓存块之间的联系,来实现更好的管理。而流特性正是一种容易发掘、有使用潜力的缓存块间联系。第二,随着片上多处理器的兴起,多线程环境下的程序流行为和缓存管理成为下一个研究的目标。SAGA 的一个优点正是可以较容易地移植到片上多处理器的共享缓存上。在共享缓存上测量程序的流行为并指导缓存的管理策略,是我们正在进行的研究工作。最后,基于流检测引擎进行数据预取是非常成熟的优化技术,因数据预取与缓存替换策略属于两个方面的内容,故本文不作深入分析,但二者之间的联系和结合,是将来的工作之一。

参 考 文 献

- [1] Lin J, Jaleel A, Chen Y, Li W, Tang Z. Memory characterization of SPEC CPU2006 benchmark suite//Proceedings of the Workshop for Computer Architecture Evaluation of Commercial Workloads (CAECW), Co-Located with HPCA. 2008
- [2] Chen Y, Jaleel A, Li W, Lin J, Tang Z. Memory characterization of emerging recognition-mining-synthesis workloads for multi-core processors//Proceedings of the Workshop for Computer Architecture Evaluation of Commercial Workloads (CAECW), Co-Located with HPCA. 2008
- [3] Jouppi N. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers//Proceedings of the 17th Annual International Symposium on Computer Architecture. Seattle, WA, USA, 1990: 364-373
- [4] Palacharla S, Kessler R. Evaluating stream buffers as a secondary cache replacement//Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, 1994: 24-33
- [5] Mohan T, Supinski B, McKee S, Mueller F, Yoo A, Schulz M. Identifying and exploiting spatial regularity in data memory references//Proceedings of the ACM/IEEE Supercomputing Conference. Washington, DC, USA, 2003: 49
- [6] Hughes C, Grzeszczuk R, Sifakis E, Kim D, Kumar S, Selle A, Chhugani J, Holliman M, Chen Y. Physical simulation for animation and visual effects: Parallelization and characterization for chip multiprocessors//Proceedings of the 34th International Conference on Computer Architecture. San Diego, California, USA, 2007: 220-231
- [7] Hur I, Lin C. Memory prefetching using adaptive stream detection//Proceedings of the 39th Annual IEEE/ACM Inter-

national Symposium on Microarchitecture. Orlando, Florida, USA, 2006: 397-408

[8] Nesbit K, Smith J. Data cache prefetching using a global history buffer//Proceedings of the 10th International Symposium on High Performance Computer Architecture. Madrid, Spain, 2004: 96

[9] Megiddo N, Modha DS. ARC: A self-tuning, low overhead replacement cache//Proceedings of the 2nd USENIX Conference on File and Storage Technologies. San Francisco, CA, USA, 2003: 115-130

[10] Jiang S, Zhang X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance//Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. Marina Del Rey, CA, USA, 2002: 31-42

[11] Qureshi M, Jaleel A, Patt Y, Steely S, Emer J. Adaptive insertion policies for high performance caching//Proceedings of the 34th Annual International Symposium on Computer Architecture. San Diego, CA, USA, 2007: 381-391

[12] Basu A, Kirman N, Kirman M, Chaudhuri M, Martinez Jose. Scavenger: A new last level cache architecture with global block priority//Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. Chicago, IL, USA, 2007: 421-432

[13] Sinharoy B, Kalla R N, Tendler J M, Eickemeyer R J, Joyner J B. POWER5 system microarchitecture. IBM Journal of Research and Development, 2005, 49(4/5): 505-521

[14] Inside Intel Core microarchitecture and smart memory access. White paper, Intel Corporation, 2006

[15] Luk C K, Cohn R et al. Pin: Building customized program analysis tools with dynamic instrumentation//Proceedings of the Programming Language Design and Implementation (PLDI). Chicago, IL, USA, 2005: 190-200

[16] Austin T, Larson E, Ernst D. SimpleScalar: An infrastructure for computer system modeling. Computer, 2002, 35(2): 59-67

[17] Perelman E, Hamerly G, Calder B. Picking statistically valid and early simulation points//Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques. New Orleans, LA, USA, 2003: 244-255



CHEN Yu, born in 1981, Ph. D. candidate. His research interests focus on cache hierarchy on multi-core processors.

LIN Jun-Min, born in 1979, Ph. D. candidate. His re-

search interests include computer architecture, and multi-core processors.

QIAO Lin, born in 1972, associate professor. His research interests include computer architecture and compiler optimizations.

TANG Zhi-Zhong, born in 1946, professor, Ph. D. supervisor. His research interests include computer architecture and advanced compiler techniques.