计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2017,26(10):36−43 [doi: 10.15888/j.cnki.csa.005990]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

CMFSim: 高可配可扩展的缓存微架构功能模拟器①

宋双洋 [1,2], 赵　姗 [1], 杨秋松 [1]

[1](中国科学院 软件研究所, 北京 100190)
[2](中国科学院大学, 北京 100049)

摘　要: 作为提高 CPU 读取和存储数据的效率, 弥补与主存之间存取速度差距的有效策略, CPU 的缓存 (Cache) 充分利用其对数据使用的局部性原理, 对最近或最常使用的数据进行暂存, 对 CPU 的性能起着决定性作用. 缓存的微架构正是决定缓存性能的关键性因素. 然而, 现代先进的 CPU 缓存都具备极为复杂的结构, 存在多种策略、多种硬件算法和多个层级等不同维度的设计, 从硬件上直接设计和论证不仅耗时而且成本很高, Cache 微架构模拟器正是用软件方法对硬件微架构进行模拟和仿真. 设计一款结构优良的缓存, 对不同微架构进行评估, 是一件具有深远意义的工作. 本文从硬件结构出发, 设计实现了一款多级、高可配、高可扩展的缓存微架构功能模拟器 CMFSim (Cache microarchitecture functional simulator), 实现了常见的缓存策略和硬件算法, 可以进行给定配置下的缓存功能的模拟, 从而分析配置参数与缓存性能间的关系.

关键词: 多级 Cache; Cache 微架构; Cache 模拟器

引用格式: 宋双洋,赵姗,杨秋松.CMFSim: 高可配可扩展的缓存微架构功能模拟器.计算机系统应用,2017,26(10):36–43. http://www.c-s-a.org.cn/1003-3254/5990.html

CMFSim: A Highly Configurable and Extensible Cache Microarchitecture Functional Simulator

SONG Shuang-Yang[1,2], ZHAO Shan[1], YANG Qiu-Song[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: As an effective strategy to improve the efficiency for CPU reading and writing, and to fill the speed gap between CPU and the main memory, the cache in CPU makes the best of the locality theory by storing the latest or the most frequently used data. It dominates the performance of a CPU, and the microarchitecture of the cache, however, dominates the cache performance. The modern advanced cache commonly constructed with very complicated structures, contain multiple cache strategies, hardware algorithm and multi-level design, making it expensive to design and verify directly with hardware for time as well as money. Thus, it is far-reaching to simulate the hardware microarchitecture by software modeling. Cache microarchitecture simulator exactly assists the design or the evaluation of an excellent cache. In this article, a highly configurable and extensible cache microarchitecture functional simulator CMFSim is developed on the basis of hardware structure. It implements the common cache strategies and hardware algorithm, which can conveniently simulate the cache microarchitecture for the given configuration and analyze the performance with the specified parameters.

Key words: multi-level Cache; Cache microarchitecture; Cache simulator

现代信息技术的发展中计算机始终处于核心地位, 不论是移动设备、嵌入式设备, 还是个人计算机、企

① 基金项目: 国家“核高基”科技重大专项 (2014ZX01029101-002)
收稿时间: 2017-01-19; 采用时间: 2017-02-15

业服务器乃至超级计算机. 所有这些计算机系统的核心部件就是其处理器[1]. 一款 CPU 的性能极大程度上决定了计算机系统的信息处理能力. 设计和研发 CPU 的最新微架构, 不论是从信息技术产业还是从国家战略上来看, 不仅能推动整个信息技术的前进, 而且能在不断发展的过程中保持自身的核心竞争力. CPU 的缓存作为用来弥补不断提高的 CPU 运算速度与主存存取速度间差距的特殊结构, 充分利用了 CPU 对数据和指令使用时的时间局部性和空间局部性原理[2], 将最近或最常使用的数据进行暂存, 从而获得处理器运算性能上极大程度的提升. 现代 CPU 的缓存经过了长足的发展, 从普通的读写策略、替换策略到不同层级及其包含特性, 乃至数据一致性问题都得到了长足的发展并提出了很多设计要点, 缓存的微架构正是从这些方面入手来构建完善的缓存系统.

在体系结构研究中, 软件模拟是一种非常有效的构建设计原型并进行评估的方式, Cache 微架构的设计当然也不例外. 在充分理解 Cache 硬件结构的基础上, 使用软件方法进行建模, 能够快速论证不同微架构在设计上的优劣, 及对性能产生的影响; 同时, 还能采用控制变量的方式对感兴趣的设计因素进行着重考察, 发现和分析出其与性能间的一般性关系, 从而能够反过来对原型设计给出指导和建议. 虽然软件模拟器使用广泛, 但单独针对 Cache 的模拟工具很少, 深藏于开源的全 CPU 模拟器中难以独立的去研究 Cache, 或只是在公司或机构内部使用而无法获取; 另外, 由于 Cache 结构的复杂性, 软件模拟一般只有某个侧重点, 可以完成所有 Cache 的功能实现模拟, 也可以考察精确的时序, 二者同时模拟会带来软件设计过于复杂以及模拟运行的速度过于缓慢的问题.

本文基于现代 CPU Cache 的硬件结构, 设计实现了一个高可配、可扩展的独立 Cache 功能模拟器 CMFSim, 对不同缓存结构、写策略、替换策略、包含特性等多个维度提供了细粒度的配置, 着重功能模拟, 针对单核情形下实现, 并与 DRAMSim2 主存模拟模块整合, 提供了统一的访存接口来进行功能模拟.

1 相关工作

现有的开源模拟器项目都集中在功能全面且庞大的单处理器的全系统模拟, 以进行整个 CPU 微架构的仿真与模拟. gem5[3]就是这样的开源全系统微架构模拟器的典型代表, 提供了系统调用模拟与全系统模拟两种工作模式, 提供 x86、ARM、MIPS、Power、Alpha 等多种指令集架构, 同时支持目录式和广播式两种 cache 一致性协议, 提供基于 TCP/IP 的网络访问模拟等诸多特性. gem5 通过社区共同维护和开发, 功能全面体系庞大, 对于进行全系统模拟而言是一个很好的工具, 但主要问题是设计结构过于复杂, 难以拆解用于独立研究某个模块.

Zesto[4]是另一个新一代的全系统模拟器, 这是一个在单周期水平上进行 CPU 微架构的模拟, 在硬件上最接近当前最先进的高性能 x86 处理器, 主要目标就是在非常底层的微架构设计上进行 x86 指令集架构下时钟周期精确的模拟. 但是, 由于过于精确的模拟, 模拟器的运行速度非常缓慢, 对于运行模拟器的巨量指令流来说其运行时间难以接受. 同时, Zesto 也是一个全系统模拟器, 也存在难以拆解的困难.

除此之外, 还有 SimpleScalar[5]由多种模拟工具构建的模拟工具家族, 这些工具已广泛使用和流行了数十年时间, 基于该工具集和衍生的模拟工具遍布于大量的学术文章中. 但随着处理器硬件工艺的不断发展, 很多模拟工具特别是时钟周期精确模拟工具的硬件设定或者假设还停留在二十年前的水平[4]. MARSS[6]是最新的开源全系统模拟器, 提供了针对 x86 架构下单核与多核的顺序与乱序支持的时钟精确模拟, 可以直接运行操作系统和 x86 架构的二进制应用程序, 是一个从处理器到内存、磁盘到其他外部设备, 从操作系统到共享库和应用程序的超全模拟器.

相对于全面、庞大的单处理器全系统模拟器而言, 针对多处理器场景也存在较多模拟器项目. 上世纪八十年代提出的高可扩展的多处理器模拟器[7], 主要模拟了各个处理器通过时序共享的单一总线进行主存一致性访问的架构. Augmint[8]填补了 CISC 架构和指令混合的模拟器空缺, 是一款执行驱动的多处理器模拟工具包. MICA[9]是新一代主要用来进行分布式共享内存的多处理器场景的模拟, 运行于多个廉价机器组成的集群上, 使用应用程序的执行流来作为输入, 提供了各核调度与内存互联的算法与接口.

多处理器模拟器主要用于分布式架构下的系统模拟, 作为体系架构的模拟在本质上与单处理器相似.

现代 CPU 体系结构的各个模块的设计都非常复杂, 各个模块的功能和结构很大程度上是可以独立考察的, 但全系统模拟器重点在整个系统的整体模拟, 因此各个独立的模块耦合过于紧密, 被淹没与整个模拟

器中, 难以去独立构建与研究.

2 系统设计与实现

2.1 Cache 硬件结构

现代主流 Cache 的基本硬件结构主要包括这几个方面的内容: Cache 组织结构、地址映射、替换策略、写策略、多级包含性等, 下面详细阐述.

2.1.1 组织结构与地址映射

Cache 的基本硬件结构是由 SRAM 构成的阵列, 阵列中每个基本存储块称为一个 cache line, 所有对 Cache 的操作都是以 cache line 为基本单位进行. 由于 Cache 的大小远不及主存和外部存储设备, 需要进行映射. 对于 Cache 中的 cache line, 有两种常见的组织方式: 一个地址对应的数据只能存储到固定的一个 cache line 中, 称为直接映射 (direct-mapped); 一个地址对应的数据可以存储在任意 cache line 中, 称为全相联 (fully associative). 前者对 Cache 的空间利用率很低, 有可能某些 cache line 一直没有数据存储和读取, 容易发生冲突, 但实现上最简单; 后者能充分利用不常用的 cache line, 空间利用率很高, 不容易发生冲突, 但硬件实现上很复杂. 因此, 将二者结合, 兼顾空间利用率、冲突概率和实现难易程度, 就是现代 Cache 主要的组织结构——多路组相连 (set associative).

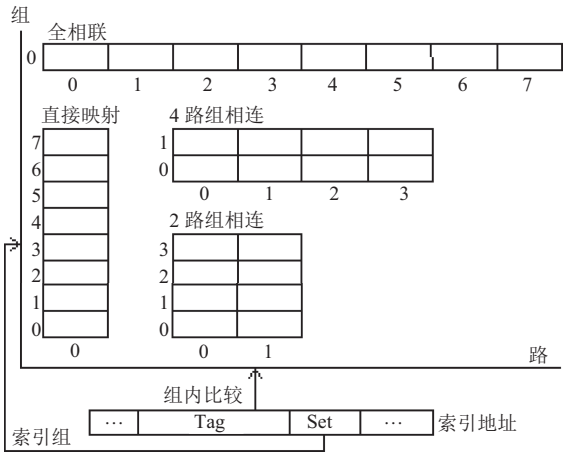

图 1 8 个 cache line 的不同组织结构示意

硬件上为了支持一个周期处理多个请求, 提出了 bank 的概念, 在上述多路组相连的 Cache 结构基础上构建为一个 bank, 然后多个 bank 再构建为一级 Cache.

在进行 cache line 查询的时候, 需要使用访问地址进行索引, 图 2 是一个 64 位物理地址下各索引划分的实例, 从低位到高位有 cache line 内的偏移、组索引、组内标签, 若分为多个 bank 则还需要放置 bank 索引.

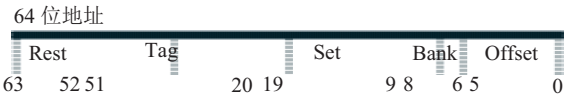

图 2 物理地址索引划分实例

为了索引查询的方便, 除 tag 外, 一般各个索引地址的比特数如果为 N, 那么对应部分总数就是 2^N. 图 2 中组数 (Set) 共 11 比特, 则总组数为 2048 组; Bank 数占用 3 个比特, 则一共有 8 个 bank; cache line 的大小一般都为 64, 则 Offset 占用 6 比特. 每次 Cache 操作都基于地址索引出各部分, 是进行相应操作的基础.

2.1.2 替换策略

由于 Cache 空间有限, 当数据填充满之后, 或者需要填充的 cache line 被之前的占用了, 那么就会发生替换, 为最新的请求开辟新的位置. 对于多路组相连的结构, 每个组通过组索引只能在组内替换, 但同一个组内的多路是可以任意选择的, 这里就存在多种替换方式的选择了, 常见策略如下:

① 随机替换: 随机选择一路 cache line 剔除;

② 先进先出[10](FIFO): 维护每一路写入的顺序, 按照先进先出的方式剔除最先进入的;

③ LRU[10]: 维护对每一路最近访问的次数, 替换时选择最近使用最少次数的;

④ Tree-based PLRU[11]: 使用一个二叉树结构来维护各路的优先级, 替换时剔除最不优先的.

2.1.3 写策略与包含性

对于 Cache 的写操作, 也存在不同的策略. 这些策略主要作用于多级 Cache 结构中, 使得上下各级 Cache 所执行的操作会有不同.

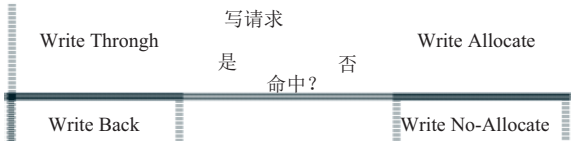

图 3 写请求的不同策略

对于写请求命中的情况下, 可以选择 write-through 直写方式也就是不写入当前 Cache 直接写入下一级, 这样当前 Cache 中数据失效, 实现起来简单, 但需要花费的周期较长; 也可以选择 write-back 方式, 写入当前

Cache 中并做好标记, 等写入的这个 cache line 发生替换时或者需要读取它的数据时, 再写入下一级, 这样可以合并最近的多次写入, 提高效率.

对于写缺失的情况下, 按照是否先分配一个 cache line 之后在进行 Cache 写操作分为 write allocate 和 write no-allocate 两种策略, 其中 write allocate 方式虽然硬件实现复杂一些但依然是绝大多数设计的选择.

当引入了多层 Cache 之后, 还有另一个需要考虑的问题就是各级 Cache 之间的包含关系, 分为 inclusive 和 exclusive 两种, 前者要求强制包含, 上一级的每个 cache line 都会在下一级存在, 为了维护这种包含性需要额外的操作而花费时钟周期, 但各级之间的数据一致性维护相对简单一些; 后者则没有这种要求, 可以上一级出现的 cache line 并不存在于下一级中, 此时对于各级 Cache 数据一致性的维护会比较复杂.

2.2 软件建模

基于前文所述的 Cache 硬件结构, 从软件建模的角度出发, 设计和定义了各个硬件结构的软件抽象对应的数据结构, 并在此基础上通过指定的配置参数, 选定相应的替换算法、写策略、包含性, 并根据相应参数实例化指定的 Cache 结构, 构建起整个 CMFSim 功能模拟器. 首先, 对一个单独 Cache 的基本软件模拟的系统架构如图 4 所示.

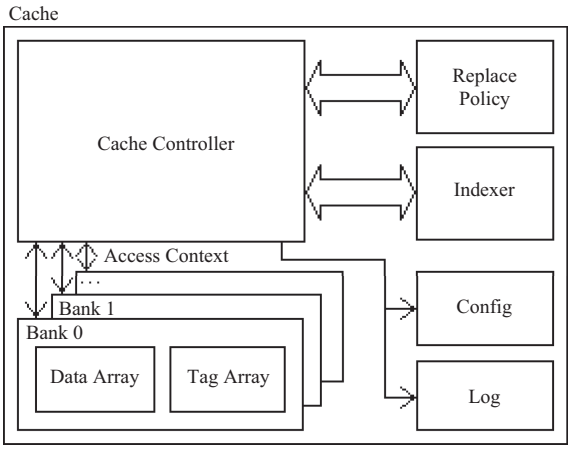

图 4 单级 Cache 软件建模架构

一个完整的 Cache 的核心为一个 CacheController, 封装为一个类, 负责控制该 Cache 的所有行为, 其所需的主要数据结构就是 DataArray 和 TagArray, 前者用来存储实际缓存的 cache line 数据, 后者则保存每个 cache line 相对应的标签、状态、有效性等信息. 与此同时如果有多个 bank, 则据此构成多个 bank, CacheController 负责控制到某个 bank 的访问, 将每个请求都以 AccessContext 对象的方式进行包装, 该对象是内部各个接口之间传递数据的通用数据结构. 每个 cache 默认为一个 bank, 通过读取用户在配置文件中给定的配置, 来实例化每个 DataArray 和 TagArray 的大小, 同时构造出相应数目的 bank; 与此同时, 还根据指定的替换算法进行选择, 决定构建相应的 ReplacePolicy 对象; 最后, 还需要依据配置中给定的索引划分起始比特与长度, 来使用 Indexer 类进行物理地址各部分的划分. 另外, 对运行过程和结果中的各项关键信息需要进行日志记录, 如总的读写次数、命中失效次数、读取和写入的数据量等信息.

2.2.1 DataArray 与 TagArray 抽象

对于 Cache 的行为可以从软件角度抽象出读和写两种接口, 分别对 DataArray 和 TagArray 两个数据结果进行操作, 这也正好对应了 Cache 硬件上的基本结构. 其详细定义见表 1.

表 1 DataArray 与 TagArray 主要接口定义

类别		定义
DataArray	Entry	CacheLine {vector<uint8_t>;};
	读	bool read(uint32_t set, uint32_t way, CacheLine &cl);
	写	bool write(uint32_t set, uint32_t way, const CacheLine &cl);

TagArray	Entry	TagArrayItem {
		addr_t tag;
		bool valid;
		bool dirty;
		};

	读	bool read(uint32_t set, addr_t tag);
	写	bool write(uint32_t set, uint32_t way, addr_t tag);
	状态	bool get/setX(uint32_t set, addr_t tag, bool ret); X={valid, dirty} bool get/setXforcely(uint32_t set, uint32_t way, bool ret); X={valid, dirty} uint32_t busy(uint32_t set);

读写接口均以 bool 方式作为返回标志以表明是否成功, 数据均以引用方式传递. DataArray 和 TagArray

都是一个类似“二维数组”的结构, 来支持多路组相连的 cache line 组织结构, 第一维为组数, 第二维为路数. 每个元素, 也就是表 1 中 Entry 的定义, 代表了“数组”中每一项的具体内容. 对于 DataArray, 其读写接口只提供组索引和路索引的方式进行执行, 因为这些操作是以 TagArray 的 Tag 和状态进行比较和判定为基础的, 直接操作的就是 Cache 中的数据. 但 TagArray 首先提供了针对 Tag 的读写接口, 同时提供了针对状态位的读写操作中, 这类状态位的操作接口以 get 与 set 开头来命名, 可以按照 Tag 进行匹配后执行, 也可以用路索引进行强制设置的方式执行, 从而满足不同场景下对 TagArray 状态的设置需求. 另外, 还提供了判断某一组 cache line 是否是全部被占用的“busy”接口, 可以用来判断是否需要对当前组内的某个 cache line 进行替换操作.

2.2.2 替换策略与索引划分

图 4 中的 ReplacePolicy 类维护了每个 cache line 相对应的状态, 这些状态也是一个二维数组的结构. 对于给定的某一组, 通过索引第一维, 得到该组内各个 cache line 的一维状态数组, 前文所述的四种替换算法均能以此为基础实现, 通过建模和验证, 以 C++模板方式实现, 只需要提供如下两个通用的接口:

① void update(vector<uint32_t>&ref, uint32_t w);

② uint32_t get(vector<uint32_t> &ref);

传入的参数 ref 表示当前访问的 cache line 所在组的各 cache line 的状态数组, update 接口用来更新指定路索引的 cache line 状态, get 接口用来获取当前组内需要剔除的 cache line 的路索引. 这两个接口能够兼容前文所述的四种算法的实现. FIFO 替换算法的 update 接口描述如下:

```
FIFO_UPDATE(ref, way):
1   let result=ref.size
2   for i=0 to result-1:
3      if ref[i]==way:
4         result=i+1
5         break
6   for j=result-1to 1:
7      ref[j]=ref[j-1]
8   ref[0]=way
```

ref 数组中每个值为路索引, 第一个元素保存的是上一次访问的, 第二个为上上一次访问的, 以此类推, 最后一个元素保存的是最早访问的. 每次更新的时候找到当前访问的路 (第 2 到 5 行), 将数组中在其之前的向后移动一个元素 (第 6 到 7 行), 然后把当前路索引放在数组第一个位置 (第 8 行). 调用 get 接口进行获取替换的路索引时, 直接返回数组最后一个元素的值即可. 对于 PLRU 算法, 则只需要使用状态数组的前 N-1 个元素 (N 为每一组的路数), 模拟一个完全二叉树, 其 update 和 get 接口的实现算法描述如下:

```
PLRU_UPDATE (ref, way):
1   let n=ref.size, deep=log2(n-1)
2   let i=0, j=deep-1
3   while i < n-1 and j >=0:
4      bit=(way >> j) & 1
5      if bit==ref[i]:
6         ref[i]=1-ref[i]
7      if bit==0:
8         i=i * 2+1
9      else:
10        i=i * 2+2
PLRU_GET (ref)
1   let n=ref.size, way=0
2   for i=0 to n-1:
3      way=2 * way+ref[i]
4      if ref[i]==0:
5         i=i * 2+1
6      else:
7         i=i * 2+2
8   return way
```

每个元素要么为 0 要么为 1, 0 往左子树走 1 往右子树走, 通过与二叉堆相似的方式, 在进行更新的时候把路索引的二进制表示在树中进行遍历, 将遍历路径上的每一个元素的值进行反转 (第 6 行). 调用 get 接口就直接依据树中每个元素的值进行遍历得到相应的二进制值就是需要替换的路索引 (第 3 行). 对于, LRU 和随机替换, 均能在状态数组的基础上实现, 在此不予赘述.

与此同时, CacheController 需要使用 Indexer 类对输入请求的物理地址进行划分, 需要执行的就是对给定的物理地址进行按位操作即可, 划分的依据是 Config 中给出的各部分起始比特和长度.

2.2.3 CacheSystem 与完整访存系统

在构建完整的单级 Cache 之后, 就可以着手构建

多级的 Cache 系统. 图 5 描述了一个指定的典型三级 Cache 系统与 DRAMSim2 构建的完整访存系统架构. 多级 Cache 之间目前仅支持对相互的包含性的设置. CacheSystem 的架构可以通过不同的配置来自行设置和定义, 配置文件以 ini 格式的语法描述并进行解析, 图 5 中的三级结构配置以及 L3 的索引配置如下:

```
[STRUCTURE]
levels=3
# 0: inclusive, 1: exclusive
level_share_1=0
level_share_2=0
cacheline=64
[STRUCTURE-L1]
banks=1
sets=512
ways=1
[STRUCTURE-L2]
banks=1
sets=1024
ways=4
[STRUCTURE-L3]
banks=4
sets=2048
ways=8
……
[INDEX-L3]
offset_start=0
offset_length=6
bank_start=6
bank_length=2
set_start=11
set_length=11
tag_start=17
tag_length=32
rest_start=49
rest_length=15
……
```

其他完整配置类似, 还包括各级的替换策略、写命中与写失效的策略等, 使用者可以依据配置文件的语法进行自定义配置, 来构建不同微架构的 CacheSystem.

CacheSystem 通过读取和解析配置文件, 依次传递给各单级 Cache 的 Config 去构建每一级 Cache, 得到的多级 Cache 实例被 CacheSystem 类操纵和使用. CacheSystem 最终在整个系统中以单例形式呈现, 并整合了开源的主存模拟模块 DRAMSim2, 对外提供了统一的访存读写接口来调用.

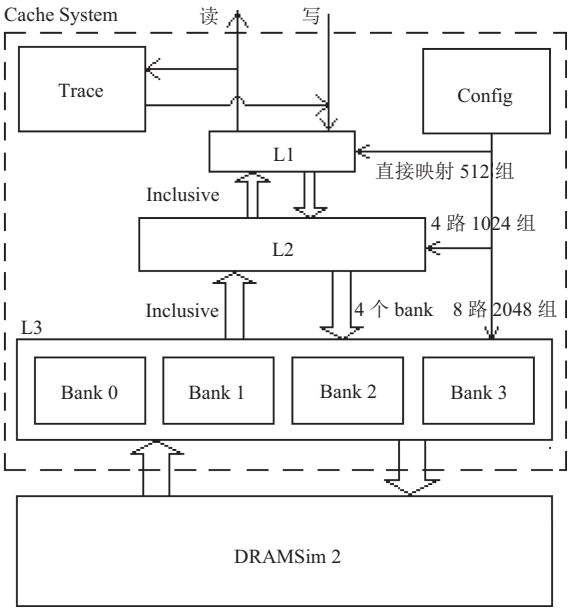

图 5 CacheSystem 与 DRAMSim2 构建的访存系统

整个系统除了提供以动态链接库的形式作为独立模块供外部调用之外, 图 5 中还给出了其提供的 Trace 模块, 可以一起编译为一个可执行文件, 来接收应用程序进行访存操作的 trace 流文件作为输入, 单独进行 Cache 微架构的研究和功能验证.

3 测试与结果

基于本文实现的 CMFSim 功能模拟器, 编译为接收访存操作的 trace 为输入的可执行文件来进行测试. 测试平台为 64 位 Ubuntu 14.04 操作系统, Intel i5 双核 CPU, 1 GB 内存.

首先使用二进制分析平台 Pin[16], 实现了获取应用程序访存操作 trace 的工具, 对 Ubuntu 系统下的 “/bin/ls”程序的访存操作 trace 进行了捕获, 共得到了 34 万多条访存操作记录. 获取的 trace 记录示例如下:

```
W 0x7ffc989b2a88 8
W 0x7ffc989b2a80 8
W 0x7ffc989b2a70 8
R 0x7f35b5afbe70 8
W 0x7f35b5afbc98 8
```

R 0x7f35b5afc000 8

…

其中的第一列的“R”和“W”分别代表读请求和写请求, 紧接着的一列为当前请求的物理地址, 最后一列为一个整数, 代表此次请求的数据长度.

3.1 Cache 结构的验证

以上述 trace 为输入, 首先构建了测试不同 Cache 结构与命中率关系的实验, 分别配置不同大小的组数来统计命中率.

图 6 为测试结果, S 表示组数 (Set), W 代表路数 (Way), 可以看出随着组数不断增加, 命中率是不断提高的, 但是当增加到一定程度, 命中率提高的幅度就不太明显了, 当命中率的提高不足以弥补硬件成本的增加时就需要进行折中, 这对特定结构的 Cache 设计具有验证和指导意义.

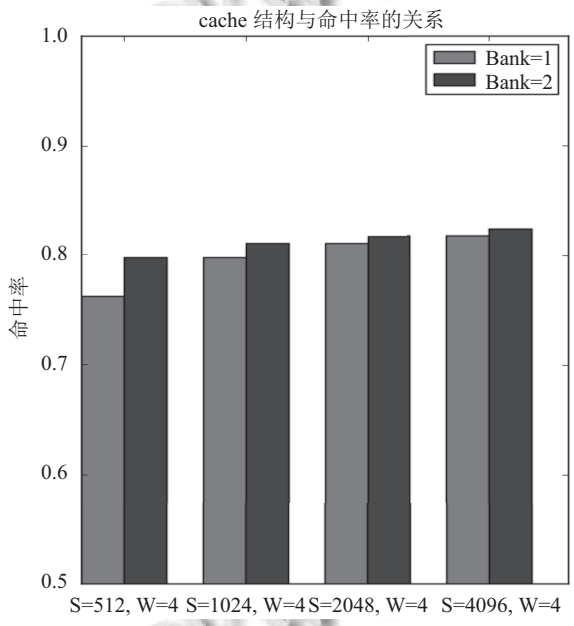

图 6 Cache 结构与命中率的关系

3.2 组索引划分的评估

对于 Cache 使用物理地址进行划分索引, 其划分位置非常深刻地影响了命中率. 因此设计了如下的实验: 从物理地址的第 6 比特开始依次往后滑动, 分别统计不同组索引位置下的命中率, 从图 7 中可以很明显的看出, 随着组索引逐渐向高比特移动命中率逐渐上升, 在第 11 比特进行划分的组索引的命中率最好, 而且两种不同 Cache 结构都是如此, 当超过 11 比特再往高比特移动命中率就开始下降了. 但是, 对于 Cache 最终设计方案, 需要综合考虑不同类型的程序, 若是针对某一特定领域的设计可以进行特殊设计, 否则针对通用领域需要进行折中, 该 Cache 模拟器可以对这些设计进行验证和评估.

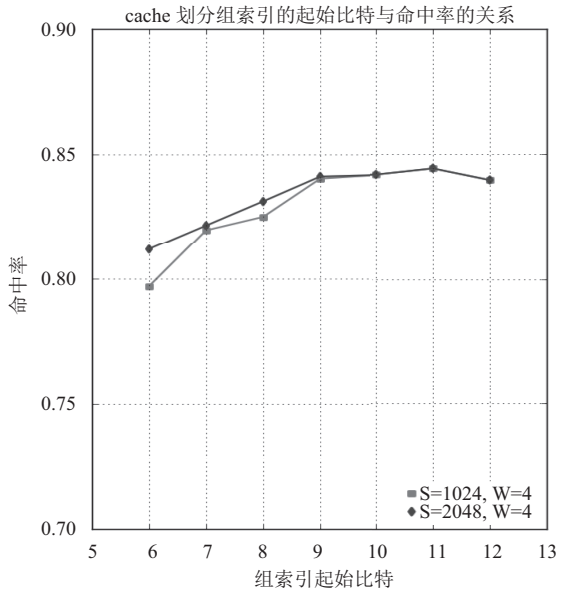

图 7 Cache 划分组索引与命中率关系

3.3 多级 Cache 系统的测试

按照图 5 所标注的配置, 构建了一个三级的 Cache 系统, L1 配置为 4 路 512 组, L2 配置为 4 路 1024 组, L3 配置为 4 个 bank, 每个 bank 8 路 2048 组, L2 包含 L1, L3 包含 L2, 其中 L1、L2 为 write through 与 write no-allocate, L3 为 write back 与 write allocate. 使用这些配置构建的访存系统, 运行相同的访存 trace, 统计的 L1、L2、L3 各自的命中率见图 8.

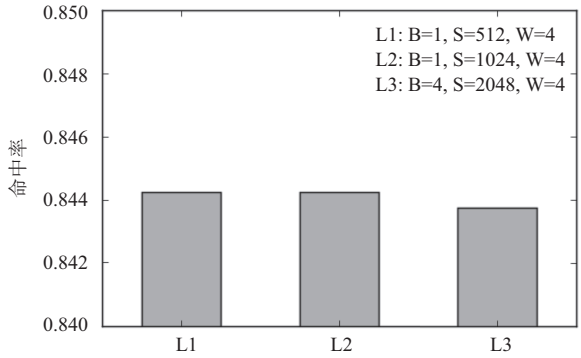

图 8 三级 Cache 系统的命中率

与图 7 中的结果对比, L1 和 L2 都达到了最优命中率但 L1、L2 的组数明显要小很多, 可以看出这种多级配置下的缓存系统对命中率的提升, 在硬件设计上

可以增加 L1、L2 的硬件成本, 而减小 L3 的硬件成本, 从而可以在保证命中率的基础上降低整体的成本.

4 总结

针对处理器体系结构中的缓存这一独立模块的模拟器大都深藏于大量的全功能模拟器中, 本文以现代先进 CPU 缓存的硬件结构为依托, 从软件建模与仿真的角度, 设计实现了一个功能完善、高可配、模块独立的单核下的多级缓存功能模拟器 CMFSim, 并与开源的 DRAMSim2 整合, 封装了统一的访存接口方便调用, 能够从功能验证的角度完成缓存微架构的模拟. 通过实验分析, 验证了缓存微架构设计与命中率之间的关系, 能够表明该独立的缓存功能模拟模块的有效性, 可以作为独立模块去构建完整的全功能模拟器.

本文仅针对单核下的功能模拟为主要目标, 因为只用来进行功能验证和性能评估, 对于多核情况只需进行相应的横向扩展, 比如以多线程方式将 CMFSim 运行多个实例的方式即可进行功能模拟. 在未来的研究与开发中, 将考虑多核下的精确时序模拟与数据一致性的模拟, 实现更加精细化的微架构建模与评估.

参考文献

1 González A, Latorre F, Magklis G. Processor Microarchitecture: An Implementation Perspective. Morgan & Claypool, 2010.

2 Patterson DA. Computer Architecture: A Quantitative Approach. 5th ed. Morgan Kaufmann, 2011.

3 Binkert N, Beckmann B, Black G, *et al*. The gem5 simulator. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1–7. [doi: 10.1145/2024716]

4 Loh GH, Subramaniam S, Xie YJ. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. Proc. of IEEE International Symposium on Performance Analysis of Systems and Software. Boston, MA, USA. 2009. 53–64.

5 Austin T, Larson E, Ernst D. SimpleScalar: An infrastructure for computer system modeling. Computer, 2002, 35(2): 59–67. [doi: 10.1109/2.982917]

6 Patel A, Afram F, Chen SF, *et al*. MARSS: A full system simulator for multicore x86 CPUs. Proc. of the 48th ACM/EDAC/IEEE Design Automation Conference. New York, NY, USA. 2011. 1050–1055.

7 Papamarcos MS, Patel JH. A low-overhead coherence solution for multiprocessors with private cache memories. ACM SIGARCH Computer Architecture News, 1984, 12(3): 348–354. [doi: 10.1145/773453]

8 Nguyen AT, Michael M, Sharma A, *et al*. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. Proc. of 1996 IEEE International Computer Design: VLSI in Computers and Processors. Austin, TX, USA. 1996. 486–490.

9 Hsiao HC, King CT. MICA: A memory and interconnect simulation environment for cache-based architectures. Proc. of the 33rd Annual Simulation Symposium. Washington, DC, USA. 2000. 317–325.

10 Dan A, Towsley D. An approximate analysis of the LRU and FIFO buffer replacement schemes. Proc. of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. Boulder, Colorado, USA. 1990.

11 Smith MB, Tresidder MJ. Pseudo-LRU cache memory replacement method and apparatus utilizing nodes. U.S., US5594886A. 1997-01-14.

12 Dubois M, Annavaram M, Stenström P. Parallel computer organization and design. Cambridge: Cambridge University Press, 2012.

13 Jacob B, Ng S, Wang D. Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann, 2007.

14 Drepper U. What Every Programmer Should Know About Memory. Red Hat, Inc., 2007.

15 Rosenfeld P, Cooper-balis E, Jacob B. DRAMSim2: A cycle accurate memory system simulator. IEEE Computer Architecture Letters, 2011, 10(1): 16–19. [doi: 10.1109/L-CA.2011.4]

16 Luk CK, Cohn R, Muth R, *et al*. Pin: Building customized program analysis tools with dynamic instrumentation. Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. Chicago, IL, USA. 2005. 190–200.