计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2024,33(2):62−71 [doi: 10.15888/j.cnki.csa.009393]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

面向 SW26010Pro 处理器的全局符号重定位优化①

钱　宏[1], 王　飞[2], 刘　沙[1], 郑天宇[3], 宋佳伟[4], 安　虹[1]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230026)
[2](清华大学 计算机科学与技术系, 北京 100084)
[3](之江实验室, 杭州 311121)
[4](国家超级计算无锡中心, 无锡 214000)
通信作者: 钱　宏, E-mail: qh1986@mail.ustc.edu.cn

摘　要: 申威异构众核处理器运算核心访问主存的延迟很大, 程序中应尽量避免运算核心代码访问主存的操作. 全局偏移表存放程序中全局变量和函数的地址, 不适合保存在珍稀的运算核心局部存储空间中, 并且其访问模式通常比较离散, 因而也不适合对其做 Cache 预取, 访问全局偏移表引入的访问主存操作对程序性能影响较大. 本文针对异构众核程序静态链接与动态链接的使用场景, 分析链接器 relaxation 优化的使用限制, 通过“gp 基地址+扩展偏移”的方法实现避免访问主存操作的全局符号重定位优化. 实验结果表明, 该重定位优化方法能够以增加少量代码为代价, 在运算核心代码调用函数与访问全局变量时有效避免访问全局偏移表引入的访问主存的操作, 提高众核程序的运行性能.

关键词: 众核处理器; 全局偏移表; 重定位; 链接器优化; 性能

引用格式: 钱宏,王飞,刘沙,郑天宇,宋佳伟,安虹.面向 SW26010Pro 处理器的全局符号重定位优化.计算机系统应用,2024,33(2):62–71. http://www.c-s-a.org.cn/1003-3254/9393.html

Optimized Global Symbol Relocations in SW26010Pro Processors

QIAN Hong[1], WANG Fei[2], LIU Sha[1], ZHENG Tian-Yu[3], SONG Jia-Wei[4], AN Hong[1]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)
[2](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)
[3](Zhejiang Lab, Hangzhou 311121, China)
[4](National Supercomputing Center in Wuxi, Wuxi 214000, China)

Abstract: The delay of the computing core access to the main memory of Shenwei heterogeneous many-core processors is very large, and thus the program should try to avoid the access of computing core code to main the memory as much as possible. The global offset table stores the addresses of global variables and functions in the program, which is not suitable to be saved in the rare local storage space of the computing core, and it is not suitable for cache prefetching because of its discrete access patterns. Therefore, accessing the main memory operation introduced by accessing the global offset table has a great influence on program performance. In view of the usage scenarios of static linking and dynamic linking of heterogeneous many-core programs, the usage limitations of linker relaxation optimization are analyzed, and a global symbol relocation optimization method is designed based on “gp address base+extended offset” to avoid accessing the main memory. Experimental results show that at the cost of adding a small amount of code, the relocation optimization method can effectively avoid the operation of accessing the main memory introduced by accessing the global offset table when the computing core code calls functions and accesses global variables, which improves the running performance of many-core programs.

Key words: many-core processor; global offset table (GOT); relocation; linker optimization; performance

① 基金项目: 国家重点研发计划 (2020YFB0204602)
收稿时间: 2023-07-16; 修改时间: 2023-08-21, 2023-09-15; 采用时间: 2023-09-26; csa 在线出版时间: 2023-11-24
CNKI 网络首发时间: 2023-11-27

© 中国科学院软件研究所 http://www.c-s-a.org.cn

近年来, 随着半导体工艺越来越接近摩尔定律的极限, 处理器主频的提升已经十分有限, 而多级流水线、核内多发射、向量宽度等提升单核心性能的技术手段也趋近发展极限, 因此单处理器核心的性能增长渐趋缓慢. 为持续提升处理器性能, 处理器结构经历了从单核到多核、众核的发展.

众核处理器因其在性能功耗上的优势, 得到了快速的发展. 现代超级计算机系统越来越多地采用了众核架构的处理器, 例如 NVIDIA 的 GPU 处理器系列[1]、AMD 的 GPU 架构[2]、Intel Xeon Phi 处理器[3]、PEZY-SC 处理器、申威众核处理器[4]、飞腾 matrix 众核处理器[5]等. 其中申威众核处理器包含 SW26010 和 SW26010-Pro 处理器, 均采用自主研发的异构众核架构.

基于 SW26010 芯片构建的“神威·太湖之光”超级计算机系统从 2016 年 6 月开始, 连续 4 次蝉联超级计算机 TOP500 排行榜冠军, 在此系统上完成的“千万核可扩展大气动力学全隐式模拟”和“非线性大地震模拟”两项应用获得高性能计算机应用领域的最高奖“戈登·贝尔奖”[6,7]. 基于 SW26010Pro 处理器构建了新一代神威超级计算机, 在此系统上完成的量子模拟器应用再次获得“戈登·贝尔奖”[8].

申威众核处理器由多个运算核组构成, 每个运算核组由一个运算控制核心 (MPE) 和一个 8×8 的运算核心阵列 (CPE Array) 组成. SW26010 处理器由 4 个运算核组构成, SW26010Pro 处理器继承和发展了 SW26010 处理器的系统架构, 由 6 个运算核组构成, 其总体结构如图 1 所示. 运算控制核心上运行操作系统和用户程序, 负责存储资源的管理, 提供消息、文件、调试、低功耗管理等服务, 也进行用户程序中难以并行化的计算; 运算核心阵列包含 64 个运算核心, 支持多种数据分布管理模式和多种运算核心间信息传递方式, 为处理器提供强大的计算能力.

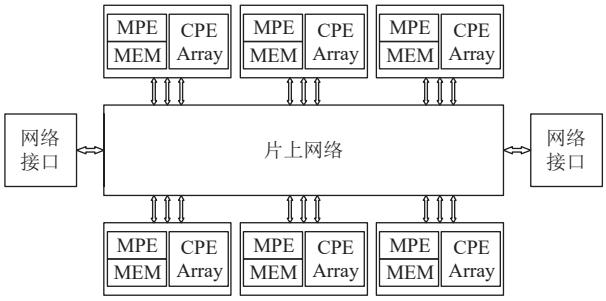

图 1 SW26010Pro 处理器总体结构

由于制造工艺、封装技术等一系列原因, 处理器速度的增长远远超过了存储器速度的增长, 导致越来越严重的“存储墙”问题[9]. 在 CPU 架构中, 通常引入多级 Cache 和预取技术来减少平均访存时间, 在一定程度上缓解“存储墙”问题. 由于芯片面积、功耗等因素的限制, 众核架构的运算核心往往采用精简的核心设计, 这导致“存储墙”问题对众核处理器性能的影响更为突出.

在 SW26010Pro 处理器上, 运算核心访问主存的延迟很大, 每个运算核心集成了一块支持高速访问的片上数据存储空间. 为高效发挥运算核心的并行加速性能, 运行在运算核心上的程序应尽量保证访问的数据保存在片上数据存储空间内, 尽可能避免产生访问主存的操作.

本文围绕 SW26010Pro 处理器运算核心访问主存延迟大的问题展开讨论, 重点研究程序中全局符号的寻址方式处理, 针对众核程序静态链接与动态链接的使用场景, 基于“gp 基地址+扩展偏移”的方式设计实现了一种全局符号重定位优化方法, 用于避免运算核心代码访问全局符号引入的访主存操作, 提升众核程序性能.

1 SW26010Pro 众核处理器存储层次介绍

SW26010Pro 众核处理器运算核组内的存储层次如图 2 所示. 运算控制核心的存储层次包含寄存器文件、大小各为 32 KB 的指令 Cache (ICache) 和数据 Cache (DCache)、大小为 512 KB 的指令和数据共用的二级 Cache、主存等. 运算核心的存储层次包含寄存器、大小为 32 KB 的 ICache、大小为 256 KB 的片上数据存储空间、主存等. 运算控制核心和运算核心访问主存空间的延迟是非常大的, 达到数百时钟周期. 运算控制核心由于一级和二级 Cache 的帮助能够有效缓解该问题, 而程序中如果运算核心代码频繁地访问主存将对程序性能产生极大的影响.

在 SW26010 处理器中, 运算核心的片上数据存储空间为软件管理的局部数据存储 LDM, 需要程序员在编程时进行显式的管理, 编程门槛较高. 在升级版的 SW26010Pro 处理器中, 片上数据存储空间可以全部配置为软件管理的 LDM, 也可以部分配置为硬件自动管理的 DCache, 降低编程门槛. SW26010Pro 处理器支持在程序运行过程中动态修改运算核心 LDM 和 DCache

© 中国科学院软件研究所 http://www.c-s-a.org.cn

的配置, 程序员可以根据课题代码的特征和实际应用需求进行选择. LDM 需要软件进行显式的管理, 程序员在进行算法设计阶段就需要将 LDM 空间的合理使用考虑进来, 例如采用双缓冲方式对数据传输和计算时间进行隐藏等, 编程要求高, 适用于对少量核心代码进行深度优化. 对于大量非核心代码, 则可以使用 DCache 模式, 依赖硬件的自动管理和编译器的预取优化, 较为方便地获得众核加速效果.

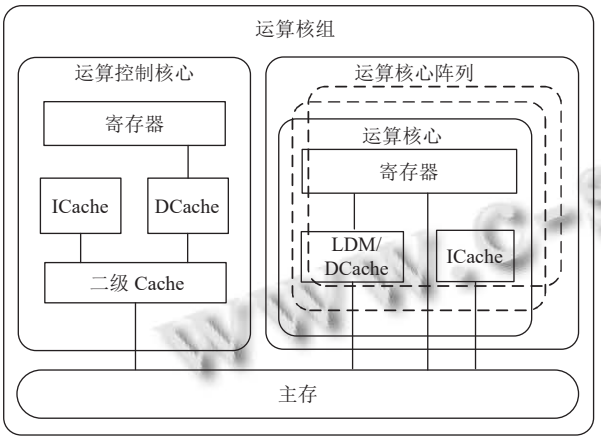

图 2 运算核组内的存储层次

SW26010Pro 处理器上, 应用程序中的用户数据可以使用__thread_local 关键字声明在 LDM 空间中, 对于非 LDM 空间中的用户数据, 可以配置一定数量的 DCache, 在循环中通过 Cache 预取指令预先取到 DCache 中, 以减小数据访问延迟. 程序中的全局变量和函数的地址是通过全局偏移表 (global offset table, GOT) 来维护的, 因 LDM 空间的珍稀性而不适合将 GOT 表保存在其中, 并且 GOT 表的访问模式通常比较离散, 因而也不适合对其做 Cache 预取. 当片上数据存储空间全部配置为 LDM 时, 访问 GOT 表不可避免地成为访主存操作; 当片上数据存储空间部分配置为 DCache 时, 初次访问 GOT 表条目为 DCache miss, 需要从主存中读取, 同时可以把该条目所在的 Cache 行取到 DCache 中, 后续对这些条目的访问可能命中 DCache, 也可能因为该 Cache 行被其他数据淘汰出 DCache 而成为访主存操作, 一定程度地影响程序性能.

2 全局符号寻址的编译链接处理

2.1 众核程序编译流程

SW26010Pro 处理器上构建了 SWGCC 众核编译系统, 支持面向异构众核处理器的加速编程模型, 提供上层并行软件和应用程序到众核可执行码的高效编译转换, 其具体的编译流程如图 3 所示.

编译负责把源代码处理成汇编语言, 是众核编译系统中最重要也是最复杂的一个步骤. SWGCC 众核编译系统复用了编译前端的词法分析、语法分析和中端的中间代码生成与通用优化, 将运算控制核心代码和运算核心代码处理成统一的中间表示. 根据运算控制核心和运算核心不同的后端机器描述进行针对性的后端优化并生成相应的汇编代码.

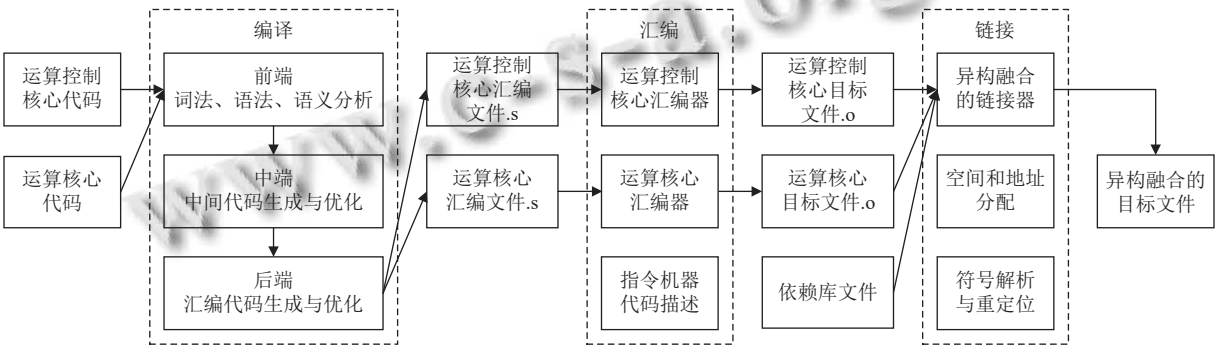

图 3 SWGCC 众核编译流程

汇编将编译输出的汇编代码转换成目标代码, 众核编译系统针对运算控制核心和运算核心汇编代码调用相应的汇编器, 根据汇编指令和指令机器代码描述一一对照翻译转化, 生成相应的目标代码.

链接则是将运算控制核心和运算核心目标代码与依赖的库文件融合链接并输出为众核目标文件的过程. 由于代码之间会有大量的调用和引用, 链接器依据符号调用关系将不同目标文件进行拼接, 并重定位程序中的数据位置, 使各个代码模块之间能够正确地衔接.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

2.2 异构融合链接

链接处理的目标文件通常有以下 3 种不同的形式: 可重定位目标文件、可执行目标文件和共享目标文件, 本文中讨论的目标文件是基于 Unix 系统的 ELF 格式的. 可重定位目标文件包含.o 文件和静态链接库.a 文件, 可以被链接成可执行文件或共享目标文件; 共享目标文件包含动态链接库.so 文件.

链接的过程主要包括空间和地址的分配、符号解析和重定位等, 最后生成可以在目标机器上运行的可执行文件或动态链接库文件. 空间和地址分配阶段, 链接器扫描所有输入目标文件, 获取它们各个段的长度、属性和位置信息, 并将其符号表中所有的符号定义和引用收集合并到全局偏移表 (GOT) 中. 链接器将输入目标文件中的各个段进行合并, 并建立映射关系. 符号解析与重定位阶段, 链接器读取输入目标文件中的数据和重定位信息, 进行符号解析与重定位, 并调整代码中的地址.

SWGCC 异构融合链接的机制如图 4 所示, 链接器将运算控制核心与运算核心目标文件中的各个段进行合并, 如将所有运算核心目标文件的代码段合并到.text1 段, 所有运算控制核心目标文件的代码段合并到.text 段, 将所有输入目标文件的全局变量合并到.data 段. 运算控制核心目标文件中的.tdata 段对应运算控制核心代码中__thread 关键字修饰的线程静态私有变量; .tdata_private 段对应运算核心代码中__thread 关键字修饰的线程静态私有变量; 运算核心目标文件中的.tdata_local 段对应运算核心代码中__thread_local 关键字修饰的静态 LDM 变量, 运算核心 LDM 空间为从零地址开始的编址空间, 在众核程序加载时将.tdata_local 段拷贝至 LDM 地址空间.

2.3 全局符号重定位处理

程序中的全局变量和函数是由 GOT 表来维护的, GOT 表里面的每个条目对应某个全局变量或函数的地址, 在编译系统执行链接操作时链接器会对这些符号信息逐个解析来完成重定位工作. 应用程序在运行过程中, 通过访问 GOT 表中对应的条目来获取全局符号的地址, 以完成对全局变量或函数地址的读取.

程序对 GOT 表条目的访问是通过 literal 重定位方式实现的, 如下给出了运算核心代码访问全局变量和函数调用的一个示例, 在运算核心函数 slave_test 中访问全局变量 a, 并调用外部运算核心函数 slave_fun.

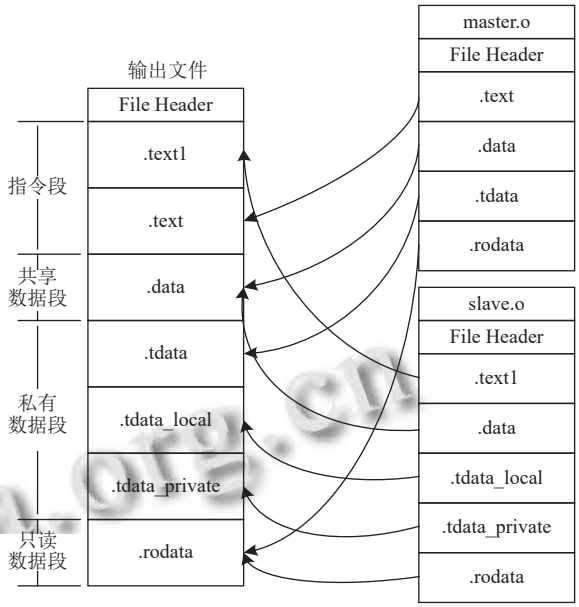

图 4 SWGCC 异构融合链接

```
extern long a;
extern void slave_fun();
int slave_test()
{
    a=0;
    slave_fun();
}
```

编译系统处理示例程序对应生成的汇编代码如图 5 所示, 程序访问全局变量 a 时, 需要首先从 GOT 表中获取 a 的地址, 再对 a 地址指向的存储空间进行读写操作, 其中$31 为恒零寄存器; 调用 slave_fun 函数时, 需要首先从 GOT 表中获取 slave_fun 函数的地址, 然后使用 jmp 指令跳转过去.

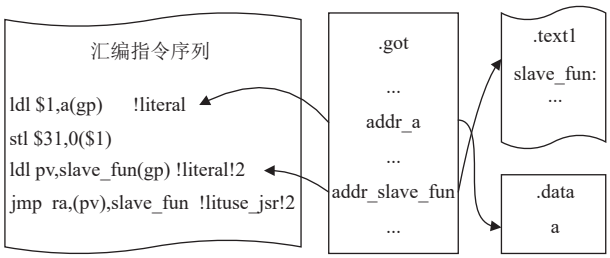

图 5 全局符号访问 GOT 表的汇编指令序列

GOT 表不适合存放在 LDM 空间中, 因为 LDM 空间容量有限, 是 SW26010Pro 众核处理器的珍稀资源, 在一些应用程序中 GOT 表里全局符号条目的数量是极为可观的, GOT 表静态占用 LDM 空间将使得众核程序无法最大化地使用 LDM 空间, 不利于发挥众核处

© 中国科学院软件研究所 http://www.c-s-a.org.cn

理器的加速效果. 因此, GOT 表存放在主存中, 运算核心代码访问 GOT 表条目会引入访问主存的操作, 应用程序中运算核心代码频繁地访问主存会极大地影响程序性能.

3 链接时优化技术

3.1 编译器链接时优化

编译器过程间分析与优化 (inter-procedural optimization, IPO) 是提升应用程序运行性能的有效编译优化手段[10]. 传统编译器通常每次针对一个处理单元进行编译分析与优化, 处理单元为一个源代码文件或模块, 编译优化相应地局限于单个文件或模块内. IPO 优化则聚焦于对整个程序代码进行分析与优化.

编译器链接时优化 (link-time optimization, LTO) 是属于 IPO 中的一种优化, 编译器将多个输入文件的中间表示整合到一起进行分析处理, 能够看到更多的代码, 统一分析获取到更多的程序信息[11,12], 因而可以发掘更多的优化机会. GCC 编译器从 4.5.0 版本开始引入 LTO 优化, 执行函数内联[13,14]、常量传播和无用代码删除等优化, 实现函数分裂以支持函数部分内联. Glek 等在 GCC 编译器里实现了函数重排序优化[11], 在 LTO 优化时可以根据函数调用顺序来指定目标码中的函数顺序, 增加代码局部性, 减少程序运行时使用的内存页表数. LLVM 编译器基于中间表示 IR 和 LTO 实现了 outlining[15]、函数合并 (function merging)[16]等优化, 能够有效减小最终生成的目标代码的大小.

编译器链接时优化进行的函数内联优化可以减少一部分函数调用, 常量传播优化可以减少对一部分全局变量的访问, SWGCC 编译器实现了这些常规的链接时优化, 但是这些优化适用的场景有限, 如内联需要考虑函数代码的大小, 无法针对已经编译好的库函数进行优化等, 不能从根本上解决运算核心获取全局符号地址访问主存的问题.

3.2 链接器优化

传统编译器在编译阶段无法获得指令和数据的虚拟地址信息, 而在静态链接阶段这些信息都已经确定, 利用这个时机链接器可以实现编译时因为信息不足而无法实现的优化. 链接器实现了 relaxation 优化[17,18], 可以在一定范围内将 literal 方式获取全局变量地址的 load 指令优化成 ldi 计算指令, 对应的限制条件是全局变量的地址到 gp (global point) 的偏移在 ldi 指令的偏移范围内, ldi 指令的偏移为 16 位有符号整数, 那么其偏移范围为±32 KB; 链接器也能够在一定跳转范围内将寄存器间接转移指令优化成 PC 偏移的无条件转移指令.

SW26010Pro 处理器运算核心支持 jmp 和 br 两条无条件转移指令, 其指令格式如下所示. 其中, jmp 指令是寄存器间接转移指令, 跳转的目的地址由寄存器 pv 提供, ra 寄存器保存本指令下一条指令的地址, 即为返回地址; br 指令为 PC 偏移的无条件转移指令, 跳转的目的地址为“本指令 PC 值+disp 偏移”, ra 寄存器同样保存返回地址, disp 偏移量为 23 位有符号整数, 可在前后各 4M 条指令的范围内实现跳转.

```
jmp ra, (pv)
br ra, disp
```

当跳转的目的地址满足与跳转指令所在地址的偏移量在 23 位有符号整数范围内, 并且跳转的目的函数不需要使用 pv 寄存器计算 gp 时, SWGCC 链接器可以将从 GOT 表获取目的函数地址的 load 指令替换成 nop 空指令, 将 jmp 指令替换成 br 指令, 并跳过计算 gp 的两条指令, 从而避免函数调用时需要从 GOT 表获取目的函数地址的访主存操作, 如图 6 所示.

链接器 relaxation 优化函数调用的限制主要在于:

(1) 链接器要能够准确地知道目的函数的地址, 并根据跳转指令所处的地址计算跳转指令的具体偏移, 以此来判断是否可以使用 br 指令替换 jmp 指令, 这通常要求程序使用静态链接, 而动态链接一般不满足这个条件.

(2) 要求主调 caller 函数与被调 callee 函数使用相同的 gp 值. 使用 load 指令将从 GOT 段获取的目的函数地址存入 pv 寄存器, 不仅用于 jmp 指令的跳转目的地址, 而且跳转到 callee 函数后需要使用 pv 寄存器重新计算 gp 值. 当 caller 函数与 callee 函数的 gp 值不相同时, 则不能把 load 指令替换成 br 指令.

gp 是专门用于访问 GOT 段的基址寄存器. 编译器处理每一个编译单元时都会在目标文件中生成一个 GOT 段, 在链接时将多个目标文件的 GOT 段进行合并处理, 但并不总是能把所有目标文件的 GOT 段都合并成一个 GOT 段的, 要求 GOT 段里的每个条目都能通过 gp 寻址方式访问到, 因此 GOT 段的大小受限于 literal 寻址方式中 load 指令的偏移范围. SW26010Pro 处理器运算核心中 64 位装入指令的偏移是 16 位有符

© 中国科学院软件研究所 http://www.c-s-a.org.cn

号整数, 即可以寻址的范围为±32 KB, 那么每个 GOT 段的大小限制在 64 KB 以内.

当多个目标文件的 GOT 段大小的和不超过 64 KB 时, 链接器将这些目标文件的 GOT 段合并成一个 GOT 段, 使用同一个 gp 值; 当合并进某个目标文件的 GOT 段导致大小超过 64 KB 时, 则不能将该目标文件的 GOT 段合并进来. 链接生成的文件中可能有多个 GOT 段, 分别使用不同的 gp 值. 所以 caller 函数与 callee 函数有可能使用不同的 gp 值, 为保证程序正确运行, 在每个函数头上需要使用 pv 寄存器计算 gp 值, 在完成函数运行返回上层 caller 函数时也需要使用 ra 寄存器重新计算 gp 值.

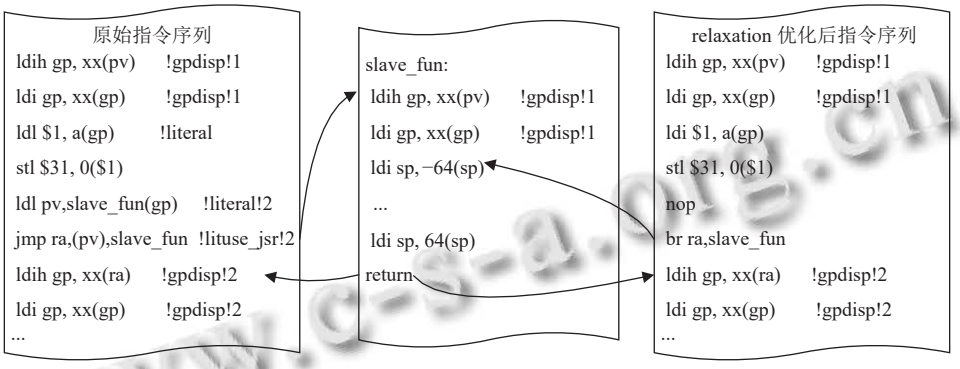

图 6 SWGCC 链接器 relaxation 优化

4 SWGCC 全局符号重定位优化

4.1 SWGCC 静态链接与动态链接

SWGCC 支持静态链接与动态链接. 静态链接对应图 4 中混合链接的输出目标文件是众核静态可执行文件, 静态链接时所有全局符号的重定位操作在链接阶段完成, 在运行时不再有重定位的开销, 是 SW26010Pro 处理器上高性能计算应用主要使用的链接方式. 静态链接时所有全局符号的虚拟地址都已经确定, 链接器能够进行 relaxation 优化, 在满足优化条件时将运算核心代码中 literal 寻址方式的访存指令进行优化替换.

随着大数据、人工智能等领域的迅速发展, 出现了高性能计算+大数据、高性能计算+人工智能等应用软件的新模式, 对超级计算机上编译系统的支持提出了新的要求, 如动态链接库的运行时加载、即时编译等需求, 而这些都是以动态链接为基础的. 为更好地服务应用, SWGCC 编译系统实现了对众核混合程序的动态链接支持, 为 Python、PyTorch[19]、LLVM 即时编译 (just-in-time compilation, JIT)、Julia[20]等软件的开发提供基础功能支撑.

与静态链接过程不同, 动态链接的链接工作通常是由动态链接器 (ld-linux.so.2) 完成的. 当动态链接的程序被装载的时候, 动态链接器将程序所依赖的所有动态链接库装载到进程的内存空间, 并进行符号决议和重定位工作.

动态链接器对全局符号的解析不是静态固定的, 而是取决于程序运行时动态链接器装载的顺序, 受链接时动态链接库的顺序和运行时的环境配置影响. 有可能在多个动态链接库里定义了相同的全局变量或函数, 动态链接器进行符号查找的顺序与动态链接库装载的顺序一致. 因此动态链接过程通常无法在链接器内对访问全局符号引入的访主存操作进行优化.

SWGCC 动态链接对应图 4 中混合链接的输出目标文件是众核动态链接库.so 文件或众核动态可执行文件. 在 SW26010Pro 处理器中, 运算核心 jmp 和 br 无条件转移指令使用的 PC 地址是 24 位短地址, 要求 jmp 和 br 指令所在的地址与跳转目的地址的 24 位以上的高位部分是一致的, 即这两个地址要处在同一个 16 MB 对齐的地址空间内, 否则运行会报 PC 溢出异常. 对于静态链接的应用程序, 链接器将所有目标文件的运算核心代码合并到可执行文件的.text1 段, 整个.text1 段的大小通常不会超过 16 MB, 没有发生过 PC 溢出异常. 对于动态链接, 动态链接器将每个动态链接库作为一个整体进行装载, 并不会对多个动态链接库内的相同段进行合并, 那么多个动态链接库的.text1 段极有可能被装载到不同的 16 MB 地址空间内. 当从某个动态链接库内的运算核心函数跳转到另一个动态链接库内的运算核心函数时, 就有可能发生 PC 溢出异常. 因此, 在 SW26010Pro 处理器的异构动态编程中, 我们要求

© 中国科学院软件研究所 http://www.c-s-a.org.cn

运算核心代码调用的函数都包含在同一个动态链接库内而不依赖于其他动态链接库, 以避免发生 PC 溢出异常.

4.2 全局符号重定位优化

在 SW26010Pro 处理器的动态链接与运行的场景中, 对于运算核心代码部分就相当于使用静态链接, 运算核心代码所有的函数调用都包含在同一个动态链接库中, 链接时所有运算核心函数的相对地址都是确定的, 因此可以针对运算核心函数调用进行 relaxation 优化, 但是同样受到 caller 函数与 callee 函数使用相同 gp 值的限制.

针对这一限制, 本文在 literal 重定位方式的基础上, 设计实现基于“gp+扩展偏移”的全局符号重定位优化. 当 caller 函数与 callee 函数使用不同的 gp 值时, 需要获取 callee 函数的地址以重新计算 gp 值, 本文在 gp 值的基础上通过“ldih+ldi”指令对计算得到 callee 函数的地址, 避免从 GOT 表 load 获取 callee 函数地址的访主存操作, 优化 literal 重定位方式的实现.

SWGCC 全局符号重定位优化如图 7 所示, 左边部分为编译器处理第 3.2 节示例代码生成的汇编代码, 编译器在 literal 重定位方式的 load 指令前对应生成一条 literalhi 重定位方式的 ldih 指令, 图右边部分为优化后的指令序列, 链接器在处理重定位阶段计算全局符号地址与 gp 的偏移, 判断该偏移值是否能够使用“ldih+ldi”指令对表示, 能够表示时则将 load 指令优化成 ldi 指令, 不能够表示时则不做替换, 具体算法如算法 1 所示.

算法 1. 全局符号重定位优化算法

1) 编译器处理运算核心代码访问全局符号时生成 literalhi 重定位指令, 具体地, 处理访问全局变量生成“ldih+ldl”的“literalhi+literal”重定位指令对, 处理函数调用生成“ldih+ldl+jmp”的“literalhi+literal+lituse_jsr”重定位指令对;
2) 链接器内新增 literalhi 重定位方式, 在进行异构众核链接时识别并处理 literalhi 重定位指令:
a) 对于识别为 literalhi 重定位方式的指令, 获取该全局符号的虚拟地址与当前的 gp 值, 两者相减得到具体的偏移值;
b) 判断步骤 a) 中计算得到的偏移值是否能够用“ldih+ldi”指令对表示, 即偏移值右移 16 位后的绝对值是否小于 32 768;
c) 如果步骤 b) 中的判断条件成立, 则可以将“ldih+ldl”指令对优化成“ldih+ldi”指令对, 将偏移值的高 16 位和低 16 位分别计算填入 ldih 和 ldi 指令的偏移位中;
d) 如果步骤 b) 中的判断条件不成立, 则保持“ldih+ldl”指令对不变, 计算全局符号在 GOT 表中的位置与 gp 值的相对偏移, 将该相对偏移的高 16 位和低 16 位分别计算填入 ldih 和 ldl 指令的偏移位中;
3) 链接器进一步进行针对函数调用的 relaxation 优化, 判断函数调用是否满足优化条件, 如满足优化条件则将“ldih+ldi+jmp”指令对优化成“nop+nop+br”指令对.

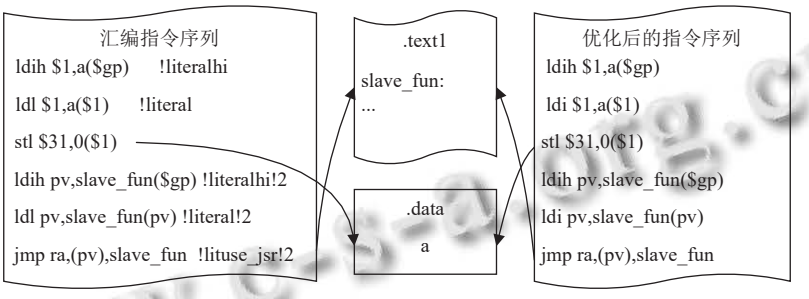

图 7 SWGCC 全局符号重定位优化示例

相比于 literal 重定位方式的 16 位有符号整数的偏移, “ldih+ldi”指令对能够表示 32 位有符号整数的偏移, 即可以在 gp 值的基础上寻址±2 GB 的地址空间. 在应用程序中通常使用 malloc、free 函数动态申请和释放内存空间, 可执行程序或一个动态链接库里的代码段与静态数据段的总大小通常都处于以上寻址空间内. 因此, 在 SW26010Pro 处理器上的静态链接和动态链接的使用场景中, 访问 GOT 表获取全局符号地址产生的访主存操作通常都可以通过“literalhi+literal”重定位优化避免, 相应的开销是每条访问增加一条 ldih 指令. 代码的总体增加量取决于程序中运算核心代码访问全局符号的代码数量, 运算核心主要用于加速并行计算, 其代码中访问全局符号的占比通常很小.

5 实验与分析

我们的工作是面向 SW26010Pro 处理器的性能优化, 所以使用 SW26010Pro 处理器作为硬件测试平台. 我们选择移植于新一代神威超级计算机平台上的“超大规模量子随机电路实时模拟 (SWQSIM)”[8]软件与深度学习框架 PyTorch[19]进行实验分析.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

5.1 SWQSIM 测试

SWQSIM 是 2021 年全球超级计算大会 (SC21) 上获得“戈登贝尔奖”的应用软件, 基于新一代神威超级计算机平台实现, 使用 4190 万处理器核心实现了 1.2 Eflops 单精度或 4.4 Eflops 混合精度的计算性能. 该应用具有优秀的并行可扩展性, 本文使用 16 个 SW26010Pro 处理器模拟 53 bit 20 cycle 谷歌悬铃木电路单个振幅 720 个子任务, 比较不同优化下的性能.

SWQSIM 软件使用静态链接, 链接时打开 relaxation 优化, 可执行文件中 GOT 段的大小约为 26 KB, 在整个程序中使用同一个 gp 值, 因此不使用全局符号重定位优化时, 链接器 relaxation 优化可以将可执行文件内所有运算核心函数调用的“ldl+jmp”指令对优化替换成“nop+br”指令对. 作为对比, 我们对是否使用 literalhi 全局符号重定位优化、是否使用 relaxation 优化两两组合的 4 种情形进行分析.

对于运算核心代码段.text1 段的大小, 不使用 literalhi 全局符号重定位优化时, relaxation 优化不改变.text1 段的大小, 大小为 412.4 KB; 使用 literalhi 全局符号重定位优化时, 大小为 414.1 KB, literalhi 全局符号重定位优化每次增加一条 ldih 指令, SWQSIM 程序总的运算核心代码段大小增加约 0.41%.

我们在 SW26010Pro 处理器上测试 4 种组合情形下 SWQSIM 程序的运算时间, 测试结果如图 8 所示, 测试结果为测试 10 次的运算时间取平均值. 其中, org 表示使用原始的 literal 重定位方式, opt 表示使用优化的重定位方式, relax 表示链接器打开 relaxation 优化, norelax 表示关闭 relaxation 优化, 4 种情形的运算时间以“org+norelax”的运算时间为基准进行了归一化.

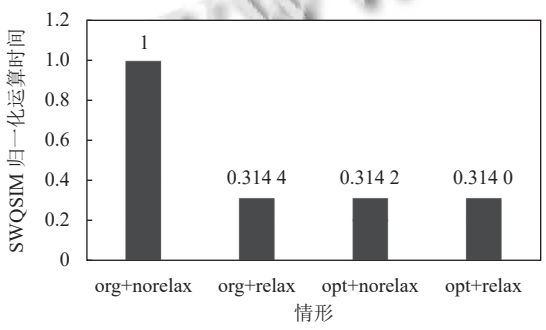

图 8 SWQSIM 不同情形下的归一化运算时间

SWQSIM 程序将运算核心局部数据存储空间全部配置成软件控制的 LDM 空间, 所以程序中运算核心访问 GOT 表的操作都是访主存操作. 从图 8 的测试结果可以看到, 使用原始的 literal 重定位方式且关闭链接器 relaxation 优化时, 程序中全局符号访问引入的运算核心访问主存操作对程序性能影响很大, 打开链接器 relaxation 优化或者使用优化的重定位方式能够显著提升程序性能.

我们进一步使用 SW26010Pro 处理器软件环境提供的性能监测 API 接口, 分别统计 4 种情形下 SWQSIM 程序运行的运算核心总指令数和运算核心访问主存 GLD 次数, 统计结果如表 1 所示. 可以看到, 使用原始的 literal 重定位方式且关闭链接器 relaxation 优化时, 运算核心访问主存的次数约占运行指令数的 0.052 6%, 虽然访问主存次数在运行总指令数中的占比很小, 但是仍然对程序性能产生了巨大的影响. 使用链接器 relaxation 优化能够避免运算核心代码中函数调用引入访主存操作. 本文提出的“literalhi+literal”优化的重定位方式, 不仅能够避免运算核心代码中函数调用引入的访主存操作, 而且能够避免访问全局变量引入的访主存操作, 所以统计数据中使用优化的重定位方式测得的访问主存次数最少.

表 1 SWQSIM 不同情形下运算核心访问主存次数统计

情形	运行指令数 ($\times10^{15}$)	访问主存次数 ($\times10^{9}$)	访问主存次数/运行指令数
org+norelax	3.634	2355.93	5.261E–4
org+relax	4.478	5.09	1.137E–6
opt+norelax	4.469	5.04	1.129E–6
opt+relax	4.461	5.04	1.131E–6

结合图 8 和表 1 的测试结果, 使用原始的 literal 重定位方式且关闭链接器 relaxation 优化时, 运算时间最长, 运行的总指令数却最少, 分析原因是该程序核心段部分为 DMA (direct memory access, 运算核心 LDM 与主存之间的直接内存访问) 带宽受限型, DMA 传输为异步操作, 运算核心代码在发出 DMA 请求后需循环读取回答字以等待数据传输完成. 当运算核心代码中访主存操作很少时, 运算核心阵列中 64 个运算核心程序运行的齐步性较好, 都会同步地发起 DMA 请求而抢占 DMA 带宽, 导致数据传输的时间较长, 约占程序运算时间的 52.0%; 而当运算核心代码中存在较多访主存操作时, 各个运算核心访问主存的延迟不一致, 导致程序运行的齐步性被打乱, DMA 操作相应错开了, 带宽竞争没有那么激烈, 使得数据传输的时间变短,

© 中国科学院软件研究所 http://www.c-s-a.org.cn

约占程序运算时间的 10.3%, 相应的循环访问回答字的指令数就大幅减少了.

5.2 PyTorch 测试

本文采用移植于新一代神威超级计算机平台的深度学习框架 PyTorch 测试动态链接下重定位优化的性能效果, 动态链接时链接器不进行 relaxation 优化. 我们在 PyTorch 框架中分别对深度残差网络模型 ResNet-50[21]、编码式模型 BERT[22]、生成式模型 GPT-2[23] 等模型使用与不使用重定位优化时的性能进行了测试, 使用的计算资源为单 SW26010Pro 处理器, 运算核心局部数据存储空间全部配置成软件控制的 LDM 空间. ResNet50 模型测试处理一张 224×224×3 的图片 img; BERT 测试采用 12 层卷积网络, hidden-size 为 512; GPT-2 测试采用 12 层卷积网络, hidden-size 为 1 024.

我们分析比较了优化前后 PyTorch 核心算子库 libswmath.so 中运算核心代码.text1 段的大小变化, 不使用 literalhi 全局符号重定位优化时, .text1 段的大小约为 2.949 MB; 使用 literalhi 全局符号重定位优化时, 大小约为 2.972 MB, 算子库总的运算核心代码段大小增加约 0.75%.

PyTorch 测试运算时间如图 9 所示, 其中 org 表示使用原始的 literal 重定位方式, opt 表示使用优化的重定位方式, 运算时间以 org 为基准进行了归一化. 可以看到, 优化的重定位方式使 ResNet50 测试的性能提升约 50%, 效果显著; 使 BERT 和 GPT-2 测试的性能小幅提升, 分别提升 2.2% 和 5.8%.

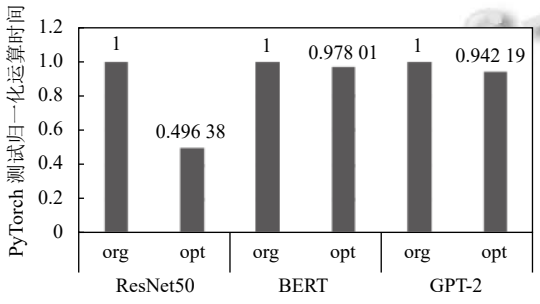

图 9 PyTorch 测试在不同情形下的归一化运算时间

我们进一步使用 SW26010Pro 处理器软件环境提供的性能监测 API 接口, 分别统计使用与不使用重定位优化情形下, PyTorch 测试运行的运算核心总指令数和运算核心访问主存 GLD 次数, 统计结果如表 2 所示. 统计结果显示, ResNet50 因存在较多的函数调用, 在不使用重定位优化时有较多的访问主存次数, 使用重定位优化后访问主存次数显著减少, 因此性能提升显著; BERT 与 GPT-2 测试中函数调用与访问全局变量在所有指令中占比较少, 使用重定位优化后访问主存次数减少量不多, 因此性能提升幅度较小.

表 2 PyTorch 测试不同情形下运算核心访问主存次数统计

情形	运行指令数 ($\times10^{11}$)	访问主存次数 ($\times10^{8}$)	访问主存次数/运行指令数
ResNet50 org	22.02	52.28	2.374E–3
ResNet50 opt	22.19	1.52	6.86E–5
BERT org	4.156	1.34	3.22E–4
BERT opt	4.154	1.25	3.02E–4
GPT-2 org	1.372	1.25	9.13E–4
GPT-2 opt	1.413	1.22	8.63E–4

6 结论与展望

本文针对申威异构众核处理器运算核心代码访问全局符号引入访问主存操作影响程序性能的问题, 分析链接器 relaxation 优化的使用限制, 设计实现了基于“gp 基地址+扩展偏移”的全局符号重定位优化方法. 实验结果证明, 在 SW26010Pro 处理器静态链接与动态链接的使用场景中, 全局符号重定位优化方法能够以增加少量代码为代价, 有效避免运算核心代码访问全局偏移表引入的访问主存的操作, 提高众核程序的运行性能. 目前的优化是在链接器内实现的, 适用于单一异构融合动态链接库的使用场景, 对于下一代申威异构众核芯片运算核心使用完整 PC 地址后, 为支持多异构融合动态链接库使用场景下的优化需求, 需研究在动态链接器中实现全局符号的重定位优化.

参考文献

1 Lindholm E, Nickolls J, Oberman S, *et al*. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 2008, 28(2): 39–55. [doi: 10.1109/MM.2008.31]

2 Macri J. AMD's next generation GPU and high bandwidth memory architecture: FURY. Proceedings of the 2015 IEEE Hot Chips 27 Symposium. Cupertino: IEEE, 2015. 1–26. [doi: 10.1109/HOTCHIPS.2015.7477461]

3 Jeffers J. Intel® Xeon Phi™ Coprocessors. Shi X, Kindratenko V, Yang CW. Modern Accelerator Technologies for Geographic Information Science. New York: Springer, 2013. 25–39. [doi: 10.1007/978-1-4614-8745-6_3]

4 胡向东, 柯希明, 尹飞, 等. 高性能众核处理器申威 26010.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

计算机研究与发展, 2021, 58(6): 1155–1165. [doi: 10.7544/issn1000-1239.2021.20201041]

5 Liao XK, Xiao LQ, Yang CQ, *et al*. MilkyWay-2 supercomputer: System and application. Frontiers of Computer Science, 2014, 8(3): 345–356. [doi: 10.1007/s11704-014-3501-3]

6 Yang C, Xue W, Fu HH, *et al*. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis. Salt Lake City: IEEE, 2016. 57–68.

7 Fu HH, He CH, Chen BW, *et al*. 18.9-Pflops nonlinear earthquake simulation on Sunway TaihuLight: Enabling depiction of 18-Hz and 8-meter scenarios. Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis. Denver: IEEE, 2017. 1–12.

8 Liu Y, Liu X, Li F, *et al*. Closing the "quantum supremacy" gap: Achieving real-time simulation of a random quantum circuit using a new sunway supercomputer. Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage and Analysis. St. Louis: IEEE, 2021. 1–12.

9 Wulf WA, McKee SA. Hitting the memory wall: Implications of the obvious. ACM Sigarch Computer Architecture News, 1995, 23(1): 20–24. [doi: 10.1145/216585.216588]

10 Cooper KD, Kennedy K, Torczon L. Interprocedural optimization: Eliminating unnecessary recompilation. Proceedings of the 1986 SIGPLAN symposium on Compiler construction. Palo Alto: ACM, 1986. 58–67. [doi: 10.1145/12276.13317]

11 Glek T, Hubicka J. Optimizing real world applications with GCC link time optimization. arXiv:1010.2196, 2010.

12 Johnson T, Amini M, Li XD. ThinLTO: Scalable and incremental LTO. Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization. Austin: ACM, 2017. 111–121. [doi: 10.1109/CGO.2017.7863733]

13 Cooper KD, Hall MW, Torczon L. An experiment with inline substitution. Software: Practice and Experience, 1991, 21(6): 581–601. [doi: 10.1002/spe.4380210604]

14 Davidson JW, Holler AM. A study of a C function inliner. Software: Practice and Experience, 1988, 18(8): 775–790. [doi: 10.1002/spe.4380180805]

15 Jessica Paquette. Reducing code size using outlining. https://www.llvm.org/devmtg/2016-11/Slides/Paquette-Outliner.pdf. (2019-10-30)[2023-06-16].

16 Rocha RCO, Petoumenos P, Wang Z, *et al*. Function merging by sequence alignment. Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. Washington: IEEE, 2019. 149–163.

17 Srivastava A, Wall DW. Link-time optimization of address calculation on a 64-bit architecture. ACM Sigplan Notices, 1994, 29(6): 49–60. [doi: 10.1145/773473.178248]

18 MaskRay. The dark side of RISC-V linker relaxation. https://maskray.me/blog/2021-03-14-the-dark-side-of-riscv-linker-relaxation. (2021-03-14).

19 高捷, 刘沙, 黄则强, 等. 基于国产众核处理器的深度神经网络算子加速库优化. 计算机科学, 2022, 49(5): 355–362. [doi: 10.11896/jsjkx.210500226]

20 Shang HH, Shen L, Fan Y, *et al*. Large-scale simulation of quantum computational chemistry on a new sunway supercomputer. Proceedings of the 2022 International Conference for High Performance Computing, Networking, Storage and Analysis. Dallas: IEEE, 2022. 1–14.

21 He KM, Zhang XY, Ren SQ, *et al*. Deep residual learning for image recognition. Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition. Las Vegas: IEEE, 2016. 770–778.

22 Devlin J, Chang MW, Lee K, *et al*. BERT: Pre-training of deep bidirectional transformers for language understanding. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Minneapolis: Association for Computational Linguistics, 2019. 4171–4186.

23 Radford A, Wu J, Child R, *et al*. Language models are unsupervised multitask learners. OpenAI, 2019, 1(8): 9.

(校对责编: 孙君艳)

© 中国科学院软件研究所 http://www.c-s-a.org.cn