计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2020,29(2):129−139 [doi: 10.15888/j.cnki.csa.007318]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

嵌入式软件编译时能耗演化优化算法①

姚万庆[1], 倪友聪[1,2], 杜　欣[1], 叶　鹏[3], 肖如良[1]

[1](福建师范大学 数学与信息学院, 福州 350117)
[2](福建师范大学 福建省公共服务大数据挖掘与应用工程技术研究中心, 福州 350117)
[3](武汉纺织大学 数学与计算机学院, 武汉 430073)
通讯作者: 倪友聪, E-mail: youcongni@foxmail.com

摘　要: 编译选项的选择优化为降低嵌入式软件能耗提供了一种可行且有效的解决方案. GA-FP 算法将频繁模式挖掘应用到演化过程中并已取得了较好的结果. 然而, GA-FP 还存在事务表规模较大、频繁选项模式的启发信息不全和时效性不好以及单点变异效率不高的缺点, 潜在地影响了解质量和收敛速度. 针对这些问题, 文中提出一种嵌入式软件编译时能耗演化优化算法 GA-MFPM. GA-MFPM 借助逐代替换参考点和事务表的机制以降低事务表大小; 在此基础上提出可获取更多启发式信息的频繁编译选项挖掘算法, 并采用逐代挖掘的策略以保持频繁选项模式的时效性; 进一步设计最大频繁模式匹配算法进行多点变异, 以提高优化质量和收敛速度. 通过与 GA-FP 在 5 个不同领域的 8 个典型案例下实验对比的结果表明: 本文 GA-MFPM 获取了较 GA-FP 更低的软件能耗 (平均降低 2.4%, 最高降低 16.1%) 和更快的收敛速度 (平均加快 57.6%, 最高加快 97.5%).

关键词: 软件能耗; 编译优化; 频繁模式挖掘; 嵌入式软件; 演化算法

引用格式: 姚万庆,倪友聪,杜欣,叶鹏,肖如良.嵌入式软件编译时能耗演化优化算法.计算机系统应用,2020,29(2):129–139. http://www.c-s-a.org.cn/1003-3254/7318.html

Evolutionary Optimization Algorithm for Embedded Software Energy Consumption at GCC Compiling

YAO Wan-Qing[1], NI You-Cong[1,2], DU Xin[1], YE Peng[3], XIAO Ru-Liang[1]

[1](College of Mathematics and Informatics, Fujian Normal University, Fuzhou 350117, China)
[2](Fujian Provincial Engineering Technology Research Center for Public Service Big Data Mining and Application, Fujian Normal University, Fuzhou 350117, China)
[3](College of Mathematics and Computer Science, Wuhan Textile University, Wuhan 430073, China)

Abstract: The optimization of compilation options provides a feasible and effective solution to reduce the energy consumption of embedded software. GA-FP algorithm applies frequent pattern mining into the evolutionary process and has achieved better results than other algorithms. However, GA-FP still has the disadvantages, such as the large size of transaction table, incomplete and obsolete heuristic information provided by frequent option patterns and inefficient single point mutation, which potentially affects the solution quality and convergence rate. Aiming to these problems, this study proposes an evolutionary optimization algorithm for embedded software energy consumption at GCC compile time, called GA-MFPM. GA-MFPM replaces the reference point and transaction table mechanism by generation to reduce size of the transaction table. Further, a frequent compilation option mining algorithm is designed to acquire more heuristic information. It adopts a generation-by-generation mining strategy to help maintain the timeliness of frequent option

① 基金项目: 福建省新世纪优秀人才项目 (2017 年); 福建省自然科学基金 (2015J01235, 2017J01498); 福建省教育厅 JK 类项目 (JK2015006); 湖北省自然科学基金 (2018CFB689)
Foundation item: New Century Talent Supporting Program of Fujian Province (Year 2017); Natural Science Foundation of Fujian Province (2015J01235, 2017J 01498); JK Science and Technology Program , Fujian Province (JK2015006); Natural Science Foundation of Hubei Province (2018CFB689)
收稿时间: 2019-07-31; 修改时间: 2019-09-02; 采用时间: 2019-09-18; csa 在线出版时间: 2020-01-16

patterns. Based on the frequent option patterns, a maximum frequent pattern matching algorithm is designed to perform multi-point mutation to improve optimization quality and convergence rate. The comparative experiments are done on 8 typical cases in 5 different fields between GA-MFPM and GA-FP. The experimental results indicate that the GA-MFPM can not only reduce the energy consumption of software more effectively (the average and maximal reduction ratios are 2.4% and 16.1% respectively), but also converge faster (the average of 57.6% faster and up to 97.5% faster) than GA-FP in this study.

Key words: software energy; compilation optimization; frequent pattern mining; embedded software; evolutionary algorithm

能耗是嵌入式系统的关键质量属性. 据报道[1], 在嵌入式系统中高达 80% 的能耗直接与软件执行活动密切相关. 因而在电量受限的执行环境中, 降低嵌入式软件的能耗具有更为重要的价值和意义[2]. 通过选择一组最佳编译选项[3]在给定的执行平台下对嵌入式软件源代码进行编译, 可以获取能耗更低的可执行代码. 与源代码重构[4]的能耗优化技术相比, 基于编译选项选择的优化不仅可以考虑硬件平台特性, 而且还能在不修改源代码的情况下保证功能语义一致性. 因此, 编译选项的选择优化为降低嵌入式软件能耗提供了一种可行且有效的解决方案.

开源编译器 GCC[5]已广泛用于嵌入式系统的开发. 针对 GCC 编译器的编译选项选择优化问题可描述为: 对于源代码 src 和执行平台 platform, 搜索选项选择方案 $X=<x_1, x_2, \cdots, x_i \cdots, x_l>$ ($x_i \in \{0,1\}$分别表示不选用或选用第 i 号选项, l 为可用的选项数), 使得目标函数 $f(Sft_{\text{src}}, X)$的值最大. f 表示按 X 的选项选择方案对 src 进行编译和链接后得到的可执行代码在 platform 上从开始至结束运行所耗的电能, 较不选用任何选项的方案在能耗上的改进百分比. GCC 编译器虽已提供了若干标准优化等级用于降低可执行代码的执行时间和大小. 但 GCC 编译器没有提供专门针对能耗的优化等级. 最近研究工作已显示了一些 GCC 已有的优化等级甚至导致嵌入式软件能耗增大的情况[6,7].

近年来, 在能耗优化这一方面, GCC 编译选项的选择问题已成为一个热门的研究话题[3], 但仍面临两个公开挑战. 一是 GCC 编译器提供了大量的编译选项, 构成了庞大且离散的选择空间. 例如针对 Hoste 等[8]提出 58 个常用于能耗优化的编译选项, 其对应的选择空间将高达 2^{58}(大于 10^{17}). 另一挑战是编译选项之间、编译选项与能耗之间存在着潜在的复杂影响, 给提高搜索效率和优化质量带来相当大的困难. 一个编译选项的选用可能触发或使其它编译选项失效, 不同的编译选项对能耗所起的效用也不尽相同. 目前已涌现出统计、机器学习和演化算法等 3 类基于 GCC 编译选项选择的嵌入式软件能耗优化方法.

统计方法[2]使用部分析因实验设计技术搜索编译选项的选择空间. 正交数组被用于定义实验组和控制组, 通过曼-惠特尼检验或克鲁斯凯-沃利斯统计检验以观测能耗是否发生显著变化, 进而确定选用或不选用的选项. 统计方法不仅难以识别任意多个编译选项之间的影响, 而且受限于搜索空间也难以找到最优的编译选项组. 机器学习方法[3,9], 通过构建预测模型以帮助 GCC 编译器或优化算法搜索出最佳编译选项组. 构建预测模型分为收集训练样本和建立预测模型两个阶段. 收集训练样本阶段是一个迭代过程: 在每个迭代步中, 首先选择一个嵌入式软件并计算其特征的值; 接着基于一定策略在选项选择空间中采样获得多个选项组; 再利用一个选项组对所选择的嵌入式软件进行编译, 并通过执行获取能耗值. 基于软件的特征值、选项组和能耗值可构建出一个训练样本. 利用收集的训练样本集, 建模阶段则根据选用的机器学习算法[9]构建相应的预测模型. 为了增加嵌入式软件特征的有用性, 静态[10]、动态[11]和混合[12]等方法被用于确定具体的特征. 然而机器学习方法的训练样本往往针对特定编译器版本和特定执行平台, 所构建的预测模型可移植性差. 另外, 受制于选项的采样空间, 机器学习方法也难以获取高质量的优化结果.

为了搜索更大的编译选项选择空间, 提高能耗优化的质量, 一些演化优化算法也被纷纷提出. 基因加权的遗传算法 GWGA[13]将权值关联到每个基因位, 用于刻画相应的编译选项对优化目标的重要性, 并在优化

过程中进行更新. 与一元分布评估算法 UMDA 类似, GWGA 仅考虑最优解中每个选项被选用的概率分布. GWGA 虽能一定程度地加快收敛速度, 但因没有考虑各个选项之间存在相互影响关系, 容易导致陷入局部最优. Tree-EDA[14]运用概率树模型捕获任意两个编译选项之间的影响, 能得到比遗传算法 (GA)、GWGA 和 UMDA 更好的解质量. 为了考虑多个编译选项之间潜在的影响, 我们提出了基于频繁模式挖掘的遗传算法 GA-FP[15]. GA-FP 在演化过程中将相对于未优化前有改进效果的个体信息 (包括选用的选项编号和获得的能耗改进) 存入全局事务表, 并在每代种群演化结束后基于全局事务表挖掘频繁选项模式集, 接着利用频繁选项模式集中各个选项的能耗标注作为启发式信息设计了“增添”和“删减”两种单点变异算子, 进而得到了比 Tree-EDA 更好的解质量和收敛速度.

然而, GA-FP 算法还存在一些不足: (1) 在演化过程中始终固定以优化前能耗值作为参考点来判断个体是否有改进, 并将有改进效果的个体信息存入全局事务表中, 容易导致事务表规模过大、存储效率不高的问题. (2) 基于全局事务表挖掘频繁选项模式, 不仅存在挖掘时间长的问题, 而且挖掘出的频繁选项模式也难以体现较好的时效性. 此外, 挖掘出的频繁选项模式仅给出单个频繁选项的能耗标注, 而多个频繁选项同时选用的次数和能耗标注也未予考虑. (3) “增添”和“删减”变异操作虽能充分利用单个频繁选项的能耗标注所体现的启发式信息进行单点变异, 但仍没有完全利用多个频繁选项同时选用的次数和能耗标注等启发式信息, 潜在地影响到解质量和收敛速度. 针对上述问题, 文中提出一种嵌入式软件编译时能耗演化优化算法 GA-MFPM. GA-MFPM 借助逐代替换参考点和事务表的机制以降低事务表大小; 在此基础上提出可获取更多启发式信息的繁编译选项挖掘算法, 并采用逐代挖掘的策略有利于保持频繁选项模式的时效性; 进一步设计最大频繁模式匹配算法进行多点变异, 以提高优化质量和收敛速度. 通过与 GA-FP 在 5 个不同领域的 8 个典型案例下实验对比的结果表明: GA-MFPM 获取了较 GA-FP 更低的软件能耗 (平均降低 2.4%, 最高降低 16.1%) 和更快的收敛速度 (平均加快 57.6%, 最高加快 97.5%).

1 GA-MFPM 算法总体流程

GA-MFPM 算法用于求解针对 GCC 编译器并用于嵌入式软件能耗优化的编译选项选择问题. 其总体流程 (算法 1) 与遗传算法 GA[16]类似. 但不同之处在于, GA-MFPM 算法在演化过程中需构建和更新带能耗改进标注的事务表 TT_E (the Transaction Table with Energe annotations of options)、挖掘和更新带能耗改进标注频繁模式集表 $TPSFO_E^+$(the Table of Pattern Set of Frequent Options with Energy annotations), 并基于 $TPSFO_E^+$进行多点变异. 下面对它们作一简要介绍.

算法 1. GA-MFPM 算法的流程

输入: 种群大小 N, 变异概率 p_m, 交叉概率 p_c, 演化代数 t, 最大演化代数 t_{max}, 嵌入式软件源代码 Sft_{src}.

输出: 最优解 X^*.

1) 演化代数 t←1;
2) 生成大小为 N 的随机种群 $Pop(t)=\{X_1,X_2,\cdots,X_k,\cdots,X_N\}$, 其中 $1\le \forall k\le N, X_k\in\Omega$;
3) 按函数 $f(Sft_{src}, X_k)$ 计算初始种群 $Pop(1)$ 中各个体 X_k 的适应度值;
4) 设置参考点值为 $Pop(1)$ 中各个体适应度的中值为参考点值 $refVal$, 并将适应度值等于或高于 $refVal$ 的个体信息存入事务表 TT_E;
5) 基于事务表 TT_E, 挖掘获取频繁模式集表 $TPSFO_E^+$;
6) While ($t < t_{max}$) Do
7) 对种群 $Pop(t)$ 中个体按概率 p_c 执行交叉操作生成临时种群 $Q(t)$;
8) For ($Q(t)$ 中每个个体 X_k) Do
9) 产生[0,1]之间的随机数 $rndVal$;
10) If ($rndVal < p_m$) Then
11) 基于模式集表 $TPSFO_E^+$并运用最大匹配算法对 X_k 进行多点变异生成新个体, 并替换 X_k;
12) End If
13) End For
14) 将事务表 TT_E 置空;
15) 按函数 $f(Sft_{src}, X_k)$ 计算种群 $Q(t)$ 中各个体 X_k 的适 应度值;
16) 设置参考点值为 $Q(t)$ 中各个体适应度的中值为参考点值 $refVal$, 并将适应度值等于或高于 $refVal$ 的个体信息存入事务表 TT_E;
17) 将模式集表 $TPSFO_E^+$置空;
18) 基于事务表 TT_E, 挖掘获取模式集表 $TPSFO_E^+$;
19) 在 $Pop(t)\cup Q(t)$ 中选择最优的 N 个体生成下一代种群 $Pop(t+1)$;
20) 将参考点值 $refVal$ 更新为种群 $Pop(t+1)$ 中各个体适应度的中值;
21) t←$t+1$;
22) End While
23) X^*←$Pop(t)$ 中的最优解;
24) 输出最优解 X^*, 并结束算法运行.

(1) 带能耗改进标注事务表的构建和更新

GA-MFPM 算法利用每代种群中优势个体的信息构造事务表, 并逐代更新事务表.

在构造事务表时, 首先按适应度值 $f(Sft_{src}, X)$(能耗改进百分比) 从小到大对种群中各个体 X 进行排序; 然后将适应度值等于或高于设定参考点值 *refVal* 的个体作为优势个体, 并把这些个体的信息存入事务表. GA-MFPM 算法借鉴了多数 EDA[17]中优势个体在种群中的占比, 将 *refVal* 设定为种群中各个体适应度值的中值. 算法 1 第 4 行构建初始事务表. 下面通过一个例子阐述事务表的构建方法.

例中, 初始种群各个体适应度的中值为 8%, 即参考点值 *refVal*=8%. 初始种群中有 5 个个体的适应度值等于或大于 *refVal*, 现以其中的一个个体 X, 其适应度的值 $f(Sft_{src}, X)$=10% 为例说明一条事务的构建. 从图 1 所示的个体 X 可知其选用 8 个选项, 其编号为: {1, 2, 3, 6, 12, 13, 15, 16}. 依据这些选用的选项及其出现次数, 并将个体 X 的适应度值作为能耗标注附加于每个选用的选项, 可分别构建 8 个三元组 (选用选项, 出现次数, 能耗标注), 如表 1 中第 2 行的事务. 类似地, 利用其它 4 个优势个体可分别构建 4 条事务, 进而获得表 1 所示的初始事务表.

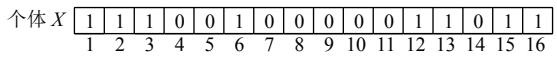

图 1 个体 X

表 1 带能耗标注的选项事务表 TT_E 的例子

事务标识 TID	带能耗标注的选项事务 T_E, T_E={<*opInfo*>|*opInfo* =(选项编号 *opNm*, 出现次数 *count*, 能耗标注 *engAno*)}
1	<(**6**,1,12%), (1,1,12%), (**3**,1,12%), (4,1,12%), (7,1,12%), (9,1,12%), (13,1,12%), (16,1,12%) >
2	<(1,1,10%), (2,1,10%), (**3**,1,10%), (**6**,1,10%), (12,1,10%),(13,1,10%),(15,1,10%), (16,1,10%) >
3	<(2,1,11%), (6,1,11%), (8,1,11%), (10,1,11%), (15,1,11%), (**3**,1,11%), (1,1,11%)>
4	<(2,1,10%), (3,1,10%), (11,1,10%), (17,1,10%), (16,1,10%) >
5	<(1,1,8%),(**6**,1,8%),(5,1,8%),(12,1,8%),(13,1,8%), (14,1,8%) >

GA-MFPM 算法在每代演化中对事务表进行更新, 如算法 1 中第 14 行和 15 行. 与对比算法 GA-FP 在演化过程中始终固定以优化前能耗值作为参考点不同, GA-MFPM 在演化过程中逐代替换参考点和事务表的机制有利于依据优势种群降低事务表大小.

(2) 带能耗改进标注频繁模式集表的挖掘和更新

算法 1 第 5 行挖掘获取初始频繁选项模式集表 $TPSFO_E^+$, 而算法 1 第 17 行至第 18 行完成在每代对模式集表 $TPSFO_E^+$的更新. $TPSFO_E^+$的具体挖掘方法将在第 2 节予以详细阐述. 相比于 GA-FP, GA-MFPM 在当前代事务表中挖掘出的频繁选项模式可较好地保持频繁选项模式的时效性.

(3) 启发式多点变异方法

算法 1 的第 11 行给出了基于模式集表 $TPSFO_E^+$ 并通过设计的最大匹配算法对个体 X_k 进行多点变异. 较 GA-FP 的单点启发式变异, GA-MFPM 的多点变异方法有利于提高收敛速度. 最大频繁模式匹配帮助的多点变异将在第 3 节进行重点阐述.

2 挖掘带能耗标注的频繁选项模式集

下面先阐述带能耗改进频繁选项模式树 FP_E 的构建, 然后给出挖掘带能耗改进标注频繁选项模式相关的定义, 最后结合例子描述带能耗标注频繁选项模式的挖掘算法 FP_E-growth$^+$.

2.1 带能耗改进频繁选项模式树 FP_E 的构建

GA-MFPM 中 FP_E 树的数据结构及其构建过程与 GA-FP 类似. 依照表 1 事务表并在最小支持度计数 C_{min}=3 下, 可构建出图 2 所示的 FP_E 树. 为了便于理解本节内容, 下面对 FP_E 树的数据结构作一简要介绍.

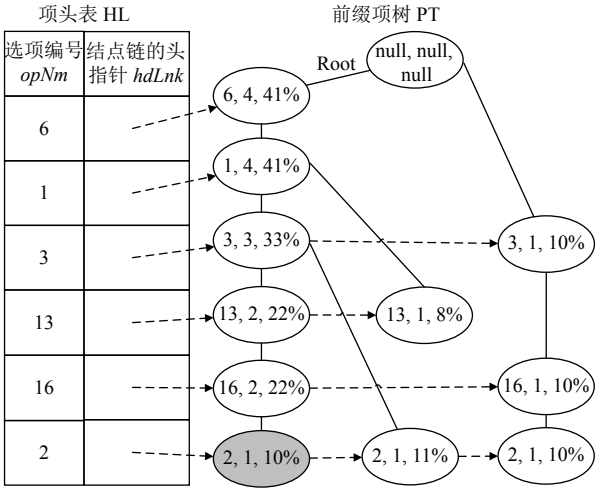

图 2 基于表 1 例子事务表 TT_E 生成的 FP_E 树

FP_E 树由前缀项树 PT 和项头表 HL 所构成. PT 树中各结点用选项编号 *opNm*、出现次数 *count*、能耗标注 *engAno*、指向父结点的指针 *parNode* 和指向下一个具有相同选项编号结点的指针 *nextNode* 五个域进行描述. 在图 2 椭圆形结点中, 用逗号分隔的是 *opNm*、*count* 和 *engAno* 的值, 而各结点的 *parNode* 和 *nextNode* 的值分别由实线和虚线弧隐含表示. 项头表

HL 中各表项由频繁选项编号 *opNm* 和结点链的头指针 *hdLnk* 两个属性表示. *hdLnk* 可将相同选项通过结点链链接起来. 图 2 中项头表 HL 最后一行的 *hdLnk* 可将所有的 2 号选项结点链接起来.

基于 FP_E 树, 可挖掘出所有带能耗标注的频繁选项模式. 下一小节将给出相关的定义.

2.2 挖掘带能耗改进标注频繁选项模式相关的定义

定义 1. 前缀路径. 若 *node* 为 FP_E 树中的一个非根结点, 则从 *root* 根节点到 *node* 结点之间的一条路径称为 *node* 的前缀路径. 而 *node* 结点称为该前缀路径的后缀结点.

前缀路径可用一个结点序列进行表示. 图 2 中灰色背景结点为后缀的前缀路径 $path_1$ 可用式 (1) 的结点序列表示.

$$path_1 =< (6,4,41\%),(1,4,41\%),(3,3,33\%),(13,2,22\%),(16,2,22\%) > \quad (1)$$

定义 2. 条件前缀路径. 设 *path* 是后缀结点 *node* 的一条前缀路径. 若将 *path* 上各结点的支持计数和能耗标注分别修改为 *node* 的支持计数和能耗标注, 而获得的路径被称为 *node* 的条件前缀路径.

条件前缀路径用于表示前缀路径上各结点对应选项与后缀结点对应选项共同出现的次数及对应的能耗标注. 例如: $path_1$ 为图 2 灰色背景结点 *node* 的前缀路径, 用 *node* 的支持计数 1 和能耗标注 10% 分别更新 $path_1$ 上各结点对应值, 可得到式 (2) 表示的 *node* 的条件前缀路径 $path_2$. 它表示选项集{6,1,3,13,16}共同出现 1 次且能耗标注为 10%.

$$path_2 =< (6,1,10\%),(1,1,10\%),(3,1,10\%),(13,1,10\%),(16,1,10\%) > \quad (2)$$

定义 3. 频繁选项的条件前缀路径集. 若 *nodes* 是 FP_E 树中所有 *i* 号频繁选项对应的结点集, 以 *nodes* 中各结点为后缀所得条件前缀路径的集合, 称为选项 *i* 的条件前缀路径集, 记为 *paths*|*i*.

FP_E 树中 *i* 号频繁选项对应的全部结点可沿结点链的头指针 *hdLnk* 遍历获取. 图 2 中 2 号选项对应的结点集, 可通过头表 HL 最后一个表项的头指针 *hdLnk* 遍历结点链得到. 图 2 中 2 号选项的条件前缀路径集由 $path_2$, $path_3$ 和 $path_4$ 所组成. 式 (2)~式 (4) 分别给出了 $path_2$~$path_4$ 的表示. 它们分别对应结点链中第 1~3 个结点的条件前缀路径.

$$path_3 =< (6,1,11\%),(1,1,11\%),(3,1,11\%) > \quad (3)$$

$$path_4 =< (3,1,10\%),(16,1,10\%) > \quad (4)$$

定义 4. 条件事务表. 以 *i* 号频繁选项的条件前缀路径集 *paths*|*i* 中各路径为事务所构建的事务表, 称为 *i* 号频繁选项的条件事务表, 记作 $TTF_E|i$.

按图 2 中 2 号频繁选项的条件前缀路径集{$path_2$, $path_3$, $path_4$}可构建表 2 所示的条件事务表.

表 2 图 2 中 2 号频繁选项的条件事务表

事务标识 TID	带能耗标注的选项事务 T_E, T_E={<*opInfo*>|*opInfo* =(选项编号 *opNm*, 出现次数 *count*, 能耗改进标注 *engAno*)}
1	<(6, 1, 10%), (1, 1, 10%), (3, 1, 10%), (13, 1, 10%), (16, 1, 10%)>
2	<(6,1, 11%), (1, 1, 11%), (3, 1, 11%) >
3	<(3,1, 10%), (16,1, 10%)>

定义 5. 条件 FP_E 树. 以 *i* 号频繁选项的条件事务表 $TT_E|i$ 作为输入所构建的 FP_E 树被称为频繁选项 *i* 的条件 FP_E 树, 记为 $FP_E|i$.

按照表 2 的 2 号频繁选项条件事务表并在最小支持计数 C_{min} =3 下, 可构建出图 3 所示的条件 FP_E 树.

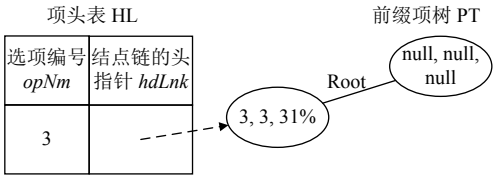

图 3 FP_E-growth$^+$以图 2 中 2 选项为后缀构建的条件 FP_E 树

定义 6. 频繁选项模式. 若 S_{op} 为频繁选项的编号集, *count* 和 *engAno* 分别为 S_{op} 中各选项共同出现的次数及对应的能耗标注, 则频繁选项模式 fop_E^+可用三元组 (S_{op}, *count*, *engAno*) 进行描述. 其中: S_{op} 的大小被称为 fop_E^+的项数, 共同出现次数 *count* 应不小于最小支持计数 C_{min}, 而 *engAno* 由式 (5) 定义. 式 (5) 中的函数 $\Pi(T_E)$ 表示从事务表 TTF_E 的一条事务 T_E 中投影出所选用选项的编号集, 而函数 $engImpr(T_E)$ 表示事务 T_E 的能耗改进百分比值. 当 fop_E^+的项数为 *k* 时, 用 $fop_E^{+(k)}$ 进行表示.

$$engAno = \sum_{T_E \in TT_E \wedge S_{op} \subseteq \Pi(T_E)} engImpr(T_E) \quad (5)$$

对比算法 GA-FP 的频繁选项模式仅给出单个频繁选项的能耗标注, 而 GA-MFPM 则定义了多个频繁

选项同时选用的次数和能耗标注. 因而, GA-MFPM 包括了新的挖掘算法.

2.3 带能耗标注频繁选项模式的挖掘算法 FP_E-growth$^+$

FP_E-growth$^+$算法基于 FP_E 树并采用模式项数渐增的递归挖掘方法, 其描述如算法 2 所示. FP_E-growth$^+$算法包括递归出口处理和循环递归处理两部分, 分别如算法 2 第 2 行至第 8 行、第 10 行至 18 行.

算法 2. FP_E-growth$^+$算法

输入: FP_E 树, 后缀结点序列 *postFixNodes*

输出: 频繁选项模式集表 $TPSFO_E^+$//全局变量, 初始为空

1) If (FP_E 树为单分支树)

2) 依据单分支路径上非根结点的序列, 通过枚举获得 所有非空结点序列的集合 $S_{nodeSeq}$;

3) For (对 $S_{nodeSeq}$ 中的每个元素 *nodeSeq*)

4) 将 *nodeSeq* 作为头部连接到 *postFixNode* 形成新序列 *newNodeSeq*;

5) 计算 *newNodeSeq* 中各结点支持计数和能耗标的最小值 *minCount* 和 *minEngAno*;

6) 用 *minCount* 和 *minEngAno* 更新 *newNodeSeq* 中各结点的支持计数和能耗标注;

7) 基于 *newNodeSeq* 生成频繁选项模式 fop_E^+, 并根据 fop_E^+的项数 *k* 将其输出到模式集表 $TPSFO_E^+$的 *k* 频繁选项模式集中;

8) End For

9) Else

10) For (*row* 从 FP_E 树项头表 HL 的尾行至头行) Do

11) 依据 *row.hdLink* 遍历获取频繁选项 *row.opNm* 的各结点并累计它们的支持计数和能耗标注, 进而获取频繁选项 *row.opNm* 的支持计数 *count* 和能耗标注 *engAno*;

12) 以 *row.opNm*, *count* 和 *engAno* 为选项编号值、支持计数值和能耗标注值, 构建新结点 *node*;

13) 将 *node* 作为第 1 个元素连接到 *postFixNodes*, 并将 *postFixNodes* 中各结点的支持计数值和能耗标注值分别更新为 *count* 和 *engAno*;

14) 基于 *postFixNodes* 生成频繁选项模式 fop_E^+, 并根据 fop_E^+的项数 *k* 将其输出到模式集表 $TPSFO_E^+$的 *k* 频繁选项模式集中;

15) 构造条件事务表 $TTF_E|$ *row.opNm*;

16) 基于条件事务表 $TTF_E|$*row.opNm*, 构造条件 FP_E 树 ($FP_E|$*row.opNm*);

17) FP_E-growth$^+$($FP_E|$*row.opNm*, *postFixNodes*);

18) End For

19) End If

20) 结束算法运行.

FP_E-growth$^+$算法的递归出口条件是当 FP_E 树为单分支的树. 当满足出口条件时, 首先依据单分支路径上非根结点的序列, 枚举获得所有非空结点序列的集合 $S_{nodeSeq}$; 然后对 $S_{nodeSeq}$ 中每个元素 *nodeSeq* 重复做以下步骤: 将 *nodeSeq* 作为头部连接到后缀结点序列 *postFixNodes* 形成新序列 *newNodeSeq*、计算序列 *newNodeSeq* 中各结点支持计数和能耗标注的最小值, 并用它们更新 *newNodeSeq* 序列中各结点的支持计数和能耗标注的值、基于 *newNodeSeq* 序列生成频繁选项模式 fop_E^+并基于项数将 fop_E^+存入模式集表的对应位置.

循环递归部分是依次对 FP_E 树项头表 HL 的尾行至头行进行递归处理, 如算法 2 第 10 行至 17 行. 递归调用前需要构建新后缀结点序列 *postFixNodes* 并输出频繁选项模式、构建条件事务表、构建条件 FP_E 树. FP_E-growth$^+$算法在挖掘频繁选项时, 采用与经典 FP 挖掘算法[18]相同的递归过程挖掘获取频繁选项. 仅在挖掘过程中增加了频繁选项的能耗标注的计算, 如算法 2 第 5 行和第 13 行. 而该步骤时间复杂性为 O(1), 故 FP_E-growth$^+$算法与经典 FP 算法相同.

图 2 所示 FP_E 树是一棵多分支的树, 在利用 FP_E-growth$^+$算法进行挖掘时, 由于不满足递归出口条件, 需要进入循环递归处理过程. 在循环中, 首先要以图 2 项头表最后一行的 2 号选项为后缀和空的后缀结点序列进行挖掘. 下面以此场景为例对 FP_E-growth$^+$算法进行解释.

(1) 递归调用的处理

1) 构建新的后缀结点序列并输出频繁选项模式

沿图 2 头表最后一行的结点链头指针进行遍历可得 2 号频繁选项的支持计数和能耗标注分别为 3 和 31%, 进而可构造一个新结点 *node*(2, 3, 31%). 将 *node* 连接到当前后缀结点序列 *postFixNodes* 时, 由于 *postFixNodes* 为空序列, 故连接后得到的 *postFixNodes*=<(2, 3, 31%)>. 根据 *postFixNodes* 生成的频繁选项模式 $fop_E^{+(1)}$=({2}, 3, 31%), 可将其输出到模式集表 $TPSFO_E^+$的 1 频繁选项集中.

2) 构建条件事务表

2 号频繁选项为后缀的条件事务表 $TT_E|$ 2 已在定义 4 中的例子给出, 如表 2 所示.

3) 构建条件 FP_E 树

根据条件事务表 $TT_E|2$ 构建条件频繁模式树 $FP_E|$2 已在定义 5 中的例子给出, 如图 3 所示. 以 $FP_E|2$ 和 *postFixNodes*=<(2, 3, 31%)>递归调用 FP_E-growth$^+$时, 由于 $FP_E|2$ 树为一单路径树满足递归出口条件, 进行递

归出口处理.

(2) 递归出口中的处理

1) 枚举获得所有非空结点序列的集合 $S_{nodeSeq}$

由于图 3 的 FP_E|2 树中仅含一个非根结点 *node*(3, 3, 31%), 故枚举生成的 $S_{nodeSeq}$={<(3, 3, 31%)>}.

2) 对 $S_{nodeSeq}$ 中各元素 *nodeSeq* 重复执行以下 3 个步骤:

① *nodeSeq* 与后缀结点序列连接获取新序列 *newNodeSeq*

$S_{nodeSeq}$ 中仅包含的一个序列与 *postFixNodes*=<(2, 3, 31%)>连接后生成新序列 *newNodeSeq*={<(3, 3, 31%)>, (2, 3, 31%)}.

② 更新序列 *newNodeSeq* 中各结点支持计数和能耗标注

由于 *newNodeSeq* 各结点的支持计数和能耗标注相同, 故更前新后的 *newNodeSeq* 序列相同.

③ 依据 *newNodeSeq* 序列生成频繁选项模式并输出

根据序列 *newNodeSeq*={<(3, 3, 31%)>, (2, 3, 31%)}可生成频繁选项模式 $fop_E^{+(2)}$=({3, 2}, 3, 31%), 可将其输出到模式集表 $TPSFO_E^+$的 2 频繁选项集中.

以图 2 的 FP_E 树和空的后缀结点序列为参数, 调用 FP_E-growth$^+$算法最终输出表 3 所示的模式集表 $TPSFO_E^+$.

表 3 例子带能耗标注的频繁选项模式集表 $TPSFO_E^+$

k 频繁选项模式集 (记 $SFOP_E^{+(K)}$)	*k* 频繁选项模式 ($fop_E^{+(K)}$)
1 频繁选项模式集 $SFOP_E^{+(1)}$	**({2},3, 31%)** ({16},3, 32%) ({13},3, 30%) ({3},4, 43%) ({1},4, 41%) ({6},4, 41%)
2 频繁选项模式集 $SFOP_E^{+(2)}$	**({3,2},3, 31%)** ({3,16},3, 32%) ({1,13},3, 30%) ({6, 13},3, 30%) ({1, 3},3, 33%) ({6,3},3, 33%) ({6,1},4, 41%)
3 频繁选项模式集 $SFOP_E^{+(3)}$	({6, 1, 13},3, 30%) ({6, 1, 3},3, 33%)

3 最大频繁选项模式匹配帮助的多点变异

不同于 GA-FP 中的单点变异, GA-MFPM 采用最大频繁选项模式匹配帮助的多点变异. 多点变异的流程如算法 3 所示. 其主要包括确定变异的位数和位置、最大频繁选项模式匹配、修改变异位值等 3 个步骤. 下面以图 4 个体 X_2={1,1,0,0,0,0,0,0,0,0,0,1,1,0,1,1}和表 3 的为例, 阐述最大频繁选项模式匹配帮助的多点变异操作流程.

算法 3. 最大频繁选项模式匹配帮助的多点变异操作流程

输入: 个体 *X*, 频繁选项模式集表 $TPSFO_E^+$

输出: 变异后个体 *X*′

1) 将存放最大匹配获取的频繁模式集 S_{fop} 置空;
2) *k*←模式集表 $TPSFO_E^+$中频繁选项模式的最大项数;
3) 在 (0, *k*]上随机产生变异的位数 *mutaBit*;
4) 随机生成 *mutaBit* 个不重复的 *X* 的下标, 并放入起始的选项编号集 S_0;
5) 在 $TPSFO_E^+$的 1 频繁选项模式集 $SFOP_E^{+(1)}$ 中查找编号集 S_0 中的各个元素, 并将查找到的元素放入选项编号集 S_1 中;
6) 以 $TPSFO_E^+$和 S_1 为输入调用算法 4 的最大匹配算法, 得到匹配的频繁模式集 S_{fop};
7) *X*′←*X*, 并以编号集 S_0 中各元素值为下标, 将 *X*′的对应位的值设为 0;
8) If (S_{fop} 不空) Then
9) 按 S_{fop} 中各元素的平均能耗标注 (*engAno*/*count*), 依概率选择其中一个频繁模式 fop_E^+;
10) 以 fop_E^+的选项编号集中各元素值为下标, 将 *X*′对应位的值设为 1;
11) End If
12) 输出 *X*′, 并结束算法运行.

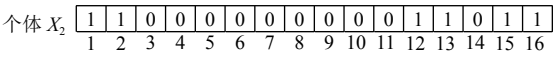

图 4 待变异的个体 X_2

(1) 确定变异的位数和位置

确定变异的位数和位置如算法 3 的第 2 行至第 4 行. 为了能基于模式集表 $TPSFO_E^+$提供的启发式信息进行变异, 规定最多变异的位数不超过 $TPSFO_E^+$中频繁选项模式的最大项数. 利用产生随机数完成变异的位数和位置的确定. 每个变异位置对应于一个选项编号, 进而得到一个选项编号集 S_0, 假设例中得到的 S_0={2, 1, 6}.

(2) 最大频繁选项模式匹配

最大频繁选项模式匹配是指将给定的选项编号集 S_0 与 $TPSFO_E^+$表中各模式的选项编号集进行最大匹配. 通过最大匹配可发现编号集 S_0 中频繁出现在优势个体中最长的编号集.

算法 3 第 5 行至第 10 行描述了最大频繁选项模式匹配的匹配过程:

① 首先基于频繁选项模式所具有的向下闭包性质[18] (*k* 频繁选项集的任何子集都应是频繁的), 在 $TPSFO_E^+$表的 1 频繁选项模式集 $SFOP_E^{+(1)}$ 中, 将各模式的 1 个编号与 S_0 中的编号进行匹配, 并将匹配的编号放入 S1 中, 进而获取 S_0 中频繁的选项编号, 例中获取的 S_1={2,1,6}.

② 然后调用算法 4 的最大频繁模式匹配算法 MFPM. MFPM 算法将输入的选项编号集赋给 S_{temp} 后, 从大小为$|S_{\text{temp}}|$的频繁选项模式集 $SFOP_E^{+(|S_{\text{temp}}|)}$ 开始对 S_{temp} 进行最大匹配, 若能匹配上则将匹配结果放入结果频繁选项模式集 S_{fop} 中并退出匹配过程, 否则在 $SFOP_E^{+(S_{\text{temp}}-1)}$ 中进行最大匹配; 直至 S_{temp} 为空结束匹配. 例中首先在 3 频繁选项模式集上对 S_{temp}={2, 1, 6}进行第 1 轮最大匹配, 没有任何匹配的频繁模式. 因而在 2 频繁选项模式集上对 S_{temp}={2,1,6}进行第 2 轮最大匹配, 获取了频繁模式集 S_{fop} ={ ({6,1},4, 41%)}.

算法 4. 最大频繁选项模式匹配算法 MFPM

输入: 频繁选项模式集表 $TPSFO_E^+$, 匹配的选项编号集 S_{opNm}

输出: 最大匹配的频繁选项模式集 S_{fop}

1) 临时选项编号集 S_{temp}←S_{opNm};
2) 匹配的频繁选项模式集 S_{fop} 置空;
3) While (S_{temp} 不空) Do
4) i←S_{temp} 的大小;
5) $TPSFO_E^+$的$|S_{\text{temp}}|$频繁模式集中查找选项编号集等于 S_{temp} 的模式 fop_E^+, 若找到将 fop_E^+放入 S_{fop} 中, 退出 While 循环;
6) For (j 从 i 递减到 1)
7) 从 S_{temp} 中删除第 j 个元素, 生成新的选项编号集 S_{temp}';
8) 在模式集表 $TPSFO_E^+$的$|S_{\text{temp}}'|$频繁模式集查找模式的选项编号集等于 S_{temp}', 若找到将该模式放入 S_{fop} 中, 并退出 For 循环;
9) End For
10) If (S_{fop} 不空) Then
11) 退出 While 循环;
12) Else
13) 删除 S_{temp} 的最后一个元素;
14) i=i–1;
15) End If
16) End While
17) 输出频繁选项模式集 S_{fop}, 并结束算法运行.

(3) 修改变异位值

算法 3 第 7 行至第 11 行负责完成个体中变异位值的修改. 首先, 将个体 X_2 赋给 X_2', 并以 S_0 (存放随机生成的各变异位置) 中各元素值为下标, 将 X_2'对应位的值设为 0. 例中由 S_0={2, 1, 6}, 可知 X_2'的第 2, 1, 6 位上的值均设为 0. 然后, 若基于最大匹配得到频繁模式集 S_{fop} 不空, 则按 S_{fop} 中各元素的平均能耗标注 (*engAno*/*count*), 依概率选择其中一个频繁模式 fop_E^+, 并以 fop_E^+的选项编号集中各元素值为下标, 将 X_2'对应位的值设为 1.

例中最大匹配获取的 S_{fop} ={({6,1},4, 41%)}不为空, 因 S_{fop} 只有一个频繁二项集, 则 fop_E^+={{6,1},4, 41%)}因而再将 X_2'第 1 和第 6 位上的值设定为 1. 最终 X_2'应如图 5 所示, 本例中编号 2 和 6 位上的值发生变异, 即 2 号选项由选用变为不选用, 而 6 号选项由不选用变为选用.

个体 X_2'	1	0	1	0	0	1	0	0	1	1	1	0	1	1	0	1
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

图 5 变异后的个体 X_2'

4 案例研究

本节给出了案例研究. 4.1 节简介了实验案例; 4.2 节提出了要验证的问题及度量标准; 4.3 节说明了实验中使用的统计方法; 4.4 节介绍了实验安装; 4.5 节展示了实验结果并进行了分析.

4.1 案例简介

本文从最新的 BEEBS 平台[6]中选用了 8 个案例, 如表 4 所示, BEEBS 是目前最大的用于嵌入式系统执行时间和能耗优化的开源基准平台. 综合考虑以下 3 个要素, 从 BEEBS 所包含的 84 个案例中选取了具有代表性的 8 个案例.

表 4 实验案例的应用领域、源代码的结构特性和操作特性

案例	应用领域	源代码的结构特性	源代码的操作特性			
			整型运算强度	浮点运算强度	分支频度	内存访问频度
CRC32	网络、通讯	循环	√		√	
Cubic	汽车			√		√
Dijkstra	网络	嵌套循环			√	√
FDCT	消费		√			
Float_Matrix	汽车、消费	数组访问, 循环	√			
Int_Matrix	汽车	数组访问, 循环		√		
Rijndael	安全	算术类型	√		√	√
SHA	安全、网络	算术类型	√			√

(1) 源代码的操作特性: 案例涵盖尽可能影响运行时能耗的各种不同操作. 如整型运算强度、浮点运算

强度、分支频度和内存访问频度等.

(2) 应用领域: 案例覆盖尽可能多的领域. 如汽车、消费、网络、安全和通讯等.

(3) 源代码的结构特性: 由于基准测试的编译和大量的编译选项的存在会极大地影响上述案例的编译效率, 所以在基准测试源代码中也会有一系列影响编译过程的特性. 其中包括: 循环、嵌套循环、不同的算术类型 (8 位、16 位和 32 位整型、浮点型等)、函数调用、字符串操作、位操作和数组访问. 表 4 给出了选用的 8 个案例的应用领域、源代码的结构特性和操作特性.

4.2 研究问题及其度量指标

问题 1. 解质量: 本文 GA-MFPM 算法较 GA-FP 算法能否得到更优的编译选项组合, 使得案例的运行能耗更低? GA-FP 是目前以能耗为优化目标并可获取较优编译选项组合的一种算法. 通过回答这一问题, 可以验证 GA-MFPM 算法的有用性.

度量指标: 将 GA-MFPM 和 GA-FP 最优解对应的能耗相对改进百分比 $I_{\triangle eng\%}$ 作为度量指标. 在案例源代码 Sft_{src} 下, $I_{\triangle eng\%}$ 的定义如式 (6) 所示, 其值越大越好. 在式 (6) 中, $X^*_{\text{GA-FP}}$ 和 $X^*_{\text{GA-MFPM}}$ 分别表示 GA-FP 和 GA-MFPM 算法获得的最优解. $f(Sft_{\text{src}},X^*_{\text{GA-MFPM}})$ 和 $f(Sft_{\text{src}},X^*_{\text{GA-FP}})$ 分别表示在案例源代码 Sft_{src} 下 GA-FP 和 GA-MFPM 算法最优解相对于-O0 等级的能耗改进百分比.

$$I_{eng\%}(Sft_{\text{src}})=\frac{f\left(Sft_{\text{src}},X^*_{\text{GA-MFPM}}\right)-f\left(Sft_{\text{src}},X^*_{\text{GA-FP}}\right)}{f\left(Sft_{\text{src}},X^*_{\text{GA-FP}}\right)} \tag{6}$$

问题 2. 收敛速度: 与 GA-FP 算法比, GA-MFPM 算法能否加快收敛速度? 通过回答这一问题进一步验证 GA-MFPM 算法的有效性.

度量指标: 为了公平比较两种算法的收敛速度, 将 GA-MFPM 相对于 GA-FP 首达最优时间的减少百分比 $I_{\triangle t\%}$ 作为度量指标. 在案例源代码 Sft_{src} 下, $I_{\triangle t\%}$ 的定义如式 (7) 所示, 其值越大越好. 式 (7) 中 $MinT_{\text{GA-FP}}(Sft_{\text{src}},X^*)$ 和 $MinT_{\text{GA-MFPM}}(Sft_{\text{src}},X^*)$ 分别表示在案例源代码 Sft_{src} 下 GA-FP 和 GA-MFPM 首达最优解 X^* 所需要的时间.

$$I_{\Delta t\%}(Sft_{\text{src}})=\frac{MinT_{\text{GA-FP}}(Sft_{\text{src}},X^*)-MinT_{\text{GA-MFPM}}(Sft_{\text{src}},X^*)}{MinT_{\text{GA-FP}}(Sft_{\text{src}},X^*)} \tag{7}$$

4.3 使用的统计方法

由于演化算法具有随机性, GA-MFPM 和 GA-FP 算法被分别独立运行 20 次, 并进行统计检验. 为了对实验数据进行统计分析, 本文采用了 Wilcoxom 秩和检验[19]方法, 并将置信水平 α 的值设置为 0.05. 为了进一步观测对比数据的差异程度 (effect size), 本文还使用 Vargha-Delaney[19]的 $\hat{A}_{12}$ 作为差异程度的直观度量. $\hat{A}_{12}$ 的值域为[0, 1], 其值越大说明差异程度越大.

4.4 实验安装

(1) 实验环境

上位机的运行环境: Intel(R) Core(TM)i5-4590, 3.30 GHz 处理器, 8 GB 内存及 Ubuntu-16.04.1 操作系统.

能耗评估系统: 基于 STM32F4 板自主研发.

案例软件优化的运行环境: STM32F103 板.

编译器和编译选项: GCC4.9.2, 并选用与 GA-FP[15] 相同的 58 个编译选项.

(2) 算法参数的设定

为了保持公平, 尽可能让 GA-MFPM 采用与 GA-FP 相同的参数设定, 如表 5 所示. 但两个算法的最小支持计数设定有所不同. 这是因为 GA-MFPM 采用逐代替换事务表的策略. 在事务表中, 若每个选项按随机方式选用, 则每个选项出现的次数应为种群大小 N 的一半. 而 GA-MFPM 期望跟踪各代种群中优势个体选用选项的特点, 故将 $C_{\min}$ 设定为 $0.6\times N$. 而不是 GA-FP 中设定的 0.1 倍全局事务表大小.

表 5 GA-FP 和 GA-MFPM 的参数设定

参数	GA-FP	GA-MFPM
种群大小	100	100
最大演化代数	50	50
变异概率	0.3	0.3
交叉概率	0.6	0.6
最小支持计数	0.1 倍事务表大小	0.6 倍事务表大小

4.5 实验结果及分析

下面具体给出各问题的实验结果及分析.

(1) 问题 1(解质量) 的实验结果及分析.

表 6 给出了 GA-MFPM 在这 8 个案例下对比 GA-FP 的能耗改进百分比 ($I_{\triangle eng\%}$) 的秩和检验结果. 从表 6 中第 2 列可知 p-value 值都远比置信水平 0.05 要小, 这表明 GA-MFPM 的 $I_{\triangle eng\%}$ 指标在这 8 个案例下的统计意义显著比 GA-FP 的好. 从表 6 中第 3 列可看出在这

8 个案例下, GA-MFPM 算法的 effect size 值以很明显的优势优于 GA-FP. 而从图 6 所示的统计盒图也能直观反映出一致结论. 从表 7 的统计量结果能知道: 8 个案例下的 $I_{\triangle eng\%}$ 指标平均值为 2.4%, 最大 $I_{\triangle eng\%}$ 指标值可达 16.1%.

表 6 GA-MFPM 较 GA-FP 在 8 个案例下能耗改进百分比 ($I_{\triangle eng\%}$) 的秩和检验结果

案例	P-value	Effect size
CRC32	4.843e–06	0.92
Cubic	3.561e–04	0.83
Dijstra	8.072e–06	0.91
FDCT	1.147e–07	0.99
Float_Matrix	5.874e–06	0.92
Int_Matrix	2.062e–07	0.98
Rijndael	2.170e–05	0.89
Nettle-SHA256	2.873e–06	0.93

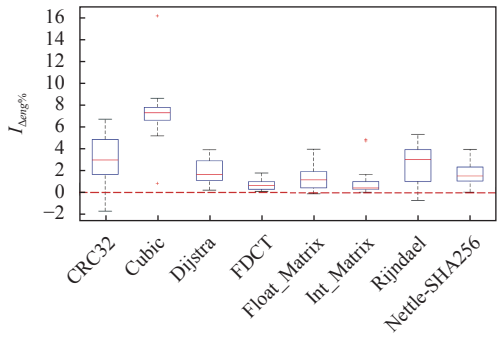

图 6 GA-MFPM 较 GA-FP 在 8 个案例下能耗改进百分比 ($I_{\triangle eng\%}$) 的统计盒图

表 7 GA-MFPM 较 GA-FP 在 8 个案例下能耗改进百分比 ($I_{\triangle eng\%}$) 的统计量结果

案例	$I_{\triangle eng\%}$ 均值 (%)	$I_{\triangle eng\%}$ 最大值 (%)	$I_{\triangle eng\%}$ 最小值 (%)	$I_{\triangle eng\%}$ 方差
CRC32	3.0	6.7	–1.7	5.4e–4
Cubic	7.4	**16.1**	0.8	7.2e–4
Dijstra	1.9	3.9	0.2	1.1e–4
FDCT	0.7	1.8	0.1	2.6e–5
Float_Matrix	1.3	3.9	–0.1	1.0e–4
Int_Matrix	1.0	4.8	0.03	1.9e–4
Rijndael	2.5	5.3	–0.7	3.7e–4
Nettle-SHA256	1.7	3.9	0.03	2.8e–4
均值	2.4	5.8	–0.17	2.9e–4

GA-MFPM 较 GA-FP 获取更好解质量的原因是: GA-MFPM 在逐代挖掘的频繁模式中保存了多个频繁选项, 同时标注选用次数和能耗, 可获取更完整、时效性更好的启发式信息, 有利于帮助提高解质量.

(2) 问题 2(收敛速度) 的实验结果及分析.

表 8 给出了在 8 个案例下收敛速度指标 $I_{\triangle t\%}$ (相对于 GA-FP, GA-MFPM 首达最优解耗时的相对减少百分比) 的秩和检验结果. 表 8 中第 2 列 p-value 值都远比置信水平 0.05 要小, 表明 GA-MFPM 的 $I_{\triangle t\%}$ 指标在 8 个案例下的统计意义显著比 GA-FP 的好. 表 8 中 GA-MFPM 首达最优解较 GA-FP 在大概率上需要更少的时间. 表 9 可知: 8 个案例下的 $I_{\triangle t\%}$ 指标平均值为 57.6%, 最大达到了 97.5%. 图 7 所示的统计盒图也能直观反映出一致结论.

表 8 在 8 个案例下收敛速度指标 $I_{\triangle t\%}$ (GA-MFPM 相对于 GA-FP 首达最优解耗时的减少百分比) 秩和检验结果

案例	P-value	Effect size
CRC32	2.343e–04	0.84
Cubic	1.505e–05	0.90
Dijstra	1.525e–04	0.85
FDCT	1.7e–05	0.9
Float_Matrix	7.838e–05	0.87
Int_Matrix	3.736e–06	0.93
Rijndael	4.416e–05	0.88
Nettle-SHA256	4.255e–06	0.92

表 9 在 8 个案例下收敛速度指标 $I_{\triangle t\%}$ (GA-MFPM 相对于 GA-FP 首达最优解耗时的减少百分比) 的统计量结果

案例	$I_{\triangle t\%}$ 均值	$I_{\triangle t\%}$ 最大值	$I_{\triangle t\%}$ 最小值	$I_{\triangle t\%}$ 方差
CRC32	50	86.8	0.8	7.8e–2
Cubic	59.1	96.0	1.6	7.4e–2
Dijstra	41.8	78.8	–6.0	7.4e–2
FDCT	61.8	**97.5**	2.0	5.8e–2
Float_Matrix	57.1	90.5	2.1	7.3e–2
Int_Matrix	61.1	95	–4.3	6.5e–2
Rijndael	58.9	96.7	–8.4	8.4e–2
Nettle-SHA256	70.6	90.1	47.6	1.6e–2
均值	57.6	91.4	4.4	6.5e–2

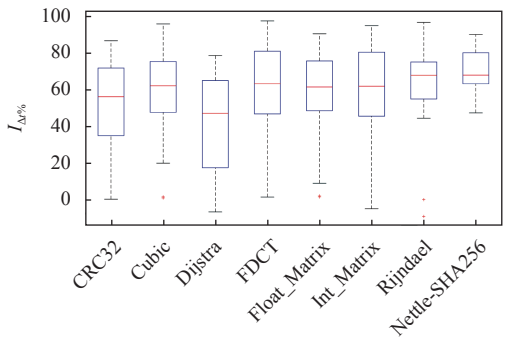

图 7 在 8 个案例下收敛速度指标 $I_{\triangle t\%}$ (GA-MFPM 相对于 GA-FP 首达最优解耗时的减少百分比) 的统计盒图

GA-MFPM 较 GA-FP 获取更好的收敛速度的原因在于: GA-MFPM 逐代替换参考点的机制可有效降低事务表的大小、借助最大频繁模式匹配帮助的多点变

异可提高变异效率, 进而有利于帮助提高收敛速度.

5 结语

本文将频繁模式挖掘和最大频繁模式匹配帮助的多点变异引入到传统遗传算法, 提出了一种用于 GCC 编译时能耗演化优化算法 GA-MFPM. GA-MFPM 采用逐代替换判定是否有能耗改进的参考点、逐代构建事务表和逐代挖掘的策略; 在此基础上提出可获取更多启发式信息的频繁编译选项挖掘算法; 进一步设计了最大频繁模式匹配帮助的多点变异方法. 通过案例研究实证了 GA-MFPM 的解质量和收敛速度在统计意义上显著优于已有的 GA-FP 算法. 未来我们将引入代理模型解决 GA-MFPM 算法中目标值计算耗时长的问题.

参考文献

1 Georgiou K, Xavier-de-Souza S, Eder K. The IoT energy challenge: A software perspective. IEEE Embedded Systems Letters, 2018, 10(3): 53–56. [doi: 10.1109/LES.2017.2741419]

2 Pallister J, Hollis SJ. Bennett J. Identifying compiler options to minimize energy consumption for embedded platforms. The Computer Journal, 2015, 58(1): 95–109. [doi: 10.1093/comjnl/bxt129]

3 Ashouri AH, Killian W, Cavazos J, *et al*. A survey on compiler autotuning using machine learning. ACM Computing Surveys, 2019, 51(5): 96.

4 Petke J, Haraldsson SO, Harman M, *et al*. Genetic improvement of software: A comprehensive survey. IEEE Transactions on Evolutionary Computation, 2018, 22(3): 415–432. [doi: 10.1109/TEVC.2017.2693219]

5 Using the GNU Compiler Collector (GCC). https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

6 Pallister J. Exploring the Fundamental Differences between Compiler Optimisations for Energy and for Performance[Ph. D. Thesis]. Bristol: University of Bristol, 2016.

7 Nobre R, Reis L, Cardoso JMP. Compiler phase ordering as an orthogonal approach for reducing energy consumption. Proceedings of the 19th Workshop on Compilers for Parallel Computing. Valladolid, Spain. 2016.

8 Hoste K, Eeckhout L. Cole: Compiler optimization level exploration. Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. Boston, MA, USA. 2008. 165–174.

9 Wang Z, O'Boyle M. Machine learning in compiler optimization. Proceedings of the IEEE, 2018, 106(11): 1879–1901. [doi: 10.1109/JPROC.2018.2817118]

10 Leather H, Bonilla E, O'Boyle M. Automatic feature generation for machine learning based optimizing compilation. Proceedings of 2009 International Symposium on Code Generation and Optimization. Seattle, WA, USA. 2009. 81–91.

11 Ashouri AH, Mariani G, Palermo G, *et al*. A Bayesian network approach for compiler auto-tuning for embedded processors. Proceedings of 2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia. Greater Noida, India. 2014. 90–97.

12 Ashouri AH, Mariani G, Palermo G, *et al*. COBAYN: Compiler autotuning framework using Bayesian networks. ACM Transactions on Architecture and Code Optimization, 2016, 13(2): 21.

13 Lin SC, Chang CK, Lin NW. Automatic selection of GCC optimization options using a gene weighted genetic algorithm. Proceedings of 2008 13th Asia-Pacific Computer Systems Architecture Conference. Hsinchu, China. 2008. 1–8.

14 Garciarena U, Santana R. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion. Denver, CO, USA. 2016. 1159–1166.

15 倪友聪, 吴瑞, 杜欣, 等. 基于频繁模式挖掘的 GCC 编译时能耗演化优化算法. 软件学报, 2019, 30(5): 1269–1287. [doi: 10.13328/j.cnki.jos.005734]

16 Holland JH. Genetic algorithms. Scientific American, 1992, 267(1): 66–73. [doi: 10.1038/scientificamerican0792-66]

17 Hauschild M, Pelikan M. An introduction and survey of estimation of distribution algorithms. Swarm and Evolutionary Computation, 2011, 1(3): 111–128. [doi: 10.1016/j.swevo.2011.08.003]

18 Han JW, Pei J, Yin YW. Mining frequent patterns without candidate generation. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. Dallas, TX, USA. 2000. 1–12.

19 Arcuri A, Briand L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. Proceedings of 2011 33rd International Conference on Software Engineering. Honolulu, HI, USA. 2011. 1–10.