doi:10.11920/xnmdzk.2020.01.006

基于 X86 平台的编译器性能优化

贺春林,赖庆宽,朱广林,何先波
(西华师范大学计算机学院,四川 南充 637009)

摘 要:编译器的性能受机器平台的影响,只有编译器与机器平台相适应配套,才能发挥出极致性能.因此,编译器优化工作显得尤为重要.在传统的编译器优化工作中,常采用的是迭代测试分析方法,但编译器优化 Pass 繁多,使用此方法致使测试任务量大、分析数据量多、优化工作难.因此,提出一种正确性粗细粒度差异化分析技术,从正确性性能摸索、粗粒度迭代测试以及细粒度核心差异化分析三方面解决编译器优化工作中的困难.最后,通过 SPEC CPU2017 实验测试,验证了该方法的实用性,为 GCC 优化提供了正确的优化方向.

关键词:编译器性能;编译器优化;迭代测试;优化 Pass;优化方向
中图分类号:TP314 **文献标志码**:A **文章编号**:2095-4271(2020)01-0033-08

Compiler performance optimization based on X86 platform

HE Chun - lin, LAI Qing - kuan, ZHU Guang - lin, HE Xian - bo
(School of Mathematics and Information, China West Normal University, Nanchong 637009, P. R. C.)

Abstract: The performance of the compiler is affected by the machine platform, and only the compiler and the machine platform can be matched to achieve the ultimate performance. Therefore, compiler optimization is especially important. In the traditional compiler optimization work, the iterative test analysis method is often used, but the compiler optimizes the Pass. The use of this method results in a large amount of test tasks, a large amount of analysis data, and difficulty in optimization. Therefore, this paper proposes a correctness and coarse - grained particle size differentiation analysis technique, which solves the difficulties in compiler optimization work from three aspects: correctness performance exploration, coarse - grained iterative test and fine - grained core differentiation analysis. Finally, the practicality of the method is verified by SPEC CPU2017 experimental test, which provides the correct optimization direction for GCC optimization.

Key words: compiler performance; compiler optimization; iterative test; optimize Pass; optimize direction

随着半导体技术的快速发展及大数据应用中数据的急速增长,在高性能计算领域中,并行计算机架构越发复杂.现今流行的高性能计算系统架构基本使用了多核处理器 MIC(Many Integrated Core Architecture),并增加 GPU(Graphics Processing Unit)或者 AI 处理器作为加速器或协处理器,以适应大量、复杂地并行计算[1-2].编译器的性能尤为重要,编译器与处理器架构和应用程序之间的关系相辅相成,是应用程序正确运行的根本保障,是处理器架构发挥极致性能的重要体现.编译器的性能受机器平台架构和编译器自身软件的影响,如机器平台架构的内存大小、缓存等级与大小是影响预取、内存对齐等局部性优化的因素,编译器的数据流分析是影响常量传播和无效代码删除的因素[3-5].不同的机器平台架构,选择不同优

收稿日期:2019-11-15
作者简介:贺春林(1971-),男,四川广安人,教授,研究方向:数据挖掘与图像处理. E - mail:chunlin_he@163.com
通信作者:赖庆宽(1994-),男,四川凉山州人,研究方向:编译器优化. E - mail: laiqingkuan@ict.ac.cn
基金项目:国家自然科学基金(61871330);西华师大英才基金(17YC150)

化能力的编译器,方可发挥出极致的性能. 因此,编译器的优化是一件非常重要的事情,编译器优化研究也是一个热点的研究方向.

编译器优化选项多样且复杂,就 GCC 编译器就有上百种编译优化选项[6]. 探究编译器性能时,测试任务量繁多且沉重,为每个测试用例在不同架构上和不同编译器间找出最优的优化选项搭配不是优化编译器的任务,因为一般用户也只关注常用的优化选项,如 O3、O2 等,但为之找出影响性能的关键优化选项是该工作的重中之重,也是评估编译器性能及优化编译器的必备前提. 迭代编译优化是一种暴力穷举的方法[7],该方法虽对所有的程序,不同的架构,不同的编译器都有很强的普适性,但它是对不同参数的自由组合,测试任务重且测试技术含量低,该技术逐渐被边缘化或者被研究者做不同方向的扩展. 并且更多的迭代编译优化是找出针对该测试用例的最优优化选项搭配,但编译器优化工作是在已有的常用编译选项中找出该测试用例的影响性能的关键优化选项.

本文提出一种编译器优化分析技术—正确性粗细粒度差异化分析技术,可快速准确定位出测试用例的关键优化选项. 首先,以 SPEC CPU2017 标准测试软件根据常用优化选项对不同编译器做正确且全面的摸底测试;其次,根据摸底测试结果选择特定测试用例,采用迭代优化分析技术,但简化了优化 Pass 的组合,只对常用优化 Pass 做粗粒度迭代测试. 因为在程序编译过程中只会选择一些常用优化选项,如 O3、过程间优化(IPO)、预取优化(prefetch)、向量化优化(vectorize)、链接时优化(LTO)等针对具体测试用例,以迭代编译技术对常用优化选项做粗粒度迭代测试,快速定位出关键优化选项组合;最后,借助 Vtune 分析工具,缩小测试用例的分析代码量,锁定热点函数区域(hot function),并细粒度核心差异化分析此区域代码的中间文件(IR),准确定位出编译器的优化方向.

1 研究背景

1.1 机器架构

现今服务器架构愈来愈复杂,多采用非统一内存访问(NUMA)架构[8-9]. 如图 1 所示,该服务系统有 4 个节点,每个节点包含 8 个 Core 处理器,每个处理器直接相连 L1 – 2 高速缓存,每组处理器共享 L3 高速缓存并直连一块存储器. 在不同的机器平台上对同一份编译器源码安装,在编译阶段会根据机器平台硬件参数,如 CPU 型号、缓存大小等因素,生成具有不同能力的编译器. CPU 型号的不同致使指令集不同,如 Intel 最新处理器 Haswell 已将向量指令 AVX 扩展至 AVX2,缓存等级与大小的不同致使编译器局部性优化能力也相差甚远[10],因此编译器的性能与硬件平台具有莫大的关系,为此优化编译器工作必须在特定平台上展开.

1.2 编译器选型

现今主流编译器有 GCC、LLVM、ICC、AOCC. GCC 是 GNU 工程中的核心工具编译器,是 Linux 操作系统默认自带的编译器,也是目前使用最为广泛的编译器之一;它具有非常突出的优点:支持众多的前端语言;支持众多的目标机器体系结构,并且移植性很强;配有完整强大的工具链;但是对于商业化公司它也有无法弥补的缺陷,它的开源许可协议为 GPL(GNU General Public Licence)很难商业化[11]. 与之相类似的编译器 LLVM 有其独特的优势:开源许可协议为 Apache 2.0,可商业化,也正是这一优点得到很多商业大公司的青睐、研究、应用并正反馈;另外还有它强大的技术优势:统一的中间表示(LLVM IR),模块化的框架结构,可重用的工具链技术;正因以上技术优势,可对不同应用(如 GPU 数据库、TVM 深度学习框架、区块链)等快速定制与之应用相匹配的编译器[12]. ICC 编译器是 Intel 为其处理器产品打造的一款高性能、高标准的编译器,也正因它是 Intel 独家主打的商业编译器,它也只能局限于 Inter@ 64 架构和 IA – 32 架构机器上运行. 与之相类似的编译器 AOCC 是 AMD 公司为其一系列处理器特别是 Zen 架构处理器开发的一款高性能、高质量的编译器,它是在 LLVM 编译器的基础上做了进一步的优化,如特定处理器的支持,全局优化,矢量化优化等[13].

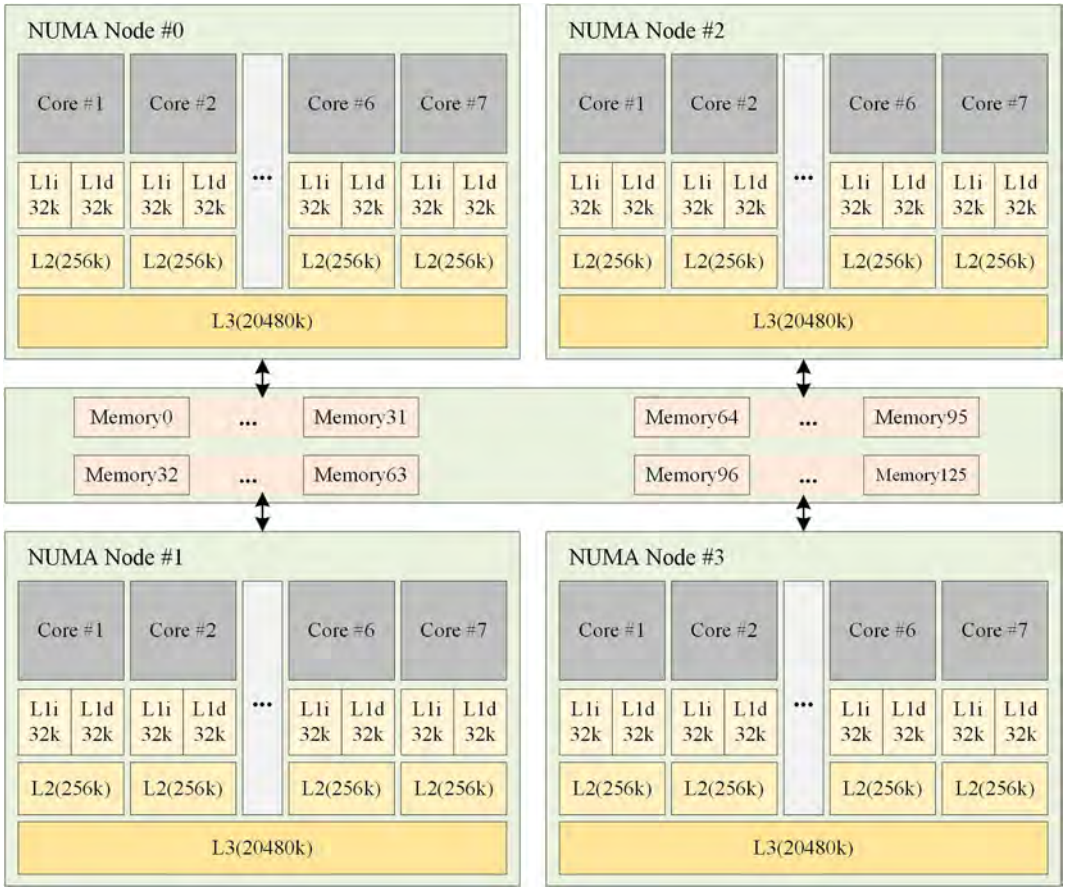

图 1　NUMA 架构示意图

Fig. 1　NUMA architecture diagram

1.3　编译器优化难点

编译器的优化工作是一件繁重且复杂的事情,更多依赖于前期的大量性能摸索实验和后期专家的优化经验. 在前期性能摸索实验中,常选定性能强势的商用编译器如 ICC 和 AOCC 为参照编译器,选定开源编译器 GCC 或 LLVM 为目标编译器,以两种编译器编译运行 SPEC CPU2017 测试用例,对比各个测试用例的性能差;选择出性能差大的测试用例,并采用迭代编译优化的方法,发现目标编译器相较于参照编译器的不足和待优化方向. 但编译器的优化 Pass 非常多,就 GCC 编译器就有上百种编译优化 Pass[6],迭代编译优化方式是一种耗时费力的选择方案. 在后期优化工作中,更多依赖于专家的优化经验. 如对计算机体系结构知识的掌握,测试用例程序行为特征的分析和编译器优化选项实现原理的了解等多方面进行.

2　正确性粗细粒度差异化分析技术

正确性粗细粒度差异化分析技术旨在降低编译器优化过程中的分析难度,快速定位出关键优化选项组合和目标编译器优化方向. 图 2 展示了正确性粗细粒度差异化分析技术框架的主要模块,其中包含三个步骤,依次是正确性性能摸索,粗粒度迭代测试和细粒度核心差异化分析. 首先,以 SPEC CPU2017 标准测试软件对不同编译器做正确且全面的摸底测试;其次针对具体测试用例,以迭代编译技术对常用优化选项做粗粒度迭代测试,快速定位出关键优化选项组合;最后,细粒度核心差异化分析应用程序的热点区域源码以及着重对比应用程序的中间文件(IR),准确定位出编译器的优化方向.

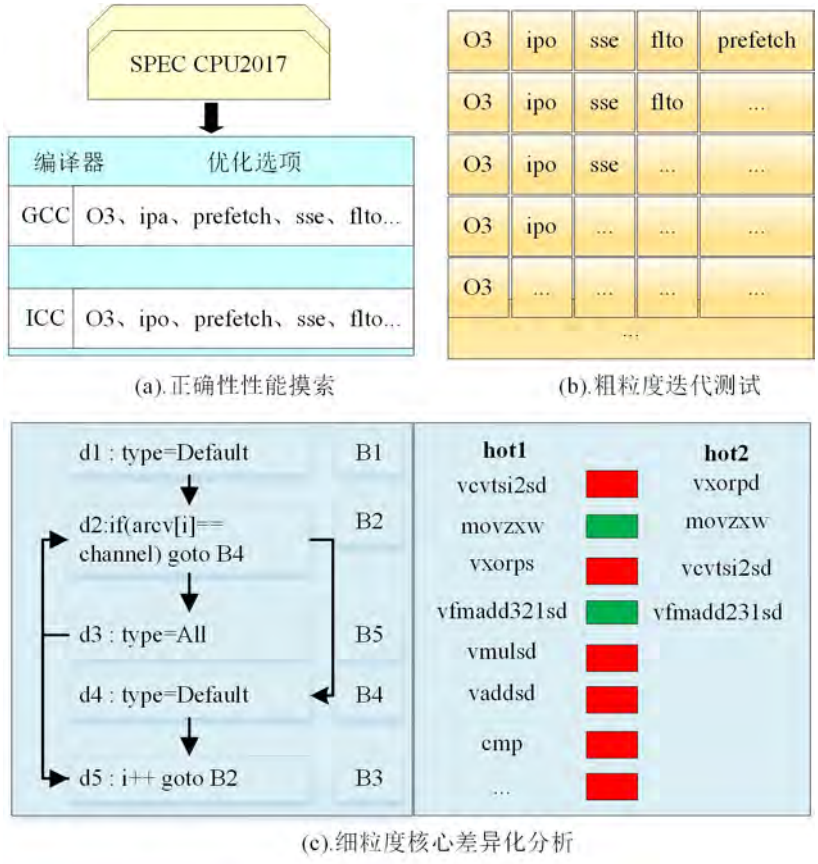

图 2 正确性粗细粒度差异化分析技术

Fig. 2 Coarse and fine – grained optimization differential analysis technology

2.1 正确性性能摸索

正确性性能摸索是该分析技术的第一步,是后续分析优化的基础. 表 1 中列举了机器平台、编译器以及常用优化选项信息. 在此过程中,根据表 1 列举的常用优化选项,用标准化 SPEC CPU2017 测试软件,对两种重要的编译器(GCC、ICC)做性能摸索. 在编译与运行各个测试用例的过程中,确保测试用例的正确性,在此基础上调整编译选项的组合顺序,将性能达到最优. 然后对测试结果做详细统计分析,确定出可以优化的测试用例.

表 1 机器平台及编译器常用优化选项信息

Table 1 Machine platform information and compiler optimization options

机器平台	编译器	版本	常用优化选项
OS:Ubuntu 14.04.5 LTS CPU:Intel Xeon E7 – 4809 v3 CPU MHz:1847.578 Memory:252GB L1i cache:32K L1dcache:32K L2 cache:256K L3 cache:2M	GCC	8.2.0	O3, – march = native, – flto, – frefetech – loop – arrays, – ffast – math, – ffinite – math – only, – mavx, – funroll – loops
	ICC	18.0.3	O3, – march = native, – no – pre – div, – flto, – qopt – prefetch, – ipo, – qopt – mem – layout – trans = 3, – ffinite – math – only

2.2　粗粒度迭代测试

粗粒度迭代测试是正确性粗细粒度差异化分析技术的关键一步,针对每个测试用例将 ICC 结果和 GCC 结果做加速比对比,确定出加速比大的测试用例,该测试用例就是待分析可做优化的例子. 依据表 2 列举的六类常用优化选项(常规优化、过程间优化、向量化优化、预取优化、链接时优化、插桩优化)对选定的测试用例做迭代测试,在测试过程中对优化选项做打开与关闭组合. GCC 和 ICC 编译器都支持以上六类常用优化,但其性能表现不一. 如预取优化是仿存优化中重要优化手段之一,它主要分为指令预取和数据预取,影响此优化的关键因素是发生预取的时间点和预取距离,做到及时、有效、低开销是预取优化的目标. 一般应用程序都会使用预取优化,但不同编译器在此优化上的性能表现有好有坏.

编译器的优化选项远远不止表中所列,但表中的优化选项是常用且重要的优化选项,限定优化选项范围,减轻迭代编译的工作量. 通过粗粒度迭代测试,可大致确定出重要优化选项组合,该优化选项是此测试用例影响性能的关键因素,并且测试出该选项在不同编译器上的优化加速比,估计出优化的价值空间.

表 2　编译器常用优化选项类别信息

Table 2　Compiler common optimization option category information

优化类别	GCC 优化选项	ICC 优化选项
常规优化	O1,O2,O3	O1,O2,O3
过程间优化	-fipa-pure-const, -fipa-functions, -fipa-reference, -fipa-sra -finline-funtions	-ipo -finline-functions
向量化优化	-ftree-vectorize, -ftree-loop-vectorize, -ftree-slp-vectorize	-vec, -vec-guard-write
预取优化	-fprefetch-loop-arrays,prefetch-latency, simultaneous-prefetches	-qopt-prefetch, -qopt-prefetch-distance
链接时优化	-flto	-flto, -ffat-lto-objects
插桩优化	-fprofile-arcs -fprofile-use	-prof-gen, -prof-use

2.3　细粒度核心差异化分析

细粒度核心差异化分析是该技术的最后一步,首先用 Vtune 工具确定应用程序的热点代码,并分析该热点代码的行为特征,如数据流图或控制流图模式,然后是将经编译器编译之后的中间文件信息(IR)做详细对比,发现在指令级别的特殊化差异. 测试应用程序经过编译器编译之后,中间文件大小会急剧增加,不同的编译器指令匹配方式不尽相同,且汇编指令相较于程序语言不容易阅读理解. 为此,需要借助 Vtune 性能分析工具,确定出热点函数区域块或做其它事件的检测,如内存的访问(memory access)或缓存的命中率(cache miss),并以此信息缩小分析代码量,快速定位出详尽可靠的优化方向.

3　实验与分析

3.1　实验平台与测试用例

正确性粗细粒度差异化分析技术是一种高效、实用的编译器优化分析技术. 根据前文对此方法的详细阐述,在实验平台 Intel 机器上,选定 GCC 为目标待优化编译器,ICC 为参照编译器,详细信息如表 1 所示. 并采用性能加速比来对比 GCC 和 ICC 的性能. 如公式 1 所示,以 GCC 为基准 base, gcc^{ratio} 代表用 GCC 编译器编译测试用例的可执行文件运行时间比,以 SPEC CPU2017 计算为准, icc^{ratio} 同理,最终算得性能加速比 speedup:

$$\text{speedup} = \frac{icc^{ratio}}{gcc^{ratio}}. \tag{1}$$

本文实验选用官方权威标准的性能测试集 SPEC CPU2017,该测试集包括地震模型等多领域不同的应用类型,每一个应用可能具有定点或浮点密集、访问密集或条件密集等不同特征[14-15].

3.2 实验结果与分析

如图3所示,是用 GCC 和 ICC 编译器分别编译运行 SPEC CPU2017 定点测试集结果. 如图中所示,625.x264 和 648.exchange2_s 性能加速比非常大,该测试用例结果显示有非常大的优化价值,可以依据对该测试用例做进一步的粗粒度迭代测试和细粒度核心差异化分析,从而锁定出 GCC 编译器相较于 ICC 编译器在此测试用例上的优化不足,已确定出优化 GCC 编译器的方向.

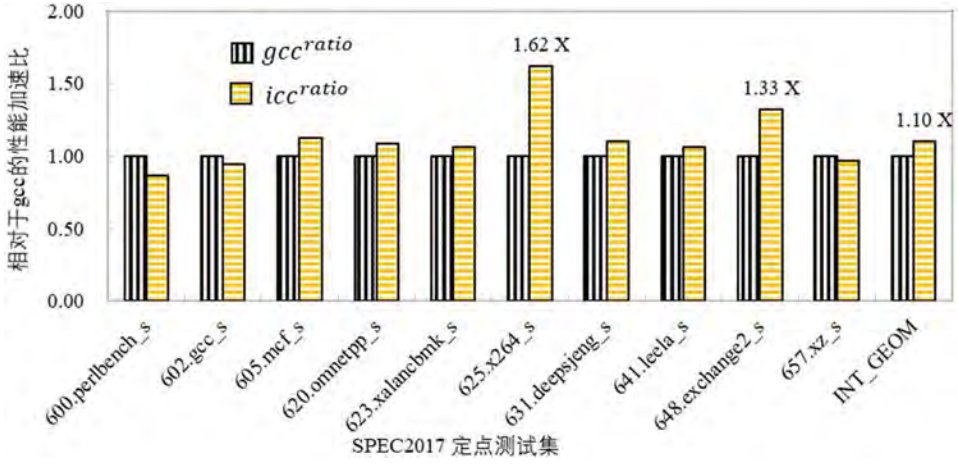

图3 Intel 平台上以 GCC 基准的定点测试

Fig. 3 Integer testing on the Intel platform with GCC as the benchmark

如图4所示,是用 GCC 和 ICC 编译器分别编译运行 SPEC CPU2017 浮点测试集结果. 其显示 619.lbm_s、638.imagick_s 等多个测试用例具有非常大的性能加速比. 相较于定点数据集测试用例,ICC 编译器在浮点数据集测试用例比 GCC 的性能高出约57%,但在定点数据集上只高出10%左右,所以浮点优化是工作中的重中之重. 依据以上结果,对每个测试用例做粗粒度迭代分析和细粒度核心差异化分析,可准确定位出 GCC 编译器的不足与优化方向.

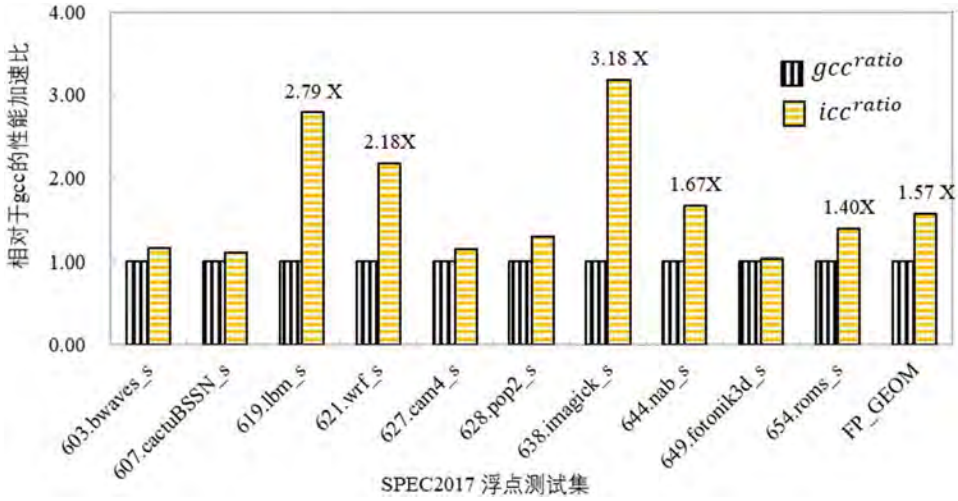

图4 Intel 平台上以 GCC 基准的浮点测试

Fig. 4 Float testing on the Intel platform with GCC as the benchmark

以 638. imagick_s 为例,对其做粗粒度迭代分析,如表 1 所示的常用优化选项,对其进行显示的打开与关闭组合,测其优化选项在该测试用例上的性能加速比. 如图 5 所示,在该测试用例上,GCC 编译器在无效代码删除(dead code elimination)上的性能加速比为 0%,ICC 编译器在此优化选项上的性能加速比为 23%,无效代码删除优化选项已在 O2 开启,并会使用过程间优化分析的信息,ICC 在该优化选项上的加速比远远高于 GCC 在该优化选项上的加速比,即该优化选项是 638. imagick_s 测试用例的重要优化选项.

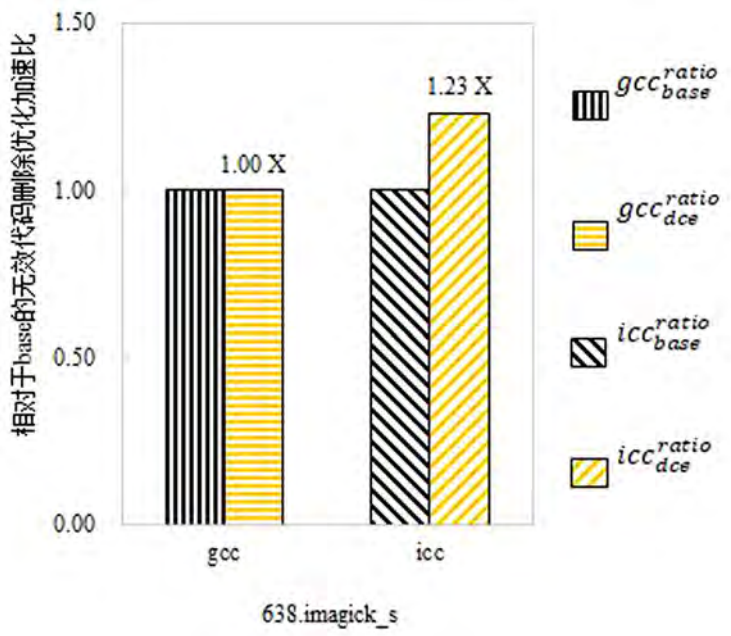

图 5　GCC、ICC 无效代码删除选项测试

Fig. 5　The test of DCE optimization with GCC and ICC

在以上实验分析基础上,借助 Vtune 工具做实验,可确定出热点函数区域,详细分析此热点函数区域代码的数据流图或者控制流图,然后对比此热点函数区域在指令级别的差异,如图 2 所示,在热点函数区域中一块代码为无效代码,并且此无效代码的指令数量上有很大的差别,GCC 对此无效代码有 50 条指令,但是 ICC 却只有 5 条指令. 因此,确定出 GCC 编译器在无效代码删除优化上相对于 ICC 编译器有很大的不同,即无效代码删除优化是 GCC 编译器的一个重要的优化方向.

综上所述,通过该技术可以快速、准确定位出编译器的重要优化选项和优化方向.

4　总结与展望

本文提出了一种正确性粗细粒度差异化编译器分析技术,通过实验验证了该技术的实用性与正确性. 正确且全面摸索了 GCC 和 ICC 的性能,并对其测试结果做详尽分析,选取出可做优化的多个测试用例,准确定位出 GCC 编译器相较与 ICC 编译器在无效代码删除方面的不足. 接下来,需要运用此分析技术,继续准确定位出其他测试用例上 GCC 的优化方向,并且依此优化方向,完善 GCC 现在各方面的不足,提升 GCC 的性能.

参考文献

[1]SUN X H, CHEN Y. Reevaluating Amdahl's law in the multicore era [J]. Journal of Parallel and Distributed Computing, 2010, 70(2): 183 – 188.

[2]伍明川, 黄磊, 刘颖, 等. 面向神威·太湖之光的国产异构众核处理器 OpenCL 编译系统[J]. 计算机学报, 2018, 41(10):64 – 78.

[3]ISLAM M, BANERJEE S, MESWANI M, et al. Prefetching as a Potentially Effective Technique for Hybrid Memory Optimization[C]// 2016.

[4] D YUSHAN, LI CHUNJIANG, YING X U . Implementation and effects of loop – array – prefetching optimization in GCC[J]. Computer Engineering & Applications, 2016:112 – 119.

[5] AMIR MORAD, LEONID YAVITS, RAN GINOSAR. GP – SIMD processing – in – memory[J]. Acm Transactions on Architecture & Code Optimization, 2015, 11(4):1 – 26.

[6] KULKARNI S, CAVAZOSJ. Mitigating the compiler optimization phase – ordering problem using machine learning[J]. Acm Sigplan Notices, 2012,47(10):147 – 162.

[7] CHEN Y, FANG S D , HUANG Y J, et al. Deconstructing Iterative Optimization[J]. Acm Transactions on Architecture And Code Optimization,2012,9(3) :30.

[8] CHRISTOPH LAMETER. NUMA (Non – Uniform Memory Access): An Overview[J]. Queue, 2013, 11(7):19 – 25.

[9] PLAUTH M , HAGEN W , FEINBUBE F, et al. Parallel Implementation Strategies for Hierarchical Non – uniform Memory Access Systems by Example of the Scale – Invariant Feature Transform Algorithm[C]// 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2016.

[10] SURESH D C , JU R , LAI M , et al. Multi – core scalability – impacting compiler optimizations[J]. Computer Science – Research and Development, 2010, 25(1 – 2):15 – 24.

[11] STALLMAN R M, DEVELOPERCOMMUNITY G. Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3[C]// 2009.

[12] GHICA L , TAPUS N. Optimized retargetable compiler for embedded processors – GCC vs LLVM[C]// IEEE International Conference on Intelligent Computer Communication & Processing. IEEE, 2015.

[13] MICHEL LARABEL. AMD Releases Optimizing C/C + + Compiler For Ryzen[EB/OL].(2017 – 5 – 16)[2019 – 12 – 27].

[14] PANDA R , SONG S , DEAN J , et al. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon? [C]// IEEE International Symposium on High Performance Computer Architecture. IEEE Computer Society, 2018.

[15] WU QINZHE, STEVEN FLOLID, SHUANGSONG,et al. Invited Paper for the Hot Workloads Special Session Hot Regions in SPEC CPU2017[C]// 2018 IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2018.

(责任编辑:张阳,付强,李建忠,罗敏;英文编辑:周序林)