

分簇处理器中分簇投机的 L0 Cache 设计

杨 兵,毛志刚,陈 晓,尹捷明

(哈尔滨工业大学 微电子中心,黑龙江 哈尔滨 150001)

摘 要: 处理器分簇技术是进一步提高超标量处理器性能的一种有效手段,实现了更大指令窗口和发射宽度的同时对 Cache 系统提出了严峻要求,需要一种访存延迟更小、扩展性更强的 Cache 结构.采用分簇投机的 L0 Cache 结构,处理器在访存时投机访问各簇内简单快速的 L0 Cache,较好地隐藏了下级 Cache 的访问延迟.仿真结果显示在 8 簇的分簇处理器中,采用 4kB,2 路组相连的分簇 L0 Cache 后处理器性能平均提升 5.6%,在部分测试程序中达到 20% 以上.

关键词: 分簇处理器;Cache 结构;投机访问

中图分类号: TP368.1

文献标识码: A

文章编号: 1000-7180(2010)07-0015-06

Speculative Clustered L0 Caches Design in Clustered Processor

YANG Bing, MAO Zhi-gang, CHEN Xiao, YIN Jie-ming

(Microelectronics Center, Harbin Institute of Technology, Harbin 150001, China)

Abstract: Clustering is an attractive technique for large monolithic superscalar processor, allowing for more in-flight instructions, wider issue width. Thus, to design a Cache structure with low memory access latency and high scalability is needed. By implementing speculative clustered L0 caches, clustered processor speculatively accesses a small, fast, and simple L0 cache in cluster so that accessing latency of low-level high-capacity cache is hidden. As a result, the latency of memory access is shortened. Simulation studies show that 4kB, 2-way set associative L0 Cache in 1x8 clustered processor provides a 5.6% IPC improvement, and in some particular programs a 20% improvement is achieved.

Key words: clustered processor; cache; speculative access

1 引言

分簇技术有效减小了硬件复杂度,连线延时^[1]等,是由于现有分簇处理器中一般采用集总的 Cache 结构,其访存延时过长问题并没有得到有效的解决.随着工艺水平的提高,处理器中能集成的存储容量可以进一步提高,然而随着处理器内核尺寸的增大,时钟周期时间的减少,Load 指令完成一次访存操作的周期数不断增加,这样希望通过使用大的片上存储系统来提高性能的处理器的访问延时增加而不能提高性能.同样,由于片内时钟频率的进一步增加,也进一步加大了片内和片外存储器性能之间的差距.在传统的分簇处理器中,多个协同工作的簇共同访问一个的共享的 Cache 大大削弱了分簇处理器的可扩展性.

越是大的片上 Cache 其延时问题越是加剧(面积增加导致连线延时增大,译码、选择逻辑延时增加),这就要求分簇处理器的 Cache 结构必须从集总结构转为分簇结构以此来缓解上述问题^[2].

分簇技术使超标量处理器中指令访存延迟成为了性能进一步提升的瓶颈.在解决这一问题的相关研究中,Manoj Franklin 提出一种 Address Resolution Buffer(ARB) 的硬件结构用于实现 Load/Store 指令的再定序^[3].

Sridhar Gopaly 等人在 ARB 的基础上提出一种投机版本的 Cache(SVC)^[4-5],这分布式的 Cache 结构消除了 ARB 中延时和带宽的问题^[6].在 SVC 的方案中,每个簇拥有一个本地独有的 L1 Cache,使用修改的一致性协议保证数据一致和正确.他们的

这种投机分布式 Cache 取代了传统的 LSQ, 解决了访存指令相关性去歧义问题, 有效克服了由于内存相关性去歧义带来的 ILP 局限, 但硬件复杂度和访问延迟较高. Dana S. Henry 在 SVC 的基础上提出了投机的簇内 Cache 结构^[7]. 这种 Cache 和 SVC 一样是投机的, 但不包括 LSQ 功能、不采用一致性协议、结构简单、访存延迟小, 有效解决了分簇处理器在访存时的延时问题. 在访问命中率和投机正确率上该结构还有进一步提高的空间.

2 分簇投机 L0 CACHE

该结构采用在分簇处理器中每一个簇添加一个私有的 L0 Cache 的方法来减小分簇超标量处理器的访存延迟, 其结构框图如图 1 所示. 处理器在访问 Cache 时同时访问簇内 L0 Cache 以及下级各簇共享的 L1 Cache. L0 Cache 的访存结果可以快速地返回给处理器, 以弥补 L1 Cache 访存延时和分簇处理器处理速度之间的差距. 但 L0 Cache 不采用一致性协议, 其返回的数据是投机的, 因此同时使用投机检验及恢复机制来保证程序执行的正确性.

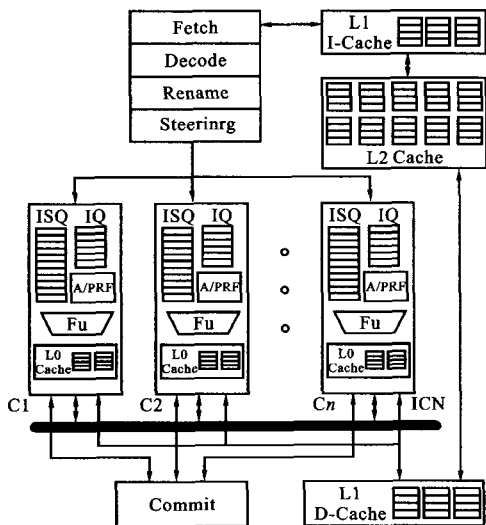


图 1 基于 L0 Cache 结构的处理器结构

2.1 组织结构

在采用分簇投机 L0 Cache 的分簇处理器中, L0 Cache 容量小、速度快, 且只保留由已经执行完的指令产生的数据. 为了保证这个 Cache 在访问时延时尽可能地小, 其结构就要尽可能简单, 故 L0 Cache 之间没有一致性协议, 没有和下级 Cache 之间的交互, 没有缺失处理. Load 指令读 L0 Cache 缺失后, L0 Cache 不会从 L1 Cache 中找回缺失的 Cache 块, Store 指令写 L0 Cache 缺失时直接将数据写入 L0 Cache, 并不会像传统的 Cache 那样从下级 Cache 中

找回该地址所对应的 Cache 块数据后完成写操作. 所以 L0 Cache 中的数据并不能保证绝对正确, 而是一个投机的值, 必须在得到结果以后验证其正确性. L0 Cache 是一个真正的簇内的结构, 不需要添加任何全局的结构来保证执行正确, 也不需要多个 L0 Cache 之间的一致性协议, 这大大简化了设计, 而且保证了访问延时.

L0 Cache 的块大小设置和 Load、Store 指令访问内存单元时所操作的最大的块大小相同. 这样做的好处是简化了硬件实现. 因为当访存指令对 Cache 进行写操作时, Cache 通常需要写缺失的处理, 从下级 Cache 中返回所操作块的其余部分. 而把块大小设置成访存指令访问内存单元时所操作的最大尺寸后, L0 Cache 可以不需要从 L1 Cache 中找回块的其余部分. 但是这种设计, 会引起一定程度的 L0 Cache 投机错误以及放弃了部分空间局部性的利用. 本课题中仿真使用的指令集为 ALPHA 指令集, 其最大访存数据长度为 8 字节, 所以 L0 的块大小设置可以为 8 字节.

为了保证 L0 Cache 数据的正确性, 需要及时更新 L0 Cache. 处理器中每个簇拥有一个的 L0 Store Buffer 用于存储 L0 Cache 的更新信息, 并在合适的时刻写入 L0 Cache. 由于本文不关注 LSQ 结构的设计, 故采用一种复制的 LSQ 结构, 但依然可采用其他文献中分立 (distribute) 的 LSQ 结构, 这是与 L0 Cache 正交的结构设计. 处理器采用各簇共享的 L1 DCache. LSQ 和 L1 Cache 的操作和实现基本不变. 当一个访存指令被送到 LSQ 中, 同时也包括了指令的动态序号, 所以 LSQ 可以确定正确的指令序.

2.2 执行过程

2.2.1 Load 指令执行过程

当某条 Load 指令经过取指、译码、重命名、分派等流水级后, 被加入所在簇的 IQ (Instruction Queue) 队列等待发射的同时使用 LSQ 去歧义并作可能的数据前递. 若需进行访存操作, 则该 Load 指令同时对簇内 L0 Cache 以及全局 L1 DCache 发送访问请求.

L0 Cache 结构简单, 容量小, 故访存延迟较小, 数据先于 L1 Cache 返回. 若该访存操作在 L0 Cache 中缺失, 则访存指令等待直到 L1 Cache 返回数据时唤醒后续相关指令. 若 L0 Cache 命中, 由于该返回的数据是投机的, 唤醒的后续指令投机执行. L1 Cache 的访存结果在若干个周期后返回. 当 L1 Cache 中数据返回时, L1 Cache 与 L0 Cache 中返回的值进行比较验证 L0 Cache 投机是否正确. 若 Load

投机错误,则处理器对 ROB(Reorder Buffer)中该 Load 指令所对应的投机错误标志位置位,重执行这条 Load 指令及其后续相关指令.在保证 L0 Cache 访问命中率和投机正确率足够高的情况下,采用 L0 Cache 可以有效减小 Load 指令的访存延迟.

当该 Load 指令到达 ROB 头部时进入提交流水级.首先检查该 Load 指令所对应的投机错误标志位.若已置位,则重执行该 Load 指令及其以后所有相关指令,并将从 L1 Cache 或再下级内存层次中取到的值更新 L0 Cache.若未置位,即表示该 Load 指令执行正确,进行提交操作.

2.2.2 Store 指令执行过程

Store 指令经过取指、译码、重命名、分派等流水级后被加入所在簇的 IQ 队列等待发射.当一个 Store 发射时,其访存地址和数据发送给 LSQ,以便解决指令间的相关性.

当该 Store 指令到达 ROB 头部时进入提交流水级,Store 指令把数据写入 L1 Cache 以及 L0 Cache,完成提交操作.

2.2.3 L0 Cache 更新过程

Load/Store 指令在提交流水段更新 L0 Cache. Load 指令在发现投机错误时更新 L0 Cache,避免 L0 Cache 下次被访问时依旧返回错误的值.为了加快 L0 Cache 的更新速度,当 L0 Cache 配置了读更新选项时,Load 指令也会在 L0 Cache 缺失时将从下级 L1 Cache 中得到的数据更新 L0 Cache.当 Store 指令进入提交流水段时,Store 指令在将数据写入 L1 Cache 的同时更新 L0 Cache.更新数据发送给簇内的 L0 Store Buffer,同时广播到其他簇中存在相应 L0 Store Buffer 中,待端口空闲时更新 L0 Cache.若 Store Buffer 已满则数据直接丢弃,仿真表明 Store Buffer 条目 6~8 个可基本保证数据不丢弃.配置 Load 指令在 L0 Cache 缺失时更新 L0 Cache 选项可加快 L0 Cache 的更新速度,进一步利用了程序局部性特性,但同时大量的 Load 指令在完成时更新 L0 Cache 会加重 L0 写端口的压力并且也会使投机错误的次数增加.

2.2.4 L0 Cache 块替换

Load/Store 指令更新 L0 Cache 时数据写入 L0 Cache 永远被认为是命中的.因为 L0 Cache 没有写缺失的处理,所以当 L0 Cache 中若没有该 Load/Store 访存地址对应的 Cache 块时,数据直接写入 L0 并置该 Cache 块为有效,即使数据没有写满整个 Cache 块也不从 L1 中找回数据.若 L0 Cache 已满则

按照最近最久未使用算法进行 Cache 块替换,替换出的数据直接丢弃.

2.3 投机错误分析及改进

在这种结构中,投机 L0 Cache 产生的错误大致分为以下三种情况:

(1) 不同的访存数据长度导致 L0 Cache 数据错误.在 ALPHA 指令集,访存指令的数据长度分别从 1 个字节到 8 个字节的.由于 L0 Cache 没有写缺失处理,所以当 L0 Cache 发生写缺失且写入数据长度小于 L0 Cache 的块大小(8 字节)时,L0 Cache 中该 Cache 块就会被置为有效,但是其中有部分数据是无效的.例如指令 I1、I2 是两条访存地址相同的访存指令,且中间没有其他该地址上的访存指令,如图 2 所示.假设初始情况下,L0 Cache 中该地址所对应的块是无效的,I1 提交时将 4 字节长度的数据写入 L1 Cache 同时也写入 L0 Cache,使该 Cache 块置为有效.但是由于 L0 Cache 没有写缺失的处理,不从下级 Cache 中找回块的其余数据,所以该 Cache 块中只有前 4 字节的数据是有效的,而后四字节的数据是无效的.这就导致了后续的 I2 读取这个块中全部 8 字节数据时,读命中却得到了错误的数,最终导致了 L0 Cache 投机执行错误.

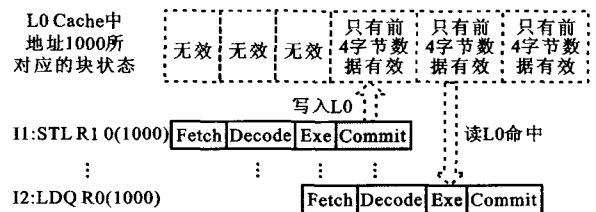


图 2 不同数据长度的访存操作带来的 L0 投机错误

(2) 未搜索 L0 Store Buffer 导致 L0 Cache 返回错误的数。为了简化 L0 Cache 的设计,L0 Store Buffer 没有使用任何搜索机制.这就可能引起当 Load 指令投机访问 L0 Cache 时,该地址上正确数据还在 L0 Store Buffer 中,没有来得及更新到 L0 Cache 中.这时 Load 指令从 L0 Cache 中读取的数据就不是旧的错误的数,结果导致了该 Load 指令投机执行错误.

(3) L0 Store Buffer 溢出导致 L0 Cache 数据错误.如上一节所描述的,L0 Store Buffer 溢出时会抛弃后续指令更新 L0 Cache 的数据.这就会使某些正确的数据不能更新到 L0 Cache 中,从而导致了 Load 指令投机执行错误.

为了保证 L0 Cache 能有效降低访存延时,尽量减少 L0 Cache 的错误次数是关键.仿真结果表明在

Store Buffer 条目数足够的情况下,绝大多数错误是由于上述第一种——不同的访存数据长度导致. LSQ 的数据前递很好地降低了第二种错误的发生.

为了减少这一投机错误的发生,采用定长读取的策略固定读取 L1 Cache 及其下级存储层次的数据长度,即当 Load 指令访问 L1 Cache 或更下级存储层次时,无论该指令读取的数据长度为多少,都以 ALPHA 指令集中最长的访存操作数据长度——8 字节来取数据.当指令提交需要更新 L0 Cache 时把处理器所需的数据部分提交给寄存器,同时把全部的 8 字节的数据存入 L0 Cache.这样就有效避免了大多数错误的发生,同时也不增加过多的额外开销.

3 仿真实验

本文在 M5 模拟器^[8]的基础上进行 L0 Cache 结构建模. M5 模拟器是密西根大学 (University of Michigan) 研究组使用 C++ 语言编写用于处理器仿真的模拟器. 与其他的处理器模拟器相比, M5 不仅仅提供了处理器核 (CPU core), 还包括了存储器、I/O 等外围设备的模块, 可以模拟出完整的系统运行过程. 在此基础上, 本文建立了周期精确、参数可配置的分簇超标量处理器, 并加入了 L0 Cache 结构. 测试采用的处理器默认是 1×8 的分簇处理器, 具体配置如表 1 所示. 同时表 1 也包括了除 L0 Cache 以外的存储结构层次, L0 Cache 的访存延迟设为一个周期.

表 1 默认的处理器的配置参数

簇的数量	8 个
单个簇的指令窗口大小	16 项
单个簇的发射宽度	1 条
单个簇的功能单元数量	IntALU、IntMult、FPALU、FPMult 各一个
单个簇的寄存器数量	整形 48 个和浮点 48 个
簇间连线延时	2 个周期
指令分派策略	基于指令相关性的调度算法
取指宽度	每周期 8 条指令(最多 2 条分支指令)
Load/Store 队列大小	64 项
重定序缓冲大小	192 项
取指队列长度	64 项
Load 投机错误恢复机制	该指令到达 ROB 头部时刷流水线
L0 Store Buffer 条目数	8 项/簇
L1 D-Cache	64kB, 4 路组相连, 块大小 64 位 命中延时 4 个周期
L1 I-Cache	32kB, 2 路组相连, 块大小 64 位 命中延时 1 个周期
L2 Cache	256kB, 4 路组相连 块大小 64 位 命中延时 10 个周期
内存延时	100 个周期

由于 L0 Cache 没有缺失处理, 所有写操作都按

命中处理, 故只做读操作的命中率统计. L0 Cache 的读命中率标志着处理器中 Load 指令从 L0 Cache 中读取数据次数的多少, 其次数越多说明 Load 指令投机执行的次数越多, 在保证正确率的前提下, Load 延时也就越小, 性能提高越明显. 其中读命中率计算公式如下:

$$\text{L0 Cache 读命中率} = \text{L0 Cache 读命中数} / \text{L0 Cache 读次数}$$

图 3、图 4 分别是整型和浮点测试程序在没有读更新 L0 Cache 的情况下不同 L0 Cache 容量和组相连的 L0 Cache 命中率. 仿真可知, 就 4kB, 2 路组相连的 L0 Cache 来说, 平均读取命中率为 22.59%, 其中整型程序平均读取命中率为 28.15%, 而浮点程序平均读取命中率为 17.83%. 整体而言, 整型程序的平均命中率要高于浮点程序. 无论是在整型还是浮点程序中, 增大 Cache 容量或者将直接映射变为 2 路组相连在绝大多数情况下可以提高 L0 Cache 的读取命中率, 但是提高的幅度并不是很大. 而不同的程序特性对 L0 Cache 的读取命中率影响很大, 而在 mcf、swim、lucas, L0 Cache 命中次数极少. 这是因为这些程序在程序局部性上表现较差, 甚至在容量 64kB 的 L1 Dcache 中命中也较低, 例如 mcf、lucas 的 L1 DCache 缺失率分别为 99% 和 47%.

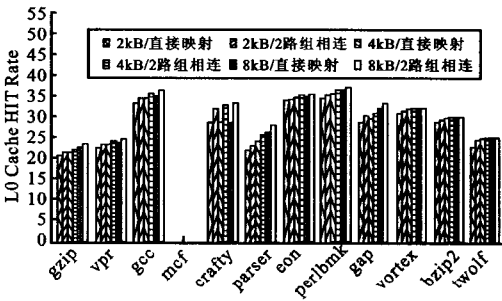


图 3 整型程序无读更新 L0 Cache 时 L0 读命中率

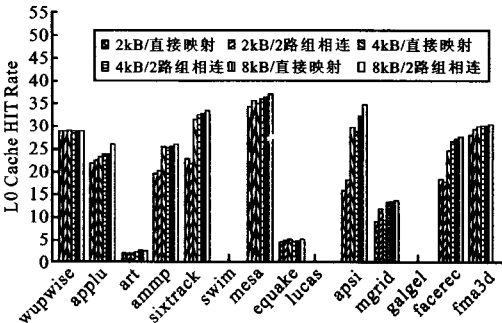


图 4 浮点程序无读更新 L0 Cache 时 L0 读命中率

为进一步提高 L0 Cache 的读取命中率, 在原有

结构的基础上加入了 L0 Cache 读更新的策略,即每一次 Load 指令读 L0 Cache 缺失时将 L1 Cache 中得到数值在 Load 提交时写回 L0 Cache 中.图 5、图 6 分别为在整型和浮点程序中加入读更新 L0 Cache 后的 L0 Cache 的读取命中率.从图中可以看出 L0 Cache 的读取命中率有了显著的提高.就 4kB,2 路组相连的 L0 Cache 而言,平均读取命中率为 34.82%.其中整型程序平均读取命中率为 40.75%,浮点程序平均读取命中率为 29.73%,同比增长分别为 44.8% 和 66.7%.读更新策略虽然大大提高了 L0 Cache 的命中率,但是同时需要考虑的是加入读更新后,对 L0 Cache 的写操作次数会大大增加,这会增加 L0 Cache 的开销和复杂度.

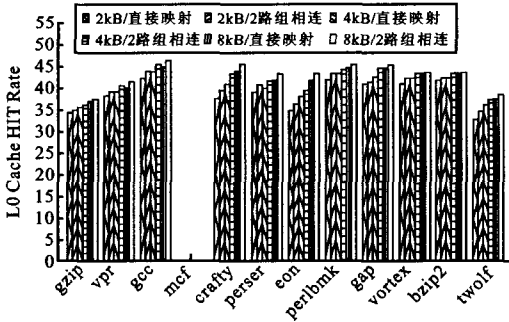


图 5 整型程序在读更新 L0 Cache 时的 L0 读命中率

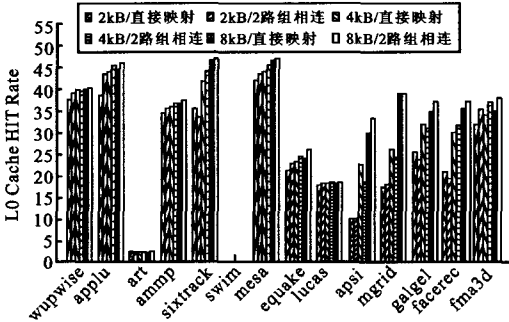


图 6 浮点程序在读更新 L0 Cache 时的 L0 读命中率

L0 Cache 的投机正确率是直接影响 L0 Cache 好坏的重要因素.因为 Load 每命中 L0 Cache 且投机执行正确一次,可以加快程序若干个周期,这取决于 L0 命中延时和 L1 命中延时的差值.而一旦投机错误,就需要刷新流水线来恢复错误,这会带来几十个到几百个时钟周期的损失.使 L0 Cache 得到一个高的投机正确率是决定 L0 Cache 结构好坏的关键.其中投机正确率的计算公式如下:

$$\text{投机正确率} = \text{投机正确次数} / \text{命中次数}$$

表 2 所示为 4kB,组相连为 2 的 L0 Cache 配置下的投机正确率,其他配置的 L0 结果基本类似.在

没有读更新的情况下,处理器执行 2 千万个周期的平均投机正确率为 99.5%.这种情况下 L0 Cache 的命中率不高,并没有最大程度的利用 L0 Cache.加入读更新机制可以大幅提高 L0 Cache 的读取命中率.加了读更新机制后的正确率下降明显,结果现实错误次数也随之增多约 6.7 倍.为了解决这一问题,在读更新基础上采用了固定读取数据长度的方法,统计发现采用该方法后正确率提高到了 99.92%.LSQ 对执行中(in-flight)的访存指令进行了去歧义和数据前递操作以及程序局部性特性很好得保证了 L0 Cache 投机访问的正确性.

表 2 4kB,2 路组相连的 L0 Cache 投机正确率

测试程序	基本结构	加入读更新机制后	固定数据长度后
gzip	99.4	82.2	100.0
vpr	97.0	97.0	99.7
gcc	99.9	99.5	99.9
mcf	98.3	98.3	100.0
crafty	99.9	99.5	99.9
parser	99.9	99.3	99.9
eon	100.0	99.9	100.0
perlbnk	99.9	98.2	99.9
gap	98.3	98.9	99.5
vortex	99.9	99.6	99.9
bzip2	97.2	89.1	100.0
twolf	99.4	97.5	99.6
mesa	99.2	97.6	99.9
equake	100.0	99.5	100.0
其余 SPEC2K	≈100.0	≈99.9	≈100.0
Average	99.51	98.27	99.92

分簇处理器 IPC 的加速比是体现投机的 L0 Cache 结构对处理器整体影响的一个指标.图 7 所示为在 1x8 的分簇处理器中采用本文所提出的 4kB,2 路组相连的 L0 Cache 与未采用 L0 Cache 的 IPC 加速比归一化图.从图中可以看出,与文献[7]的结果相比,本文的 L0 Cache 结构使分簇处理器的性能得到了进一步地提高. IPC 平均提高了 5.583%,其中整型程序提高了 7.421%,而浮点程序提高了 4.113%,个别程序如 eon,art 甚至提高了 20%以上.加入 L0 Cache 结构后对整型程序的性能提升要明显优于浮点程序.相对于整型程序,容量较小、块大小较小的 Cache 较难利用浮点程序的空间和时间局部性,因而对性能提升较小.图 8 所示为在整型程序中不同容量不同策略的 L0 Cache 结构对性能提升的影响. L0 Cache 容量的增加对整体性能的提升并不显著,而加入读更新和固定读取数据长

