软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(12):5387−5401 [doi: 10.13328/j.cnki.jos.007406] [CSTR: 32375.14.jos.007406]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向编译优化结果不一致的代码高效定位*

于恒彪[1], 易 昕[1], 范小康[1], 唐 滔[1], 黄 春[1], 尹帮虎[2], 王 戟[1]

[1](国防科技大学 计算机学院, 湖南 长沙 410073)

[2](国防科技大学 系统工程学院, 湖南 长沙 410073)

通信作者: 易昕, E-mail: yixin09@nudt.edu.cn

摘 要: 编译器是程序开发人员最依赖的性能优化工具之一. 然而, 受限于浮点数有限精度编码问题, 很多编译优化选项会改变浮点计算的语义, 进而导致程序计算结果不一致. 定位程序中导致编译优化结果不一致的语句对于程序性能优化和结果可复现具有重要意义. 当前最先进的方法 PLiner 采用基于语句精度增强的二分搜索来定位导致编译优化结果不一致的代码段, 受限于对多源问题代码的定位支持不够和搜索效率不高问题. 提出一种浮点指令差异性引导的 Delta 调试定位方法 FI3D, 利用 Delta 调试中的回溯机制更好地支持多源问题代码定位, 基于不同编译优化选项下函数浮点指令序列的差异性来引导定位. 使用 NPB 基准测试集中的 6 个应用、GSL 数学库中的 10 个程序和 floatsmith 混合精度测试集中的 2 个程序对 FI3D 进行了评测, 实验结果显示 FI3D 能够成功定位 PLiner 失效的 4 个测试用例, 且对 PLiner 成功定位的 14 个测试用例获得了平均 26.8% 的性能提升.

关键词: 编译优化; 结果不一致; 浮点指令差异性; Delta 调试

中图法分类号: TP311

中文引用格式: 于恒彪, 易昕, 范小康, 唐滔, 黄春, 尹帮虎, 王戟. 面向编译优化结果不一致的代码高效定位. 软件学报, 2025, 36(12): 5387–5401. http://www.jos.org.cn/1000-9825/7406.htm

英文引用格式: Yu HB, Yi X, Fan XK, Tang T, Huang C, Yin BH, Wang J. Efficient Localization for Codes Causing Compilation Optimization-induced Result Inconsistency. Ruan Jian Xue Bao/Journal of Software, 2025, 36(12): 5387–5401 (in Chinese). http://www.jos.org.cn/1000-9825/7406.htm

Efficient Localization for Codes Causing Compilation Optimization-induced Result Inconsistency

YU Heng-Biao[1], YI Xin[1], FAN Xiao-Kang[1], TANG Tao[1], HUANG Chun[1], YIN Bang-Hu[2], WANG Ji[1]

[1](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

[2](College of Systems Engineering, National University of Defense Technology, Changsha 410073, China)

Abstract: The compiler is one of the most relied-upon performance tuning tools for program developers. However, due to the limited precision encoding of floating-point numbers, many compiler optimization options can alter the semantics of floating-point calculations, leading to result inconsistency. Locating the program statements that cause compilation optimization-induced result inconsistency is crucial for performance tuning and result reproducibility. The state-of-the-art approach employs precision enhancement-based binary search to locate the code snippets causing result inconsistency but suffers from insufficient support for multi-source localization and low search efficiency. This study proposes a floating-point instruction difference-guided Delta-Debugging localization method, FI3D, which utilizes the backtracking mechanism in Delta-Debugging to better support multi-source problem code localization and exploits the differences in floating-point instruction sequences under different compiler optimization options to guide the localization. FI3D is evaluated using 6 applications from the NPB benchmark, 10 programs from the GNU scientific library, and 2 programs from the floatsmith mixed-precision benchmark. Experimental results demonstrate that FI3D successfully locates the 4 applications where PLiner fails and achieves an average 26.8% performance improvement for the 14 cases successfully located by PLiner.

Key words: compiler optimization; result inconsistency; floating-point instruction difference; Delta-Debugging

---

* 基金项目: 国家重点研发计划 (2023YFB3001600); 国家自然科学基金 (62272471, 62202488)
收稿时间: 2024-09-23; 修改时间: 2024-12-25; 采用时间: 2025-01-31; jos 在线出版时间: 2025-06-04
CNKI 网络首发时间: 2025-06-05

编译优化会将程序代码段转换为功能等价的更高质量的代码段, 已经成为开发人员优化程序性能最主要的途径之一. 然而, 开发人员往往并不知道每个编译优化选项背后所执行的具体优化策略, 特别是认识不到相同的编译选项在不同编译器内的优化含义也是不同的, 例如 Intel 编译器 ICC 与 GNU 编译器 GCC 在-O3 选项下会执行不同的优化策略. 很多激进编译优化策略会改变浮点操作的语义, 导致程序的计算结果发生改变, 这种现象称为编译优化结果不一致性 (compilation optimization-induced result inconsistency)[1].

浮点数是实数在计算机内部存储表示的事实标准. 受限于浮点数的有限精度编码问题, 舍入误差 (rounding error) 不可避免. 浮点运算不满足实数运算相关的优化理论, 例如结合律和分配率. 浮点融合乘加 (fused multiply add, *fma*) 是高性能计算领域广泛应用的性能优化手段, 然而 *fma* 操作比 *fmul* 和 *fadd* 序列少了一个浮点舍入步骤, 会导致计算结果产生差异性. 为了解决融合乘加所带来的结果差异性, 工程实践中常通过-ffp-contract=off 选项来禁用浮点表达式收缩优化. 为了提升程序性能, 程序开发人员在编译时会使用一些 IEEE-Unsafe 的激进编译优化选项 (例如-freciprocal-math 和-fassociative-math)[2,3], 这也会导致结果不一致性. 例如 GCC 编译运行单精度表达式 1.23+1E10–1E10 时, -O0 选项的结果为 0 (1.23 会在第 1 步求和时被舍入), 而-O3 -ffast-math 选项会将表达式直接优化为 1.23.

有效发现编译优化结果不一致性并定位导致结果不一致性的问题代码对于科学计算的结果可复现性和性能优化具有重要意义[4]; Varity[5]使用随机差分测试来检测编译优化结果不一致性, CIV[6]在输入空间内进行启发式搜索获得触发编译优化结果不一致性的输入. 编译优化结果不一致性问题代码定位主要基于代码精度增强或者编译选项回归的二分搜索 (例如 FLiT[2,3]、PLiner[4]和 CIEL[7]). 在定位到导致编译优化结果不一致性的代码后, 程序开发人员可以仅对该部分问题代码进行高精度浮点执行 (例如 PLiner[4]) 或者浮点表达式精度增强重写 (例如 Herbie[8]、AutoRNP[9]和 ACESE[10]), 使得程序既能获得高级别编译优化所带来的性能提升也满足计算结果的可复现性.

本文主要关注如何更高效地定位导致编译优化结果不一致性的代码段. 当前最先进的定位工作是 PLiner[4], 一种基于浮点语句精度增强的二分搜索定位工具; PLiner 会将可疑代码段中的浮点语句提升为高精度 (long double) 执行, 如果程序精度增强变换后编译优化结果不一致性消失, 则表示触发结果不一致性的问题代码位于被精度提升的代码段中, PLiner 会进一步采用二分搜索来缩小问题代码范围; PLiner 会在函数级、循环级、基本块级和语句级不断采用二分精化搜索来定位问题代码.

然而, PLiner 主要存在两方面不足: (1) 不能有效支持多源问题代码段的定位. 例如当导致编译优化结果不一致性的问题代码分布在两个不同的函数内时, 可能会导致二分搜索定位失效; (2) 对搜索域内的所有代码进行二分搜索导致问题代码定位效率不高. 激进编译优化对不同代码段的影响程度不同, 考虑到高精度浮点的巨大执行开销, 将浮点计算行为未发生改变的浮点代码段也转换为高精度模式并执行会显著增加问题代码定位的时间开销.

为了解决现有编译优化结果不一致性问题代码定位方法存在的问题, 本文提出了一种浮点指令差异性引导的 Delta-Debugging[11]定位方法 FI3D (floating-point instruction difference guided Delta-Debugging). 一方面, 与传统的二分搜索定位不同, FI3D 采用 Delta-Debugging 搜索策略, 借助其内部的回溯机制解决了现有方法对多源问题代码定位支持不够的问题; 另一方面, FI3D 基于函数在编译优化前后的浮点指令序列差异性对函数进行过滤排序, 过滤掉那些编译优化前后浮点计算行为未发生改变的函数, 并优先对浮点指令序列差异性大的函数进行搜索, 显著提升了问题代码定位效率. 在开源 NPB 测试集 (https://www.nas.nasa.gov/software/npb.html)、GSL 数学库[12]和 floatsmith[13]混合精度测试集上的对比实验显示, FI3D 的定位能力和定位效率都显著优于当前最先进工具 PLiner.

本文的主要贡献可归纳如下.

(1) 提出了一种基于 Delta-Debugging 搜索的编译优化结果不一致性问题代码定位方法, 与当前最先进工具 PLiner 相比, 该方法能够有效支持多源问题代码的定位.

(2) 提出了一种基于编译优化前后函数浮点指令序列差异性的代码过滤和排序机制, 能够有效提升问题代码的搜索定位效率.

(3) 基于 NPB 测试集中的 6 个典型应用、GSL 数学库中的 10 个函数和 floatsmith 混合精度测试集中的 2 个程序对本文方法进行了评测, 实验结果表明: 本文方法能够成功定位 PLiner 失效的 4 个测试用例, 且对 PLiner 定

位成功的 14 个测试用例能够获得平均 26.8% 的性能提升.

本文第 1 节介绍相关工作, 包括基于高精度浮点差异测试的误差检测、编译优化结果差异性分析和浮点精度修复. 第 2 节介绍本文所需的基础知识, 包括编译优化结果差异性概念和基于浮点精度增强的问题代码二分搜索. 第 3 节介绍本文提出的浮点指令差异性引导的 Delta-Debugging 搜索定位方法. 第 4 节通过对比实验展示本文方法的有效性和高效性. 最后总结全文.

1 编译优化结果差异性分析相关工作

本节详细介绍编译优化结果差异性分析的相关工作, 包括基于高精度浮点差异测试的误差检测、触发编译优化结果差异性的输入生成方法、面向编译优化结果差异性的问题代码定位方法和基于程序变换的浮点精度问题修复技术.

(1) 基于高精度浮点差异测试的浮点误差检测. FPDebug[14]和 Herbgrind[15]在实际执行程序每条浮点指令时会同时基于高精度数学库 (MPFR[16]) 做影子执行, 通过对比影子执行和实际执行结果的差异来评估浮点操作的舍入误差; BGRT[17]、EAGT[18]、LGSA[19]和 DEMC[9]设计了不同的启发式搜索策略在输入域内采样获取触发高浮点误差的输入, FPGen[20]将触发浮点误差的输入生成问题转换为程序分支覆盖问题, 并基于符号执行[21]进行求解. 这些方法都会执行搜索得到的输入并与高精度模式下的执行结果进行对比, 依此来评估浮点输入所产生的误差. 与上述方法直接将高精度浮点执行作为测试预言 (test oracle) 不同, FI3D 是根据代码段的高精度版本是否会解决结果不一致性问题来判定该段代码是否是触发编译优化结果差异性的问题来源. 此外, FI3D 基于编译优化前后浮点指令差异性优先对浮点计算行为改变大的代码段进行精度增强测试, 有效减少了高精度浮点代码变换和执行的开销.

(2) 触发编译优化结果差异性的输入生成方法. Varity[5]能够生成随机测试用例, 对随机采样得到的输入对比不同编译优化选项下的执行结果, 评估该输入是否触发了编译优化结果不一致性; CIV[6]基于输入空间划分和马尔可夫链蒙特卡洛 (Markov chain Monte Carlo, MCMC)[22] 采样快速产生触发编译优化结果差异性的输入; CIGEN[23]基于浮点输入域划分和覆盖率引导的输入采样来产生触发编译优化结果不一致性的输入区间. 与这些触发编译优化结果不一致性的输入生成方法不同, FI3D 是对已发现的编译优化结果不一致性进行问题代码定位; Bintuner[24]研究编译优化选项对编译生成的二进制指令序列的影响, 并基于遗传算法来产生触发二进制指令序列最大差异性的编译优化配置. 与 Bintuner 关注通过编译优化配置来最大化二进制指令序列差异性不同, FI3D 关注面向编译优化结果差异性的问题代码定位.

(3) 编译优化结果差异性的问题代码段定位. FLiT[2,3]通过模糊测试发现触发两个编译优化选项结果不一致的输入, 然后通过对文件和函数的二分搜索来定位导致结果差异性的文件和函数; FLiT 在搜索过程中观察将给定文件/函数的编译选项恢复成基础编译选项后结果不一致性是否消失来判定问题代码是否在当前文件/函数列表内. 显然 FLiT 只能在文件/函数级别定位问题代码, 且不适用于代码文件中包含大量函数或者函数体语句较多的场景; PLiner[4]是一种基于浮点精度增强的二分搜索定位工具; PLiner 在搜索过程中会将可疑问题代码段自动转换为高精度版本, 然后使用激进编译优化选项编译执行, 如果结果不一致性消失则表示问题代码包含在提升精度的代码中, 会对该段代码进一步进行二分搜索; PLiner 会在函数级、循环级、基本块级和语句级分层进行精化搜索, 适用场景更全面且代码定位粒度更细; PLiner 的二分搜索策略不能有效支持多源问题代码的定位, 且二分搜索的效率不高问题会加重定位过程中高精度浮点运算引入的巨大执行开销; FI3D 从两个方面对 PLiner 进行提升, 一方面采用 Delta-Debugging 搜索来支持多源问题代码的定位, 另一方面基于编译优化前后的浮点指令差异性来引导搜索, 提升问题代码定位效率; CIEL[7]是一个面向 GPU 程序编译优化结果差异性的问题代码定位工具, 定位思想与 PLiner 基于语句增强的二分搜索类似. 与 CIEL 不同, FI3D 面向传统的 CPU 程序, 且 FI3D 浮点指令差异性指导的 Delta-Debugging 搜索方法可以直接用于优化 CIEL.

(4) 基于程序变换的浮点精度修复技术. 程序变换被广泛应用于浮点精度问题修复, 最直接方式是将程序转换为高精度浮点执行; AutoRNP[9]会针对给定输入下的浮点误差生成补丁代码, 基于插值技术拟合高精度计算结果; Herbie[8]为了提升浮点表达式的计算准确性, 会将输入区间划分成不同子区间, 然后在不同输入子区间内使用精度

更高的浮点表达式替换重写; ACESE[10]基于误差微结构和统计信息来获取浮点误差分布, 并通过程序综合生成多项式补丁代码来修复精度问题. 在 FI3D 定位到导致编译优化结果不一致性的问题代码段后, 可以使用上述浮点精度修复技术对这些代码段进行重写, 使得程序既可以获得高级别编译优化带来的性能提升又能够确保结果可复现性. 需要特别指出的是高精度浮点执行在绝大部分情况下可以解决浮点舍入误差问题, 但是并不适用于那些罕见的浮点特定精度 (precision-specific)[25]运算场景. 例如, GLIBC 库的 exp 函数定义了双精度浮点变量 n=6755399441055744.0, 使用语句 $(x+n)-n$ 来对双精度浮点数 x 取整, 显然该取整功能只面向双精度浮点数, 将变量 x 和 n 提升为更高精度则会破坏程序原有语义.

2 背景知识

本节主要介绍浮点表示、编译优化结果差异性和基于精度增强的编译优化结果差异性定位.

2.1 浮点表示和编译优化结果差异性

浮点数是实数在计算机内部的近似表示. IEEE-754[26]是应用最广泛的浮点数编码标准, 一个浮点数可以表示为: $(-1)^S\times M\times 2^E$. 符号位 S 为 1 表示负数, 0 表示正数; $M=m_0.m_1m_2\ldots m_n$ 是尾数, 其中 m_0 为隐藏位, $.m_1m_2\ldots m_n$ 是 n 位尾数位; $E=e-bias$ 是 p 位阶码, 其中 e 是按位存储的偏置阶码, 偏置值 $bias$ 为 $2^{p-1}-1$. 表 1 给出了不同精度浮点数 IEEE-754 的编码格式.

表 1 IEEE-754 浮点编码格式

精度	符号位	阶码	尾数
fp16	1	5	10
float	1	8	23
double	1	11	52

由于浮点数采用有限精度编码, 舍入误差不可避免. 当一个实数不能被浮点数精确编码表示时, 会根据系统的舍入模式 (例如就近舍入) 对其进行近似表示. 假定 R 和 F 分别表示实数和浮点数集合, 给定实数 x ($x\in R$), 它的浮点表示为 x_f ($x_f\in F$), 对应的舍入误差为 $x-x_f$. 绝对误差 (absolute error, Err_{abs}) 和相对误差 (relative error, Err_{rel}) 被广泛用来衡量舍入误差. 给定浮点函数 $f(x)$, 假定 $f_r(x)$ 表示函数的精确计算结果, 绝对误差 Err_{abs} 和相对误差 Err_{rel} 的定义如下:

$$Err_{\text{abs}}(f(x),f_r(x))=|f_r(x)-f(x)| \tag{1}$$

$$Err_{\text{rel}}(f(x),f_r(x))=\left|\frac{f_r(x)-f(x)}{f_r(x)}\right| \tag{2}$$

由于浮点舍入误差的存在, 浮点运算并不满足实数运算的相关优化理论, 因此编译器的激进优化会改变浮点计算行为, 例如常用的-ffast-math 优化选项; Intel 科技报告[1]中专门介绍了 ICC 编译器各种编译优化选项可能导致的结果差异性. 需要特别指出的是相同的编译选项在不同编译器中的优化含义有可能不同, 例如-O3 选项在 IBM XLC 编译器中会执行 IEEE-754 不安全的优化遍, 而在 GCC 编译器中则不会.

美国劳伦斯利弗莫尔国家实验室 (LLNL) 在使用 XLC 编译器和-O3 选项编译运行流体动力学应用 Laghos (https://github.com/CEED/Laghos) 时产生了明显不一致的能量计算结果, 开发人员耗费了大量的精力来分析汇编代码查找-O3 选项产生结果不一致性的原因. 图 1 给出了 Laghos 应用中提取的触发编译优化结果不一致性的核心代码示例[4]. 当使用 XLC 编译器基于-O2 选项编译运行该示例程序时输出−1.350 7, 而 XLC 编译器基于-O3 选项编译运行则输出-inf. 通过检查代码生成的汇编代码发现第 6 行代码中的除法操作在-O3 选项下被优化为了求倒数操作和乘法操作. 由于 gradv10 是一个非规格化浮点数, 2*gradv10 求倒数时发生了浮点上溢出异常. 实验室开发人员分析后通过将代码中的浮点变量由 double 类型提升为 long double 类型在-O3 选项下编译运行也能够产生与-O2 一致的能量计算结果.

```
1  #include <stdio.h>
2  int main ( ) {
3      double gradv11 = −3.935e−309;
4      double gradv00 = 1.430e−309;
5      double gradv10 = 1.986e−309;
6      double zeta = (gradv11− gradv00)/(2*gradv10);
7      printf ( “zeta = %.5g\n ”, zeta);
8      return 0;
9  }
```

图 1 Laghos 应用中提取的编译优化结果不一致性示例程序

2.2 PLiner 编译优化结果差异性定位

高效定位导致编译优化结果差异性的代码对于数值程序的结果可复现性和性能优化具有重要意义, 即开发人员可以仅对定位到的代码片段进行基于精度提升或者表达式重写的浮点精度修复, 使得程序在进行高级别编译优化时仍具有结果的可复现性; PLiner[4]是现有最先进的编译优化结果差异性定位工具, 能够进行语句级问题代码定位.

图 2 给出了 PLiner 框架图; PLiner 的出发点是浮点表达式在提升为高精度后, 浮点舍入误差问题会极大缓解, 浮点运算行为接近相应的实数运算, 从而避免了激进编译优化对浮点计算行为的影响 (忽略特殊的 precision-specific 操作). 因此, PLiner 基于浮点表达式精度增强来不断搜索定位导致编译优化结果不一致性的代码.

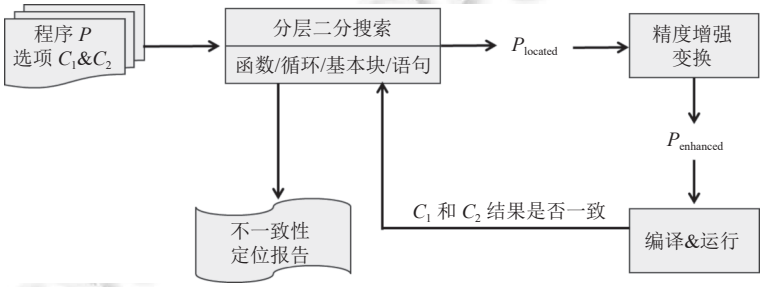

图 2 PLiner 编译优化结果差异性问题代码定位框架图

PLiner 的输入包括待分析的程序 P、导致计算结果不一致的编译优化选项 C_1 和 C_2 (假定 C_2 为高级别编译优化选项); PLiner 会对程序进行分层二分搜索定位. 首先, 在函数层面会将 P 中所有函数都定位为可疑代码 P_{located}; 然后, 调用 LLVM 优化遍 (optimization pass) 自动将可疑代码中的浮点操作提升为 long double 类型, 生成精度增强后的程序 P_{enhanced}. 为了便于扩展支持更多的编译器和硬件架构, PLiner 在源代码层面做语句精度提升变换; 接着, 会采用激进编译优化选项 C_2 编译运行 P_{enhanced}, 如果结果不一致性消失, 则表明问题代码段位于当前定位的可疑代码 P_{located} 内, 会在后续搜索时将 P_{located} 中函数列表均分, 再重复该定位过程, 否则返回最近通过精度提升解决结果不一致性的函数列表. 在定位到问题函数后, PLiner 会依次对问题函数内部的循环体、基本块、基本块内的语句依次采用类似的基于精度提升的二分搜索进一步定位, 最终得到导致编译优化结果不一致的问题语句集.

PLiner 存在如下 3 个主要不足.

(1) 缺乏对多源问题代码的定位支持. 假设程序 P 包含 4 个函数<$func_1$, $func_2$, $func_3$, $func_4$>且问题代码位于函数 $func_2$ 和 $func_3$ 中. 4 个函数都转换为高精度能够修正结果不一致性问题, 二分搜索会将函数列表均分, 而对均分后的两个函数列表<$func_1$, $func_2$>和<$func_3$, $func_4$>分别提升精度均不能修正结果不一致性问题, 最终会导致 PLiner 返回全部 4 个函数的列表<$func_1$, $func_2$, $func_3$, $func_4$>, 不能有效精确定位导致结果不一致性的函数 $func_2$ 和 $func_3$.

(2) 搜索效率不高. 二分搜索效率依赖函数列表的排序布局, 由于缺乏每个函数浮点计算行为受激进编译优化的影响程度信息, 默认的函数排序会导致问题函数定位效率低. 例如, 假设程序 P 包含 4 个函数<$func_1$, $func_2$, $func_3$, $func_4$>且问题代码位于函数 $func_4$ 内, 二分搜索过程中会尝试 5 次精度增强转换和编译执行才会定位到 $func_4$. {<$func_1$>, √}表示提升 $func_1$ 精度成功解决了结果不一致性问题, {<$func_1$>, ×}表示提升 $func_1$ 精度没有解决结果不一致性问题; PLiner 对程序 P 的精度提升定位过程为 {<$func_1$, $func_2$, $func_3$, $func_4$>, √} → {<$func_1$, $func_2$>, ×} → {<$func_3$, $func_4$>, √} → {<$func_3$>, ×} → {<$func_4$>, √}. 而如果列表顺序变为<$func_4$, $func_1$, $func_2$, $func_3$>, 二分搜索过程

只需要 3 次精度增强转换和编译执行即可定位到 $func_4$. 精度提升定位过程为{<$func_4$, $func_1$, $func_2$, $func_3$>, √} → {<$func_4$, $func_1$>, √} → {<$func_4$>, √}.

(3) 缺乏对多问题文件的精化搜索支持. 当前 PLiner 的实现机制中没有很好地支持多文件程序, 其搜索逻辑中没有设计如何对定位到的多个问题代码文件进行精化定位. 考虑到实际浮点应用往往包含多个源代码文件, 从提高工具实用性的角度需要扩展支持多问题文件的精化搜索.

3 浮点指令差异性引导的 Delta-Debugging 搜索定位

3.1 方法框架介绍

为了有效应对现有编译优化结果差异性定位工具存在的不足, 本文给出了浮点指令差异性引导的 Delta-Debugging 搜索定位方法 FI3D. FI3D 的核心思想是使用 Delta-Debugging 中的回溯机制实现对多源问题代码定位的支持, 使用函数不同编译选项下浮点指令序列差异性系数来引导搜索进而提升定位效率. 图 3 给出了 FI3D 的框架图; FI3D 的输入包含待分析程序 P (程序输入为了简洁性略去)、编译优化选项 C_1 和 C_2; $Execute(P, C)$ 表示基于选项 C 编译运行程序 P 的执行结果. 编译优化结果不一致性是指 $Execute(P, C_1)$≠$Execute(P, C_2)$. 需要指出的是判定浮点结果一致性会评估两种编译优化模式下运行结果的相对误差是否超过给定的相对误差阈值. 首先, FI3D 内部的指令记录优化遍会记录程序 P 中每个函数 $func_i$ 基于编译优化选项 C_1 和 C_2 生成的浮点指令序列 $trace_i$ 和 $trace_i'$; 接着通过指令序列差异性评估模块计算每个函数的浮点指令序列在不同编译选项下的差异性系数, 过滤掉那些不存在差异性的函数, 并将函数按照差异性系数从大到小排列; 最后调用 Delta-Debugging 搜索模块对输入的函数列表进行搜索定位, 获取导致编译优化结果差异性的代码段.

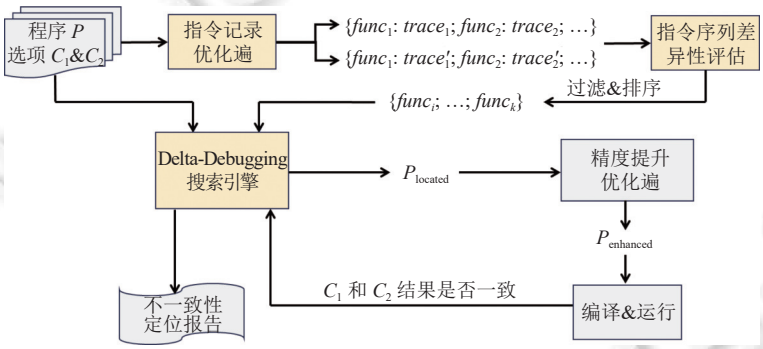

图 3 FI3D 编译优化结果差异性问题代码定位框架图

在 FI3D 的框架中, 浮点指令记录优化遍和浮点指令序列差异性评估模块能够过滤掉那些浮点计算行为没有被激进编译优化改变的文件/函数, 并将浮点计算行为改变程度大的文件/函数放置待搜索文件/函数列表的前部, 这会提升 Delta-Debugging 搜索定位效率 (详情参见第 3.2 和 3.3 节); Delta-Debugging 搜索引擎中的回溯机制能够很好地解决多源问题代码的定位问题, 内部的精化机制能够不断缩小可疑代码的范围 (详情参见第 3.3 节). 需要特别指出的是 Delta-Debugging 分层搜索引擎中扩展了对多问题文件的精化定位支持 (详情参见第 3.4 节).

3.2 编译优化浮点指令序列差异性评估

激进编译优化导致程序结果发生改变, 根本原因是改变了程序中的浮点计算行为. 浮点指令序列是浮点计算行为最直接的描述. 浮点计算行为改变程度越大的函数越有可能是导致程序结果发生改变的原因. 无论是二分搜索还是 Delta-Debugging, 搜索定位效率与文件/函数的排列顺序有直接关系. 为了提升搜索定位效率, 本文对待搜索文件/函数进行过滤排序.

以函数级为例, 算法 1 给出了过滤排序过程, 输入包括目标程序 P、函数列表 $flist$、编译优化选项 C_1 和 C_2, 输出包括过滤和排序后的新的函数列表 $list$. 列表 $list$ 和字典 $score\_dict$ 初始为空. 首先, 分别基于编译优化选项

C_1 和 C_2 在编译程序 P 时记录每个函数的浮点指令序列 (第 3, 4 行). 需要说明的是本文通过静态扫描函数指令序列的方式来获取函数的浮点指令序列, 并非在运行时检查记录浮点指令. 因为包含循环体的程序动态执行时会产生大量浮点指令, 极大增加浮点指令序列差异性系数的计算开销. 接着, *Diff* 函数会计算每个待搜索函数在不同编译优化选项下的浮点指令序列差异性系数, 并作为该函数的优先级评分 (第 5 行). 差异性为 0 表示该函数在不同编译优化选项下浮点计算行为未发生改变, 直接过滤掉 (第 6, 7 行). 最后 *sort*(*list*, *score_dict*) 会按照差异性系数的降序对列表 *list* 进行排序 (第 9 行). 函数的差异性系数高意味着该函数的浮点计算行为受高级别编译优化改变的程度大, 将这些函数放置在待定位函数列表的前部, 会被 Delta-Debugging 优先探索定位.

算法 1. 基于浮点指令序列差异性的函数过滤和排序 *FilterSort*(P, *flist*, C_1, C_2).

输入: 目标程序 P、函数列表 *flist*、编译优化选项 C_1 和 C_2;
输出: 过滤排序后的待搜索定位函数列表 *list*.

FilterSort(P, *flist*, C_1, C_2)
1. *list* = [], *score_dict* = {}
2. for *func* in *flist* do
3. tf_1← *compileRecord*(P, *func*, C_1)
4. tf_2← *compileRecord*(P, *func*, C_2)
5. *score_dict*[*func*] = *Diff*(tf_1, tf_2)
6. if *score_dict*[*func*] ≠ 0 then
7. *list.add*(*func*)
8. end for
9. *sort*(*list*, *score_dict*)
10. return *list*

下面详细介绍浮点指令差异性系数计算函数 *Diff*. 给定两个浮点指令序列 tf_1 和 tf_2, 函数 *Ratio*(tf_1, tf_2) 会计算两个浮点指令序列的相似值; *Diff*(tf_1, tf_2)=1.0–*Ratio*(tf_1, tf_2). *Ratio*(tf_1, tf_2) 计算基于经典的最长匹配迭代计算策略 (https://docs.python.org/3/library/difflib.html#sequencematcher-objects), 具体步骤如下.

(1) 队列 *QS* 包含待计算相似值的序列对, 初始为{(tf_1, tf_2)}. 队列 *MS* 存放计算得出的匹配序列, 初始为空队列. 执行步骤 2.

(2) 判定 *QS* 是否为空, 若为空则跳转执行步骤 4; 否则, 从 *QS* 中取出一个待匹配指令序列对 (t_1, t_2), *Match*(t_1,t_2) 会获取 t_1 和 t_2 中的最长匹配序列, 记为 *ms*, 并将 *ms* 加入到 *MS* 中. 执行步骤 3.

(3) *ExcludeL*(t_1, t_2, *ms*) 计算序列 t_1 和 t_2 排除掉 *ms* 后得到的左侧指令序列, 记录为 t_1^l 和 t_2^l, 若果 t_1^l 和 t_2^l 均不为空, 则将 (t_1^l, t_2^l) 加入到 *QS* 中. 类似的, *ExcludeR*(t_1, t_2, *ms*) 计算序列 t_1 和 t_2 排除掉 *ms* 后得到的右侧指令序列, 记录为 t_1^r 和 t_2^r, 若果 t_1^r 和 t_2^r 均不为空, 则将 (t_1^r, t_2^r) 加入到 *QS* 中. 跳转至步骤 2.

(4) 计算 *MS* 中所有指令序列的长度和, 记为 *M*. 计算 tf_1 和 tf_2 的指令长度和, 记为 *T*; *Ratio*(tf_1, tf_2)=2×*M*/*T*.

假定存在两个浮点指令序列 tf_1=<*fsub*, *fadd*, *fma*, *fmov*, *fdiv*, *fmul*>, tf_2=<*fadd*, *fma*, *fdiv*, *fmul*>. 首先计算得出最长匹配序列为<*fadd*, *fma*>加入到 *MS* 中, 此时 t_1^l=<*fsub*>, t_2^l=<>, t_1^r=<*fmov*, *fdiv*, *fmul*>, t_2^r=<*fdiv*, *fmul*>. 因为 t_1^r 和 t_2^r 均非空, 故将 (t_1^r, t_2^r) 加入到 *QS* 中; 接着从 *QS* 取出待计算匹配对 (<*fmov*, *fdiv*, *fmul*>, <*fdiv*, *fmul*>), 得出最长匹配对<*fdiv*, *fmul*>, 将其加入到 *MS* 中, 此时 t_1^l=<*fmov*>, t_2^l=<>, t_1^r=<>, t_2^r=<>, 因此 *QS* 不会再加入新的待计算匹配对并变为空. 最终 *MS*={<*fadd*, *fma*>, <*fdiv*, *fmul*>}, 其所有序列的长度和 *M* 为 4, tf_1 和 tf_2 的指令长度和为 10, 因此 *Ratio*(tf_1, tf_2)=2×4/10=0.8. 计算得到相似值 *Ratio* 后, 差异值 *Diff*(tf_1, tf_2)=1.0–*Ratio*(tf_1, tf_2)=0.2.

3.3 基于 Delta-Debugging 的编译优化结果不一致性搜索定位

Delta-Debugging[11]由 Zeller 教授提出, 用于在软件迭代开发过程中精确定位导致测试用例测试失效的代码改

动位置. 搜索配置是一组代码改动位置的集合, Delta-Debugging 假定搜索配置满足单调性 (若配置 c 导致了测试失效则任何包含 c 的配置都会导致测试失效)、无歧义性 (若配置 c_1 和 c_2 均能导致测试失效则 $c_1 \wedge c_2$ 也能导致失效, 即要求产生失效的改动集是唯一的) 和一致性 (任何配置的测试结果是确定的, 即通过或失效).

给定原始程序 P 和对 P 进行代码改动的位置集合$\{l_1, l_2,\ldots, l_n\}$更新后测试失效. Delta-Debugging 调试 $DD(P, \{l_1, l_1,\ldots, l_n\})$ 会返回导致测试失效的最小代码改动位置集合. 如果 n=1 则返回 l_1. 否则令 $P_1 \leftarrow P \oplus \{l_1,\ldots, l_{n/2}\}$, $P_2 \leftarrow P \oplus \{l_{n/2+1},\ldots, l_n\}$, 其中 $\oplus$ 表示代码合并. 如果 P_1 测试失效返回 $DD(P, \{l_1,\ldots, l_{n/2}\})$, 如果 P_2 测试失效返回 $DD(P, \{l_{n/2+1},\ldots, l_n\})$, 否则返回 $DD(P_2, \{l_1,\ldots, l_{n/2}\}) \cup DD(P_1, \{l_{n/2+1},\ldots, l_n\})$. Delta-Debugging 本质是一种带回溯的分治搜索过程.

本文将 Delta-Debugging 思想移植应用到导致编译优化结果差异性的问题代码定位上. 搜索过程仍然是沿用 PLiner 的分层搜索, 即先定位问题文件、函数列表, 再依次定位函数内部的循环、基本块和语句. 为了简洁性, 本文以函数级的 Delta-Debugging 搜索为例进行说明.

算法 2 展示了基于 Delta-Debugging 的函数级代码定位过程. 算法输入包括程序 P、函数列表 *flist* 和触发结果不一致性的不同编译选项 C_1 和 C_2, 这里假定 C_2 为高级别编译优化选项, 输出包括定位是否成功标识和触发编译优化结果不一致性的问题函数列表. 给定函数列表 $list_1$ 和 $list_2$, $list_1 \cup list_2$ 表示将两个列表中的函数序列求并集后生成的函数列表. 函数 *transform*(*P*, *list*) 会将程序 P 中在列表 *list* 里的函数都提升为高精度, 生成精度提升版程序. 如果在高级别编译优化选项 C_2 下执行所有函数都精度增强后的程序仍然存在结果不一致性则返回定位失败和 *flist* (第 2, 3 行), 否则基于 Delta-Debugging 来搜索定位 *flist* 中触发结果不一致性的函数.

DDFuncLoc 定位是一个递归分治过程. 首先, 如果当前成功修复结果不一致性问题的增强函数列表 *flist* 只包含一个函数, 则直接返回 true 和该列表 (第 4, 5 行). 接着, 函数 *partition*(*flist*) 会将 *flist* 中的函数均分成 left 和 right. 如果增强 left 中的函数精度能解决结果不一致性, 则递归调用 *DDFuncLoc*(*P*, left, C_1, C_2) 来对 left 列表继续精化搜索 (第 7–10 行). 类似的, 如果增强 right 中的函数精度能解决结果不一致性, 则递归调用 *DDFuncLoc*(*P*, right, C_1, C_2) 来对 right 列表继续精化搜索 (第 11–14 行). 当列表 left 和 right 单独增强精度都不能解决结果不一致性时, 按照 Delta-Debugging 思路通过回溯递归来搜索定位. 具体而言, 将 P 中 left 列表所包含的函数精度增强得到新的代码基准 P_{left}, 递归调用 *DDFuncLoc*(P_{left}, right, C_1, C_2) 来搜索 right 列表中导致结果不一致性的函数 (第 16 行). 类似的, 将 P 中 right 列表所包含的函数精度增强得到新的代码基准 P_{right}, 递归调用 *DDFuncLoc*(P_{right}, left, C_1, C_2) 来搜索 left 列表中导致结果不一致性的函数 (第 17 行). 最后, 定位成功情况为左右两侧定位情况的逻辑与 $flag_1 \wedge flag_2$, 定位结果为左右两侧定位结果的并集 $list_1 \cup list_2$ (第 18–20 行).

算法 2. 基于 Delta-Debugging 的函数级搜索定位算法 *DDFuncLoc*.

输入: 目标程序 P、函数列表 *flist*、编译优化选项 C_1 和 C_2;
输出: 定位是否成功标识、触发编译优化结果差异性的问题函数列表.

DDFuncLoc(*P*, *flist*, C_1, C_2)
1. $P_{\text{enhanced}} \leftarrow$ *transform*(*P*, *flist*)
2. if *Execute*(P_{enhanced}, C_2) ≠ *Execute*(*P*, C_1) then
3. return false, *flist*
4. if *size*(*flist*)==1 then
5. return true, *flist*
6. left, right ← *partition*(*flist*)
7. $P_{\text{left}} \leftarrow$ *transform*(*P*, left)
8. if *Execute*(P_{left}, C_2) == *Execute*(*P*, C_1) then
9. *flag*, *list* = *DDFuncLoc*(*P*, left, C_1, C_2)

10. return *flag*, *list*
11. $P_{\text{right}} \leftarrow transform(P, \text{right})$
12. if $Execute(P_{\text{right}}, C_2) == Execute(P, C_1)$ then
13. $flag, list = DDFuncLoc(P, \text{right}, C_1, C_2)$
14. return *flag*, *list*
15. else
16. $flag_1, list_1 = DDFuncLoc(P_{\text{left}}, \text{right}, C_1, C_2)$
17. $flag_2, list_2 = DDFuncLoc(P_{\text{right}}, \text{left}, C_1, C_2)$
18. $flag = flag_1 \wedge flag_2$
19. $list = list_1 \cup list_2$
20. return *flag*, *list*

回到二分搜索不能有效定位多源问题代码的例子, 即程序 P 包含 4 个函数<$func_1, func_2, func_3, func_4$>且问题代码位于函数 $func_2$ 和 $func_3$ 中; Delta-Debugging 在判定元组 (<$func_1, func_2$>, <$func_3, func_4$>) 和 (<$func_3, func_4$>, <$func_1, func_2$>) 不能消除不一致性后, 会回溯递归搜索. 首先, 在增强左侧列表<$func_1, func_2$>精度的前提下, 递归搜索右侧列表<$func_3, func_4$>. 显然元组 (<$func_1, func_2, func_3$>, <$func_4$>) 修复成功, 因此基于左侧精度增强的右侧搜索会返回 true 和<$func_3$>. 类似的, 在增强右侧列表<$func_3, func_4$>精度的前提下, 递归搜索左侧列表<$func_1, func_2$>. 显然元组 (<$func_3, func_4, func_2$>, <$func_1$>) 修复成功, 因此基于右侧精度增强的左侧搜索会返回 true 和<$func_2$>. 因此, Delta-Debugging 搜索最终会成功返回 true 和<$func_2, func_3$>.

3.4 多文件项目支持

现有最先进编译优化结果不一致性定位工具 PLiner 不支持多问题文件的精化搜索定位; FI3D 在文件层面扩展了 Delta-Debugging 搜索定位机制. 首先, 基于第 3.3 节介绍的 Delta-Debugging 搜索定位到触发结果不一致性的文件列表 $filelist$=<$file_1, file_2, \ldots, file_n$>. 接着按照如下策略进行每个文件上的精化搜索 (i 初始取值为 0):

(1) 如果 $i>n$ 跳转执行步骤 4, 否则, 精化搜索 $file_i$. 对于列表中其他文件 $file_j$ ($j \neq i \wedge 1 \leqslant j \leqslant n$), 首先对其进行备份, 接着如果存在已经定位好的 $file_j^l$, 则使用 $file_j^l$ 替换 $file_j$, 否则直接使用 $file_j$ 的高精度版本进行替换.

(2) 对 $file_i$ 进行单文件精化搜索定位, 即依次在函数级、循环级、基本块级和语句级依次进行精化搜索, 记录只将 $file_i$ 最终定位的问题语句进行精度提升的文件版本为 $file_i^l$.

(3) 将文件 $file_j$ ($j \neq i \wedge 1 \leqslant j \leqslant n$) 恢复成备份的原始文件, 执行 $i=i+1$ 并跳转执行步骤 1.

(4) 多文件最终的搜索定位结果记录为 $\{file_i^l \mid 1 \leqslant i \leqslant n\}$ ($file_i^l$ 和 $file_i$ 差异性即为每个文件的定位结果).

显然, 多文件精化搜索定位的过程是一个增量式控制变量定位过程, 每次都将当前搜索定位文件外的其他文件替换为针对定位结果进行高精度修复的版本或者完全高精度版本, 然后再对当前文件进行单文件上的精化搜索定位. 实验中选取了多文件用例进行了评测, FI3D 成功定位到了多个目标文件中触发编译优化结果不一致性的代码, 显示了本文方法的有效性.

4 实验分析

4.1 工具实现

本文基于 LLVM 编译器、Python difflib 模块和经典编译优化结果不一致性定位工具 PLiner 实现了 FI3D. 如图 4 所示, FI3D 主要包括 3 个模块: 不同编译选项下的浮点指令记录模块、浮点指令序列差异性计算模块、基于 Delta-Debugging 的代码搜索定位模块; FI3D 支持用户指定待搜索文件/函数列表, 并且设置差异性系数阈值来过滤待搜索函数/文件 (默认情况 FI3D 会过滤掉差异性系数为 0 的函数/文件).

(1) 浮点指令记录模块. 本文在 LLVM 17.0.1 中实现了一个函数优化遍 fpids, 该优化遍能够在 LLVM IR 层面

记录每个函数的浮点指令序列. 给定待分析程序 P 和编译选项 C_1 和 C_2, 首先调用 LLVM 中的 Clang 前端生成不同编译优化选项对应的 IR 文件 (当有多个源代码文件时会基于 llvm-link 将其链接为一个 IR 文件). 接着, 使用 LLVM 中的 opt 工具调用优化遍 fpids 解析 IR 文件, 生成不同编译优化选项下每个函数对应的浮点指令序列.

(2) 浮点指令序列差异性计算模块. 本文基于 Python (版本 3.8.10) difflib 模块的 sequenceMatcher 类来计算不同函数浮点指令序列的差异性系数; FI3D 会过滤掉那些差异性系数为零的文件/函数, 并将文件/函数按照差异性系数降序进行排列.

(3) Delta-Debugging 代码搜索定位模块. 本文在 PLiner 中扩展实现了 Delta-Debugging 搜索定位机制, 复用了 PLiner 的语句精度增强变换模块. 此外, 为了提高 FI3D 的可用性, 在 PLiner 中扩展了对多问题文件的精化搜索支持, 解决了 PLiner 不支持单精度浮点类型的问题, 修正了 PLiner 精度增强变换接口在实现上的一些缺陷 (例如 PLiner 对浮点数组精度增强存在缺陷).

图 4 FI3D 实现架构

4.2 研究问题

本文实验主要回答如下 2 个研究问题.

(1) 研究问题 1: 本文工具 FI3D 是否能有效定位触发编译优化结果差异性的代码, 特别是多源问题代码?

(2) 研究问题 2: 本文工具 FI3D 定位触发编译优化结果差异性的代码是否比当前最先进工具 PLiner 更高效?

4.3 实验设计

表 2 给出了实验所使用的测试用例.

表 2 实验程序集

程序	#文件/#函数个数	行数	功能简介
CG.A	1/7	1 066	共轭梯度不规则存取和通信 (A规模)
CG.B	1/7	1 066	共轭梯度不规则存取和通信 (B规模)
MG.A	1/15	1 385	多重网格长、短距离通信 (A规模)
SP.A	13/13	3 440	标量五对角线求解器 (A规模)
SP.B	13/13	3 440	标量五对角线求解器 (B规模)
LU.A	13/13	3 926	下上高斯-赛德尔求解器 (A规模)
sin	1/5	286	正弦计算函数
sinc	1/6	351	辛格计算函数
cos	1/5	295	余弦计算函数
dilog	1/8	340	二重对数计算函数
expint_E1	1/6	482	指数积分E1计算函数
expint_E2	1/7	522	指数积分E2计算函数
clausen	1/6	278	克劳森积分计算函数
Ci	1/8	647	余弦积分计算函数
bessel_J1	1/6	335	贝塞尔J1计算函数
bessel_y0	1/6	307	贝塞尔y0计算函数
arclength	1/3	75	弧长计算函数
dft	1/3	86	离散傅里叶变换
总计	54/138	18 329	18个开源测试用例

表 2 的 4 列分别给出了程序名、程序包含的代码文件/函数个数、程序代码行数和程序功能简介. 我们选取了美国航空航天局 NASA 推出的高性能计算基准测试集 NAS parallel benchmarks (NPB) 中的 6 个应用 (涵盖了

PLiner 使用的 CG 和 SP 应用)、开源 GNU scientific library (GSL) 中能够触发编译优化结果差异性的 10 个函数和 floatsmith 混合精度测试集中能够触发编译优化结果差异性的 2 个程序, 共计 18 329 行代码. 其中 CG、MG、SP 和 LU 来自 NPB 测试集, 分别为共轭梯度不规则存取和通信、多重网格长短距离通信、标量五对角线求解器和下上高斯-赛德尔求解器. 需要特别指出的是 NPB 应用不同规模下的浮点计算行为和结果校验均不同. 函数 sin、sinc、cos、dilog、expint_E1、expint_E2、clausen、Ci、bessel_J1 和 bessel_y0 来自 GSL 数学库中 Specfunc 类函数, 分别为正弦、辛格、余弦、二重对数、指数积分 E1、指数积分 E2、克劳森、余弦积分、贝塞尔 J1 和贝塞尔 y0 计算函数; arclength 和 dft 来自 floatsmith 测试集, 分别用于弧长计算和离散傅里叶变换计算. 实验选取的测试用例均为多文件或多函数程序; PLiner 实验所使用的其他 100 个简单合成用例都是数十行代码的单一函数, FI3D 可直接定位这些简单用例, 考虑到 FI3D 是面向实际数值程序的编译优化结果差异性定位, 为了简洁性, 未在实验中列出这些简单函数的定位情况.

实验基于 LLVM 编译器 (版本 17.0.1) 编译运行测试程序, 触发测试用例结果不一致性的基础编译优化选项和激进编译优化选项分别为-O0 和-O3 -ffast-math. 因为-O0 是最基础的编译选项, 会关闭几乎全部编译优化策略, 能够最直接反映程序计算行为, 而-O3 -ffast-math 是开发人员最常用的激进编译优化选项, 会开启绝大部分浮点激进编译优化 (例如-freciprocal-math 和-fassociative-math 等). -O0 和-O3 -ffast-math 便于触发编译优化结果不一致性来评测 FI3D; PLiner 实验中也是选取了该组编译选项来进行实验.

NPB 程序和 GSL 函数结果不一致性的定义如下.

(1) NPB 程序: NPB 程序运行结束后会将计算结果与预存的标准结果进行校验, 如果相对误差不超过给定阈值则输出 successful, 否则输出 fail. 因此, NPB 程序编译优化结果不一致性是指-O0 选项下编译运行程序结果校验为 successful, 而-O3 -ffast-math 选项下编译运行结果校验为 fail; NPB 程序内部为不同规模定义了对应的输入和标准计算结果. 为了更好地评估 FI3D 的有效性, 与 PLiner 类似, 本文调整了 NPB 程序的相对误差阈值, 使得-O0 和-O3 -ffast-math 两组选项能够产生结果不一致性. 误差阈值的设置是基于二分搜索获取的, 即在误差阈值范围区间内不断二分搜索直到获取的误差阈值能够使得-O0 返回 successful, 而-O3 -ffast-math 返回 fail.

(2) GSL 函数和 floatsmith 函数: 实验为这些函数定义了相对误差阈值, 函数的标准结果采用 long double 模式和-O0 编译运行得出并在函数中预存, 当函数运行结果与标准结果的相对误差超过阈值则返回 fail, 否则返回 successful. 误差阈值的设置策略与 NPB 类似.

本文实验的硬件运行环境为 12 代酷睿 i7 处理器, 拥有 16 个 CPU 核心, 主频 2.5 GHz. 软件环境为 Ubuntu 20.04.6 操作系统.

4.4 实验结果

(1) 研究问题 1: 有效性

表 3 给出了编译优化结果不一致性的定位结果, [$func_1$:$line_1$–$line_2$] 表示问题代码位于函数 $func_1$ 中的 $line_1$ 行到 $line_2$ 行. 第 2 列和第 3 列分别是 PLiner 和 FI3D 的定位结果. FI3D 能够有效定位所有程序触发结果不一致性的代码段, 而 PLiner 则存在 4 个程序 (CG.A、MG.A、dilog 和 bessel_J1) 不能有效精确定位. 不难发现, PLiner 定位失败的程序, 触发结果不一致性的问题代码分布在多个函数内, 这也反映了 PLiner 的二分搜索定位存在的不足和 FI3D 的 Delta-Debugging 搜索的有效性. 对于 PLiner 定位成功的程序, FI3D 的定位结果与其一致, 这也反映出了 FI3D 实现的正确性. 实验发现浮点乘加优化、结合律优化和倒数优化等优化模式容易导致浮点程序结果的不一致性.

研究问题 1: FI3D 能够有效定位 NPB 应用、GSL 数学函数和 floatsmith 测试程序触发编译优化结果不一致性的代码, 对于 PLiner 定位失效的 4 个包含多源问题代码的程序 CG.A、MG.A、dilog 和 bessel_J1, FI3D 能够进行有效定位.

(2) 研究问题 2: 高效性

浮点指令差异性引导对搜索效率的提升主要来自两个方面原因: 通过过滤计算行为未被编译优化改变的文件/函数来缩小问题代码搜索空间和通过将浮点计算行为改变程度大的文件/函数放置在待定位列表的前部来引导定

位优先搜索. 图 5 给出了每个程序中 FI3D 基于函数浮点指令差异性分析过滤掉的函数数目所占程序中总的函数个数的比例; FI3D 平均能够过滤 44.1% 的函数, 过滤比例范围为 7.7%–66.7%, 这能有效缩小后续 Delta-Debugging 的搜索空间.

表 3　编译优化结果不一致性定位结果

Program	Localization results		Time (s)		
	PLiner	FI3D	PLiner	FI3D	Improvement (%)
CG.A	failed	[sparse, sprnvc]	NA	8.3	NA
CG.B	[sparse:741-795]	[sparse:741-795]	369.8	256.4	30.7
MG.A	failed	[resid:499, psinv:1 191]	NA	76.8	NA
SP.A	[exact_solution:44]	[exact_solution:44]	282.8	184.9	34.6
SP.B	[exact_solution:44]	[exact_solution:44]	1 331.3	847.1	36.4
LU.A	[blts:109-197]	[blts:109-197]	546.8	445.1	18.6
sin	[sin_e:254]	[sin_e:254]	2.4	1.9	20.8
sinc	[sin_e:319]	[sin_e:319]	2.7	2.2	18.5
cos	[cos_e:262]	[cos_e:262]	2.0	1.5	25.0
dilog	failed	[xge0:234-245, series_2:195]	NA	2.5	NA
expint_E1	[expint_E1_impl:419]	[expint_E1_impl:419]	3.6	3.1	13.9
expint_E2	[cheb_eval_e:all]	[cheb_eval_e:all]	1.5	1	33.3
clausen	[angle_res_pos_err_e]	[angle_res_pos_err_e]	1.5	0.9	40
Ci	[sin_e:601]	[sin_e:601]	2.9	2.5	13.8
bessel_J1	failed	[cheb_eval_e:216, bessel_J1_e:290]	NA	3.9	NA
bessel_y0	[cos_e:230]	[cos_e:230]	1.9	1.5	21.1
arclength	[do_fun]	[do_fun]	2.6	1.2	53.8
dft	[dft:82]	[dft:82]	3.4	2.9	14.7

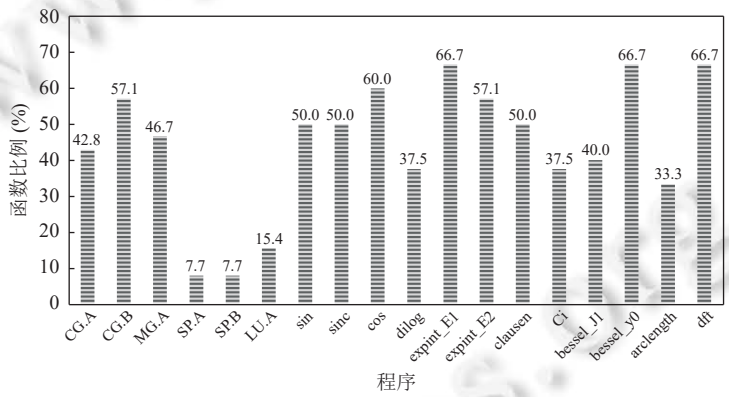

图 5　FI3D 过滤函数比例示意图

表 3 中 Time 列给出了 PLiner 和 FI3D 的定位时间开销, Improvement 列给出了 FI3D 对于 PLiner 的性能提升百分比 (PLiner 定位时间减去 FI3D 定位时间再除以 PLiner 定位时间). 对于 PLiner 定位失效的情况, 时间开销和性能提升比例均设置为 NA. 对于 PLiner 定位成功的 14 个测试用例, FI3D 能够获得平均 26.8% 的性能提升, 性能提升比例范围为 13.8%–53.8%. 这有效反映了浮点指令差异性引导对问题代码搜索定位效率提升的作用.

表 4 给出了每个测试程序中问题函数在所有函数浮点指令差异性系数中的排名, m/n 表示问题函数在所有 n 个函数的浮点指令差异性系数中位于第 m 位, 多个问题函数使用&连接. 所有程序的问题函数均排在函数指令差异性系数序列的前部, 特别是 MG.A、SP.A、SP.B 和 LU.A, 其问题函数在全部 10 余个函数中排名前几位. 排名较差的 CG 和 Ci, 其问题函数也位于差异性系数排名的前半部分. 这充分说明基于函数编译优化前后浮点指令序列的差异性系数对待定位文件/函数进行排序, 能够有效引导 Delta-Debugging 快速定位问题代码.

消融实验: 为了进一步评估浮点指令差异性引导对搜索定位的贡献, 本文进一步实验了将浮点指令差异性引导直接作用于 PLiner, 记为 PLiner$^+$. 图 6 给出了 PLiner$^+$ 相比 PLiner 在定位成功的 14 个测试用例上的性能提升情况, PLiner$^+$ 能够获得平均 21.5% 的性能提升. 除了 Ci 函数外, 其余 13 个测试用例均能获得性能提升, 提升范围为 8.3%–53.8%. 对于 Ci 函数, 尽管浮点指令差异性过滤掉了 3 个冗余函数, 但是考虑到其问题函数本身就位于二分搜索容易定位的位置, PLiner$^+$ 由于指令差异性分析存在一定的开销, 相比 PLiner 有 1.7% 的轻微性能下降. 总体来说, 实验结果显示了浮点指令差异性引导能够有效提升问题代码的搜索定位效率.

表 4　问题函数的浮点指令差异性系数排名情况

问题函数	排名	问题函数	排名
CG.A	3/7&4/7	dilog	2/8&3/8
CG.B	3/7	expint_E1	2/6
MG.A	3/15&4/15	expint_E2	1/7
SP.A	1/13	clausen	2/6
SP.B	1/13	Ci	4/8
LU.A	2/13	bessel_J1	1/5&3/5
sin	2/5	bessel_y0	2/6
sinc	2/6	arclength	1/3
cos	2/5	dft	1/3

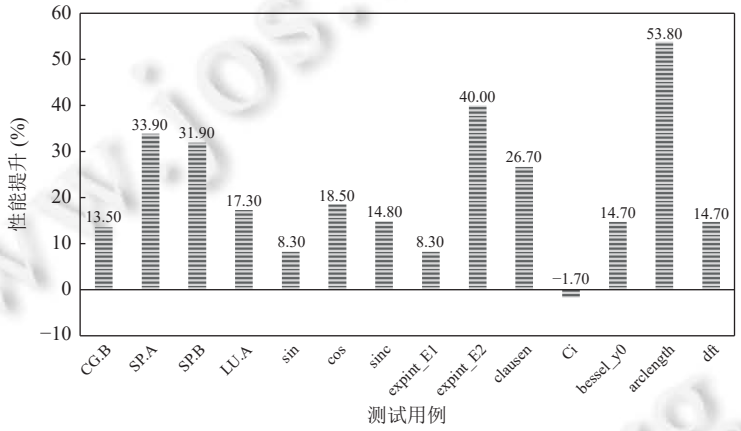

图 6　PLiner$^+$相对 PLiner 的性能提升情况

研究问题 2: FI3D 能够比当前最先进的工具 PLiner 更高效地定位 NPB 应用、GSL 数学函数和 floatsmith 测试程序触发编译优化结果不一致性的代码, 且对于 PLiner 定位成功的 14 个测试用例, FI3D 能够获取平均 26.8% 的性能提升. 将浮点指令差异性引导直接作用于 PLiner, 能够带来平均 21.5% 的性能提升.

4.5　有效性威胁

FI3D 的有效性威胁主要分为外部威胁和内部威胁两类. 外部威胁主要来源是选取的实验对象和对比的编译优化选项有限. 尽管选取了有限的测试用例, NPB 测试集是高性能计算领域应用最广泛的性能评测程序集之一, 而 GSL 数学库则被广泛用于浮点分析工具有效性评测, floatsmith 测试集被广泛用于混合精度优化工具评测, 这些程序被广泛用于编译优化相关工具的评测[4,6,27]. 本文所选取的触发编译优化结果不一致性选项为-O0 和-O3 -ffast-math, 这是因为-O0 会关闭几乎所有编译优化策略, 而-O3 -ffast-math 是开发人员使用最广泛的激进编译优化选项, 开启了绝大部分激进编译优化策略, 这组选项能够更好地触发结果不一致性和评测工具定位问题代码的能力; FI3D 的内部威胁来自工具的实现正确性, 为了控制内部威胁, FI3D 在开发过程中进行了多轮内部正确性测试, 且 FI3D 实验的结果也都经过了人工验证. 另一方面, FI3D 与 PLiner 在 14 个测试应用上定位结果的一致性也反映了 FI3D 实现的正确性.

5 总 结

激进编译优化会改变程序中的浮点计算行为, 导致计算结果不一致性问题. 本文提出了一种浮点指令序列差异性引导的 Delta-Debugging 定位方法 FI3D, 能够高效定位程序中导致编译优化结果不一致性的代码. 一方面, FI3D 基于 Delta-Debugging 中的回溯搜索机制有效支持了多源问题代码的定位. 另一方面, FI3D 利用函数编译优化前后浮点指令序列的差异性过滤掉冗余代码并优先搜索浮点计算行为改变程度高的代码, 提升了问题代码搜索定位效率. 本文基于 LLVM 编译器、Python difflib 模块和编译优化结果不一致性代码定位工具 PLiner 实现了 FI3D, 并基于 NPB 测试集、GSL 数学库和 floatsmith 测试集中的典型用例进行了评测. 相比当前最先进的工具 PLiner, FI3D 能够有效定位 PLiner 失效的 4 个用例, 且对 PLiner 成功定位的 14 个用例 FI3D 获得了平均 26.8% 的性能提升. 实验结果显示了 FI3D 的有效性和高效性. 未来主要工作是将 FI3D 的分析定位思想移植支持其他开源编译器, 例如 GNU 编译器 GCC 等.

References:

[1] Corden MJ, Kreitzer D. Consistency of floating-point results using the intel compiler or why doesn't my application always give the same answer. 2009. https://software.intel.com/sites/default/files/article/164389/fp-consistency-102511.pdf

[2] Sawaya G, Bentley M, Briggs I, Gopalakrishnan G, Ahn DH. FLiT: Cross-platform floating-point result-consistency tester and workload. In: Proc. of the 2017 IEEE Int'l Symp. on Workload Characterization. Seattle: IEEE, 2017. 229–238. [doi: 10.1109/IISWC.2017.8167780]

[3] Bentley M, Briggs I, Gopalakrishnan G, Ahn DH, Laguna I, Lee GL, Jones HE. Multi-level analysis of compiler-induced variability and performance tradeoffs. In: Proc. of the 28th Int'l Symp. on High-performance Parallel and Distributed Computing. Phoenix: ACM, 2019. 61–72. [doi: 10.1145/3307681.3325960]

[4] Guo H, Laguna I, C. Rubio-González. PLiner: Isolating lines of floating-point code for compiler-induced variability. In: Proc. of the 2020 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Atlanta: IEEE, 2020. 1–14. [doi: 10.1109/SC41405.2020.00053]

[5] Laguna I. Varity: Quantifying floating-point variations in HPC systems through randomized testing. In: Proc. of the 2020 IEEE Int'l Parallel and Distributed Processing Symp. NewOrleans: IEEE, 2020. 622–633. [doi: 10.1109/IPDPS47924.2020.00070]

[6] Yu HB, Yi X, Yin BH, Li F, Chen ZB, Huang C. Efficient generation of floating-point inputs for compiler-induced variability. In: Proc. of the 2023 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Macao: IEEE, 2023. 224–235. [doi: 10.1109/SANER56733.2023.00030]

[7] Miao D, Laguna I, Rubio-González C. Expression isolation of compiler-induced numerical inconsistencies in heterogeneous code. In: Proc. of the 38th Int'l Conf. on High Performance Computing. Hamburg: Springer, 2023. 381–401. [doi: 10.1007/978-3-031-32041-5_20]

[8] Panchekha P, Sanchez-Stern A, Wilcox JR, Tatlock Z. Automatically improving accuracy for floating point expressions. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland: ACM, 2015. 1–11. [doi: 10.1145/2737924.2737959]

[9] Yi X, Chen LQ, Mao XG, Ji T. Efficient automated repair of high floating-point errors in numerical libraries. Proc. of the ACM on Programming Languages, 2019, 3(POPL): 56. [doi: 10.1145/3290369]

[10] Zou DM, Gu YC, Shi YF, Wang MZ, Xiong YF, Su ZD. Oracle-free repair synthesis for floating-point programs. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA2): 159. [doi: 10.1145/3563322]

[11] Zeller A. Yesterday, my program worked. Today, it does not. Why? In: Proc. of the 7th European Software Engineering Conf. Held Jointly with the 7th ACM SIGSOFT Symp. on the Foundations of Software Engineering. Toulouse: Springer, 1999. 253–267. [doi: 10.1007/3-540-48166-4_16]

[12] Galassi M, Davies J, Theiler J, Gough B, Jungman G, Alken P, Booth M, Rossi F. GNU Scientific Library Reference Manual. 3rd ed., Godalming: Network Theory Ltd., 2009. 1–572.

[13] Lam MO, Vanderbruggen T, Menon H, Schordan M. Tool integration for source-level mixed precision. In: Proc. of the 3rd Int'l Workshop on Software Correctness for HPC Applications. Denver: IEEE, 2019. 27–35. [doi: 10.1109/CORRECTNESS49594.2019.00009]

[14] Benz F, Hildebrandt A, Hack S. A dynamic program analysis to find floating-point accuracy problems. In: Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation. Beijing: ACM, 2012. 453–462. [doi: 10.1145/2254064.2254118]

[15] Sanchez-Stern A, Panchekha P, Lerner S, Tatlock Z. Finding root causes of floating point error. In: Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Philadelphia: ACM, 2018. 256–269. [doi: 10.1145/3192366.3192411]

[16] Fousse L, Hanrot G, Lefèvre V, Pélissier P, Zimmermann P. MPFR: A multiple-precision binary floating-point library with correct rounding. ACM Trans. on Mathematical Software (TOMS), 2007, 33(2): 13–es. [doi: 10.1145/1236463.1236468]

[17] Chiang WF, Gopalakrishnan G, Rakamaric Z, Solovyev A. Efficient search for inputs causing high floating-point errors. In: Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Orlando: ACM, 2014. 43–52. [doi: 10.1145/2555243.2555265]

[18] Yi X, Chen LQ, Mao XG, Ji T. Efficient global search for inputs triggering high floating-point inaccuracies. In: Proc. of the 24th Asia-Pacific Software Engineering Conf. Nanjing: IEEE, 2017. 11–20. [doi: 10.1109/APSEC.2017.7]

[19] Zou DM, Wang R, Xiong YF, Zhang L, Su ZD, Mei H. A genetic algorithm for detecting significant floating-point inaccuracies. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 529–539. [doi: 10.1109/ICSE.2015.70]

[20] Guo H, Rubio-González C. Efficient generation of error-inducing floating-point inputs via symbolic execution. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul: IEEE, 2020. 1261–1272. [doi: 10.1145/3377811.3380359]

[21] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 209–224.

[22] Kaas RE, Carlin BP, Gelman A, Neal RM. Markov chain Monte Carlo in practice: A roundtable discussion. The American Statistician, 1998, 52(2): 93–100. [doi: 10.1080/00031305.1998.10480547]

[23] Miao D, Laguna I, Rubio-González C. Input range generation for compiler-induced numerical inconsistencies. In: Proc. of the 38th ACM Int'l Conf. on Supercomputing. Kyoto: ACM, 2024. 201–212. [doi: 10.1145/3650200.3656618]

[24] Ren XL, Ho M, Ming J, Lei Y, Li L. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 142–157. [doi: 10.1145/3453483.3454035]

[25] Wang R, Zou DM, He XR, Xiong YF, Zhang L, Huang G. Detecting and fixing precision-specific operations for measuring floating-point errors. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 619–630. [doi: 10.1145/2950290.2950355]

[26] IEEE. IEEE Std 754-2019 IEEE standard for floating-point arithmetic. 2019. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8766229 [doi: 10.1109/IEEESTD.2019.8766229]

[27] Zhou H, Xue JL. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In: Proc. of the 2016 Int'l Symp. on Code Generation and Optimization. Barcelona: ACM, 2016. 59–69. [doi: 10.1145/2854038.2854054]

于恒彪(1990－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为高性能计算, 程序分析.

黄春(1973－), 女, 博士, 研究员, 博士生导师, 主要研究领域为高性能计算, 编译系统.

易昕(1992－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为高性能计算, 程序分析.

尹帮虎(1989－), 男, 博士, 高级工程师, 主要研究领域为高可信软件, 系统建模与仿真.

范小康(1987－), 男, 博士, 副研究员, 主要研究领域为高性能计算, 编译优化.

王戟(1969－), 男, 博士, 研究员, 博士生导师, CCF 会士, 主要研究领域为软件方法学, 软件分析与验证, 并行与分布计算.

唐滔(1984－), 男, 博士, 副研究员, 主要研究领域为并行编程模型, 编译优化.