Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2021-CJ-001

2021-CJ-001

一种经验库制导的浮点程序优化加速策略

Anxiang Xiao, Xiaoshuo Zhang, Enyi Tang, Xin Chen, Linzhang Wang

Technical Report 2021

一种经验库制导的浮点程序优化加速策略

肖安祥[1),2)]　张硕骁[1),2)]　汤恩义[1),2)]　陈　鑫[1)]　王林章[1)]

[1)](南京大学计算机软件新技术国家重点实验室, 江苏南京 210023)
[2)](南京大学软件学院, 江苏南京 210093)

摘　要　为了保障数值程序的准确与高效, 浮点程序自动优化成为了近年来学术界关注的一项新兴技术. 该技术的核心思想是将经典的数值分析理论总结成程序转换规则, 并利用规则将浮点程序以计算过程更为稳定的算法进行自动重写, 从而使数值程序稳定高效且易于维护. 然而, 现有浮点程序自动优化方法的优化效率是其主要瓶颈. 随着越来越多的数值程序转换规则被发现与总结, 自动优化框架的规则库会越来越大, 传统优化方法在规则库中遍历所有优化规则的过程也变得越来越困难. 本文提出了一种经验库制导的浮点程序优化加速策略, 该策略基于浮点误差成因的相似性原理, 将已成功优化浮点程序对应的符号与结构特征抽取出来, 并以散列的方式将优化该程序涉及到的规则序列保存在一个优化经验库中. 当新的浮点数值程序需要进行自动优化时, 本文算法首先计算其符号结构特征与经验库中各记录的相似度. 符号结构相似度较高的记录所对应的规则序列会被优先用于浮点程序重写, 从而得到优化程序. 随着优化程序的增多, 经验库的规模会逐渐增大, 经验库的散列化分区存储设计保证了其检索与匹配效率. 在浮点程序优化基准用例集 FPBench 和开源物理引擎 OpenRelativity 上的实验表明, 在优化后浮点程序精度与传统方法类似的前提下, 本文的加速策略能使平均优化速度相对于传统浮点程序优化方法提升 6.04 倍, 显著提高了浮点数值程序自动优化技术的可用性.

关键词　加速策略；程序优化；浮点数值程序；经验库
中图法分类号　TP391

An Experience-Guided Acceleration Strategy for Floating-Point Optimization

XIAO An-Xiang[1),2)]　ZHANG Shuo-Xiao[1),2)]　TANG En-Yi[1),2)]　CHEN Xin[1)]　WANG Lin-Zhang[1)]

[1)](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)
[2)](Software Institute, Nanjing University, Nanjing 210093, China)

Abstract　To ensure the correctness and efficiency of floating-point programs, automatic optimizing techniques for floating-point programs have been attached attention from academia in recent years. The core idea is to summarize transformation rules from classical numerical analysis theory and rewrite floating-point programs automatically with more stable algorithms, which produces more stable, efficient and maintainable programs. However, efficiency becomes the bottleneck of these techniques. As more and more transformation rules are discovered and summarized, the rule base used by the optimizing framework becomes larger and larger. It is more difficult to scan the whole rule base as usual. In this paper, we propose an experience-guided acceleration strategy for floating-point optimization. Based on the principle of similar causes of floating-point errors, the strategy extracts the corresponding symbolic and structural features of successfully optimized floating-point programs, and saves the rule sequences involved in the optimization of the programs into a hash-based optimization experience database. When optimizing a floating-point program, the optimizer first calculates the similarity between symbolic and structural features of the program and that of records in the experience base. Rule sequences associated with records having higher similarity are preferred to be used in rewriting and optimizing the program. With more and more programs to be optimized, the experience base increases gradually and a hashed-partition design ensures the efficiency of record-searching. The above approach has been evaluated on the FPBench benchmark and the open source engine OpenRelativity. The results show that compared to the traditional optimizing approach, it can achieve a 6.04x speedup in average. In addition, it improves accuracy for floating-point programs effectively as well as the traditional optimizing approach does. Therefore, our approach improves the usability of automatic optimizing techniques significantly.

Key words　acceleration strategy; program optimization; floating point programs; experience base

收稿日期：2022-01-05；修改日期：2022-01-05 本课题得到国家自然科学基金 (62172210, 61772260) 资助. **肖安祥**, 硕士, 主要研究领域为浮点程序优化与软件测试. E-mail: anxiang.xiao@foxmail.com. **张硕骁**, 博士研究生，主要研究领域为浮点程序优化与区块链. E-mail:732376572@qq.com. **汤恩义**(**通信作者**), 博士, 副教授, 主要研究领域为软件工程, 新型软件测试方法, 程序分析方法. E-mail: eytang@nju.edu.cn. **陈鑫**, 博士, 副教授, 主要研究领域为软件工程, 软件测试, 验证技术. E-mail: chenxin@nju.edu.cn. **王林章**, 博士, 教授, 主要研究领域为为模型驱动的软件测试与验证, 安全测试, 软件测试自动化. E-mail: lzwang@nju.edu.cn.

1 引言

许多应用于关键领域的软件系统常常结合领域应用的特点而伴随有大量的浮点数值计算. 例如在智能计算领域, 神经网络的训练过程和计算过程本身就是一个复杂的数值计算算法. 这些数值计算算法的实现往往要求既准确又高效: 一方面, 数值误差累积所导致的程序错误可能会引起灾难性后果[1–6], 这要求浮点数值程序在数值误差累积上得到有效控制; 另一方面, 很多算法要求对应的浮点程序尽可能高效. 如果使用更大内存来提高程序中各浮点值的表示精度, 数值程序的计算量会以几十倍到几百倍的程度显著增加[7]. 例如, 它会使得本来就需要训练数个月的深度学习程序因需要增加数百倍的训练时间而变得不可行. 在业界, 为了保持浮点程序的准确高效, 往往会在限制浮点数表示精度的同时, 修改程序的数值算法使之成为一个稳定算法.

浮点数值程序自动优化是一项新兴静态分析与转化技术[8,9], 该技术将数值程序的定义域按照数值特性自动划分成子空间, 并在各个子空间内将程序执行流程替换成高效、稳定的数值算法. 其转化核心在于将数值分析领域[10,11] 的知识总结成程序转换规则, 并将它们组织成一个规则库. 通过遍历规则库中的规则, 可以自动将浮点数值程序重写为更稳定的算法, 从而减少其误差累积, 最终保障程序的正确性. 由于这一过程并未改变浮点程序中各浮点值的表示精度, 优化后的程序仍然能够保持高效. 另外, 由于优化过程完全自动进行, 使用这一技术的软件开发人员仅仅需要维护不需要考虑算法稳定性的数值程序. 这进一步降低了数值程序维护难度, 提高了浮点数值程序的可维护性.

现有浮点程序自动优化方法的优化效率是其主要瓶颈. 以相关工作[8] 的实验数据为例, 平均代码行数为 5.07 行的 28 个用例, 在双核 2.3 GHz Intel Core i5 平台上的平均优化时间为 68.42 秒. 一方面, 由于浮点程序自动优化算法关于代码规模的时间复杂度高于线性关系, 因而在大规模浮点数值程序上直接应用已有的优化方法十分困难; 另一方面, 优化一个程序的过程往往需要多次搜索规则并反复对多个步骤尝试进行算法重写, 而工业级的浮点程序自动优化技术往往需要更为庞大的程序转换规则库, 这将会进一步增加规则的搜索时间, 加剧该优化效率问题.

为了解决这一问题, 本文提出了一种经验库制导的浮点程序优化加速策略. 该策略基于浮点误差成因的相似性原理. 其核心思想在于: 计算过程相似浮点程序的误差累积成因也往往是类似的, 因此在很大概率上可以用类似的规则序列来进行重写优化, 而不必在规则库中重新搜索. 举例来说: 浮点程序 $\sqrt{x+1}-\sqrt{x}$ 可以通过规则 $a-b \rightarrow (a^2-b^2)/(a+b)$ 转换成优化形式 $1/(\sqrt{x+1}+\sqrt{x})$, 与之相似的浮点程序 $\sqrt{x+1}-\sqrt{x}+x$ 同样也可以使用类似的规则进行优化, 得到优化形式 $1/(\sqrt{x+1}+\sqrt{x})+x$. 因此后一程序在自动优化时可以优先尝试的前一程序的优化规则, 而不必在规则库中查找, 以提高优化效率.

为了实现这一原理, 本文的加速策略设计了一个基于散列化分区存储的经验库. 该经验库将已成功优化浮点程序的符号特结构特征以及优化规则序列记录下来. 当需要对浮点数值程序进行优化时, 加速策略会在经验库中搜索相似程序在优化过程中使用过的的规则序列, 并利用它们来制导规则的选取过程, 从而避免了规则库的多次搜索以及对重写算法反复进行评估, 达到优化加速的目的. 随着优化程序的增多, 经验库的规模会逐渐增大, 散列化分区存储设计保证了其检索与匹配效率能够适应大规模经验库的要求.

我们在 FPBench[12] 浮点程序基准用例集上部署了实验, 以评估本文加速策略的有效性. 实验结果表明, 本文加速策略在经验库仅积累 15 条优化经验的条件下, 平均优化速度达到了传统浮点程序优化方法[8] 的 6.04 倍. 与此同时, 实验结果显示我们的加速方法在误差优化效果上与已有的低速优化基本一致, 均能显著降低数值程序的累积误差, 保证程序准确高效. 我们进一步在 MIT 开发的相对论场景物理模拟引擎 OpenRelativity 及其依赖库上实施了评估实验, 实验结果表明本文算法可用于大型实际用例, 且能成功实现现有技术无法完成的数值优化.

本文的主要贡献如下:

1. 本文提出了一种经验库制导的浮点数值程序优化加速策略. 该策略基于浮点误差成因相似性原理, 利用经验库中的优化经验制导并加速程序优化过程.

2. 本文设计并实现了经验库制导的浮点程序优化加速策略. 它主要由三部分组成: 基于分区存储结构的散列化经验库, 程序相似度分析, 以及经验库制导优化算法.

3. 本文在 FPBench 浮点程序基准用例集和 OpenRelativity 开源物理引擎上开展了评估实验. 实验结果表明, 在优化后程序精度效率与传统方法类似的情况下, 本文加速策略的平均优化速度可以达到传统浮点程序优化方法的 6.04 倍, 优化效果具有有效性.

2 浮点程序优化

2.1 浮点数值与误差表示

在计算机内部, 浮点数的表示通常遵循 IEEE754 标准[13]. 它由三部分组成, 分别是符号位 (S), 指数域 (E) 和尾数域 (M). 浮点数值可以由公式 $v=(-1)^S\times e^E\times M$ 计算得到. 其中, 符号位 S 指示了浮点数的正负性. 当 S=0 时, 表示浮点数值为正数, 而当 S=1 时表示浮点数值为负数. 指数域 E 决定了该浮点值的数量级浮动范围. 尾数域 M 表示的该浮点值的有效数字, 可用 $b_0.b_1b_2...b_m$ 来表示, $b_i(0\leqslant i\leqslant m)$ 为其中的一个比特位.IEEE 标准定义了不同精度的浮点数, 常见的有 32 位单精度浮点数和 64 位双精度浮点数. 有限的精度表示不可避免地引入了舍入误差 (roundoff error). 如果浮点程序的实现不当, 舍入误差会被放大[14], 导致最终计算结果与预期结果偏离甚远.

在判断一个浮点程序是否存在误差时, 我们需要借助一些误差度量方法. 常见的误差度量方法有三种: 绝对误差, 相对误差, 以及 ulp 误差. 当 $f(x)$ 为浮点程序的理论准确值,$f'(x)$ 为浮点程序的实际输出值时, 绝对误差的定义为 $E_{abs}(f(x),f'(x))=|f'(x)-f(x)|$; 相对误差的定义为 $E_{rel}(f(x),f'(x))=|(f'(x)-f(x))/f(x)|$. 绝对误差和相对误差在数学上的定义清晰直接, 但在实践上为了更好地处理上溢 (inf), 下溢 (-inf), 除零异常 (nan) 等特殊值, 浮点程序优化框架定义了 ulp 误差[8,10,14]. 它的定义如下:

$$E_{ulp}(f(x),f'(x))=log_2|\{z\in FP|\ l\leqslant z\leqslant u\}| \quad (1)$$

$$l=min(f(x),f'(x)),\ u=max(f(x),f'(x)) \quad (2)$$

式中 FP 代表所有浮点数的集合,$z\in FP$ 表明 z 是一个浮点数. 这里的 ulp 误差从直观上来说即为理论准确值 $f(x)$ 与实际输出值 $f'(x)$ 之间的浮点数个数差距所对应的二进制数量级. 简单来说,ulp 误差计算了浮点程序的理论准确值与实际输出值有多少个比特位不同, 即实际程序输出结果中有多少比特位是不准确的. 对于 32 位单精度浮点程序来说,ulp 误差的范围为 [0,32], 64 位双精度浮点程序的 ulp 误差范围为 [0,64]. 误差为 0 表示没有误差, 数值越大则误差越大.

2.2 浮点优化框架

自动化浮点程序优化方法[8,9] 的基本原理是将数值分析领域的知识与理论总结成程序转换规则, 并利用规则对浮点程序进行重写, 得到一个与原始程序在理论上等价但误差积累更少的优化程序. 最基本的程序转换规则由两部分组成: 匹配模式和生成模式. 匹配模式定义了运用这条规则的浮点程序须满足的形式要求, 生成模式定义了转换后浮点程序的生成方式. 为了描述上的清晰, 在本文中我们以

匹配模式 → 生成模式

的形式来表示一条程序转换规则. 这里我们以规则 $a-b\rightarrow(a^2-b^2)/(a+b)$ 为例,$a-b$ 是规则的匹配模式, 其中 a 和 b 是占位符, 它不一定仅指代某个变量, 也可以是一个程序段或一个函数. 一个程序如果能通过变量代换的方式匹配规则输入, 就可以使用这条规则. 例如 $sin(x)-cos(y)$ 可以匹配 $a-b$. 当 $sin(x)$ 匹配 a,$cos(y)$ 匹配 b 后, 可以利用对应的生成规则做变量代换, 将它重写成 $(sin^2x-cos^2y)/(sin(x)+cos(y))$. 重写后的程序和原始程序在数学理论上等价, 但由于采用了不同的计算过程, 引入的误差累积也会不同.

许多浮点算法在定义域的某些子空间能够保持稳定, 而在另一些子空间上误差积累较为严重. 对于这一问题, 优化方法会通过静态程序分析推导浮点程序的各个子空间, 并将同一浮点程序的各子空间分别优化成不同的稳定算法, 从而保障在运行时的各种不同程序输入均能交由在该子空间运行稳定的数值算法进行处理.

各个定义域子空间内部的浮点优化过程是一个迭代过程. 当优化框架优化一个浮点数值程序时, 会反复遍历与匹配规则库中的程序转换规则, 并利用匹配上的规则对程序进行重写. 对于重写后的每一个程序, 优化框架会进一步判断其浮点误差累积是否低于当前程序, 并以误差累积较低的程序来替换当前程序. 然后优化框架会进一步在重写后的程序上继续遍历与匹配程序转换规则, 持续这一迭代过程, 从而不断搜索找到误差更小的高效重写程序. 对于规则数量较小的规则库来说, 这一迭代过程能够取得很好的优化效果. 然而随着规则库规模的不

断扩大, 以及待优化数值程序复杂性的加剧, 该方法的优化效率会成为其主要瓶颈.

举例来说, 当优化框架优化一元二次方程的求解程序 $(-b+\sqrt{b^2-4ac})/(2a)$ 时, 会遍历规则库并找到所有与其匹配的规则, 比如规则 $a+b \rightarrow (a^2-b^2)/(a-b)$, $x \rightarrow log(e^x)$ 等. 使用这些规则可以变换生成不同的重写程序. 对这一过程进行反复迭代后, 会得到一个数量庞大的重写程序集. 最后, 优化算法会分析该浮点程序的定义域, 将定义域划分成子空间 (或者在一元自变量时直接划分成子区间), 并为每个子空间指定误差最小的浮点算法程序段. 通过将这些子空间和程序段以分段函数的形式进行整合, 实现定义域的每个子空间均能保持数值稳定. 该例子优化后含有三个子区间, 输出程序为:

$$\begin{cases} \frac{-b+\sqrt{b^2-4ac}}{2a} & -10^{154} < b < 0 \\ \frac{2c}{-b-\sqrt{b^2-4ac}} & 0 \leqslant b < 10^{154} \\ -\frac{c}{b} & |b| \geqslant 10^{154} \end{cases} \tag{3}$$

3 加速算法示例

在本章, 我们将会用一个具体的例子来阐述本文加速策略如何对优化过程进行加速. 在优化之前, 假定经验库中已存储了部分程序的优化经验, 其中包括了程序 P : $(-b+\sqrt{b^2-4ac})/(2a)$ 的优化经验, 如表 1所示.

表 1 程序 $(-b+\sqrt{b^2-4ac})/(2a)$ 的优化经验

待优化程序	规则序列	优化后程序
$\frac{-b+\sqrt{b^2-4ac}}{2a}$	$a+b=\frac{a^2-b^2}{a-b}$	$\frac{2c}{-b-\sqrt{b^2-4ac}}$
$\frac{-b+\sqrt{b^2-4ac}}{2a}$	$\sqrt{1+x} \approx 1+\frac{1}{2}x$	$-\frac{c}{b}$

假定现在需要优化另一个程序 Q: $(-b-\sqrt{b^2-4ac})/(2a)$. 在优化之前, 加速策略首先会查找经验库, 看是否存在相似程序的优化经验. 如果存在, 则利用这些经验来指导并加速优化过程. 为了提高查询的效率, 经验库被设计成一种类似于散列表的分区存储形式. 它由多个分区组成, 每个分区会维护一个散列区间, 分别为 [0,50),[50,100),[100,150), ⋯, 以此类推. 查询之前, 首先根据待优化程序的符号结构等信息计算出一个散列值 (具体计算方式将在 4.2 中详细介绍), 然后根据散列值来确定各分区的查找优先级. 分区的查找优先级由程序散列值和各分区 (对应散列区间) 的距离决定. 两者的距离越近, 则对应分区的查找优先级越高. 在这个例子中, 程序 Q 的散列值为 981, 落入 20 号分区的散列区间 [950,1000), 因此 20 号分区具备最高的查找优先级. 另一方面, 优化经验的存储采用了类似的机制. 一个程序的优化经验会存储在离该程序最近的分区内. 比如对于程序 P 而言, 它的程序特征值为 975, 同样落入 20 号分区的散列区间 [950,1000), 因此它的优化经验被存储于分区 20. 类似于散列表的原理, 当不存在散列值冲突时, 经验库的查询会以最高效的状态进行; 即使散列值发生冲突, 造成多种不同类型的经验被保存在了同一个分区. 后续优化仍然可以正常执行, 不同之处在于进一步的分区内查找会额外多匹配一次该分区内的其它经验. 在进一步进行分区内查找时, 待优化程序会和分区内各优化经验的前件程序匹配相似度, 若发现存在相似性, 则取出其该优化经验对应的优化后件 (即优化过程) 进行进一步尝试. 可以发现, 程序 P 和 Q 是相似的 (相似度计算方式将在 4.3 中详细介绍), 因此 P 的优化经验被取出并用于 Q 的优化. 本文加速策略沿用了传统优化算法的迭代优化框架. 在每轮迭代中, 加速策略会搜索和匹配优化后件中的程序转换规则, 并用这些规则重写当前程序, 以生成误差累积更小的程序. 与传统优化算法不同的是, 加速策略会直接使用从经验库检索得到的规则序列, 而不需要遍历规则库. 在这个例子中, 规则 $\sqrt{1+x} \approx 1+x/2$ 被用于重写程序 Q, 生成了程序 $-c/b$. 然而在使用规则 $a+b=(a^2-b^2)/(a-b)$ 时, 发现程序 Q 并不能匹配这条规则的匹配模式, 因而无法直接使用规则对程序进行重写. 遇到这种情况时, 加速策略会从规则库中找到与当前规则相似度最高的且可匹配规则. 在这个例子中, 加速策略找到了替代规则 $a-b=(a^2-b^2)/(a+b)$, 对程序 Q 做转换得到 $(2c)/(-b+\sqrt{b^2-4ac})$. 最终, 本文加速后的优化策略通过对定义域划分子空间而得到最终的优化结果.

$$\begin{cases} \frac{-b-\sqrt{b^2-4ac}}{2a} & 0 \leqslant b < 10^{154} \\ \frac{2c}{-b+\sqrt{b^2-4ac}} & -10^{154} < b < 0 \\ -\frac{b}{a}+\frac{c}{b} & |b| \geqslant 10^{154} \end{cases} \tag{4}$$

回顾程序 Q 的优化过程可以发现, 相比于传统优化算法, 加速算法避免了对规则库进行遍历匹配. 随着越来越多规则被总结和发现, 规则库的规模会

日益庞大, 而加速算法的优化效率依旧能保持高效, 这显著提高了浮点误差自动优化技术的可用性. 同时, 经验库的设计采用了散列化分区存储结构. 在大规模数据下, 经验库依旧能保持较好的查询性能, 而不会成为优化框架的性能瓶颈. 本章的示例重点描述了加速策略的优化主流程, 并隐藏了部分算法细节, 这部分内容将在第四章进行介绍.

4 优化加速策略

4.1 概述

本文提出了一种经验库制导的浮点程序优化加速策略, 其优化流程如图 1所示. 对于一个给定的输入程序, 优化加速策略首先会判断是否能使用已有的优化经验来加速当优化过程, 然后再决定使用哪种优化策略. 优化开始前, 会尝试从优化经验库中检索与输入程序相似度较高的程序, 并取出优化该程序时使用过的规则序列. 如果检索成功, 则使用这些规则序列加速优化过程. 反之, 如果检索失败, 加速策略则会退化成相对低效的贪心策略 (具体使用哪种策略取决于要加速的对象, 其基本优化思路可参见第二章). 待优化结束后, 如果没有用到加速策略, 说明当前经验库缺乏这类程序的优化经验, 此时会将当前程序的优化规则序列存储在经验库中, 以便将来遇到相似程序时能加速其优化过程. 随着越来越多的程序被优化, 经验库会累积大量优化经验. 此时, 经验库的搜索成功率会逐渐提高, 更多程序的优化过程将会得到加速.

优化加速策略需要解决三个关键问题. 第一, 随着经验库规模的不断增长, 如何保证检索效率不会成为性能瓶颈. 第二, 如何准确衡量程序相似度. 第三, 如何利用优化经验来加速优化过程. 我们将在下面小节中对其进行一一介绍, 并在本节最后对该算法的可并行性进行探讨.

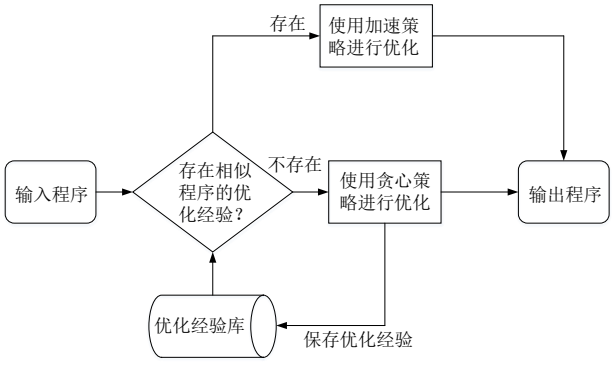

图 1 优化加速策略的优化流程

4.2 优化经验库

为了解决经验库在大规模数据下的查询效率问题, 本文将经验库设计成一种分区存储结构. 如图 2所示, 经验库以分区的形式存储, 每个分区内部只存储一部分优化经验. 此外每个分区会维护一个散列区间, 当对经验库发起查询请求时, 会先根据输入程序的符号结构特征计算出一个散列值, 并根据散列值确定各个存储分区的查找优先级. 查找算法会优先查找优先级最高的分区, 如发生散列冲突, 会导致优先级最高的分区内保存着一些用不了的优化经验, 额外增加一些分区内的匹配时间. 在分区内匹配时, 通过分析存储于该分区的程序和待优化程序的相似程度, 取出相似程序的优化经验来指导并加速当前程序的优化过程.

在我们的经验库中, 每一条优化经验由前件和后件两部分组成. 其前件是一个以前被成功优化过的数值程序, 该程序在被整体保存到经验库之前会进行标准化, 去除和数值无关的代码, 对数值运算过程中涉及到的变量进行标准化重命名, 使其数值运算的特性得到集中保留. 其后件记录了该程序的优化步骤, 即优化该程序用到的程序转换规则序列. 在优化时, 我们首先匹配待优化程序和经验的前件程序是否匹配, 如果能匹配上, 则优先尝试应用经验后件的转换规则对代优化程序进行转换和重写, 即可得到对应的重写程序.

图 2 优化经验库的散列化分区存储结构

在散列化分区查找时, 查找优先级和输入程序的散列值有关. 其计算公式如下:

$$\text{散列值} = \sum_{\text{符号}\in\text{程序}} \text{符号编码} \times \text{位置编码} \quad (5)$$

程序的符号是构成程序的基本元素, 有三种类别, 分别是输入参数, 常量以及运算符. 比如, 程序 $f(x) = x + 1$ 共有 3 个符号, 其中 x 是输入参数,+ 是运算符,1 是常量. 符号的编码值根据其类型的不同采用了不同的编码规则. 输入参数的编码值由它在参数列表中的声明顺序决定. 参数列表中第一个参数的编码值为 1, 第二个参数的编码值为 2, 以此类推. 常量的编码值等于常量本身. 运算符的编码值通过查询编码表 [1] 得到. 编码表为每种运算符都预先定义

[1]符号编码表参见https://software.nju.edu.cn/njdx/DFS//file/2020/09/11/

了一个编码值. 其编码的基本原则是: 如果两个运算符的语义是相似的, 则它们的编码值相近. 反之, 如果两个运算符的语义差异较大, 则它们的编码值会有显著差异. 这种设计使得符号语义相似的程序能计算得到相似的散列值. 在计算散列值时, 还需要计算符号的位置编码值, 其编码函数的定义如下:

$$\text{位置编码值} = \text{程序高度} - \text{符号深度} + 1 \quad (6)$$

每个程序都能转换成一个与之等价的二叉树形式, 其中非叶子用于表示运算符, 叶子节点用于表示操作数, 比如输入参数或者常量. 程序高度等于其对应二叉树的高度 (树根的高度为 0), 而符号深度等于该符号对应节点在二叉树中和树根的距离 (树根的深度为 0). 直观来看, 散列值是对程序符号结构特征的一种抽象表示. 其中符号编码抽象了程序的符号特征, 而位置编码抽象了程序的结构特征. 一般情况下, 符号结构特征越相似的程序, 它们计算的散列值会越接近. 以程序 P_1 : $\sqrt{x+1}-\sqrt{x}$ 和 P_2 : $\sqrt[3]{x+1}-\sqrt[3]{x}$ 为例. 根据符号编码的编码定义, 上述符号的编码值分别为 $E(x)=1, E(+)=10, E(-)=11, E(\sqrt{})=60, E(\sqrt[3]{})=61$. 可以看出, 在预定义的符号编码值中, 语义接近的 $+$ 和 $-$, $\sqrt{}$ 和 $\sqrt[3]{}$ 有着较为接近的编码值. 再由散列值计算公式 5,P_1 的散列值等于 $4\times E(-)+3\times E(\sqrt{})+3\times E(\sqrt{})+2\times E(+)+2\times E(x)+1\times E(x)+1\times E(1)=428$. 类似地,$P_2$ 的散列值为 434. 可以看出, 它们的散列值是比较接近的. 而对于一个在符号和结构上与 P_1, P_2 均不相似的程序, 例如 P_3 : $(x-y)^2$, 其散列值为 190, 和 P_1, P_2 的散列值相差较大.

值得注意的是散列值可能会因为发生冲突而造成检索效率下降, 当我们的经验库中保存了 n 条经验数据, 散列值编码由 d 比特的数据位构成, 则理论最大编码值为 2^d. 这里我们假设经验库中各优化经验及其编码相互独立, 并将每一条经验的编码分布近似看作均匀分布的随机编码, 并在此基础上计算冲突概率 $p(n)=1-q(n,d)$. $q(n,d)$ 是不发生冲突的概率, 即在 d 比特编码下, n 条优化经验的散列值均互不相同的概率. 由于理论上存在 2^d 条编码数据, 在编码分布相互独立时, 某一条优化经验可能成为一个特定编码的概率为 $1/2^d$, 因而该经验与另外 $n-1$ 条经验散列值不同的概率为 $1-\frac{n-1}{2^d}$, 经过递归运算, 我们有:

$$\begin{aligned} q(n,d) &= \left(1-\frac{n-1}{2^d}\right)\cdot q(n-1,d) \\ &= \left(1-\frac{n-1}{2^d}\right)\cdot\left(1-\frac{n-2}{2^d}\right)\cdots\left(1-\frac{1}{2^d}\right) \end{aligned}$$

当 $d=64$ 时, 即使经验库规模 n 达到 10^9 量级, 冲突概率 $p(n)<0.0267$, 因此, 本文方法的冲突概率很低.

经验库被分成了多个分区, 每个分区对应一个散列区间, 分别为 [0, 50), [50, 100), [100, 150), …, 以此类推. 每个散列区间的中点值被视作该分区的唯一标识符. 例如,1 号分区的标识符为 25,2 号分区的标识符为 75, 以此类推. 在存储已优化浮点程序的优化经验时, 首先会根据该程序的符号结构特征计算出其散列值, 然后将相关数据存储于最近的分区内. 程序和分区的距离等于 |程序散列值 − 分区唯一标识符|. 由于散列值在一定程度上是对程序符号结构特征的抽象, 处于相同分区的程序, 其相似度往往会高于不同分区程序的相似度. 类似地, 在查找过程中分区的查找优先级由其标识符与待优化程序的距离决定. 两者的距离越小, 则分区的查找优先级越高. 如果分区的标识符接近于散列值, 其存有相似程序优化经验的概率更高. 查找经验库时, 只需在优先级最高的分区进行查找, 而无需遍历整个经验库, 最终通过程序相似度分析即可找到相似程序的优化记录. 我们在不同的经验库规模下验证了经验查询时间, 结果如表 2所示, 可以看到查询时间实际受经验库规模的影响很小. 出现这一现象的主要原因在于:1) 理论上查询散列表的时间复杂度是常数复杂度 O(C), 应当不受经验库规模的影响;2) 实践中散列表可能会发生散列值冲突, 随着经验库规模的增大, 发生冲突的概率会略有增加, 根据前面的推导可知, 即使经验库规模 n 达到 10^9 量级, 其冲突概率 $p(n)<0.0267$, 完全能够满足算法要求.

表 2 不同经验库规模的查询时间

经验库规模	100	1000	10000
查询时间	2.18s	2.37s	2.79s

4.3 程序相似度分析

在查找经验库的某个分区时, 我们需要找到相似程度的优化经验, 此时需要计算分区内各优化经验的前件程序和待优化程序之间的相似程度. 本文从符号特征和结构特征两个维度出发, 设计了下列

程序相似度计算策略:

$$\text{程序相似度} = 0.5\times(\text{符号相似度}+\text{结构相似度}) \quad (7)$$

符号相似度的定义如下:

$$S_{sym}(P_1, P_2) = \frac{2\times|shared|}{|syms(P_1)|+|syms(P_2)|} \quad (8)$$

$$shared = \{s|s \in syms(P_1) \wedge s \in syms(P_2)\} \quad (9)$$

其中, $syms(p)$ 表示程序 p 的符号集, $|s|$ 表示集合 s 的元素个数. 根据定义, 符号相似度的取值范围为 [0,1]. 数值越高, 则表示两个程序在符号这个维度上更相似. 符号相似度的基本思想是: 若程序共享的符号数量占总符号数量的比例越高, 则它们的相似度越高. 以计算程序 $P_1: e^x-1$ 和 $P_2: e^{ax}-1$ 的符号相似度为例, 它们的共享符号集为 $\{e, x, -, 1\}$. 不同的点在于, 程序 P_2 还多了一个符号 a. 根据定义, 两者的符号相似度为 $2\times4/(4+5)=0.89$. 可以看出, 符号相似度是计算次序不敏感的. 即在计算程序相似度时, 我们并不考虑符号所处的位置, 而只考虑它是否存在. 为了弥补这一点, 我们引入了结构相似度. 它的定义如下:

$$S_{struct}(P_1, P_2) = \frac{|es|}{min(|syms(P_1)|, |syms(P_2)|)} \quad (10)$$

$$es = \{s_1|eq(s_1, s_2) \wedge s_1 \in syms(P_1) \wedge s_2 \in syms(P_2)\} \quad (11)$$

根据定义, 结构相似性的取值范围为 [0,1]. 数值越高, 则表示两个程序在结构上越相似. 在这个定义中,eq 函数用于判断两个符号是否相等. 符号相等需要满足两个条件. 其一, 它们处在两个程序的相同位置. 在计算符号位置时, 需要先将程序转成等价的二叉树形式, 根节点到符号结点的路径唯一标识了该符号的位置. 例如, 程序 $x+y$ 的 $+$ 和程序 $a\times b$ 的 $\times$ 都位于各自程序二叉树形式的根节点处, 因此它们所处的位置相同. 其二, 两个符号处在同一分组. 我们预先将常用符号进行了分组 [2]. 分组的基本原则是将一些语义上相似的符号放在同一分组, 而语义差异明显的符号放在不同分组. 例如, 符号 $+$ 和 $-$ 放在同一分组, 而符号 sin 和 cos 放在另一分组. 结构相似度计算的是相等符号对占总符号数的比例. 所占比例越高, 说明两个程序在结构这个维度上越相似. 以计算程序 $P_3: (1-cos(x))/(x\times x)$ 和 $P_4: (1-e^x)/(y\times z)$ 的结构相似度为例, 共有 7 个相等符号对, 分别为 $\{1,1\}, \{-,-\}, \{x,x\}, \{/,/\}, \{x,y\}, \{\times,\times\}, \{x,z\}$. 根据定义, 两者的结构相似度为 $7/min(8,8)=0.875$. 可以看出, 结构相似度是计算次序敏感的, 和计算次序不敏感的符号相似度形成了互补关系.

[2]符号分组定义参见https://software.nju.edu.cn/njdx/DFS//file/2020/09/11/20200911192211454pkdkc9.pdf

4.4 优化加速算法

本文提出了一种经验库制导的浮点程序优化加速算法, 其详细定义参见算法 1. 其输入通常是一个带有累积误差的浮点数值程序, 而输出是一个与输入程序语义等价且浮点误差累积更小的程序. 除输入参数外, 算法还需要依赖两个全局变量, 分别是优化经验库 *OptExp* 和程序候选集 *Candidates*. 优化经验库的作用是存储过往浮点数值程序的优化经验, 并用于指导并加速当前程序的优化过程 (参见 4.2). 程序候选集的作用是收集程序转换过程中生成的中间程序, 用于结束阶段的区间划分和优化结果合并. 算法定义主要包含了两个函数, *Optimize* 和 *FastOptimize*.

Optimize 函数是整个优化过程的入口, 定义了优化的主流程. 在优化开始阶段, 首先从优化经验库检索相似程序的优化经验 (第 2 行). 如果检索成功, 则使用加速策略进行优化 (第 4-7 行). 优化经验包含了一系列优化规则序列, 它可以指导优化算法快速选取合适的程序转化规则对程序进行重写, 从而生成误差累积更小的程序. 这些程序会被加入到候选程序集 *Candidates* 中, 用于最终的区间划分和结果合并. 如果优化经验检索失败, 优化算法会退化成传统的贪心策略 (第 9-11 行). 优化成功后, 相关优化经验会被保存, 从而帮助经验库累积更多优化经验, 以加速后续优化过程.

FastOptimize 函数的输入是一个程序和一个规则序列. 它会优先使用规则序列中的规则重写程序, 而不是重新遍历规则库, 从而达到快速优化的目的. 规则序列使用链表进行存储, 通过递归调用的方式依次取出每条规则 (第 22 行), 将其用于重写程序. 遍历完规则链表后, 将重写生成的新程序加入到候选程序集中 (第 24 行). 在应用规则的过程中, 有可能会出现程序与规则的输入模式不匹配的情况 (第 18 行). 此时会从规则库中寻找一个与该规则相似度最高且与当前程序相匹配的规则 (第 19 行), 用它来重写当前程序 (第 21 行).

算法 1 经验库制导的浮点误差优化加速算法

输入: 带累积误差的浮点数值程序
全局变量: 优化经验库 $OptExp$
全局变量: 候选程序集 $Candidates$
输出: 与输入程序等价且浮点误差累积更小的程序

```
1:  function Optimize(p)
2:      ruleSeqList ← OptExp.getRuleSequences(p)
3:      if ruleSeqList ≠ null then    //使用加速策略
4:          for ruleSeq in ruleSeqList do
5:              FastOptimize(p, ruleSeq)
6:          end for
7:          return Candidates.combine()
8:      else   //退化成传统的贪心策略
9:          experience ← GreedyOptimize()
10:         OptExp.save(experience)
11:         return experience.result
12:     end if
13: end function
14:
15: function FastOptimize(program, ruleSeq)
16:     if ruleSeq ≠ null then
17:         rule ← ruleSeq.value
18:         if program 和 rule 不匹配 then
19:             rule ← getSimilarRule(program, rule)
20:         end if
21:         next ← transform(program, rule)
22:         FastOptimize(next, ruleSeq.next)
23:     else
24:         Candidates.add(program)
25:     end if
26: end function
```

4.5 并行优化探讨

随着多核平台的普及, 利用多核架构来进一步提高优化速度是一种有效的加速方式. 本文经验库制导的加速优化策略是一个可并行算法, 其可并行任务按照粒度可以分为三个层次: 不同优化经验的并行利用、经验内部实施过程的并行优化、以及重写规则执行过程内的并行优化.

不同优化经验的并行利用是指在对经验库做相似度匹配时, 多条经验的前件程序可以并行匹配, 对应多条经验匹配成功后后件处理也可以并行执行. 该粒度下的并行性瓶颈取决于实际需要匹配的经验数量, 最终对应于有效优化经验在算法实施中的优先级. 从表面上看, 并行算法可以应用到经验库的每一条经验上, 似乎表明该粒度下的并行瓶颈取决于经验总数量, 然而实际并不是这样. 尝试大量低优先级的无效经验会消耗大量的非必要系统资源, 尽管此时看起来 CPU 满载, 但其实这些负载是可以节省下来的. 因此, 实际运行时仅需要并行利用高优先级的优化经验即可.

经验内部实施过程的并行优化是指当匹配到某一条经验后，对该经验对定义域划分的各个子空间优化过程进行并行执行，以及对每个子空间的并行执行不同的重写规则. 该粒度下的主要并行瓶颈为该经验的定义域子空间数量, 以及每个子空间的重写规则数.

重写规则执行过程内的并行优化是指对于一些复杂的重写规则, 其执行过程本身可以实施规则内部的并行优化. 例如泰勒展式重写规则, 其展开过程是可以对各展开项的重写与评估来实施并行优化. 对于简单规则来说, 该粒度的并行优化难以实施. 该粒度下的主要并行瓶颈是当前规则的可并行重写任务数. 由于各粒度的并行能力可以相互叠加, 因此实际算法在整体上的并行瓶颈可以认为是各粒度并行任务上限之积.

从负载均衡的角度来说, 本文方法需要实施动态负载均衡. 由于优化任务的特殊性, 各程序优化任务及子任务负载和很多因素有关, 如经验库的经验分布、规则库的规则顺序、待优化程序的执行次序、甚至输入形式等因素都会对优化任务负载产生重要影响. 因此我们很难在优化执行前完整估计其各个并行子任务负载状况, 也就很难在优化前静态分配各并行单元的运行负载. 我们可以在并行算法实施时, 从不同粒度来划定可并行任务, 将它们动态加入待执行队列, 并在运行时动态分配给各个 CPU 计算资源, 当某个 CPU 空闲时动态将队列中剩余的任务分配给它, 以实现负载均衡.

5 实验评估

我们在 FPBench[12] 的基准浮点用例集上部署了实验来评估本文加速策略的有效性.FPBench 涵盖了各种不同类型的经典浮点程序, 并同时给出数值分析专家的人工优化方案, 其具体说明如表 3所示. 用例集中的人工优化方案大部分来源于权威著作 Hamming's Numerical Methods for Scientists and Engineers[11], 由于该用例集十分典型, 现有数值优化评估方法[8,9] 也常以该基准用例集作为主要实验评估用例. 除此之外, 为了评估本文算法在大型实际用例上的优化情况, 本文进一步在 MIT 开发的相对论场景物理模拟引擎 OpenRelativity 及其依赖库上实施了评估实验.

本文实验运行于 Macbook Pro, CPU 为 2.30GHz

的 Intel Core i5, 内存为 8GB 的 DDR3, 操作系统为 Mac OSX High Sierra, 开发语言为 Racket v7.0，此外我们实现了简单的单机环境多核并行优化, 以评估本文算法在多核环境下的优化效果.

通过实验, 本文拟回答以下三个关键的研究问题:

- **研究问题 1**: 经验库制导的优化加速策略能达到怎样的加速效果?
- **研究问题 2**: 加速后数值优化方法的优化精度如何?
- **研究问题 3**: 本文加速方法在大型实际数值应用上是否可用?

5.1 实验设计

表 3 FPBench 用例程序说明

用例程序	说明
$2atan$	用 $arctan(N+1)-arctan(N)$ 对定积分 $\int_N^{N+1}\frac{dx}{1+x^2}$ 求值
$2cbrt$	对 $\sqrt[3]{x+1}-\sqrt[3]{x}$ 求值
$2sin$	对 $sin(x+\epsilon)-sin(x)$ 求值
$2tan$	对 $tan(x+\epsilon)-tan(x)$ 求值
$cos2$	对 $(1-cos(x))/x^2$ 求值
$cubic$	计算一元二次方程较大根的变体, $(-b+\sqrt{b^2-3ac})/(3a)$
$exp2$	对 e^x-2+e^{-x} 求值
$expax$	对 $e^{ax}-1$ 求值
$expq2$	对 $e^x/(e^x-1)$ 求值
$logq$	对 $log(\frac{1-\epsilon}{1+\epsilon})$ 求值
$math.sin$	计算复数正弦值的虚部
$quad2m$	计算一元二次方程较小根的变体, $(-b-\sqrt{b^2-ac})/a$
$quad2p$	计算一元二次方程较大根的变体, $(-b+\sqrt{b^2-ac})/a$
$quadp$	求解一元二次方程的较大根

本文的加速策略需要经验库来制导. 我们构造了一个基于 C 库基础数学函数的优化经验库, 以此来分析评估实验效果. 用于积累经验库的 C 库基础数学函数如表 4 所示.

表 4 基于 C 语言标准数学库的经验库初始化程序集

C 库函数	初始化程序	最大 ulp 误差
$sqrt$	$(-b-\sqrt{b^2-4ac})/(2a)$	33 比特
$sqrt$	$\sqrt{x+1}-\sqrt{x}$	30 比特
exp	e^x-1	59 比特
exp	$e^{-x}-e^x$	58 比特
exp	$(e^x-1)/(e^x+1)$	39 比特
log	$log((1+x)*(1-x))$	30 比特
log	$log(1-x)-log(1+x)$	58 比特
log	$log(1-x)+log(1+x)$	29 比特
sin	$x-sin(x)$	10 比特
sin	$sin(x-y)$	15 比特
cos	$cos(x-y)$	15 比特
cos	$cos(x+y)-cos(x)$	40 比特
tan	$x-tan(x)$	10 比特
sin, tan	$(x-sin(x))/(x-tan(x))$	31 比特
sin, exp	$0.5\times sin(x)\times(e^{-y}-e^y)$	43 比特

在实验过程中为了保证评估的公平性, 每完成一个实验用例的优化后, 会把经验库重置. 即保证每个用例的实验结果是在相同经验库下完成了, 与优化次序无关. 另外, 本文所有数据均为双精度浮点数的优化结果.

本文以无加速策略的 Herbie 算法[8] 和平衡精度加速策略[15] 作为评估对比方法. Herbie 算法是近年来业界最著名的数值程序优化算法之一, 在 FPBench 等用例上优化效果显著, 而平衡精度加速策略通过降低数值优化精度来提高优化速度, 是到目前为止加速效果最好的数值优化算法.

5.2 加速效果评估

我们对 FPBench 的各数值用例程序分别运行三种不同的优化算法: 未加速的 Herbie 原始算法、平衡精度的加速优化策略、和本文的经验库制导加速优化策略, 并对比记录它们在本文实验平台上的优化时间. 图 3和表 5分别展示了实验结果及其对应的细节数据. 从实验结果来看, 加速前后的时间差距显著: 在平均情况下,Herbie 算法的优化时间是本文加速策略的 6.04 倍, 即两者加速比为 6.04. 而目前为止加速效果最好的平衡精度加速策略仅能达到 1.95 倍的加速比. 这里的加速比定义为 $\rho=t_o/t_n$,

其中 t_o 为加速前的优化时间,t_n 为加速后的优化时间, 而平均加速比由平均优化时间计算所得. 这里的优化时间统计的是优化算法将输入程序优化成浮点误差累积更小的输出程序的总时间. 因此, 它包括了规则搜索, 规则匹配, 程序重写等所有优化相关子过程的时间开销.

表 5进一步给出了各用例在规则库中搜索匹配规则的相关数据. 由于没有经验库的制导,Herbie 原始算法以及平衡精度策略需要在优化过程中反复搜索规则库、匹配规则、并评估转换后的程序. 由于平衡精度策略仅仅调整了优化过程中的计算精度而并未改变规则的搜索匹配过程, 因此其规则搜索与匹配数量和原始 Herbie 算法相同, 此时仅能达到 1.95 倍的平均加速; 本文算法由于经验库制导而大大减少了规则搜索和匹配数量, 对应地减少了重写程序的高精度评估次数, 因而能够达到平均 6.04 倍的加速效果. 在平均情况下,Herbie 原始算法和平衡精度策略尝试的重写规则数是本文加速策略的 45.06 倍. 本文将该比值定义为重写规则比 $\sigma = m_o/m_n$. 其中 m_o 为原来的重写规则数量, m_n 为本文加速后的重写规则数量. 表 5的实验数据中各用例的重写规则比一般是明显高于本文策略的加速比, 这一现象的主要原因在于: 加速前后算法除了需要执行规则匹配与程序重写操作外, 还需要执行包括用例输入生成, 误差计算等额外操作. 无论是否采用经验库制导的加速策略, 这些额外操作都是必不可少的. 因此在加速前后的程序优化时间里都包含了这些额外操作所消耗的时间. 这导致优化时间的比值 (即加速比) 会比重写规则数量的比值 (即重写规则比) 更加趋近于 1.

另外我们还观察到, 不同程序用例的加速比并不相同. 有的用例 (例如 2*cbrt*) 能够达到 93.42 倍的加速比, 而另一些用例 (例如 *cubic*) 仅达到 3.04 倍的加速比. 在用例 2*cbrt* 中, 造成数值稳定性问题的核心浮点计算过程是 $\sqrt[3]{x+1} - \sqrt[3]{x}$. 在 x 较大时该计算过程存在大数相消, 从而造成程序不稳定. 在没有经验库制导的条件下, 该程序的优化过程会匹配一些处理起来非常耗时的规则 (如泰勒展开规则), 并尝试用它们来做优化. 而在经验库制导下, 本文加速策略能预先绕过这些无效但复杂的规则尝试, 从而节省了大量的优化时间. 而在另一些用例上本文加速策略绕过的重写规则较为简单, 例如在 *cubic* 上主要绕过一些对优化无效的简单规则, 比如加法交换律规则, 加速效果相对来说不够明显, 仅能达到 3.04 倍的加速比. 因此本文加速策略带来的加速效果在不同类型的用例上会有所不同.

我们进一步评估了本文算法在多核环境下的并行优化效果. 表 5的最后 2 列描述了在 4 核环境下, 达到同样的优化精度所需的平均优化时间和加速比. 实验结果表明, 在多核平台下本文加速策略的性能得到了进一步的提升, 加速比达到了 23.34 倍, 平均优化时间减少了 74.1%, 基本能够充分利用 4 核平台的性能.

5.3 优化精度评估

本文进一步分析了未加速的 Herbie 原始算法、平衡精度的加速优化策略、以及本文的经验库制导加速优化策略在优化前后的数值精度变化, 对于每一个用例, 我们随机生成输入来分别运行各不同加速策略优化前后的数值程序. 这些随机生成的输入首先被缓存起来, 然后分别用于驱动各种不同加速策略优化前后的用例程序, 从而保证各用例程序由完全相同的随机输入驱动. 由这些用例驱动下的程序输出与基准用例集的高精度标准输出计算 ulp 误差 (该误差由公式 1 定义). 通过计算不同随机输入下的 ulp 误差均值, 我们可以得到各程序的平均 ulp 误差, 并以此来评判程序的数值稳定性. 平均 ulp 误差越低, 则表示程序的数值稳定性越好.

在浮点程序的传统评估方法中, 并不是通过在区间 [MIN_DOUBLE,MAX_DOUBLE] 均匀采样来随机生成程序输入. 这里 MIN_DOUBLE 和 MAX_DOUBLE 分别指双精度浮点类型能够表示的最小值和最大值. 因为 [MIN_DOUBLE,MAX_DOUBLE] 的表示范围较大, 在这一区间内均匀采样很难得到绝对值较小的程序输入. 因此本文通过比特位采样的方式来随机获得程序输入. 该方法的基本思想是将表示浮点数的每一个二进制比特位看作独立数据, 分别有 50% 的概率取到 0, 也有 50% 的概率取到 1. 将双精度浮点数的每一个比特位进行随机采样后组合, 产生随机输入. 由于浮点数的表示特性, 这种方式既能保证获得绝对值较大数据的概率, 也能保证获得绝对值较小数据的概率.

图 4 显示了精度变化的实验结果, 由其数据可以看出, 平衡精度的加速优化策略明显降低了优化程序的优化精度, 其优化后的平均 ulp 误差大约为 8 比特, 而本文加速策略能够基本达到和未加速的原始优化算法相同的精度, 优化后的平均 ulp 误差均为 3 比特. 出现这一现象的根本原因仍然是因为平

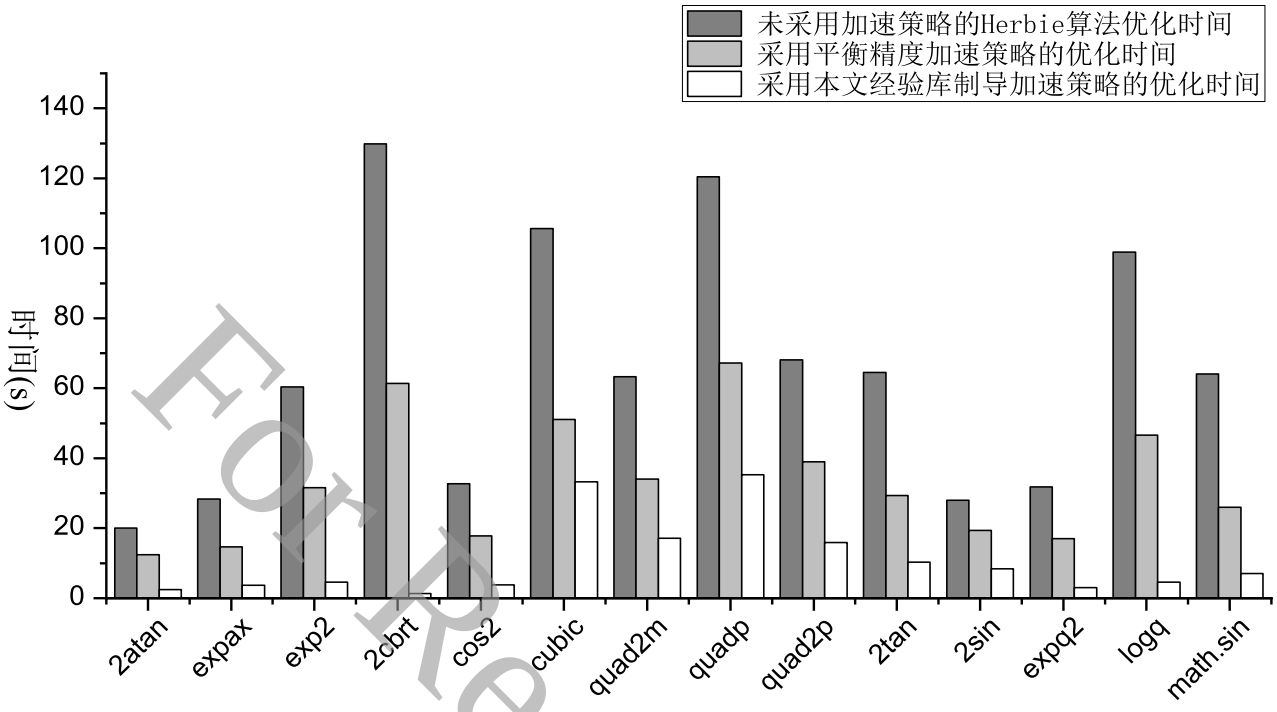

图 3　不同加速策略加速前后的 FPBench 基准用例优化时间

表 5　经验库制导加速前后的重写规则应用数目与优化时间对比

FPBench 用例程序	原始/平衡策略规则数	本文策略规则数	重写规则比	未加速优化时间	平衡精度优化时间	本文策略优化时间	平衡精度加速比	本文策略加速比	四核并行优化时间	四核并行加速比
2atan	222	4	55.5	20.05s	12.47s	2.47s	1.61x	8.12x	0.78s	25.71x
2cbrt	639	2	319.5	129.86s	61.36s	1.39s	2.12x	93.42x	0.55s	236.11x
2sin	204	4	51.0	28.02s	19.38s	8.37s	2.32x	3.35x	3.48s	8.06x
2tan	275	4	68.75	64.5s	29.2s	10.3s	2.83x	6.26x	3.18s	20.31x
cos2	293	4	73.25	32.74s	17.83s	3.82s	1.84x	8.57x	1.05s	31.18x
cubic	287	13	22.08	101.19s	51.06s	33.3s	1.98x	3.04x	8.5s	11.86x
exp2	236	7	33.71	60.34s	31.56s	4.56s	1.91x	13.23x	1.15s	52.47x
expax	203	1	203.0	28.39s	14.7s	3.7s	1.93x	7.67x	1.0s	28.39x
expq2	230	2	115.0	31.8s	17.08s	3.08s	1.86x	10.32x	0.75s	42.4x
logq	107	3	35.67	98.91s	46.63s	4.63s	2.12x	21.36x	1.45s	68.21x
math.sin	271	3	90.33	64.12s	26.04s	7.04s	2.46x	9.11x	1.98s	32.47x
quad2m	306	13	23.54	63.27s	34.11s	17.11s	1.85x	3.7x	5.33s	11.88x
quad2p	316	13	24.31	68.13s	38.96s	15.96s	1.75x	4.27x	3.65s	18.67x
quadp	286	13	22.0	120.41s	67.24s	35.25s	1.79x	3.42x	6.25s	19.27x
平均值	276.79	6.14	45.06	65.12s	33.41s	10.78s	1.95x	6.04x	2.79s	23.34x

衡精度加速策略通过降低非关键运算的精度来提高优化速度, 因而使得加速后的程序精度在整体上会有不同程度的下降, 本文算法绕开了这一问题, 通过减少搜索与匹配规则数量而进行的优化不会影响优化精度. 这里很多用例在优化前的误差相当大, 优化前的平均 ulp 误差约有 36 比特, 对于 math.sin 在优化前存在 58 比特误差. 这意味着在程序输出值的浮点数表示中, 用于表示有效数字的 52 个比特位全是无效的, 且程序输出值与理论精确值在指数位上还有 6 个比特的数量级差距. 进行自动化数值优化后, 各用例的误差明显减小.math.sin 在平衡精度的加速策略优化后存在 9 比特的 ulp 误差, 在本文加速策略优化后仅存在 1 比特的 ulp 误差, 这意味着输出精度可以达到小数点后第 51 个比特位.

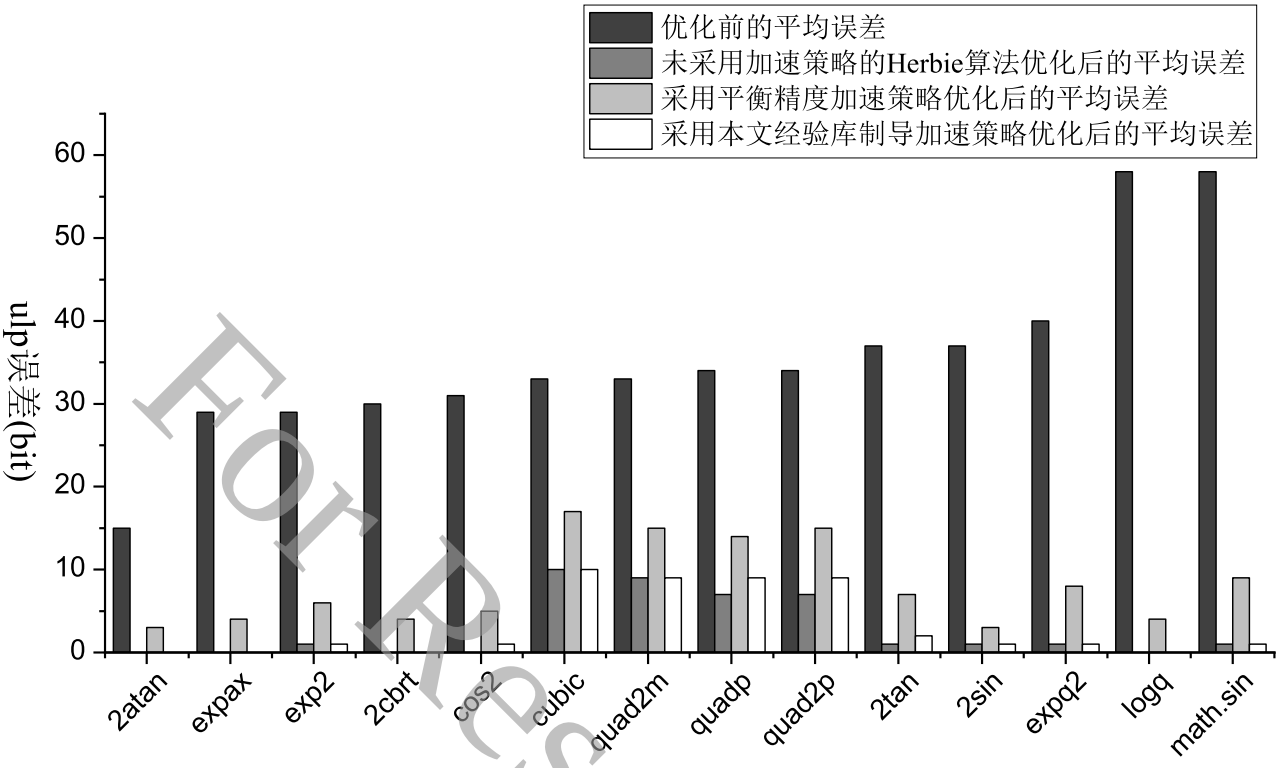

图 4 使用不同加速策略优化前后的 FPBench 程序平均数值误差

(2atan,expax,2cbrt,cos2,logq 等部分用例优化后可能会使输出结果精度达到双精度浮点表示精度, 即 ulp 误差小于 1 比特, 在图中显示为空柱)

通过对比图 4的对应数据可以看出, 在优化有效的情况下, 运用加速策略后的优化效果与不采用加速策略类似. 在 14 个 FPBench 用例程序的优化中, 使用本文加速策略与否的优化效果仅在 4 个用例程序上表现不同, 而优化效果的平均差异仅为 1.5 比特. 这 4 个用例程序分别为:$cos2, 2tan, quad2p$ 和 $quadp$. 我们进一步分析了优化效果不同的用例程序的优化细节, 发现运用加速策略后采用的重写规则与传统遍历算法采用的重写规则在这几个用例上是不同的. 以 $quad2p$ 为例, 在传统遍历算法中, 该用例匹配到了泰勒展开规则, 而基于经验库的规则未能匹配到该条规则, 而是匹配了 $a-b \to (a^2-b^2)/(a+b)$ 这样的规则来进行优化, 故而优化效果有 2 比特的不同.

5.4 大型用例评估

OpenRelativity [3]是由 MIT 开发的用于模拟相对论场景的物理引擎, 其底层依赖于 3D 引擎 Unity 的实现库 [4]. 我们在 OpenRelativity 及其相关依赖库上实施了本文的数值优化加速策略, 检测到一个原始 Herbie 优化算法和平衡精度加速策略无法优化的高误差数值程序段, 而能够在本文的加速策略下成功完成优化, 使得原始库中在平均情况下 25.7 比特的平均 ulp 误差降低到 3 比特.

经相关专业人员确认, 该误差在计算相对论效应下两个基本粒子在不同参考系下的湮灭寿命比时出现. 由于爱因斯坦相对论在理论上较为复杂, 为了便于读者理解, 我们将这种情况简化为如下二维时空下的描述. 假设两种基本粒子由同一个发射源发出, 发出时的坐标 $x_0=0$, 时间 $t_0=0$, 其中一个粒子在 (x_1,t_1) 时发生湮灭, 另一个粒子在 (x_2,t_2) 时发生湮灭. 基本粒子的受力情况和运动过程在实验上不能测量, 现在由开源相对论引擎来计算在速度为 v 的参考系下, 这两个粒子在相对论效应下的寿命比 t_1'/t_2'.

按照相对论计算中普遍采用的闵可夫斯基空间和四维矢量算法, 令 $w=ict$(这里 c 为光速, 即 $w_1=ict_1$, $w_2=ict_2$), 并参照洛伦兹变换, 我们有 $w_1'=(w_1-ix_1v/c)/\sqrt{1-v^2/c^2}$, $w_2'=(w_2-ix_2v/c)/\sqrt{1-v^2/c^2}$. 因此寿命比

$$\frac{t_1'}{t_2'}=\frac{w_1'}{w_2'}=\frac{w_1-ix_1v/c}{w_2-ix_2v/c} \tag{12}$$

其中光速 $c=3.00e+8$, 当使用该算法计算 $v=$

[3]https://github.com/MITGameLab/OpenRelativity

[4]https://github.com/Unity-Technologies/UnityCsReference

$2.97e+8, w_1 = 1.45e-202, w_2 = -2.58e-295, x_1 = 4.60e-111, x_2 = 7.67e+173$ 时存在巨大误差, 其 ulp 误差为 59 比特, 相对误差为 100%.

由于公式 (12)所示的这类运算大量存在于 OpenRelativity 的使用过程, 因此对应程序段的平均 ulp 误差很高. 而原始 Herbie 优化算法和平衡精度加速策略在优化该程序段时能够匹配到大量的重写规则, 而造成匹配重写规则中正确的优化序列变得非常困难. 经过 2 小时的运行, 我们的实验平台无法通过原始 Herbie 优化算法以及平衡精度加速策略获得任何优化结果. 本文的加速策略在经验库的制导下能大大降低重写规则的匹配数量, 在我们平台上约需要 33 分钟可以获得正确的优化结果, 优化后的 ulp 平均误差降低为 3 比特.

6 相关工作

6.1 浮点误差优化方法

在早期, 对浮点程序的误差优化大多依赖于专家知识[10,11]. 这类数值分析方法的缺点是优化成本较高, 难以应用于大规模浮点程序的优化. 随着研究的深入, 依靠计算机的自动化浮点误差优化方法成为一项新兴技术.Herbie[8] 是这类方法的一个典型代表. 它在不改变原有程序语义的情况下使用程序转换规则重写程序, 从而生成误差累积更小的浮点程序. 全局优化框架[9] 也采用了类似的策略, 不过它的侧重点在于从全局视角出发, 而不局限于程序的局部细节. 通过随机且带有优先级的程序重写策略来优化误差. 由于基于规则重写的优化方法无法解决病态问题[16–18],AutoRNP[19] 另辟蹊径, 使用函数逼近的方式来优化浮点数值程序的计算误差. 除上述工作之外, 还有一些工作致力于解决部分浮点优化领域特有的问题, 比如优化由精度特定操作 (precision-specific operations)[20] 引起的误差问题. 本文的优化加速策略也是一种行之有效的浮点误差优化技术, 但其更关注于提升浮点优化效率, 进而提高浮点优化技术的可用性.

6.2 浮点误差检测与误差输入搜索

在进行误差优化之前, 我们通常需要做浮点误差检测, 以确定浮点程序是否具备优化的价值.FPDebug[21] 是著名的动态误差检测工具. 它会在动态执行原有浮点程序机器指令的同时, 执行一条和原有指令语义相同但使用了更高精度浮点数的指令 (shadow execution). 通过对比两者执行结果的差异即可定位数值计算不稳定的指令. 这项技术的实现依赖了 MPFR[22] 和 Valgrind[23].Hergrind[24] 比 FPDebug 更进一步, 它不仅能判断浮点程序是否存在较大误差, 还能定位浮点误差产生的源头. 这个特性极大方便了开发者的调试与优化工作. 在某些应用领域, 我们不仅需要知道浮点程序是否存在较大误差, 更需要知道具体是哪些输入值触发了这些误差.BRGT[25] 使用了二分搜索来搜索误差输入. 它的基本原理是使用启发式算法对输入域进行迭代切分, 在输入域不断缩小的过程中找到有问题的输入.LSGA[26] 通过遗传算法来搜索能触发较大误差的输入.DEMC[19] 算法是 AutoRNP[19] 的一个重要模块. 它结合了蒙特卡罗马尔科夫链 (MCMC)[27] 和差分进化算法 (differential evolution)[28], 用于搜索能触发病态问题的输入. 误差检测与误差输入搜索是进行误差优化的重要基础. 上述工作的改进可帮助本文优化加速算法取得更好的优化效果.

7 总结

本文提出了一种经验库制导的浮点程序优化加速策略. 该策略基于浮点误差成因的相似性原理, 将已经成功优化浮点程序的优化经验保存在一个优化经验库中, 从而利用这些经验制导和加速其他浮点数值程序的优化过程. 随着优化经验的不断累积, 经验库的规模会逐渐增大, 散列化分区存储结构的设计使其检索匹配效率不会成为系统的性能瓶颈. 为了验证本文加速策略的有效性, 我们在浮点程序优化基准用例集 FPBench 和开源物理引擎 OpenRelativity 上开展了评估实验. 实验结果表明, 在优化后浮点程序精度效率与传统方法类似的前提下, 本文的加速策略能使平均优化速度相对于传统浮点程序优化方法提升 6.04 倍, 从而显著提高了浮点数值程序自动优化技术的可用性.

参考文献

[1] Kelley T B. Rounding problems in commercial data processing. Communications of the ACM, 1964, 7(11):654−-656.

[2] Frechtling M, Leong P H. Mcalib: Measuring sensitivity to rounding error with monte carlo programming. ACM Transactions on Programming Languages and Systems, 2015, 37(2):1-25.

[3] Moore R E. Order relations and rigor in computing. Proceedings of the 2005 ACM Symposium on Applied Computing. Santa Fe, New Mexico, 2005: 1431−-1433.

[4] Skeel R. Roundoff error and the patriot missile. SIAM News, 1992, 25

(4):11.

[5] McCullough B D, Vinod H D. The numerical reliability of econometric software. Journal of Economic Literature, 1999, 37(2):633-665.

[6] Babuška I, Söderlind G. On roundoff error growth in elliptic problems. ACM Transactions on Mathematical Software, 2018, 44(3):1-22.

[7] Tang E, Barr E, Li X, Su Z. Perturbing numerical calculations for statistical analysis of floating-point program (in) stability. Proceedings of the 19th international symposium on Software testing and analysis. Trento, Italy, 2010: 131-142.

[8] Panchekha P, Sanchez-Stern A, Wilcox J R, Tatlock Z. Automatically improving accuracy for floating point expressions. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. Portland, USA, 2015: 1-11.

[9] Wang X, Wang H, Su Z, Tang E, Chen X, Shen W, Chen Z, Wang L, Zhang X, Li X. Global optimization of numerical programs via prioritized stochastic algebraic transformations. Proceedings of the 41st International Conference on Software Engineering. Montreal, Canada, 2019: 1131-1141.

[10] Goldberg D. What every computer scientist should know about floating-point arithmetic. ACM computing surveys, 1991, 23(1):5−-48.

[11] Hamming R. Numerical methods for scientists and engineers. Courier Corporation, 2012.

[12] Damouche N, Martel M, Panchekha P, Qiu C, Sanchez-Stern A, Tatlock Z. Toward a standard benchmark format and suite for floating-point analysis. International Workshop on Numerical Software Verification. Toronto, Canada, 2016: 63-77.

[13] Lopez G A, Taufer M, Teller P J. Evaluation of ieee 754 floating-point arithmetic compliance across a wide range of heterogeneous computers. Proceedings of the 2007 Conference on Diversity in Computing. Orlando, Florida, 2007: 1-4.

[14] Zou D, Zeng M, Xiong Y, Fu Z, Zhang L, Su Z. Detecting floating-point errors via atomic conditions. Proceedings of the ACM on Programming Languages, 2019, 4(POPL):1-27.

[15] Saiki B, Flatt O, Nandi C, Panchekha P, Tatlock Z. Combining precision tuning and rewriting. 28th IEEE Symposium on Computer Arithmetic. Lyngby, Denmark, 2021: 1-8.

[16] Higham N. Accuracy and stability of numerical algorithms. SIAM, 2002.

[17] Bao T, Zhang X. On-the-fly detection of instability problems in floating-point program execution. Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. Indiana, USA, 2013: 817-832.

[18] Tang E, Zhang X, Müller N T, Chen Z, Li X. Software numerical instability detection and diagnosis by combining stochastic and infinite-precision testing. IEEE Transactions on Software Engineering, 2016, 43(10):975-994.

[19] Yi X, Chen L, Mao X, Ji T. Efficient automated repair of high floating-point errors in numerical libraries. Proceedings of the ACM on Programming Languages. Cascais, Portugal, 2019: 1-29.

[20] Wang R, Zou D, He X, Xiong Y, Zhang L, Huang G. Detecting and fixing precision-specific operations for measuring floating-point errors. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Seattle, USA, 2016: 619-630.

[21] Benz F, Hildebrandt A, Hack S. A dynamic program analysis to find floating-point accuracy problems. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. Beijing, China, 2012: 453-462.

[22] Fousse L, Hanrot G, Lefèvre V, Pélissier P, Zimmermann P. Mpfr: A multiple-precision binary floating-point library with correct rounding. ACM Transactions on Mathematical Software, 2007, 33(2):13−-es.

[23] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. California, USA, 2007: 89-100.

[24] Sanchez-Stern A, Panchekha P, Lerner S, Tatlock Z. Finding root causes of floating point error. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. Philadelphia, USA, 2018: 256-269.

[25] Chiang W, Gopalakrishnan G, Rakamaric Z, Solovyev A. Efficient search for inputs causing high floating-point errors. Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Florida, USA, 2014: 43-52.

[26] Yi X, Chen L, Mao X, Ji T. Efficient global search for inputs triggering high floating-point inaccuracies. Proceedings of the 24th Asia-Pacific Software Engineering. Nanjing, China, 2017: 11-20.

[27] Andrieu C, De Freitas N, Doucet A, Jordan M I. An introduction to mcmc for machine learning. Machine learning, 2003, 50(1):5-43.

[28] Storn R, Price K. Differential evolution - A simple and efficient heuristic for global optimization over continuous spaces. Journal of global optimization, 1997, 11(4):341-359.

XIAO An-Xiang, M.S..His research interests include floating-point program optimization and software test.

ZHANG ShuoXiao, Ph.D. candidate.His research interests include floatingpoint program optimization and blockchain.

TANG EnYi, Ph.D.,associate professor.His research interests include numerical system, program analysis, software testing, and AI infrastructure.

CHEN Xin, Ph.D.,associate professor.His research interests include software engineering, software testing, verification technology.

WANG LinZhang, Ph.D.,professor.His research interests include model driven software testing and verification, security. testing, software testing automation.

Background

Floating-point programs are common both in scientific and industrial applications. adrieu2003They approximate real numbers with precision-limited floating-point numbers according to the IEEE standard, which introduces roundoff errors inevitably. Without careful implementation, tiny roundoff errors can cause program failures. As floating-point programs are executed, roundoff errors can be amplified and accumulated, consequently leading to unexpected outputs. In life-critical systems, program failures can lead to catastrophes. Therefore, it is essential to ensure the correctness of floating-point programs.

Recently, rewrite-based approaches aiming to improve accuracy for floating-point programs have been proposed. They make use of transformation rules summarized from classical numerical analysis theories to rewrite floating-point programs with more stable numerical algorithms, while maintaining input-output equivalent semantics. With program rewriting, inaccurate programs can be automatically transformed into ones with fewer accumulating errors. However, efficiency can be a bottleneck for such techniques. On the one hand, to improve their performance on accuracy improvement, more transformation rules need to be discovered and added to the optimizing framework, which scales up the rule base gradually. On the other hand, rewrite-based techniques have to scan the whole rule base iteratively to find appropriate transformation rules, which is time-consuming in a large rule base.

To alleviate the problem, we propose an experience-guided acceleration strategy for floating-point optimization. The strategy is based on an observation that similar floating-point programs can suffer from similar inaccuracy problems, which usually result from similar causes. Therefore, rules used to improve accuracy for floating-point programs can also help to reduce accumulating errors for programs similar to the previous ones. Following the observation, optimizing experiences of similar programs can guide the optimizing framework to search rules efficiently. Instead of scanning the whole rule base, the optimizing framework only considers transformation rules that are similar to those used to improve accuracy for similar programs, thus avoiding wasting much time on examining useless rules. We have evaluated our approach on FPBench and OpenRelativity. The results show that compared to the traditional optimizing approach, it can achieve a 6.04x speedup in average. In addition, it improves accuracy for floating-point programs effectively as well as the traditional optimizing approach does. Therefore, our approach improves the usability of automatic optimizing techniques significantly.