Intel®64体系结构的数据预取机制及效果

董钰山 李春江
(国防科学技术大学计算机学院软件研究所 长沙 410073)

摘 要 数据预取是为缓解微处理器与DRAM之间速度差异而出现的隐藏访存延迟的方法。当前Intel各系列处理器都采用多种预取机制来加速数据和代码向Cache的移动,从而提升程序的性能。通过对Intel®64体系结构存储层次的分析,剖析了X86/X64体系的数据预取机制,包括硬件预取和软件预取,并且分析了编译器对软件预取机制的支持。最后测试了Intel®64体系结构数据预取对科学计算程序中紧嵌套循环性能的影响,总结出了影响数据预取有效性的几个因素。此项工作对在Intel平台上进行循环数组预取优化有指导意义。

关键词 Intel 64,Cache,硬件预取,软件预取,GCC,ICC
中图法分类号 TP314 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.5.006

Mechanism and Capability of Data Prefetching in Intel®64 Architecture

DONG Yu-shan LI Chun-jiang
(Software Institute,School of Computer Science,National University of Defense Technology,Changsha 410073,China)

Abstract Data prefetching is an approach to reducing cache miss latencies,which can appropriately fill the speed gap between the microprocessor and DRAM. Recently,Intel processor families employ several prefetching mechanisms to accelerate the movement of data or code to Cache,and improve performance. By a brief analysis of the memory hierarchy of Intel®64 architecture,data prefetching mechanism of X86/X64 architecture,including hardware prefetching and software prefetching,was deeply dissected,and then the compiler support for software prefetching mechanism was analyzed. After testing the performance of data prefetcher of Intel®64 architecture for nested loop,we concluded several factors affecting the effect of data prefetching. These works provide a valuable contribution for the research and development of the loop-array-prefetching optimization on the Intel platform.

Keywords Intel 64,Cache,Hardware prefetching,Software prefetching,GCC,ICC

1 引言

存储器是"冯·诺依曼"计算机体系的重要组成部分。然而,长久以来,存储器访问速度的增长总是不及处理器性能的提高。虽然基于程序局部性原理的层次式存储结构有效地弥补了性能的损失,且自2005年以来单核心处理器的工作频率不再提高,但二者的性能差距还是很大[1]。近些年,单端信号所提供的数据传送带宽受到了各种制约,使得差分信号闪亮登场,然而差分信号的使用却进一步扩大了访问延时[2]。另一方面,多核多线程处理器的发展,进一步提高了访存带宽需求。处理器与存储器二者性能的不一致性使得具体应用程序根据存储结构作出相应调整变得尤为重要,其主要方法有:1)大量的数据存取从处理器Cache(高速缓存)中得到满足;2)尽量利用高内存带宽,从而有效地隐藏访存延时。

预取是一种将访存和计算重叠起来,通过隐藏访存延迟来提升访存性能的重要手段。目前,Intel®(英特尔®)系列处理器采用下列几种预取机制来促进数据或代码移动以提升性能[3]:

- 硬件指令预取
- 软件数据预取
- 硬件数据或指令的Cache行预取

随着制造工艺的不断提升,Intel® 486™处理器中第一次增加了8kB片上L1-Cache;Intel® Pentium Pro处理器第一次增加了256kB片上L2-Cache;Intel® Pentium II处理器将L1-Cache分为数据Cache(Data Cache,DC)和指令Cache(Instruction Cache,IC);Intel® Pentium处理器至尊版(Extreme Edition)第一次实现了单芯片的多核技术;现在许多多核处理器都有共享的片上L3-Cache[4,5]。随着数据预取逻辑(Data Prefetch Logic)的引入,以及Intel® Smart Memory Access技术和指令乱序执行的有效结合,硬件预取(Hardware Prefetcher)在降低Cache失效上发挥了巨大作用。Intel® Pentium III处理器第一次将SSE(Streaming SIMD Extensions)[6]指令集带入了IA-32体系结构,而软件预取指令(Software Prefetch Instruction)作为SSE的一部分开始被诸多软件开发者使用。

到稿日期:2015-03-17 返修日期:2015-06-27 本文受国家自然科学基金项目:多核多线程处理器SIMD扩展的编程模型及编译优化关键技术研究(61170046),863计划项目:面向国产飞腾处理器的并行程序综合优化系统(2012AA010903)资助。
董钰山(1990—),男,硕士生,主要研究方向为计算机体系结构、编译及优化技术,E-mail:yushandong@hotmail.com;李春江(1974—),男,博士,副研究员,硕士生导师,CCF高级会员,主要研究方向为计算机体系结构、编译及优化技术。

本文主要针对当前 Intel® 64 体系结构处理器，在简单分析存储层次结构的基础上，对 X86/X64 CPU 的数据预取(Data Prefetching)——硬件预取和软件预取机制作了详细剖析，并且介绍了 GNU 编译器集合[7](GNU Compiler Collection，GCC(本文简称 GCC 或 GCC 编译器))和英特尔 C++编译器[8](Intel C++Compiler，ICC(本文简称 ICC 或 ICC 编译器))中软件预取的应用程序接口(Application Program Interface，API)，为编程人员手动添加预取指令提供了指导。基于上述分析，通过对紧嵌套循环在特定平台上的测试，分析得出影响 Intel® 64 体系结构的数据预取有效性的几个因素，为进一步在多重循环中手动添加预取指令指明了方向，对循环进行预取优化有重要意义。

2 Cache 层次结构

为了有效地弥补处理器和主存储器之间巨大的速度差距带来的性能损失，基于程序局部性原理，计算机引入片上 Cache(高速缓冲存储器)，并采用层次式 Cache 结构。Intel® 64 CPU 由于微体系结构的不同，其 Cache 层次也存在差异，但可总结为如图 1 所示的基本层次结构。多核多线程技术以及制造工艺的不断发展，将多个处理器核集成到同一个 CPU 芯片中，而且每一个处理器核有独立的 L1-ICache、L1-DCache 和 L2-Cache，进一步提高了单个处理器的执行能力，缓解了处理器与内存之间的速度差异，并增大了访存带宽。

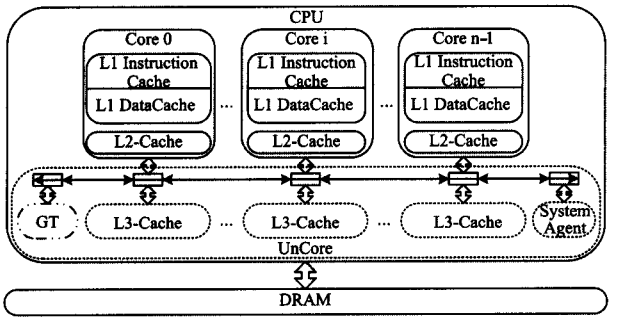

图 1 x86/x64 体系结构存储层次

2.1 基本组成

对于 Sandy Bridge、Ivy Bridge、Haswell、Haswell-E、Broadwell 等微体系结构(Microarchitecture)的处理器[4]，CPU 内部可分为 IA Cores 和 Uncore Subsystem 两部分[3]。其中，每一个 IA Core 除了拥有独立的计算单元之外，还有独立的 L1 级指令高级缓存(L1-ICache)、L1 级数据高级缓存(L1-DCache)和 L2 级高级缓存(L2-Cache)；Uncore Subsystem 包括一个系统代理(System Agent)、图形单元(Graphics Unit，GT)和 LLC(Last Level Cache)。IA Cores 与 Uncore Subsystem 之间通过高带宽双向环总线(High Bandwidth Bi-Directional Ring Bus)连接，从而使得 Uncore 单元被每一个 IA Core 平等共享，且 CPU Core 与 Ring Bus 通过 Ring Node 互联。Haswell、Haswell-E、Broadwell 微体系结构的 Cache 参数如表 1 所列，Sandy Bridge、Ivy Bridge 微体系结构的 Cache 参数如表 2 所列[3]。

表 1 Haswell 微体系结构的 Cache 参数

层次	L1-DCache	L1-ICache	L2-Cache	L3-Cache (LLC)
容量	32kB	32kB	256kB	可变
关联性 (ways)	8	8	8	可变
行大小 (Byte)	64	64	64	64
最小延迟[1)]	4cycle		11cycle	
吞吐率 (clocks)	0.5[2)]		可变	可变
峰值带宽 (bytes/cyc)	64(Load)+32(Store)		64	
更新策略	写回(WB)		写回(WB)	写回(WB)

注：1)软件可见延迟，依赖于存取模式和其他因素。
2)L1-DCache 每个时钟周期执行两个载入微操作，每个微操作至多载入 32byte 数据。

表 2 Sandy Bridge 微体系结构的 Cache 参数

层次	L1-DCache	L1-ICache	L2-Cache	L3-Cache (LLC)
容量/关联性	32kB	32kB	256kB	可变
关联性(ways)	8	8	8	可变
行大小(Byte)	64	64	64	64
最快延迟 (cycles)[1)]	4		12	26−31[2)]
吞吐率(clocks)	0.5		可变	可变
峰值带宽 (/core/cycle)	2×16bytes		1×32bytes	1×32bytes
更新策略	写回(WB)		写回(WB)	写回(WB)

注：1)当在其他核的 L2-Cache 和 L1-DCache 可用时，如果干净命中(clean hit)，则延迟 43 cycles；如果脏命中(dirty hit)，则延迟 60 cycles。
2)LLC 的延迟依赖于具体产品编号。

L3 级高级缓存(L3-Cache)，在此又可称作 LLC(Last Level Cache)，由多个 Cache 片(Slice)组成，每一个 CPU 核对应一个 Cache 片。这种局部的 Cache 片降低了 Cache 层次的设计难度，提高了 LLC 的总线带宽和可扩展性，避免了潜在的 Cache 冲突。每一个 Cache 片由两部分组成：逻辑部分和数据阵列部分。

- 逻辑部分具有处理数据相关、内存排序、数据阵列部分的存取、LLC 缺失和内存写回等功能。
- 数据阵列部分主要用于存储 Cache Line。

处在 LLC 数据区数据的物理地址通过一个哈希函数映射到 Cache 片中，所以内存地址在 LLC 数据区是均匀分布的。一个数据块的数据区是 4/8/12/16 路，可对应 0.5M/1M/1.5M/2M Cache 块大小。但是，从软件角度来看，由于 Cache 块中的物理地址分布方式与 LLC 数据区不同，因此 LLC 数据阵列表现得并不像一个常规的 N 路 Cache。

然而，Intel® Xeon E5[9] 系列处理器并不包括 GT，取而代之的是具有大容量和窥探功能的 LLC 以支持多处理器结构、Intel QPI[10](Quick Path Interconnect)接口以支持多插座平台、电源管理控制硬件、一个具有支持在内存与 I/O 设备之间高带宽传输的系统部件。但对于 P4 和 NetBurst 微体系结构的 Xeon 处理器还有一个 12k μops、8 路组相联的 Trace Cache。

英特尔集成众核架构[11,12](Intel Many Integrated Core，MIC)协处理器(如 Intel® Xeon Phi™)是一种高密度集成大量 MIC 架构内核的 PCI-e 接口形式扩展卡。每个微处理器核(MIC Core)都是基于 x86 架构的计算核心，基本指令集采

用英特尔架构(Intel Architecture,IA)x86 指令集,并有部分的指令扩展,但其核心与 Intel 桌面 CPU 还有一定差别。MIC 的缓存组织与 CPU 类似:每核有 32kB L1-ICache(Instruction Cache)和 32kB L1-DCache(Data Cache);每核拥有全局可见的 L2-Cache,大小 512kB。L2-Cache 是内核环总线(Core-Ring Interface)功能模块的一部分,含有流硬件预取单元,能够有选择地预取代码、读、RFO(Read-For-Ownership)Cache 进入 L2-Cache。L2-Cache 一共有 16 个硬件预取流(Stream),能够预取 4kB 数据页,且在流方向确定后同时发起多达 4 个预取请求。有关 MIC 缓存组织的其他详细内容请查阅 Inter 相关网站和资料,在此不再叙述。

2.2 Cache 层的相互关系

在处理器系统中,不同级别的 Cache 所使用的设计原则并不类同。每一级 Cache 都有各自的主要任务,并不是简单的容量与延时的匹配关系。Cache 层之间的相互关系有 3 种:包含(Inclusive)、独有(Exclusive)和 NI/NE Cache。

在此,先引入两个概念[2]:Inner Cache 和 Outer Cache。Inner Cache 指在微架构之内的 Cache,如 Sandy Bridge 微架构中含有 L1 和 L2 两级 Cache,且均属于私有 Cache;Outer Cache 指微架构之外的 Cache,如 LLC。由于 Inner Cache 在有些处理器系统中由多个层次组成,因此 Inclusive 和 Exclusive 概念首先出现在 Inner Cache 中,之后才是 Inner Cache 和 Outer Cache 之间的联系。为了便于叙述和理解,在此假设 Inner Cache 和 Outer Cache 都只有一级。

在处理器系统中,采用 Inclusive Cache 结构时,Inner Cache 是 Outer Cache 的一个子集,在 Inner Cache 中出现的 Cache 块(Cache Block)在 Outer Cache 中一定具有副本。Inclusive Cache 层次结构较为明晰,并在单 CPU Core 的环境下得到了广泛的应用。然而在多核环境下,多级 Cache 层次结构中使用 Inclusive 方式,不仅浪费空间,还会增加 Cache 一致性(Cache Coherency)的复杂度,从而使得严格的 Inclusive 在实现上更加困难。另外,在 Inclusive Cache 设计中存在竞争条件(Race Condition),且 Inner Cache 与 Outer Cache 的深度耦合加大了这些 Race Condition 的解决难度。

Exclusive Cache 是相对纯粹 Inclusive Cache 的另一个极端,从设计实现的角度而言仍然是紧耦合结构。采用这种 Cache 结构时,一个 Cache Block 存在于 Inner Cache 或 Outer Cache 中,但不能同时存在于这两种 Cache 中。与 Inclusive Cache 相比,Inner Cache 和 Outer Cache 之间避免了 Cache Block 重叠而产生的浪费。从 CPU Core 的角度而言,这种结构相当于提供了一个容量更大的 Cache(有效容量是 Inner Cache 与 Outer Cache 容量之和),从某种程度上提高了 Cache 层次的整体命中率。另外,在 Exclusive Cache 中存在 Cache Block 的淘汰(Eviction)操作,不同的 Eviction 机制会影响 Cache 的命中率。

NI/NE Cache 是 Exclusive Cache 和 Inclusive Cache 的折中,Inner Cache 和 Outer Cache 间没有直接关联,但是在实现时需要设置一些特殊的状态位表明各自的状态。采用 NI/NE Cache 结构时,Inner Cache 与 Outer Cache 的部分内容将重叠,与 Inclusive Cache 结构相比,Cache 利用率相对较高,但仍然不及 Exclusive Cache 结构,且因为偶然命中(Accidentally Hit)的原因,NI/NE Cache 容量利用率与 Inclusive Cache 相比,提高得较为有限。另外,与 Inclusive 和 Exclusive Cache 相比,采用这种方式使得 Inner Cache 和 Outer Cache 间的耦合度得到较大的降低,从而降低了 Cache 层次的设计难度。

现代 CMP(Chip Multiprocessors)处理器在多级 Cache 的设计中更多是使用 Exclusive 或者 NI/NE 的结构,纯粹的 Inclusive 结构在具有 3 级或以上的 Cache 层次中并不多见。Intel 从 P6 处理器到 Sandy Bridge 处理器,一直使用 NI/NE Cache 结构,但是该结构并不是 Intel x86 处理器的全部。Nehalem 和 Sandy Bridge 处理器在使用 NI/NE Cache 的同时,也使用了 Inclusive Cache。其中,L1-Cache 和 L2-Cache 是 Inner Cache,采用 NI/NE 结构;所有 CPU Core 共享的 L3-Cache(或 LLC)是 Outer Cache,采用 Inclusive Cache 结构,即该 Cache 中含有所有 CPU Core L1-Cache 和 L2-Cache 的数据副本。在 LLC 中的每一个 Cache Block 中都含有一个由 4 位组成的有效向量(Valid Vector)字段,用来表示 LLC 中包含的副本是否存在于各个 CPU Core 的 Inner Cache 中。

2.3 数据预取

在进行数据预取时,首先需要重点关注的是预取时机,不宜过早也不宜过晚,如果再考虑多处理器系统,无论采用硬件预取还是软件预取,做到恰到好处都是巨大挑战。其次,需要考虑预取的数据放置到 Cache 层次的哪一级,是 L1、L2,还是 LLC,以及所预取的数据是私有数据还是共享数据。然后,还需考虑预取数据的粒度,是以字节、字、Cache 块还是多个 Cache 块。最后,还要进一步考虑采用的预取方式,是硬件预取、软件预取,还是混合预取。

这一切增加了预取的实现难度,也造成在某些情况下,采用预取机制反而会降低效率。何时采用预取机制,关系到处理器系统结构的各个环节,需要结合软硬件资源统筹考虑。只有充分了解处理器提供的用于实现预取的软件和硬件资源,才能使系统程序员更加合理和巧妙地使用预取机制。下文将详细介绍 Intel® 64 架构的硬件预取和软件预取机制。

3 硬件预取

3.1 硬件预取概述

Pentium M、Intel® Core Solo、Intel® Core Duo、Intel® Xeon 处理器以及基于 Intel Core 微体系结构和 Intel NetBurst 微体系结构的处理器都提供硬件预取(Hardware Prefet-ching)机制,使用硬件来监控应用程序的数据存取模式,从而达到自动预取数据的目的。

硬件数据预取的目的是自动预测程序将要使用哪些数据,并且当这些数据不处在靠近程序执行核的 Cache 中时,可以将数据由下一层 Cache 或内存预取到内层 Cache 中。硬件数据预取的特点[3]包括:

(1)预取时机,当 LLC 发生两次连续 Cache 失效时,将触发硬件数据预取。

(2)在数据存取模式上的一些规则:若数据存取模式有常量的步幅,在存取步幅小于硬件预取触发距离的一半时,硬件预取有效;若存取步幅不是常量,在两次成功的 Cache 失效的步幅都小于触发的极限距离时,自动硬件预取可以有效隐藏内存延迟;若两次成功的 Cache 失效都小于触发的极限距离

并且接近 64Bytes，则自动硬件预取最有效。

(3)硬件预取将会对超越数组末尾的一些不被使用的数据发出预取请求，从而浪费总线带宽。另外，该行为将会影响下一个数组开始阶段的读取。

(4)预取不会超过 4kB 页边界。在一个硬件预取从一个新的页开始预取之前，程序需要一个对新页初始的载入请求。

(5)当一个应用的内存传输有明显的区域特性，即 Cache 失效步幅大于硬件预取触发距离时，硬件预取将会消耗大量额外的系统带宽。

(6)对当前应用程序的有效性取决于应用程序内部内存传输中小步幅和大步幅所占的比例。

(7)某些情况下，可以通过数据存取顺序的重排序变换将大步幅的 Cache 失效变换成小步幅的 Cache 失效，从而益于自动硬件数据预取。

3.2 数据预取到 L1-DataCache

在下列条件下，硬件数据预取将被载入操作触发[3,4]：

- 从 WB 内存类型载入。
- 预取数据在 load 指令执行的同一个 4kB 页内；
- 处理流水线内没有栅栏；
- 处理过程中没有太多载入缺失；
- 没有连续的存储流。

两种硬件预取策略将数据载入到 L1-DCache：

(1)数据缓存单元(Data Cache Unit，DCU)预取

这种预取方式也称作流预取，由升序存取不同载入数据触发。

(2)基于指令指针(Instruction Pointer，IP)的步幅预取

程序执行过程中记录单个的 load 指令，当检测到 load 指令存在有规律的步幅时，则发送预取。此预取可以向前或向后预取，能最高检测 2kB 步幅。

3.3 数据预取到 L2-Cache 和 LLC

将数据从内存取到 L2-Cache 和 LLC 也有两种硬件预取策略[3]：

(1)空间预取(Spatial Prefetcher)

该预取将两个 Cache 行组成 128 位对齐的块，然后取到 L2-Cache 中。

(2)流预取(Streamer)

该预取用于监管来自 L1-Cache 对上升或下降地址序列的读请求，包括 L1-DCache 请求和 L1-ICache 请求。其中，L1-Dcache 请求是由载入和存储操作或硬件预取机制发起，L1-ICache 请求主要针对代码读取。当检测到向前或向后的请求流时，目标 Cache 行将被预取。预取 Cache Line 必须在同一个 4k 页。

这两种预取方式在将数据预取到 LLC 的同时也会将数据取到 L2-Cache，除非 L2-Cache 因过度载入而发生所需请求丢失。流预取扩展还包括以下特点：

- 对每个 L2 检查同时发送两个预取请求，最高可达到 20 行；
- 动态调整每一个核未解决请求的数目：若未解决请求数目较少，流预取将向前预取更多；若未解决请求数目过多，则只预取到 LLC 以及向前预取较少；
- 检测和保持最高 32 个数据存取流，并且对于每一个 4kB 的页，可以同时保持一个向前流和一个向后流。

4 软件预取

4.1 软件预取概述

软件预取(Software Prefetching)主要是通过某种方式(程序员手动或编译器自动)在合适的位置插入预取指令，从而发送预取提示，完成数据预取，实现隐藏存储延迟，达到提高程序性能的目的。

软件预取具有如下特点：

- 处理不能触发硬件预取的不规则存取模式；
- 使用比硬件预取更少的总线带宽；
- 软件预取必须被添加到新的代码中，可能会对现存的应用无益。

软件预取最大的缺点就是在多处理机的系统上，当代码被共享时，使用软件预取将导致明显的性能开销。

4.2 PREFETCH*h* 指令

Intel 在 i486 处理器中使用的 Dummy Read 指令是后来 x86 处理器中 PREFETCH*h* 指令的雏形。当前，PREFETCH*h* 指令作为 SSE(Streaming SIMD Extensions)指令集的一部分被几乎所有 Intel 处理器所拥有，从而在合适的时间为应用程序提供软件预取支持。PREFETCH*h* 指令主要由 PREFETCHT0、PREFETCHT1、PREFETCHT2、PREFETCHNTA 4 个具体指令组成，每个指令的具体实现功能如表 3[13] 所列。

表 3 PREFETCH*h* 指令及功能

类型	PREFETCHh 指令	功能
时间局部性	PREFETCHT0	取数据到所有 Cache 层： • Pentium Ⅲ：L1-Cache 或 L2-Cache • Pentium4 和 Intel® Xeon 处理器：L2-Cache
	PREFETCHT1	取数据到 L2-Cache 或更高： • Pentium Ⅲ：L2-Cache • Pentium4 和 Inte®l Xeon 处理器：L2-Cache
	PREFETCHT2	取数据到 L2-Cache 或更高： • Pentium Ⅲ：L2-Cache • Pentium4 和 Intel® Xeon 处理器：L2-Cache
非时间局部性	PREFETCHNTA	取数据到接近处理器的地方，且最小化 Cache 污染： • Pentium Ⅲ：L1-Cache • Pentium4 和 Intel® Xeon 处理器：L2-Cache

在使用 PREFETCH*h* 指令过程中需注意以下几点：

- 源操作数是一个字节大小的内存位置(memory location)。
- 若所选择的行已经在距离处理器较近的 Cache 层中，则不需要进行数据移动，该预取被忽略。
- PREFETCH*h* 指令只是一种提示来触发硬件预取，并不影响程序的正常行为。
- 预取位置提示的实现依赖于具体的实现，可以在处理器的具体实现中被重载或忽略。预取数据的数量也是依赖于处理器的具体实现，但是最小是 32kB。
- 处理器的推测取是自发的，并从指定内存类型(允许推测读，WB、WC、WT)的系统内存区域缓存数据。一条 PREFETCH*h* 指令只是这种推测行为的一种提示，因为这种推测取可以发生在任何时间点而不需要指令的执行去为之贴标签。一条 PREFETCH*h* 指令不能够被栅栏(fence)指令命令或锁内存引用。一条 PREFETCH*h* 指令也不能够被

CLFLUSH 指令、其他 PREFETCH*h* 指令或其他一般的指令所控制，但可以被诸如 CPUID、WRMSR、OUT、MOV CR 等串行指令控制。

• 如果使用 LOCK 前缀，将产生 # UD 异常。除此之外，PREFETCH*h* 指令的使用不会产生任何异常和错误。

然而在下列几种情况下，PREFETCH*h* 不执行数据预取[3]：

• PREFETCH*h* 导致 DTLB 失效。

• 对特定地址存取时导致错误或异常。

• 内存子系统用尽 L1-Cache 和 L2-Cache 之间的请求缓冲。

• PREFETCH*h* 目标是一个不可高速缓冲的内存区域。

• 使用 LOCK 前缀，导致无效操作异常。

4.3 GCC 编译器支持

对于 C/C++等高级语言中预取指令的使用，GCC 编译器提供了如下两种用户接口函数，以实现指定数据的预取。

(1)void _mm_prefetch(char * p,int i)

参数"p"用于指定预取字节的地址(与 Cache 行一致)；"i"值用于指定预取操作的类型，有_MM_HINT_T0、_MM_HINT_T1、_MM_HINT_T2、_MM_HINT_NTA 这 4 种取值选择，分别对应于 PREFETCH*h* 指令的 PREFETCHT0、PREFETCHT1、PREFETCHT2、PREFETCHNTA 指令。

(2)void __builtin_prefetch(const void * addr,…)

该函数是 GCC 编译器的一个内置函数，通过对数据手工预取来减少数据读取延迟，但需要 CPU 支持。其中，参数 addr 是个内存指针，指向要预取的数据对象。该函数还有两个可选参数，即 rw 和 locality：

• rw 是编译时的常数，取值 1 或 0，取 1 时表示写(w)，取 0 时表示读(r)。

• locality 是编译时的常数，也称为"时间局部性(temporal locality)"，该值的范围在 0～3 之间。0 表示没有时间局部性；3 表示被访问的数据或地址具有高时间局部性；值 1 和 2 则分别表示具有低时间局部性和中等时间局部性。该值默认为 3。

GCC 源码 gcc/config/i386/xmmintrin. h 文件指明了二者的联系，如图 2 所示。

```
#ifder_OPTIMIZE_
    extern_inline void_attribute_((_gnu_inline_,_always_inline_,_
artificial_))
    _mm_prefetch(const void * _P,enum_mm_hint_I)
    {
        _builtin_prefetch(_P,(_I & 0x4)>>2,_I & 0x3);
    }
#else
    #define_mm_prefetch(P,I)_builtin_prefetch((P),((I & 0x4)>
>2),(I & 0x3))
#endif
```

图 2 GCC 两种接口函数的联系

对_builtin_prefetch()函数进行编译时，常数 locality(即参数_I)有如图 3 所示的取值定义。

```
enum_mm_hint
{
/ * _MM_HINT_ET is _MM_HINT_T with set 3rd bit. * /
  _MM_HINT_ET0=7
  _MM_HINT_ET1=6
  _MM_HINT_T0=3
  _MM_HINT_T1=2
  _MM_HINT_T2=1
  _MM_HINT_NAT=0
};
```

图 3 GCC 中编译时常数 locality 的取值

4.4 ICC 编译器支持

对于 C/C++等高级语言中预取指令的使用，Intel® ICC 编译器为用户提供了如下两种数据预取指令插入方式。

(1)用户接口函数

void _mm_prefetch(char * p,int i)，参数和功能与 GCC 相应函数一致。

(2)编译指导的数据预取

pragma prefetch

pragma prefetch [var1 [:hint1 [:distance1]] [,var2 [:hint2 [:distance2]]]…]

pragma noprefetch [var1 [,var2]…]

其中，参数 var 是预取字节的内存引用；hint 用于指定预取操作的类型：_MM_HINT_T0、_MM_HINT_NT1、_MM_HINT_NT2、_MM_HINT_NTA；distance 用于指定预取的循环迭代数。

ICC 用户手册[14]指出，这种预取语法仅被 Intel® Itanium®(英特尔®安腾®)处理器和 Intel® MIC 架构协处理器支持。

4.5 PREFETCHW 指令

PREFETCHW 指令[13]也是 SSE 指令集中的一条区别于 PREFETCH*h* 指令簇的指令，主要功能是写预期的预取数据到 Cache。该指令与 PREFETCH*h* 指令的主要区别在于该指令针对写预期的数据进行预取操作。C/C++编译器与该指令等价的 API 为 void _m_prefetchw(void *)。

GCC 编译器对上述接口函数的定义如图 4 所示。可见，在 GCC 编译器中该接口函数是通过_builtin_prefetch(const void * addr,…)函数在编译时常数 rw 为 1 且 locality 为 3 时定义的，且为一内敛函数。因此，在实际使用中该函数也可以直接用_builtin_prefetch(const void * addr,1,3)代替。

```
#ifnder_PRFCHWINTRIN_H_INCLUDED
#define_PRFCHWINTRIN_H_INCLUDED
extern_inline void _attribute_((_gnu_inline_,_always_inline_,_artifi-
cial_))
_m_prefetchw(void * _P)
{
    _builtin_prefetch(_P,1,3,/ * _MM_HINT_T0 * /);
}
#endif/ * _PRFCHWINTRIN_H_INCLUDED * /
```

图 4 GCC 中 PREFETCHW 指令的 API 实现

5 数据预取效果测试和分析

5.1 实验环境

5.1.1 软件环境

本文测试使用 GCC 编译器(版本 GCC4. 9. 0[15])，实验环境安装的操作系统为 RedHat Enterprise Linux Server release 6. 2。采用手工编写的特征程序段进行测试。

5.1.2 硬件环境

本文使用一台 Intel X86_64 至强工作站，该工作站使用双路 Xeon(R)E5-2670 处理器(主频 2.60GHz)；每个处理器芯片上包含 8 个处理器核，每个处理器核包含一个 L1 级高级缓存(32kB 指令、32kB 数据)和一个局部 L2 级高级缓存(256kB)，8 个处理器核共享片上 L3 级高级缓存(20MB)，并在 L3 级高级缓存维护一致性；处理器间的 L3 级高速缓存间通过 QPI(Quick Path Interconnect)直连，使两个处理器构成 SMP(共享主存多处理机)架构(并不是完全理想的 SMP，有非常轻微的 NUMA(非一致存储器访问)[16]效应)。该实验平台处理器的架构如图 5 所示，各级 Cache 特征参数值如表 4[3,17]所列。

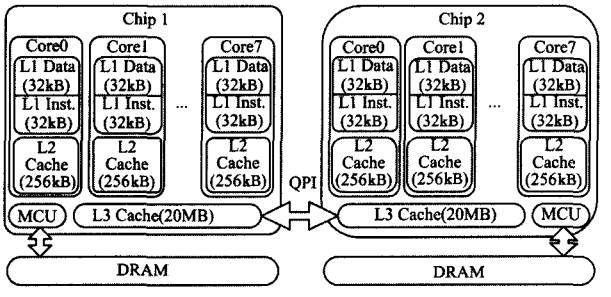

图 5 实验环境的处理器架构

表 4 Intel Xeon E5-2670 处理器各级 Cache 特征

单元	大小	组织方式	Cache 行大小
L1-ICache	32kB	8 路组相联	
L1-DCache	32kB	8 路组相联	64Bytes
L2-Cache	256kB	8 路组相联	64Bytes
L3-Cache	20MB		64Bytes

5.2 实验内容及结果分析

5.2.1 软件预取效果

由于硬件数据预取是通过程序行为自动触发的，因此对于高级程序语言编程者而言，自动触发的硬件数据预取具有不可控性，故测试主要针对软件预取对程序性能的影响。测试用例如图 6 所示，其中图 6(a)是一维三点 Jacobi 迭代法(3-point-Jacobi)，图 6(b)是二维五点 Jacobi 迭代法(5-point-Jacobi)。

```
for(t=0;t<T;t++){
  for(i=0;i<N;i++)
      a[i]=b[i];
  for(i=1;i<N-1;i++)
      b[i]=(a[i-1]+a[i]+a[i+1])/3.0;
}
```

(a)3P-Jacobi

```
for(t=0;t<T;t++){
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      a[i][j]=b[i][j];
  for(i=1;i<N-1;i++)
    for(j=1;j<N-1;j++)
      b[i][j]=(a[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1]+
4*a[i][j])/8.0;
}
```

(b)5P-Jacobi

图 6 测试用例

尽管预取指令只需要最少的时钟和内存带宽，但它们在总线周期、机器周期和资源上并不是完全无花费的。因此，过度的预取可能会导致性能开销，因为机器前端和内存子系统的资源竞争都会带来开销。这种影响在目标循环很小或目标循环流限制的情况下可能会更严重。所以需要先对测试用例作简单的循环变换，然后在合适位置插入预取指令，结果如图 7 所示。

```
for(t=0;t<T;t++){
for(i=0;i<N;i++)
a[i]=b[i];
prefetch(a[0],b[0]);
for(i=1;i<N-1-4;i+=4)
{
prefetch(a[i+4],b[i+4]);
b[i]=(a[i-1]+a[i]+a[i+1])/3.0;
b[i+1]=(a[i-1+1]+a[i+1]+a[i+1+1])/3.0;
b[i+2]=(a[i-1+2]+a[i+2]+a[i+1+2])/3.0;
b[i+3]=(a[i-1+3]+a[i+3]+a[i+1+3])/3.0;
}
b[i]=(a[i-1]+a[i]+a[i+1])/3.0;
b[i+1]=(a[i-1+1]+a[i+1]+a[i+1+1])/3.0;
b[i+2]=(a[i-1+2]+a[i+2]+a[i+1+2])/3.0;
b[i+3]=(a[i-1+3]+a[i+3]+a[i+1+3])/3.0;

}
```

(a)3P-Jacobi

```
for(t=0;t<T;t++){
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      a[i][j]=b[i][j];
  prefetch(a[0][0],a[1][0],a[2][0]);
  for(i=1;i<N-1;i++){
    for(j=1;j<N-5;j+=4){
      prefetch(a[i-1][j+4],a[i][j+4],a[i+1][j+4]);
      b[i][j]=(a[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1]+
      4*a[i][j])/8.0;
      b[i][j+1]=(a[i-1][j+1]+a[i][j-1+1]+a[i+1][j+1]+a[i]
      [j+1+1]+4*a[i][j+1])/8.0;
      b[i][j+2]=(a[i-1][j+2]+a[i][j-1+2]+a[i+1][j+2]+
      a[i][j+1+2]+4*a[i][j+2])/8.0;
      b[i][j+3]=(a[i-1][j+3]+a[i][j-1+3]+a[i+1][j+3]+a[i]
      [j+1+3]+4*a[i][j+3])/8.0;
    }
    prefetch(a[i][0],a[i+1][0],[i+2][0]);
    b[i][j]=(a[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1]+
    4*a[i][j])/8.0;
    b[i][j+1]=(a[i-1][j+1]+a[i][j-1+1]+a[i+1][j+1]+
    a[i][j+1+1]+4*a[i][j+1])/8.0;
    b[i][j+2]=(a[i-1][j+2]+a[i][j-1+2]+a[i+1][j+2]+
    a[i][j+1+2]+4*a[i][j+2])/8.0;
    b[i][j+3]=(a[i-1][j+3]+a[i][j-1+3]+a[i+1][j+3]
    a[i][j+1+3]+4*a[i][j+3])/8.0;
  }
}
```

(b)5P-Jacobi

图 7 添加预取指令的测试用例

由 GCC 编译器编译运行，两个测试用例在循环变换和添加预取指令之后的加速比如图 8 所示。

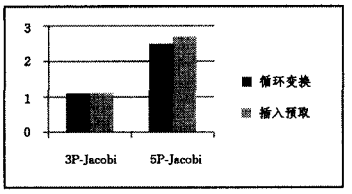

图 8　循环变换和插入预取程序加速比

由此可见，两个测试用例在循环变换和添加预取指令之后的性能都有提升，且 2D-Jacobi 性能提升较明显。另外，单纯从插入预取指令对程序性能影响而言，5-point-Jacobi 约有 8.5%的性能提升，而 3-point-Jacobi 性能几乎没有提升。分析图 7 所示程序发现，图 7(a)中，一维数组元素的存取更易触发硬件数据预取，另外，循环体计算部分计算量较小，未能和预取很好地并发执行；图 7(b)中，核心循环体涉及加法、乘法和除法 3 种运算，预取操作能够与计算操作很好地并行，从而使得预取操作更好地隐藏访存延迟，提升了程序性能。

此外，还测试了无循环变换而直接插入预取指令时程序的性能，结果表明这样会导致插入过多的预取指令，使得在机器前端和内存子系统出现大量的资源竞争，从而使程序性能下降。由此可见，插入预取指令前，数据局部性分析和适当的循环变换是必要的。除此之外，不排除插入预取指令对其他编译优化的影响。

5.2.2　预取指令的预取度

预取度[18]是指一次预取操作取回的数据量，一般以 Cache line 为基本单位。预取度过大，预取操作会导致严重的 Cache 污染，加剧访存竞争，增大平均访存延迟；预取度过小，则预取效率很低。在 3.3 节提到，对于 Intel® 64 体系结构的数据预取，由于空间预取行为的存在，预取度为两个 Cache Line。在此，参考文献[19]所述方法，将一维数组的加法程序作简单改进(见图 9)，其中数组均为双精度浮点(double)。硬件预取是内存顺序预取，对预取的数据没有针对性；软件预取是特定数据预取。

```
for(j=0;j<N;j+=D){
    c[j]=a[j]+b[j];
}
```

(a)无预取指令

```
Prefetch(a[0],b[0],C[0]);
for(j=0;j<N;j+=D){
    Prefetch(a[j+1],b[j+1],,c[j+1]);
    c[j]=a[j]+b[j];
}
```

(b)手动加预取指令

图 9　数据以步幅 D 访问的数组加法

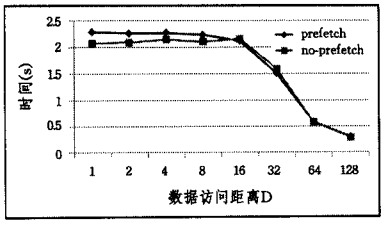

图 10　程序运行时间与数据访问距离的关系

如图 10 实验结果显示，步幅 D 从 1 到 16 变化时，程序的运行时间几乎没有变化；但是从 16 之后，运行时间随 D 的增大明显减少。由此可见，CPU 从内存存取数据的大小为 128B，又由于 Cache Line 大小为 64B，因此每次存取为 2 个 Cache Line。

结束语　循环数组在高性能应用程序和评测程序中占有相当大的比重，循环数组预取优化的研究对高性能程序性能和高性能计算机资源利用率的提高有着重要作用，尤其是对物理、生物、气象等领域计算密集型应用程序性能的提高意义重大。多核多线程多存储层次 CPU 和多核或众核架构协处理器在高性能计算领域的广泛使用，虽然提升了数值计算能力，但也加大了对内存访问带宽的需求。

X86/X64 体系 CPU 的数据预取，特别是自动触发的硬件预取，对提升数组元素的存储顺序访问效率有非常重要的作用。然而，自动触发的硬件预取是内存顺序预取，对预取对象没有针对性，所以对于非顺序访存数据，软件预取具有更大的优势。插入软件预取指令，需在局部性分析的基础上解决 3 个基本问题：什么时候预取，预取对象是什么，预取度是多少。所以，紧嵌套循环中数据局部性分析和多种循环变换综合运用，在提升数据访问的空间局部性和时间局部性的同时，还对数据预取指令的添加有指导作用。因此，Loop Tiling、Time Skewing 等循环变换技术的研究和应用对在嵌套循环中插入数据预取指令有重要意义。

本文虽然只测试了 Intel 平台上数据预取对串行程序性能提升的影响，但对平台结构和数据预取行为的剖析为共享存储器多核多线程处理器上并行程序中采用数据预取进行访存优化提供了一定的指导方向。

参 考 文 献

[1] Hennessy J L, Patterson D A. Computer architecture: a quantitative approach [M]. Elsevier, 2012

[2] Sailing. 浅谈 Cache Memory[EB/OL]. (2011-10-03)[2015-3-17]. http://blog.sina.com.cn/s/blog_6472c4cc0102dw61.html

[3] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual [EB/OL]. [2015-03-05]. http://www.intel.com/content/www/ us/en/processors/architectures-software-developer-manuals.html

[4] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture [EB/OL]. [2015-03-05]. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[5] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Documentation Changes [EB/OL]. [2015-03-05]. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[6] Intel Corporation. Intel Instruction Set Architecture Extensions [EB/OL]. [2014-12-31]. https://software.intel.com/en-us/intel-isa-extensions

[7] Free Software Foundation, Inc. GCC, the GNU Compiler Collection [EB/OL]. (2014-12-23)[2015-03-05]. https://gcc.gnu.org

[8] Intel Corporation. Intel® Parallel Studio XE 2015 Composer Edition C++Release Notes [EB/OL]. (2014-06-25)[2015-03-05]. https://software.intel.com/en-us/articles/intel-parallel-studio-xe-2015-composer-edition-c-release-notes

[9] Intel Corporation. Intel® Xeon® Processor E5-1600/E5-2600/E5-46 00 Product Families Datasheet Volume One [EB/OL]. [2015-03-05]. http://www.intel.com/products/processor%5Fnumber/

[10] Intel Corporation. An Introduction to the Intel QuickPath Interconnect[EB/OL]. [2009-01-30]. http://www.intel.com

[11] 王恩东,等. MIC 高性能计算编程指南[M]. 北京:中国水利水电出版社,2012

[12] Jeffers J, Reinders J. Intel Xeon Phi coprocessor high performance programming[M]. Newnes, 2013

[13] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume2(2A, 2B & 2C): Instruction Set Reference, A-Z [EB/OL]. [2015-03-05]. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[14] Intel Corporation. Intel® C++ Compiler User and Reference Guides [EB/OL]. [2015-03-05]. http://www.intel.com

[15] Free Software Foundation. Inc. GCC 4.9 Release Series[EB/OL]. [2014-07-16]. http://gcc.gnu.org/gcc-4.9/

[16] Manchanda N, Anand K. Non-Uniform Memory Access(NUMA)[OL]. http://cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf

[17] Intel Corporation. Intel® 6 4 and IA-32 Architectures Software Developer's Manual Volume 3(3A, 3B & 3C): System Programming Guide[EB/OL]. [2015-03-05]. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[18] Feng Q Y. Research on Data Prefetching Techniques for Loop-Level Array References[D]. Changsha: National University of Defense Technology, 2008(in Chinese)
冯权友. 面向循环级数组访问的数据预取技术研究[D]. 长沙:国防科学技术大学,2008

[19] Igor Ostrovsky Blogging. Gallery of Processor Cache Effects [EB/OL]. http://igoro.com/archive/gallery-of-processor-cache-effects

(上接第 33 页)

结束语 本文提出了一种基于热例程的系统级动态二进制翻译优化方法,该方法以频繁执行的例程作为优化单位,分别通过虚拟寄存器机制和扩展基本块划分来消除动态二进制翻译引入的块内和块间冗余。相比基于踪迹的优化方法而言,该方法具有优化单位发现开销更小、代码区域更大、无重复优化翻译等优点,更适用于系统虚拟机中操作系统代码的优化。在跨平台系统虚拟机 ARCH-BRIDGE 平台上的测试表明,通过对内核代码实施该优化方法,SPEC CPUINT 2006 程序的效率提升了 3.5%~14.4%,相比基于踪迹的优化,性能最大提升了 5.1%。

参 考 文 献

[1] Chen Wei. Research on Dynamic Binary Translation based Co-Designed Virtual Machine[D]. National University of Defense Technology, 2010

[2] Hu W, Wang J, Gao X. Godson-3: A scalable multicore RISC processor with X86 emulation[J]. Micro, IEEE, 2009, 29(2): 17-29

[3] Heng Yin, Song D. TEMU-Binary Code Analysis via Whole-System Layered Annotative Execution[R]. Berkeley: UC Berkeley, 2010

[4] Wang Rong-hua. Research on Dyanmic Binary Translation Optimization[D]. Hangzhou: Zhejiang University, 2013(in Chinese)
王荣华. 动态二进制翻译优化研究[D]. 杭州:浙江大学,2013

[5] Slechta B, Crowe D. Dynamic optimization of micro-operations [C]// Proceedings. The Ninth International Symposium on High Performance Computer Architecture, 2003 (HPCA-9 2003). IEEE, 2003: 165-176

[6] Bellard F. QEMU, a fast and portable dynamic translator[C]// USENIX annual technical conference, FREENIX Track. 2005: 41-46

[7] Hong D Y, Hsu C C, Yew P C. HQEMU, a multi-threaded and retargetable dynamic binary translator on multicore[C]// Proceedings of the Tenth International Symposium on Code Generation and Optimization, ACM, 2012: 104-113

[8] Cao Hong-jia, Tang Yu-xing, Zhou Xing-ming. Parallel Dynamic Binary Translation and its Cache Maintanance[C]// Proceedings of National Conference on Information Storage Technology. Xi'an, 2004(in Chinese)
曹宏嘉,唐遇星,周兴铭. 并行动态二进制翻译及其缓存维护[C]// 全国信息存储技术学术会议论文集. 西安,2004

[9] Dehnert J C, Grant B K, Banning J P, et al. The Transmeta Code Morphing? Software: using speculation, recovery, and adaptive retranslation to address real-life challenges[C]// Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and Runtime Optimization. IEEE Computer Society, 2003: 15-24

[10] Ebcioglu K, Altman E, Gschwind M, et al. Dynamic binary translation and optimization[J]. IEEE Transactions on Computers, 2001, 50(6): 529-548

[11] Bala V, Duesterwald E, Banerjia S. Dynamo: a transparent dynamic optimization system[J]. ACM SIGPLAN Notices, ACM, 2000, 35(5): 1-12

[12] Hsu C C, Liu P, Wu J J, et al. Improving dynamic binary optimization through early-exit guided code region formation[C]// Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ACM, 2013: 23-32

[13] Huang Cong-hui, Chen Jing, Gong Shui-qing, et al. Research of Method for Virtualizing 64-bit Windows Application Binary Interface[J]. Computer Science, 2014, 41(1): 39-42(in Chinese)
黄聪会,陈靖,龚水清,等. 64 位 Windows ABI 虚拟化方法研究[J]. 计算机科学,2014,41(1):39-42

[14] Duesterwald E, Bala V. Software profiling for hot path prediction: Less is more[J]. ACM SIGOPS Operating Systems Review, ACM, 2000, 34(5): 202-211