基于 GCC 编译器的流式存储优化方法

高秀武，黄亮明，姜军

引用本文

高秀武，黄亮明，姜军. 基于 GCC 编译器的流式存储优化方法[J]. 计算机科学, 2022, 49(11): 76-82.

GAO Xiu-wu, HUANG Liang-ming, JIANG Jun. Optimization Method of Streaming Storage Based on GCC Compiler[J]. Computer Science, 2022, 49(11): 76-82.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.211200252

基于 GCC 编译器的流式存储优化方法

高秀武　黄亮明　姜　军
江南计算技术研究所　江苏 无锡 214083
(wxgcs@163.com)

摘　要　针对流式存储访问引起的缓存污染与强制性缺失问题,部分高性能通用处理器平台提供了不经过缓存而直接访问存储器的专用通路及配套指令支持。在常见的流式存储应用场景中,合理采用直访主存方式可以提高芯片存储器系统的整体性能。然而,判断何时使用直访主存能够获得收益对于程序员来说是一项十分繁琐且容易出错的任务,一种行之有效的方法是通过编译器自动实现。因此,文中在深入分析流式存储访问模式使用不同类型访存操作性能收益的基础上,提出了基于 GCC 编译器的流式存储优化方法。该方法由编译器自动实现对程序员透明,在 GCC 编译器 SSA-GIMPLE 阶段对程序循环中具有流式访问特征的连续写或者跨步写进行识别,并根据收益分析与依赖关系筛选优化对象,最后在编译器后端匹配指令模板生成直访主存指令。使用连续/跨步写用例与 STREAM 测试集及变体在申威国产处理器平台上进行实验评估,结果表明,文中提出的优化方法能够显著缩短流式存储应用程序的执行时间,优化后 STREAM 测试集的平均加速比为 1.31。另外,文中实现的流式存储优化与循环展开优化一起使用效果更好,STREAM 测试集的平均加速比能达到 1.45。

关键词:GCC 编译器;直访主存;编译优化;代码生成;国产处理器
中图法分类号　TP311

Optimization Method of Streaming Storage Based on GCC Compiler

GAO Xiu-wu, HUANG Liang-ming and JIANG Jun
Jiang Institute of Computing Technology, Wuxi, Jiangsu 214083, China

Abstract　To solve the problem of cache pollution and mandatory loss caused by streaming memory access, some high-performance general-purpose processor platforms provide a dedicated path and supporting instructions for accessing memory directly without accessing the cache. The overall performance of chip memory system can be improved by using direct memory access in common application scenarios such as streaming storage. However, it is a tedious and error-prone task for programmers to determine when direct access to main memory is beneficial, and an effective way is to implement it automatically through the compiler. Therefore, based on the in-depth analysis of the benefits of different types of access operations under the streaming storage access mode, this paper proposes a streaming storage optimization method based on GCC compiler. In the SSA-GIMPLE stage of GCC compiler, the continuous write or step write with stream access characteristics in the program loop is recognized, and optimization objects are screened according to the benefit analysis and dependency relationship. Finally, the direct access main memory instructions are generated by matching instruction templates at the back end of compiler. The continuous/step-write case and STREAM test set and their variants are used for experimental evaluation on SW domestic processor platform. The results show that the optimized method can significantly reduce the execution time of STREAM storage applications, and the average acceleration ratio of STREAM test set after optimization is 1.31. Additionally, in conjunction with loop unwinding optimization, the STREAM test set has an average acceleration ratio of 1.45.

Keywords　GCC complier, Direct memory access, Compiler optimization, Code generation, Domestic processor

1　引言

现代处理器系统普遍采用冯·诺依曼体系结构,指令与数据存储在同一存储器中,系统运行速度严重依赖处理器对存储器的访问速度。然而,由于制造工艺的不同,处理器与储存器的性能增长呈现不均衡的发展,导致了存储器的读取速度严重滞后于处理器处理速度的“存储墙”问题[1-2]。为了解决“存储墙”问题,研究人员针对程序局部性原理[3-4]设计了基于快速缓存的层次存储结构[5-6]来提高存储系统的性能。同时,针对局部性较好的应用程序,通过预取技术[7-10]将数据

到稿日期:2021-12-22　返修日期:2022-04-29
基金项目:国家重点研发计划(2020YFB0204602);综合研究项目(针对申威处理器的编译优化提升技术)
This work was supported by the National Key Research and Development Project(2020YFB0204602) and Comprehensive Research Project(Research on Compilation Optimization Improvement for Sunway Processor).
通信作者:黄亮明(liangming_huang@126.com)

提前放到缓存中，提高缓存命中率以缩短应用程序的访问延迟，从而提高应用程序的整体性能。

然而，随着大数据、云计算以及人工智能等新兴领域的快速发展，应用程序中数据的局部性变得越来越复杂，其呈现的访存模式也越来越多样化，给基于快速缓存的层次存储结构带来了挑战。Jaleel 等[11]将应用程序中常见的访存模式分为 4 种：友好访问模式、流式访问模式、颠簸访问模式和混合访问模式。友好访问模式指短时间内会重复访问某一段数据，该模式具有良好的时间或空间局部性，能够充分发挥高速缓存的效率；流式访问模式指某段数据块被连续访问且只访问一次，该模式时间局部性差，会引起高速缓存强制性缺失；颠簸访问模式指周期性地访问某段长度的数据块，具有一定的时间或者空间局部性，但当访问数据块的长度超过缓存容量范围时，会因为缓存冲突而导致数据块未被访问就被淘汰出缓存；混合访问模式是上述 3 种模式综合混合的结果，往往局部性不高，对缓存的放置与淘汰策略是一个较大的考验。其中，流式访问模式和颠簸访问模式在一定程度上具有相同的特征，即数据顺序访问且在缓存容量范围内不可重用，可以称之为具有流式存储特征的数据访问模式。该模式在基于高速缓存的层次存储结构系统中会带来大量的写不命中时的读后写操作与严重的缓存污染[12]，大大降低了高速缓存的效率。

针对流式存储访问带来的缓存污染问题，处理器设计者提出直访主存操作[13-14]，即在层次结构的存储器系统中访存操作不经过缓存，而通过旁路直接访问存储器。直访主存操作的数据不进入缓存，避免了对缓存已有数据造成污染，且在存储之前不需要进行繁琐的 RFO 操作，可以提高存储器系统整体的访存性能。当前主流处理器厂商 Intel 和 ARM 等都提供了丰富的直访主存指令，如 Intel SSE 中的 MOVNTDQA，MOVNTDQ，MOVNTQ 等指令[15]，ARM 体系中的 LDNP，STNP 指令[16]。一般来说，直访主存指令可以由程序员通过汇编编程、调用编译器内置函数接口等方式优化程序，但判断何时使用直访主存指令能够带来收益对程序员来说是一件繁琐且容易出错的任务，一种更加高效可行的方法是编译器通过算法自动生成直访主存指令。各类编译器在直访主存指令自动生成方面存在较大差距，商业编译器如 ICC[17-18] 可以依据程序特征高效地生成直访主存指令，而开源编译器 GCC 支持较弱，面对大部分程序不能很好地生成直访主存访存指令。因此，本文基于流式访存模式的读写操作收益分析研究直访主存指令自动生成的算法，提出了一种基于 GCC 编译器的流式存储优化方法，并在国产申威平台上进行了实现。

2 研究背景

2.1 GCC 编译器系统结构

GCC[19]（GNU Compiler Collection）是 GNU 工程中的核心工具软件，支持多种前端的编程语言，包括 C，C++，Java，Ada，Go 和 Fortran 等，其编译生成的目标代码可以在几乎所有的处理器平台上运行。由于 GCC 具有支持多语言、多目标平台、源代码开发以及提供了丰富的代码优化方法等优点，是目前使用与研究最为广泛的编译器系统之一。GCC 主要包括前端、中端、后端 3 个部分，其逻辑结构如图 1 所示。

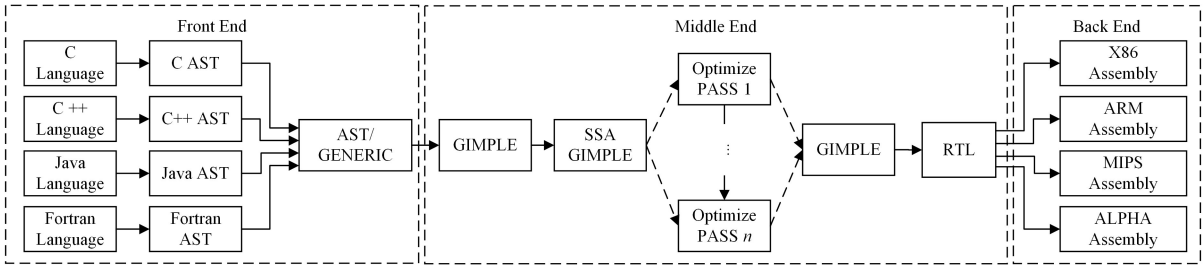

图 1 GCC 逻辑结构

Fig. 1 GCC logical structure

由图 1 可知，GCC 编译流程可以分为 3 个阶段：前端高级语言分析及标准化、中端中间语言（GIMPLE 与 RTL）处理与优化和后端代码生成。前端对使用 C/C++，Java，Ada，Go，Fortran 等高级语言编写的程序进行词法/语法分析、AST 标准化等操作，将其转换成规范化的 AST 中间表示（AST/GENERIC）。中端将 AST/GENERIC 转化为与前端语言无关的 GILPLE 中间表示，并进行与目标机器无关的优化，然后将经过优化的 GIMPLE 中间语言转化为与目标机器相关联的 RTL 中间表示同时进行与目标机器相关的优化。编译器后端结合目标机器指令模板将 RTL 中间语言转换为目标机器的汇编代码。

2.2 中间语言优化模块管理

GCC 在将高级程序源代码翻译为目标机器汇编代码的过程中，主要使用了 AST，GIMPEL 以及 RTL 这 3 种中间表示。AST 即抽象语法树（Abstract Syntax Tree），用于对前端高级语言的源代码进行规范的抽象表示。GIMPLE 中间表示是为了对各种语言的 AST 进行语言无关处理与优化而引入的，对 GIMPLE 中间表达式可进行对大多数处理器都有效的优化变换，如函数内联、冗余代码删除、数据预取以及循环变化等。RTL 即寄存器传输语言（Register Transfer Language），其是为了将与机器无关的 GIMPLE 中间表示转换成与机器相关的汇编语言而引入的，主要进行与目标机器相关的优化变化，包括循环优化、指令调度、寄存器分配、窥孔优化等。

GCC 目前对中间语言已经实现数百种优化方法，使用一个基于遍（Pass）的优化管理器进行管理，如图 2 所示。该

优化遍管理器将优化方法划分为一个个优化遍，每个优化遍执行一种特定的优化，其输出结果将作为下一个优化遍的输入。在 passes.c 源码文件中详细描述了优化遍执行顺序，并通过链表组织在一起。

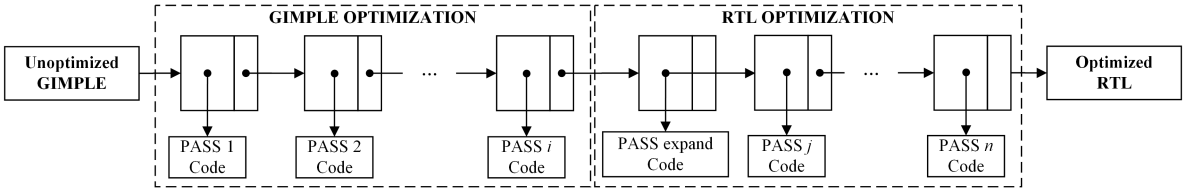

图 2 GCC 优化遍管理器

Fig. 2 GCC optimize passes manager

2.3 后端代码生成模块

GCC 编译器能够支持多种后端的目标机器，其中机器描述起到了重要作用。机器描述将目标机器的特性引入编译器中，指导编译器根据目标机器的特性进行 insn 生成与优化，并最终完成目标代码的生成。因此，每一个目标机器在 GCC 中都有属于自己的机器描述用于指导指令生成，其主要包括机器描述(Machine Description，MD)文件、机器描述头文件与 c 文件两部分。机器描述头文件与 c 文件主要描述了与目标机器相关的变量声明与函数实现。MD 文件则描述了目标机器所支持的每条指令的指令模板，指令模板定义包含了指令模板名称、RTL 模板、条件、输出模板及属性等 5 部分内容的详细规定，图 3 给出了一个运算指令模板示例。

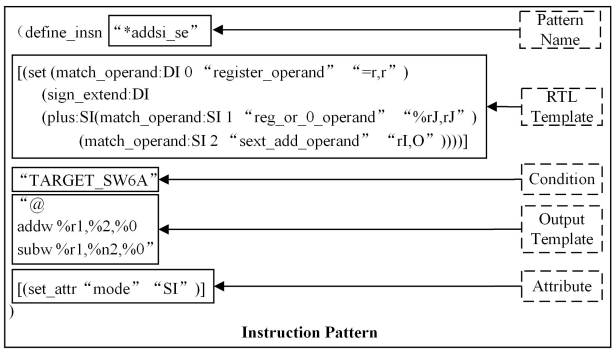

图 3 指令模板示例

Fig. 3 Instruction template example

3 直访主存优化收益分析

3.1 处理器访存方法分析

现代处理器系统普遍采用基于缓存的层次存储结构来缩减存储器与处理器之间的性能差距。处理器访问内存需经过多级缓存，即数据总是先进入缓存而不直接读取主存。如图 4(a)所示，步骤①②描述了一次读请求不命中缓存的执行过程，访存部件收到读请求后，首先依次在各级缓存中均找不到副本(即不命中缓存)，然后将主存中包含所需数据的 Cache 块逐级拷贝到 L1 高速缓存，并将所需数据读取到寄存器。命中缓存时则直接从对应缓存拷贝 Cache 块到 L1 高速缓存。相比读请求，写请求执行过程更复杂，通常为了最大化发挥程序局部性潜能还会采用写分配与写回两种缓存机制[20]。如图 4(a)所示，步骤①③④描述了写请求不命中缓存的执行过程，前期与读请求操作类似，根据写分配策略将包含旧数据的 Cache 块从主存逐级拷贝到 L1 高速缓存，并更新相应位置的数据，然后遵循写回机制直到数据所在缓存块被替换时才逐级淘汰写回主存。基于缓存的常规访存的优点是可以充分地利用程序局部性原理，减少对主存访问的实际次数，从而提升存储系统的性能；缺点是对于程序局部性较差的应用，不但不能命中缓存，还会造成缓存污染，而且在多核环境下维护 Cache 一致性还会花费大量的时钟周期。

相比常规访存操作数据总是先进入缓存，直访主存操作通过旁路直接存取主存，针对局部性差的应用程序不但可以避免 Cache 污染还可以节省存储系统的带宽。图 4(b)给出了直访主存操作的过程，读/写数据均不需要进入高速缓存而通过旁路直接访问存储器空间，即高速缓存不需要进行任何操作。因此，对于局部性较差的流式访问与颠簸访问等应用场景，使用直访主存指令进行访问更加高效。

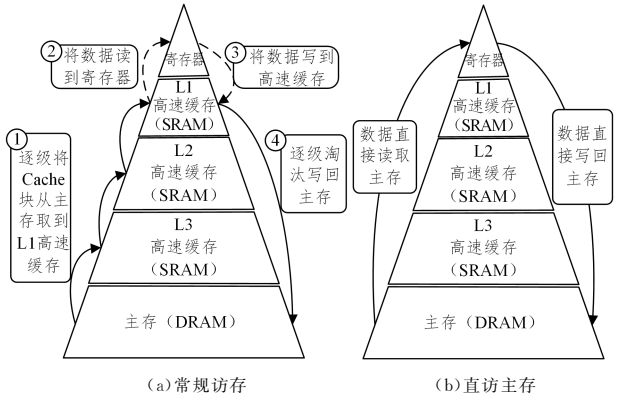

图 4 常规访存与直访主存

Fig. 4 Routine access anddirect memory access

3.2 流式模式下的直访主存收益分析

直访主存操作数据不进入 Cache 直接存取主存，其优点是能够减少 Cache 污染与节省存储带宽，缺点是无法发挥程序访存的局部性潜能。程序局部性较差时使用直访主存操作能够获得更好的性能，而程序局部性较好时则反之。因此，根据程序访存特征合理使用直访主存指令是充分发挥直访主存操作作用的关键。本节对流式访问模式下不同读写场景进行收益分析，以确定适合使用直访主存操作的情形。

(1)连续读模式

命中 Cache 时常规访存访问高速缓存，其性能优于直访主存；不命中 Cache 时，连续读因空间局部性，常规访存处理一个 Cache 行数据只需访存一次。直访主存直接读取主存，当多次读请求能及时合并时也只需访存一次。但一般情况下因处理器资源的有限性与读操作的即时性，访存请求队列很少

能够及时合并多条直访主存的读请求，因此不命中 Cache 情况下直访主存也比常规访存性能低。

(2)连续写模式

命中 Cache 时，常规访存直接将数据写入 L1 高速缓存后根据写回机制逐级写回主存，处理一个 Cache 行数据只需一次写主存；不命中 Cache 时，常规访存需先进行写分配然后写回，处理一个 Cache 行数据需要两次访存。直访主存操作直接访问主存，如果一个 Cache 行数据的多个写请求能够合并，处理一个 Cache 行数据只需一次访存；如果写请求不能有效合并，则处理一个 Cache 行的数据对象需写主存两次以上。因此，若连续写同一 Cache 行数据的写请求能有效合并，则直访主存的性能明显优于常规访存；若不能有效合并，因直访主存执行周期更短，直访主存与常规访存相比性能略高或者相当。

(3)跨步读/写模式

跨步读/写相比连续读/写来说，一个 Cache 行所需访问数据数量成倍减少，缓存效率降低，访存请求被合并的概率增大。读操作随着跨步距离增大会逐渐缩短直访主存与常规访存的性能差距，而写操作随着跨步距离的增大直访主存逐渐优于常规访存。

综上所述，连续写操作和跨步写操作是直访主存性能优于常规访存的常见场景，这两种模式在新兴的三维图像[12]、人工智能[21]、天气预报[22]、计算流体力学[22]等应用程序中应用较多。因此，本文提出了基于 GCC 编译器的流式存储优化方法，在 GCC 编译器中对连续、跨步写操作两种访存模式实现自动识别与代价分析，并在后端代码生成中匹配生成直访主存指令，从而生成更高质量的目标码。

4 针对流式存储的直访主存编译优化实现

前文介绍了 GCC 编译器的基本结构与编译优化管理策略，本文提出的流式存储优化方法主要在编译器中端和后端进行实现。中端通过优化管理器在优化遍链表中添加一个流式存储优化遍，实现访存模式的自动识别与收益分析，最终确定直访主存的优化对象；后端则在机器描述文件中添加直访主存指令模板，并在代码生成时匹配生成的直访主存指令与存储器栏栅指令，在确保优化的同时保证存储访问顺序的正确性。

4.1 流式存储优化遍

4.1.1 优化选项与优化遍定义

在 GCC 优化管理器中添加一个优化遍，首先需要在 common.opt 文件中定义一个优化选项。流式存储优化遍选项的定义如图 5 所示，其中 fstreaming-store 为流式存储优化遍选项名称，GCC 编译程序时以此来控制是否执行流式存储优化遍；以 Common Report Var 开头的第二行内容定义了优化选项开关，这是连接定义与实现的桥梁，Init(1)表示默认打开；第三行为优化遍选项含义的描述。

```
fstreaming-store
Common Report Var(flag_streaming_store) Init(1) Optimization
Generate streaming nocache store instructions, if available, for arrays in loops
```

图 5 流式存储优化遍选项定义

Fig. 5 Streaming store optimization option definition

流式存储优化遍属于 GIMPLE 优化遍，其具体定义如算法 1 所示。其中，pass_data_loop_ streaming_store 定义了优化遍的数据结构，包括优化遍类型、名称、优化标识、处理时间、执行条件、TODO 标识以及各种属性参数等信息；gimple_opt_pass 定义了优化遍的所有成员函数，gate 函数实现了优化选项开关，当判断 flag_ streaming_store 大于 0 时执行该优化遍，execute 函数指定了流式存储优化遍入口函数 tree_ssa_streaming_store_arrays。

算法 1 流式存储优化遍的定义

```
1. namespace {/* 优化遍命名空间 */
2. const pass_data pass_data_loop_streaming_store={
   GIMPLE_PASS,/* type */
   "streaming_store",/* name */
   OPTGROUP_LOOP,/* optinfo_flags */
   TV_TREE_STREAMING_NOCACHE_STORE,/* tv_id */
   ( PROP_cfg | PROP_ssa ),/* properties_required */
   0,/* properties_provided */
   0,/* properties_destroyed */
   0,/* todo_flags_start */
   0,/* todo_flags_finish */
   };
3. class pass_loop_ streaming_store:public gimple_opt_pass {
   public:pass_loop_streaming_store(gcc::context *ctxt):gimple_opt
   _pass:(pass_data_loop_streaming_store,ctxt) {}
   virtual bool gate(function *) {
   return flag_streaming_store > 0;}
   virtual unsigned int execute(function *){ return tree_ssa_strea-
   ming_store_arrays();}
}.
```

4.1.2 流式存储优化算法的流程

流式存储优化算法的框架如图 6 所示，包括访存信息收集、重用性分析、收益分析等。

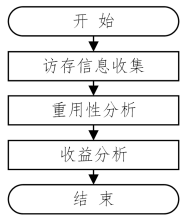

图 6 直访主存优化算法框架

Fig. 6 Framework of direct memory access optimization algorithm

访存信息收集是流式存储优化的基础，包括内存引用信息的收集与访存模式判断，其具体的实现过程如算法 2 所示，自顶向下依次扫描循环基本块的每条 GIMPLE 语句，通过 GCC 提供的 REFERENCE 接口判断 GIMPLE 语句是否存在数组访问。如果存在数组访问，则采用标量演化算法对数组地址进行分析。若数组地址可用线性表达式 $addr=base+step\times iter$ 进行解析，则成功识别数组访问的访存模式为连续写或者跨步写，否则丢弃该内存引用。其中，*base* 表示数组起始地址，*step* 表示前后两次访问元素的距离，*iter* 表示循环归纳变量。

算法 2 访存信息收集

```
输入:(loop)/*待优化的循环*/
输出:(refs)/*循环内的访存信息*/
1. for i:=0 to loop->num_nodes
2.   bsi<-gsi_start_bb(body[i])
3.   while(gsi_end_p(gsi)) do
4.     stmt<-gsi_stmt(gsi)
5.     xhs<- gimple_assign_xhs(stmt) //分别获取 stmt 语句左表达式与右表达式
6.     if(REFERENEC_CLASS_P(xhs)) //判断表达式是否存在访存引用
7.       gather_memory_references_ref(loop,&refs,xhs,stmt)//收集访存信息,并存储到 refs 数据结构中
8.     end if
9.     end while
10. end for
```

文献[23]将循环级数组访问重用性分为时间重用性、空间重用性、组重用性 3 种类型。时间重用性指同一内存位置在循环中被多次访问,空间重用性指同一内存引用在循环中访问同一 Cache 块的不同内存位置,组重用性指不同内存引用在循环中访问同一 Cache 块。针对本文的流式存储优化,借鉴其思想采用时间重用性与组重用性对内存引用进行优化对象筛选。具体实现过程如算法 3 所示,使用嵌套循环依次分析算法 2 收集的访存信息相互之间是否存在写后读与写后写两种依赖关系,若存在则丢弃,否则保留。

算法 3 流式存储优化对象筛选

```
输入:(refs)/*待筛选的访存对象*/
输出:(streaming_store_refs)/*筛选后访存对象*/
1. ref1<-refs
2. ref2<-ref1->next
3. while(ref1 && ref1->write) do
4.   streaming_store_flag<-true
5.   while(ref2) do
6.     if(rely_analysis(ref1,ref2))//依赖分析
7.       streaming_store_flag<-false
8.       break
9.     else ref2<-ref2->next
10.    end if
11.  end while
12.  if(streaming_store_flag)
13.      add(streaming_store_ref,ref1)
14.  end if
15. end while
```

流式存储优化方法收益分析如算法 4 所示,首先根据内存引用的时间重用性计算重用距离,然后与预设阈值(DMA_DISTANCE)进行比较。若重用距离大于阈值,则将直访主存标记 direct_memory_access_flag 置 true,并用直访主存中间表达式替换常规访存中间表达式;否则不改变访存对象中间表达式,因此不会对采用本文优化无收益的非流式访存产生影响。另外,由于硬件不能保证对相同存储器空间直访主存与常规访存之间的顺序,还需通过插入存储器栏栅指令来保证两者之间的访问顺序,插入存储器栏栅指令的具体实现根据直访主存标记来判断是否在循环的结尾处插入存储器栏栅指令。

算法 4 收益分析

```
输入:(streaming_store_refs)/*筛选后的访存信息*/
输出:(direct_memory_access_flag)/*直访主存标记*/
1. direct_memory_access_flag <- false
2. ref<-streaming_store_refs
3. while(ref) do
4.   dist<-self_reuse_distance(dr,loop_data_ size,n,loop)//计算重用距离
5.   if(dist > DMA_DISTANCE)
6.     direct_memory_access_flag <- true
7.     gimple_assign_set_direct_memory_store(ref->stmt, true)// 直访主存中间表达式替换常规访存中间表达式
8.   end if
9. end if
10. if(direct_memory_access_flag)
11.   emit_mfence_after_loop(loop)
12. end if
```

4.2 直访主存指令代码生成

直访主存指令代码生成主要在编译器后端完成,具体流程如下:针对直访主存中间表达式,通过 SPN 模板构造相应 INSN,并匹配固定的直访主存指令模板生成汇编代码。为了简化直访主存指令生成,用 define_expand 与 define_insn 分别对直访主存指令进行定义。如算法 5 所示,define_expand 类型指令模板用于构造 SPN 并生成 INSN;如算法 6 所示,define_insn 类型指令模板用于匹配生成最终的汇编代码。

算法 5 defin_expand 类型指令模板

```
1.(define_expand "storentdi"
2.  [(set(match_operand:DI 0 "nonimmediate _operand")(unspec:DI [(match_operand:DI 1 "input _operand")] UNSPEC_NTDI))]
3.    "flag_streaming_nocache_store==1"
4.    {
5.      if(sw_64_expand_mov(DImode,operands))
6.        DONE
7.    }
8.)
```

算法 6 defin_insn 类型指令模板

```
1.(define_insn "*storentdi"
2.    [(set(match_operand:DI 0 "nonimmediate_ operand" "=m")(unspec:DI[(match_operand:DI 1 "input_operand" "rJ")] UNSPEC_NTDI))]
3.    "register_operand(operands[0],DImode) || reg_or_0_operand(operands[1],DImode)"
4.    "stl_nc %r1,%0"
5.    [(set_attr "type" "ist")(set_attr "isa" "*")(set_attr "usegp" "*")]
6.)
```

5 实验

本文的实验平台采用国产申威多核处理器系统,基于开源编译器 GCC(版本 GCC7.1.0)实现流式存储优化方法,测试集使用连续/跨步写用例与业界广为流行的综合性内存带宽性能测试集 STREAM(5.10 版本)。

5.1 连续/跨步写评测与分析

连续/跨步写用例如图 7 所示，对一个无符号长字整数类型数组进行循环赋值，并通过 STEP 宏控制跨步距离，使用 gettimeofday 函数来计时。

```
double mysecond(){
   struct timeval tp;
   struct timezone tzp;
   gettimeofday(&tp,&tzp);
   return ((double) tp.tv_sec + (double) tp.tv_usec * 1.e-6);
}
void main(){
   unsigned long sum=0,i;
   unsigned long a[ARRAY_SIZE];
   double t,t1,t2;
   for (i=0;i<ARRAY_SIZE;i+=STEP) a[i]=1;
   t1=mysecond();
   for(i=0;i<ARRAY_SIZE;i+=STEP) a[i]=i;
   t2=mysecond();
   t=t2-t1;
   for (i=0;i<ARRAY_SIZE;i+=STEP)
      sum+=a[i];
   printf("sum=%ld,t=%lf\n",sum,t);
}
}
```

图 7 只写存储测试用例代码

Fig. 7 Write only stored test case code

对连续/跨步写测试用例进行编译时，基准测试采用-O2 选项编译，优化测试只额外打开流式存储优化选项，其他配置与基准测试完全一样。以连续写用例为例，优化前与优化后的汇编代码如图 8 所示，优化前使用常规访存指令 stl 进行数据存储，而优化后使用直访主存指令 stl_nc 进行数据存储，并在循环末尾添加存储栅栏指令 memb。

```
$L5:
  stl  $1,0($2)
  ldi  $1,1($1)
  ldi  $2,8($2)
  ldih $3,-3584($1)
  bne  $3,$L5
```

(a)优化前汇编代码

```
$L5:
  stl_nc  $1,0($2)
  ldi  $1,1($1)
  ldi  $2,8($2)
  ldih $3,-3584($1)
  bne  $3,$L5
  memb
```

(b)优化后汇编代码

图 8 连续写存储测试用例优化前与优化后的汇编代码

Fig. 8 Write only stored assembly code before and after test case optimization

图 9 给出了连续/跨步写用例优化前后的运行时间，可以看出，使用直访主存指令测试用例的运行时间明显低于使用常规访存指令，说明本文提出的优化方案可以有效提升连续/跨步写等流式存储应用场景的性能。为了进一步探索直访主存的优化潜能，结合循环展开技术对用例进行优化，从图中可知，循环展开对常规访存毫无效果，而对直访主存有一定提升。这是因为常规访存普遍采用写回策略，即数据写到缓存后并不立即写回主存，而是根据淘汰策略依次写回主存；而直访主存通过循环展开可以增加写请求合并的机会，减少对主存访问的次数，从而提升程序性能。

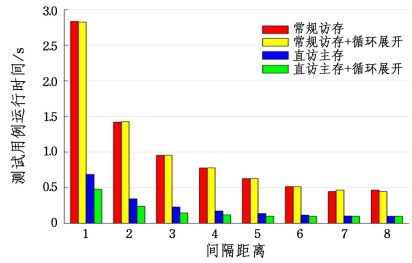

图 9 连续/跨步写测试用例的运行时间

Fig. 9 Running time of continuous writing and stride to write

5.2 STREAM 评测与分析

STREAM 通过 C 语言编写完成，针对双精度浮点数组进行 Copy，Scale，Add，Triad 这 4 种操作，具体如表 1 所列。显而易见，STREAM 测试集中的 4 个测试项都具有明显的流式存储特征，即连续访存但数据在 Cache 容量范围内不会被重用。另外，为了充分验证本文实现的流式存储优化对跨步写操作的优化效果，我们对 STREAM 的 4 个测试项进行了改造，将循环变量 j 的变化步长从 1 逐步增加到 8 进行测试，对应测试包统一命名为 STREAM-Sx，x 表示跨步步长。

表 1 STREAM 测试项

Table 1 Test items of STREAM

Test item	Operation	Memory access
Copy	$c[j]=a[j]$	1R1W
Scale	$b[j]=\text{scalar}*c[j]$	1R1W
Add	$c[j]=a[j]+b[j]$	2R1W
Triad	$a[j]=b[j]+\text{scalar}*c[j]$	2R1W

对 STREAE-Sx 测试程序优化效果的评测方法与连续/跨步写用例相同，首先评测直访主存的优化效果，其次结合循环展开进一步探索优化的潜能。图 10 给出了循环展开、直访主存以及直访主存与循环展开相结合这 3 种优化配置对 STREAM-Sx 程序的性能提升情况。从图中可知，直访主存优化总体上对性能提升明显，STREAM-Sx 的平均性能提升了 31.6%，STREAM-S8 的性能提升程度最高，为 70%。另外，还发现循环展开对课题性能几乎没有效果，而与直访主存优化相结合后可大大提升课题的性能，STREAM-Sx 的平均性能提升了 44.9%，并超过了直访主存优化。因此，通过循环展开技术可以增加直访主存请求合并，减少对主存的访问次数，从而进一步提高 STREAM-Sx 课题的性能。

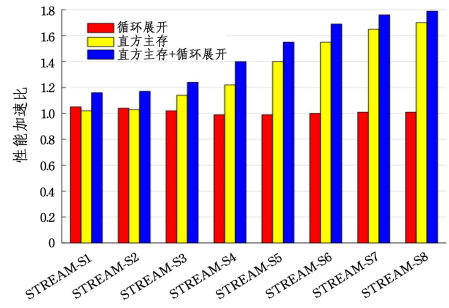

图 10 STREAM 性能提升的比例

Fig. 10 Percentage of STREAM performance improvement

针对 STREAM-Sx 程序的性能评测，直访主存优化均有较为明显的性能提升，这说明本文实现的流式存储优化对具

有流式存储的应用有较好的适应性。另外，从图 10 中可以看出，STREAM-Sx 性能提升程度的排序为 STREAM-S8＞STREAM-S7＞…＞STREAM-Sx＞…＞STREAM-S1。出现此现象的原因是，当跨步增大时一个 Cache 块的写操作减少，其合并的可能性增大，对主存该问次数减少。当然，如果跨步距离大于或等于 Cache 块大小，一个 Cache 块只有一次写，则性能收益不再增长而是一个稳定的提升比例。

结束语 本文针对日益严峻的“存储墙”问题，在流式应用场景下对常规访存与直访主存两种访存的操作原理与性能收益进行了对比分析，以此为基础在 GCC 编译器中实现了针对流式存储的编译优化，并进行了实验验证。首先，在编译器 GIMPLE 阶段自动识别程序循环中具有流式存储特征的连续写与跨步写操作，然后将两种操作下的常规访存中间表示替换为直访主存中间表示，最后在编译器后端代码生成过程中匹配生成直访主存指令。该方法通过编译器自动优化实现，用户不需增加额外操作，且对程序中其他访存模式下的指令生成无影响。实验数据表明，本文实现的优化能够解决普通访存指令在相关访问模式下效率不高的问题，在相关典型应用课题中获得了良好的优化效果。下一步的工作主要是研究直访主存优化与数据预取技术协同使用的问题。

参 考 文 献

[1] WULF W A, MCKEE S A. Hitting the memory wall: implications of the obvious [J]. ACM Sigarch Computer Architecture News, 1995, 23(1): 20-24.

[2] NOWATZYK A, PONG F, SAULSBURY A. Missing the Memory Wall: The Case for Processor/Memory Integration [J]. ACM Sigarch Computer Architecture News, 1996, 24(2): 90-101.

[3] DENNING P J. The Locality Principle [J]. Communications of the ACM, 2005, 48(7): 19-24.

[4] PING L. Analysis and Development of the Locality Principle [J]. Advances in Intelligent and Soft Computing, 2012, 133(7): 211-214.

[5] BRYANT R, O'HALLARON D. Computer systems: a programmer's perspective [M]. Upper Saddle River: Prentice Hall, 2003.

[6] VENKATESAN R, KOZHIKKOTTU V J, SHARAD M, et al. Cache Design with Domain Wall Memory[J]. IEEE Transactions on Computers, 2016, 65(4): 1010-1024.

[7] BAER J L, CHEN T F. An effective on-chip preloading scheme to reduce data access penalty [C]// IEEE Conference on Supercomputing. ACM, 1991.

[8] DONG Y S, LI C J. Mechanism and Capability of Data Prefetching in Intel © 64 Architecture [J]. Computer Science, 2016, 43(5): 34-41.

[9] TIMOTHY S A, JONES M. Software Prefetching for Indirect Memory Accesses[C]// 2017 IEEE/ACM International Symposium on Code Generation and Optimization(CGO). ACM, 2017.

[10] WANG J H, LI J, LU D D, et al. Hardware prefetching mechanism based on double step data stream[J]. Computer Engineering, 2019, 45(6): 115-118, 126.

[11] JALEEL A, THEOBALD K B, STEELY S C, et al. High performance cache replacement using re-reference interval prediction(RRIP)[C]// International Symposium on Computer Architecture. ACM, 2010.

[12] ZHUANG X T, LEE H. A hardware-based cache pollution filtering mechanism for aggressive prefetches[C]// 2003 International Conference on Parallel Processing. IEEE, 2003.

[13] PALANCA S, PENTKOVSKI V, TSAI S, et al. Method and apparatus for implementing Nontemporal stores. U. S. Patent 6, 205, 520[P]. 2001.

[14] SANDBERG A, EKLOV D, HAGERSTEN E. Reducing cache pollution through detection and elimination of Nontemporal memory accesses[C]// Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010: 1-11.

[15] Intel. Intel © 64 and IA-32 Architectures Software Developer's Manuals, Volume 2B: Instruction Set Reference [Z]. September 2016.

[16] ARM. ARM Architecture Reference Manual, ARMv8, for ARMv8-Aarchitecture profile [Z]. September 2016.

[17] KRISHNAIYER R, KULTURSAY E, CHAWLA P, et al. Compiler-Based Data Prefetching and Streaming Nontemporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor[C]// 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. IEEE, 2013: 1576-1586.

[18] Intel © C++ Compiler Classic Developer Guide and Reference [Z]. Version 2021.1, December 2020.

[19] Free Software Foundation, Inc. GCC, the GNU compiler collection [EB/OL]. (2017-05-02). https://gcc.gnu.org/.

[20] MILLER D W, III D. Performance analysis of disk cache write policies [J]. Microprocessors & Micro-systems, 1995, 19(3): 121-130.

[21] SPEC CPU2006 [EB/OL]. (2011-10-20). https://www.spec.org/cpu2006/Docs.

[22] SPEC CPU2017 [EB/OL]. (2021-04-07). https://www.spec.org/cpu2017/Docs.

[23] MOWRY T C, LAM M S, GUPTA A. Design and Evaluation of a Compiler Algorithm for Prefetching[J/OL]. Aplos, 1992. https://dl.acm.org/doi/epdf/10.1145/143365.143488.

GAO Xiu-wu, born in 1992, postgraduate. His main research interests include architecture-oriented performance analysis and optimization, compiler optimization, etc.

HUANG Liang-ming, born in 1988, Ph.D, assistant professor, is a member of China Computer Federation. His main research interests include architecture-oriented performance analysis and optimization, compiler optimization, etc.

（责任编辑：喻藜）