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

Technical Report No. NJU-SEG-2023-TR-004

2023-TR-004

基于多线程并行的符号执行引擎设计与实现

周彭,左志强

Technical Report 2023

计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220920
60(2): 248−261, 2023

基于多线程并行的符号执行引擎设计与实现

周　彭[1,2]　左志强[1,2]

[1](南京大学计算机科学与技术系　南京　210023)

[2](计算机软件新技术国家重点实验室(南京大学)　南京　210023)

(522022330105@Smail.nju.edu.cn)

Design and Implementation of a Parallel Symbolic Execution Engine Based on Multi-Threading

Zhou Peng[1,2] and Zuo Zhiqiang[1,2]

[1] (*Department of Computer Science and Technology*, *Nanjing University*, *Nanjing* 210023)

[2] (*National Key Laboratory for Novel Software Technology* (*Nanjing University*), *Nanjing* 210023)

Abstract　Symbolic execution, as an effective test generation technique, has been increasingly used for software testing and vulnerability discovery, etc. However, the number of execution paths in a program rises exponentially with the number of branches, and symbolic execution can hardly be performed efficiently on large-scale programs, leading to poor scalability. Considering that multi-processed parallel symbolic execution approaches not only suffer from high communication overhead, but also fail to use the existing constraint solving optimization techniqnes. To tackle the above problems, we devise a novel work-stealing algorithm that requires no centralized nodes. In addition, to exploit the existing constraint solving optimizations, we enable different worker nodes to share the constraint solving information so as to accelerate the solving process. Based on the above mechanisms, we implement our parallel engine PKLEE (parallel KLEE) on the top of KLEE and conduct the experimental evaluations on it. In the exhaustive execution path scenario, the average time consumed by KLEE is three to four times longer than that of PKLEE with 8 threads available, and the effective workload executed by PKLEE is on average three times higher than that of KLEE in the same period of time.

Key words　symbolic execution; load balance; constraint solving; parallel acceleration; scalability

摘　要　符号执行作为一种高效的测试生成技术,被广泛应用于软件测试、安全分析等领域.然而,由于程序中的执行路径数量随着分支数量的增加而指数级上升,符号执行往往无法在大规模程序上进行高效执行,缺乏可扩展性.已有的基于多进程的并行化方法具有较大的额外通信开销,并且缺乏对现有约束求解优化技术的利用.提出了基于多线程并行处理模型的符号执行加速方法.具体来讲,为解决并行符号执行中的不同工作节点负载不平衡问题,设计了不需要中间节点参与的工作窃取算法.为充分利用现有约束求解优化技术,提出了让不同工作节点共享约束求解信息的加速求解方法.基于符号执行引擎(KLEE)实现了多线程并行化符号执行方案,从而形成多线程并行化符号执行引擎(PKLEE).实验验证表明,在穷尽执行路径场景下,KLEE平均耗时是在给定8个线程下PKLEE的3~4倍;在同样的时间内,PKLEE执行的有效工作负载平均是KLEE的3倍.

收稿日期:2022−11−03;**修回日期**:2022−12−14

基金项目:国家自然科学基金项目(62272217)

This work was supported by the National Natural Science Foundation of China (62272217).

通信作者:左志强(zqzuo@nju.edu.cn)

关键词 符号执行；负载均衡；约束求解；并行加速；可扩展性

中图法分类号 TP311

符号执行(symbolic execution, SE)[1]是通过采用抽象的符号值代替具体输入分析程序执行情况的程序分析技术. 由于SE可以有效探索程序路径，并生成对应路径的输入，其被广泛应用于软件测试[2-3]、安全分析[4]等一系列重要领域. 经过多年的发展，工业界、学术界存在一系列相对成熟的符号执行工具，例如微软的SAGE[2]、斯坦福开源的KLEE[3]等.

现有的SE工具依然存在诸多问题. 如待分析程序中的循环、分支使得符号执行中产生指数级数量的路径，该问题被称作路径爆炸. 路径爆炸问题限制了SE的可扩展性. 此外，SE中需要通过SMT求解器判定大量路径条件(path condition, PC)，随着对SE的深入研究，约束求解的耗时也逐渐增加. 约束求解的求解效率也是影响符号执行可扩展性的一个关键因素.

针对路径爆炸以及约束求解开销过大的问题，研究人员提出了使用并行化的方式来提高SE的可扩展性. 已有的并行化SE工作[5]往往采用多进程并行模型，每个进程中单独运行一个SE引擎实例并独立探索不相交的程序状态空间，不同进程之间通过多进程通信原语来传递需要求解的路径等信息，不共享其他信息.

然而已有的基于多进程的并行化方法仍存在明显不足. 首先，传统的基于多进程的并行SE方法为了完成负载均衡，不仅需要引入极大可能成为性能瓶颈的中间节点，还需要获取新执行路径的进程重放已有的执行路径来还原执行路径的上下文. 此外，已有的顺序SE引擎(如KLEE)为了缓解求解约束条件的开销，引入了大量的约束求解优化技术，例如重复利用已经求解的约束来避免重复求解. 而已有的基于多进程的并行化SE由于路径的分散性，使得原有的约束求解优化技术难以发挥作用，这极大限制并行化加速的效果.

基于以上观察与分析，本文提出基于细粒度多线程并行的SE方法. 通过对SE算法进行细粒度多线程并行，从而规避进程间通信引入的高开销，同时利用多线程共享内存的便利性来复用已有的一系列约束求解优化技术. 本文以符号执行引擎(KLEE)作为基础，设计并实现了多线程并行化符号执行引擎(PKLEE).

本文的主要贡献包括4个方面:

1) 设计了一种基于多线程的并行符号执行方法(PKLEE)，通过多个工作线程并行进行SE求解，缓解路径爆炸问题.

2)设计了多线程场景下不需要中间协调节点的工作负载窃取方法，有效规避了中间协调节点导致的性能瓶颈问题.

3)基于现有的约束求解优化技术，提出不同工作线程共享约束求解缓存的优化方法，有效提高了约束求解效率.

4)在KLEE基础上实现了PKLEE，从而开发完成原型工具PKLEE.在小规模数据集以及真实规模数据集上进行实验评估，表明PKLEE取得较高的可扩展性，并行化效率较已有并行化方法具有明显提升.

1 技术背景

本节介绍传统SE过程以及KLEE[3]的组成模块.

1.1 符号执行算法

每一个程序上所有可能的执行路径可以生成一棵符号执行树(symbolic execution tree, SET). SET的根节点表示程序的起始状态，每一个根节点对应程序执行中的一个中间状态以及程序可以执行到该节点需要满足的路径条件. SET上的一条从根节点到叶节点的路径，对应程序的一条执行流. 图1是一个简单程序以及其对应SET的示意图.

算法1给出了传统KLEE中的SE算法.

算法1.标准SE算法.

输入：当前路径状态 *ES*;

输出：执行过程中生成的测试样例.

① while *states* 非空 && !*HaltExecution*

② *ES*←*getState*(*ES*); /* 获取下一条执行路径 */

③ *inst*←*ES.inst*; /* 获取下一条待解释指令 */

④ *stepInst*(*ES*); /* 更新当前路径的指令状态 */

⑤ *execInst*(*ES, inst*); /* 解释执行当前路径上的最新指令 */

⑥ *updateStates*(*ES*); /* 更新待执行路径 */

⑦ end while

算法1的输入为初始的路径执行状态 *ES*(包括当前路径的全部约束条件、待执行指令，以及当前所有路径上所有内存对象的映射)，一般是程序中的第1条指令对应的执行状态. 算法1中出现的 *states* 变

```
int get_sign (int x) {
 if (x==0)
    return 0;

 if (x<0)
    return -1;
 else
    return 1;
}
```

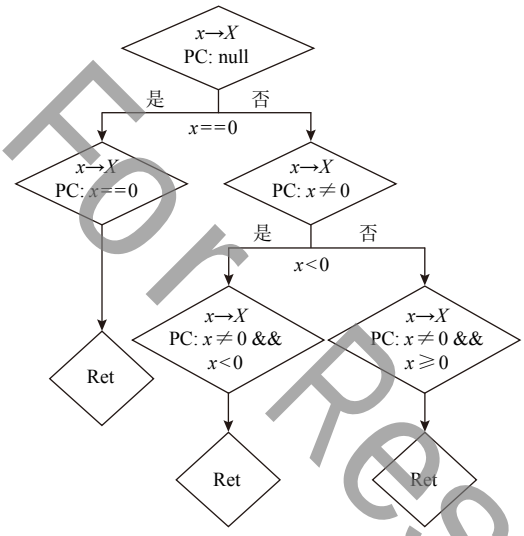


Fig. 1 Illustration of SET

图 1 SET 示意图

量为当前所有待探索的路径, *HaltExectution* 为一个全局标识符, *HaltExecution* 为真时终止当前算法. 由于同一时间可能存在多条路径可以选择探索, 绝大多数 KLEE 均提供了诸多搜索算法来选择其中一条路径探索. 算法 1 中行②将获取当前状态的过程抽象为 *getState* 方法, 返回当前搜索策略所选择路径的执行状态 *ES*.

算法 1 中的行⑤是 SE 的核心, *execInst* 对当前路径上的当前指令解释执行. 在一般的 KLEE 中, *execInst* 往往实现为一个巨大的 switch-case, 为每一条支持的指令做了不同的处理. 本文为了描述方便, 进行一些必要的简化. 一般而言, 所有的指令实际上可以抽象为分支(branch)、返回(ret)、中止(halt)以及一般的赋值(assign)语句. 算法 2 中为了屏蔽不必要的细节, 对于使用的函数均给出了其含义.

算法 2.单条指令解释执行算法 *execInst*.

输入: 当前路径状态 *ES*、当前执行指令 *inst*;

输出: 执行过程中产生的测试样例.

① if *inst* 是分支类型

② *TrueSucc*←*SMTEval*(*ES.PC*∧PC_{br}); /* 判断当前路径是否可行 */

③ *FalseSucc*←*SMTEval*(*ES.PC*∧¬PC_{br});

④ *fork*(*TrueSucc, FalseSucc*); /*拷贝生成新路径 */

⑤ else if *inst* 代替 *ret* && *isLastInst*(*inst*)

⑥ *generateTestCase*(*ES*); /* 到达执行流尾部, 生成测试样例 */

⑦ else

⑧ *InterpretInst*(*ES, inst*); /* 解释执行指令并更新状态 */

⑨ end if

对于分支语句, KLEE 会根据分支对应的路径条件 PC_{br} 以及路径状态 *ES* 上原本的路径条件 *PC* 判断可行性: *ES.PC*∧PC_{br} 对应分支可行, *ES.PC*∧¬PC_{br} 则对应分支不可行. 如果 2 个分支均可行, 则 KLEE 会复制当前路径状态 *ES*, 并将对应的路径条件 *PC* 更新, 否则直接复用当前的 *ES*(算法 2 行④).

对于返回语句 (此处仅考虑从最底层的函数调用返回的情况), KLEE 会根据路径状态 *ES* 的路径条件 *PC* 求解并生成一个具体的输入(算法 2 行⑥), 然后结束该路径的求解.

一般的 KLEE 实现中对于中止 (例如 assert 被违反) 等指令的处理与返回语句类似, 此时生成的样例往往对应程序中的一个错误.

对于赋值语句, 本文可以将其看作除了赋值、返回之外的一切不会改动路径条件 *PC* 但可能改变内存对象映射的指令, KLEE 会根据该指令的语义, 解释执行并更新内存对象映射(算法 2 行⑧), 即更新内存建模中对应变量的值或新建变量等.

1.2 KLEE 基本模块

KLEE[3] 为由斯坦福大学于 2008 年提出的符号执行引擎, 用 C++编写, 使用 LLVM IR 作为中间表示, 能够对 C、C++以及 Rust 等语言进行 SE. 多年以来, KLEE 作为动态符号执行引擎的 state-of-the-art, 在学术界被广泛应用于各类对 SE 的测试与扩展中, 例如 Cloud9[5].

KLEE 核心组件为:

1)符号虚拟机(symbolic virtual machine). KLEE 输入为 SE 程序的 LLVM bitcode 格式, 它在内存中使用 LLVM IR[6] 对程序进行符号化的解释执行. KLEE 使用执行状态(execution state, ES)抽象一条探索中的执行路径, ES 上保存了路径的约束条件、内存映射(symbolic memory store, SMS)等信息. 因为 ES 之间没有数据上的依赖, KLEE 在 SE 过程中可以同时保存多条执行路径.

2)约束求解器链(solver chain). KLEE 为了方便对约束求解进行优化, 将约束求解的过程划分成若干个阶段, 提高了约束求解优化流程的可扩展性.

KLEE 默认情况下的约束求解链依次为计时求解器(Timing Solver)、独立性化简求解器(Independence Solver)、缓存求解器(Caching Solver)以及反例缓存求解器(Cex Caching Solver). 待求解的约束会按照约束求解链依次传递, 如果中途已经得出结果, 则不会继续传递; 只有当底层求解器之前的阶段全部失效时, KLEE 才会调用底层的 SMT 求解器. KLEE 在文献 [3] 中着重介绍了反例缓存, 反例缓存基于许多约束之间十分类似的观察, 一个约束的子集的不可解等价于自身的不可解。同理, 超集的可解等价于自身的可解. 反例缓存会缓存已有的约束求解中的约束以及解, 利用约束之间的子集、超集关系推导来避免调用底层 SMT 求解器[7-9].

下面通过一个简单的例子介绍反例缓存的工作原理. 假设反例缓存中当前存在一个不可满足的约束$\{x>2, x<0\}$, 以及一个可满足的约束$\{x>2, y<1\}$, 对应的一个解为$\{x=3, y=0\}$. 对于约束$\{x>2, x<0, y>3\}$, KLEE 查询反例缓存发现该约束存在一个不可满足的子集$\{x>2, x<0\}$, 直接判定该约束不可满足; 对于约束$\{x>2\}$, KLEE 查询反例缓存发现该约束的一个超集$\{x>2, y<1\}$可满足, 其中变量x的赋值依然满足当前约束, 直接判定该约束可满足; 对于约束$\{x>2, y<1, x\neq 4\}$, KLEE 通过查询反例缓存发现该约束的一个子集$\{x>2, y<1\}$可满足, 尽管这不能直接说明当前约束可满足, 但是 KLEE 通过将已有解带入新约束中进行判定, 如果成功, 则能够避免一次 SMT 求解器的调用.

3)运行时环境模拟层. 符号执行中的一大难题是模拟外部环境, 例如操作系统中的文件在一系列读写操作下应保持一致. KLEE 对 POSIX/Linux 运行环境进行了细致的建模, 并在符号虚拟机对外部环境进行更改时使用该模拟层进行拦截, 避免了黑箱调用, 提升了 SE 的质量.

本文以 KLEE 为基础实现了 PKLEE. PKLEE 在保留了 KLEE 中核心组件的同时, 对 KLEE 底层数据结构进行了一系列改写以支持并行化路径探索, 同时使其支持并行约束求解以及动态负载均衡.

2 相关工作

为了提升 SE 的可扩展性, 近年来一直有相关工作[5,10-14] 力求通过并行化手段对之进行提升.

Siddiqui 等人[10] 提出了 ParSym, 一种基于传统 SE 算法的并行化方案. ParSym 直接将每一次的路径探索并行化, 并且使用中心节点进行任务的分配. 然而, ParSym 的负载均衡方式增加了工作节点之间的通信次数, 产生了极大的额外开销, 一定程度上限制了可扩展性.

Staats 等人[11] 提出的简单静态切分方法(simple static partition), 使用预先计算的先置条件(pre-condition)来切分 SET, 不同工作节点被赋予了不同的先置条件, 并且只执行路径条件满足其先置条件的路径, 静态地完成了 SET 的划分, 不需要像 ParSym 一样引入中间节点频繁地进行通信, 极大地降低了通信开销. 然而, 简单静态切分方法会导致不同工作节点重复探索一部分路径, 导致了一定计算资源的浪费.此外, 简单静态切分方法本质上还是一种近似的对 SET 静态均分的方法, 没有彻底解决 SET 在运行期形状不可预知的负载不均衡问题.

Bucur 等人提出的 Cloud9[5] 与 ParSym 的并行化架构类似, Cloud9 使用 KLEE 作为底层符号执行引擎, 在无共享(share-nothing)的集群上的各个节点上运行 KLEE 的独立线程实例. Cloud9 将 SET 中的叶子结点切分成不相交的部分, 并以此作为负载均衡的单位. Cloud9 使用一个中心服务器进行负载均衡, 根据不同节点上当前待处理任务数量的标准差以及当前平均待处理任务长度进行负载是否合理的判断. 在新的任务传输到工作节点上时, 由于不共享内存, 工作节点会在本地重放新的任务对应的执行路径. Cloud9 中心化负载均衡的方式容易让中心服务器成为任务的瓶颈. 此外, 有对 Cloud9 的实证分析[15] 指出它没有充分利用 KLEE 内置的诸多约束求解优化技术, 例如 1.2 节提到的反例缓存优化技术.

Siddiqui 等人提出的 Ranged Analysis[16] 使用范围分析 (ranged analysis) 的方法, 通过人工设计或随机生成的测试输入来划分符号执行的搜索树, 进而可以在划分之后的不同子树上并行进行 SE. 该方法通过测试样例对不同的搜索树进行了隔绝, 一定程度上保证了不同工作节点之间的工作负载类似. Ranged Analysis 支持随机生成样例并对搜索树进行划分, 然而, 在这种情况下无法有效保证不同工作节点的工作负载类似.

Wei 等人提出了 LLSC[17], LLSC 将基于多阶段编程(multi-stage programming)的高级语言编写的 SE 解释器转化为对应的编译器. LLSC 生成的程序相比 KLEE 等传统符号执行引擎解释执行的方式, 没有任何解释执行的开销, 直接进行符号执行、合理调用 SMT 求解器. 同时, 由于 LLSC 生成的代码使用了适

合并行化的结构，其天然支持非确定的并行 SE. LLSC 不仅降低了 SE 中探索每一条路径的成本，同时也通过并行的方式提升了 SE 的可扩展性. 不过由于 LLSC 仍然在开发阶段，其现在生成代码中并行化 SE 的方式仍然粒度较高.

除了并行化提升 SE 可扩展性，也有相关工作通过对 SET 进行剪枝[18-20]，去除 SE 中冗余的路径、减少需要探索的状态空间；也有通过改进现有 SE 的核心搜索算法来提升可扩展性的工作[21-22]，旨在使用特殊的搜索算法来专注搜索对于提升程序整体覆盖率有利的路径.

由于传统基于多进程并行符号执行方法存在通信开销大、中心化负载均衡的问题，本文提出了基于多线程并行的符号执行方法. 通过更细粒度的线程并行，不同工作线程可以高效共享已有的路径信息、约束求解信息，同时可以在不引入中间节点的情况下进行负载均衡.

3 PKLEE

本节介绍 PKLEE 的框架.

3.1 方法概述

并行化 SE 主要存在动态负载均衡、系统执行开销 2 个方面的挑战.

SE 的工作负载是动态生成的，为了维持不同工作节点之间的工作负载平衡，现有工作[5,10] 主要采取基于主从(Master-Worker)的中心化架构.在主从架构下，主节点(Master)不断向工作节点(Worker)发送心跳并监视工作节点的待执行路径数量，通过转移待执行路径来动态地实现不同工作节点间负载的平衡. 在多线程场景下，工作线程可以通过共享内存直接共享内存中的待执行路径信息，直接省去了待执行路径通信的开销以及路径中带有的路径条件等额外信息；此外，本文注意到 SE 过程中工作节点在少量路径负载时依然能够保持高 CPU 利用率，本文设计了一种惰性的负载均衡策略，尽量减少负载均衡占用的时间.

对于系统执行开销，SE 的开销主要来自约束求解以及执行过程中产生的大量待执行路径所占据的内存. 传统的多进程并行方法中每一个进程独自运行一个 KLEE 实例，不同 KLEE 之间重复维护诸如内存建模等数据结构，造成了额外内存开销. 此外，不同的 KLEE 之间无法共享已有的约束求解优化信息，例如 KLEE 中的反例缓存. 在本文多线程模型下，诸多数据结构可以直接在不同线程中共享.

基于这 2 点挑战，本文提出了如图 2 所示的 PKLEE 整体框架. PKLEE 底层同时运行多个工作线程，每个工作线程独立解释执行 IR、调用 SMT 求解器计算约束可行性. 不同工作线程同时共享 KLEE 原有的约束求解优化信息来降低约束求解开销，例如反例缓存.3.2 节对应图 2 中的工作线程的生成以及具体执行的算法，3.3 节对应图 2 中的负载均衡，3.4 节对应图 2 中的共享约束求解缓存信息.

3.2 多线程 SE

SE 算法存在天然的并行性，理论上每一条路径的探索求解彼此独立，不存在互相依赖的关系. 本文

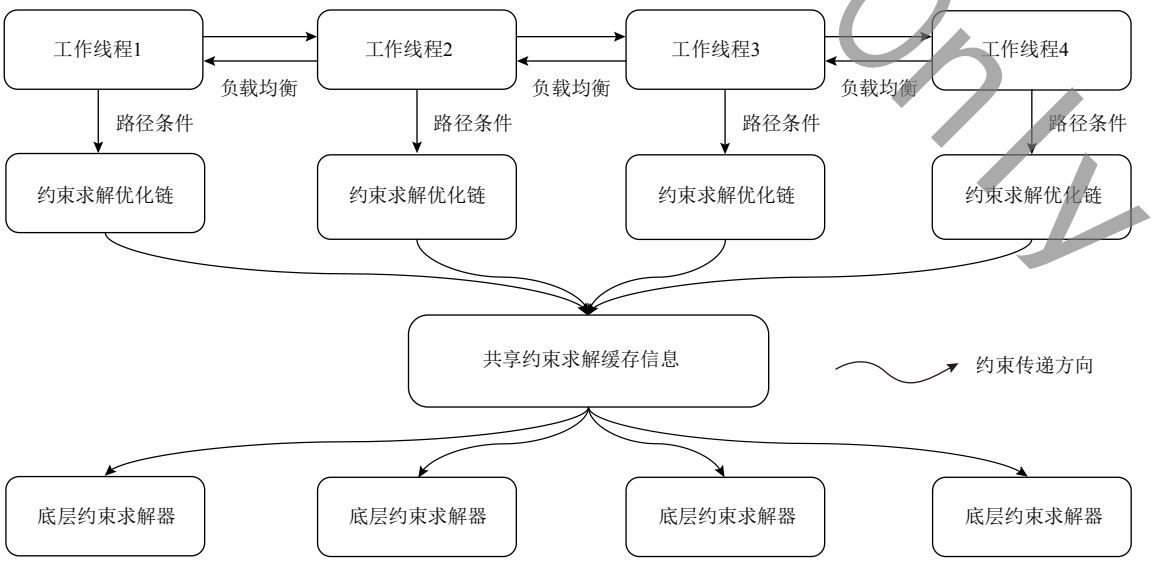


Fig. 2 PKLEE architecture

图 2 PKLEE 框架

提出的并行化方法中以每一个工作线程作为符号执行的最小工作单元，同时保留一个协调线程保证在所有工作线程探索全部路径或到达指定执行时间后结束符号执行. 协调线程算法如算法 3 所示.

算法 3.协调线程算法.

输入：待分析程序 *program*、开始生成工作线程阈值 *threshold*、工作线程数量 *worker_num*;

输出：初始化的工作线程 *workers*.

① while *states*.size() < *threshold* /* 符号执行直到路径数量达到开始生成的阈值 */

② symbolic execute *program*;

③ end while

④ allocate *states* to *worker_num workers*;

⑤ wait until all workers done

在并行化符号执行开始阶段，协调线程会进行一段时间的串行符号执行探索，直至当前待执行路径数量达到用户设定的阈值(算法 3 行①). 在路径数量达到用户设定阈值之后，协调线程生成工作线程并将当前所有待执行路径平均分配给工作线程(算法 3 行④). 此后，协调线程不再执行任何操作，直至所有工作线程完成全部路径探索，结束本次符号执行(算法 3 行⑤). 如果在执行过程中工作线程使用的总内存达到了用户设定的总内存上限或工作线程总执行时间超过了用户设定的时间，协调线程会主动通知其他线程结束执行.

每一个工作线程的执行算法与算法 1 中提到的标准符号执行算法类似，如算法 4 所示.

算法 4. 工作线程算法.

输入：待执行路径队列 *work_queue*、其他工作线程列表 *worker_list*、协调线程 *coordinator*、最低工作负载阈值 *steal_threshold*;

输出：执行过程中产生的测试样例.

① register global vars of thread; /* 注册必要的全局变量*/

② get initial work load from *coordinator*;

③ while *work_queue*.size() ≠ 0 || !*coordinator*.Halt() /* 依然存在待执行路径 */

④ if *work_queue*.size() < *steal_threshold*

⑤ *work_queue*.push(*work_steal*(*worker_list, coordinator*)); /* 当前待执行路径数量小于最低工作负载阈值，进行工作窃取 */

⑥ end if

⑦ *path* ← *getState*(*work_queue*); /* 根据本地搜索策略获取执行路径 */

⑧ execute *path* and add new paths to *work_queue*; /* 解释执行当前路径并添加新路径 */

⑨ end while

在工作线程开始探索路径前，它首先会将一些重要的线程局部变量在全局的内存对象映射中进行注册(算法 4 行①). 不同工作线程各自维护待执行路径队列，工作线程获取可执行路径后将会独立求解，直至线程本地维护的待执行路径队列为空或协调线程通知结束符号执行(算法 4 行③). 探索路径过程中，工作线程会首先根据本地的搜索策略寻找下一条应该执行的路径，如果待执行路径队列中剩余的待执行路径数量小于用户设定的最低工作负载阈值，则进入空闲状态，并进行工作窃取(算法 4 行⑤)；反之，工作线程会像标准符号执行算法一样执行当前路径，更新本地统计信息(执行时间、执行指令数量). 在完成当前路径的探索之后，工作线程将本轮执行过程中所生成的新路径直接加入线程本地的待执行路径队列中(算法 4 行⑧).该实现方法的好处是：如果待符号执行的程序本身规模较大，那么每一个工作线程几乎可以在没有任何通信的情况下分别高效地进行 SE.

3.3 动态负载均衡

SE 过程中的工作负载往往动态变化，因此良好的负载均衡策略相当重要. 本文设计了一种不需要中间协调线程参与的工作窃取算法，尽可能减少不同工作线程之间的通信次数. 由于不同工作线程的待执行路径队列中的待执行路径均位于堆上，除了不同线程之间同步的开销，其他工作线程可以几乎没有开销地直接获取待执行路径.

算法 5.工作窃取算法.

输入：其他工作线程列表 *worker_list*, 协调线程 *coordinator*, 最低工作窃取阈值 *work_threshold*;

输出：获取的待执行路径.

① *worker_to_steal* ← ∅;

② *cur_max_work* ← 0;

③ while !*coordinator*.Halt() && *worker_to_steal* is empty /* 当前仍未找到待执行路径 */

④ for *worker* in *worker_list*

⑤ if *worker*.work_num() > *cur_max_work* && *worker*.work_num() > *work_threshold*

⑥ *worker_to_steal* ← *worker*;

⑦ *cur_max_work* =*worker*.work_num();

⑧ end if

⑨ end for

⑩ wait for moment;

⑪ end while

⑫ steal work from *worker_to_steal* if it's not empty.

/*从 *worker_to_steal* 处窃取路径 */

当启动工作窃取(即工作队列中待执行路径数量小于指定阈值)时，当前工作线程轮询其他具有待执行路径的工作线程(算法 5 行③和④)，从其中待执行路径数量最多的工作线程上获取待执行路径(算法 5 行⑤). 值得注意的是，为了防止发生工作线程被窃取后工作负载过低，算法 5 只窃取待执行路径数量大于最低工作窃取阈值的工作线程(算法 5 行⑤). 实践中，最低工作窃取阈值至少应该设置为最低工作负载阈值的两倍. 每一个工作线程为了支持被询问当前的待执行路径数量，需要对搜索算法以及路径探索等模块进行一定的同步，这带来了一定的开销，因此在算法中通过每次轮询并且没有获取到待执行路径时，等待一段时间以避免频繁的同步操作(算法 5 行⑨). 同时，在每一个工作线程均进入工作窃取状态时，说明全局已经没有任何工作负载，协调线程将会通知所有工作线程本次运行已经完成.

3.4 共享约束求解优化

已有的 KLEE 往往使用约束求解缓存技术来加速约束求解[3]. KLEE 在文献 [3] 中提出了反例缓存的约束优化方式，其基本思路在第 1.2 节已经做了简要阐述，实质上是通过缓存求解过的约束解来加速之后的约束求解，在 SE 过程的不同路径求解模式类似时可以极大提升求解速率. 通过 KLEE 在文献 [3] 的实验以及工业界实践，反例缓存往往可以提升十倍及以上的约束求解效率.

然而，本文发现，由于不同工作线程的约束求解缓存彼此独立，相比于串行执行，并行执行过程中的单次约束求解开销往往会更大. 同时，不同工作线程对应的 SET 的部分往往差异较大，在某个工作线程窃取了待执行路径之后开始的新求解，可能会产生较高的约束求解开销，类似于计算机中的 Cache miss 现象.

基于以上观察，本文在并行化算法基础上，将不同工作线程的缓存优化进行共享，每一个工作线程都会将其求解过的约束放入全局约束求解缓存中，并且已有的约束求解信息对其他线程可见.

4 工具实现

本节介绍 PKLEE. 图 3 给出了 PKLEE 中工作线程的执行框架.

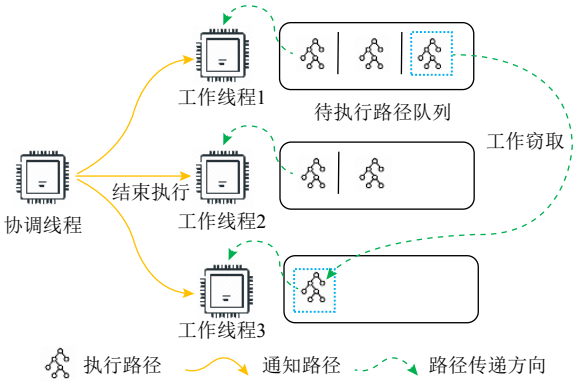


Fig. 3 Illustration of PKLEE's worker threads

图 3 PKLEE 工作线程示意图

4.1 KLEE 并行化改造

由于 KLEE 为单进程单线程执行模型设计，支持并行化符号执行存在一定难度. 为了实现这一目标，本文对 KLEE 底层数据结构、上层执行逻辑进行了诸多改造.

PKLEE 使用路径上下文(execution context, EC)解决 KLEE 无法进行多路径并行执行的问题. 路径上下文记录执行过程中不同路径的增删路径信息，它使得不同路径执行过程互相隔离. 每一个不同的路径上下文调用不同的 SMT 求解器进行求解.

PKLEE 也对 KLEE 中诸如引用计数类 ref 等重要的底层数据结构进行了改写，以保证 PKLEE 在并发执行路径场景下正确工作. 对于 KLEE 执行逻辑中的数据竞争问题，PKLEE 目前使用互斥访问的方式进行简单的保护.

4.2 工作线程实现

PKLEE 中的每一个工作线程对应 4.1 节中的一个路径上下文，每一个工作线程同时维护包含所有未执行路径的队列. 为了实现 3.3 节中的基于工作窃取的动态负载均衡算法，每一个工作线程在被初始化时获取当前其他工作线程的路径上下文，并且可以通过路径上下文获取其他工作线程当前的待执行路径队列中的路径. 如图 3 所示，工作线程 3 当前待执行路径队列为空，因此它将执行工作窃取. 工作线程 3 访问其他所有工作线程的待执行工作队列，发现工作线程 1 具有最多待执行路径，因此从工作线程 1 处窃取了一条路径并执行. 在所有工作线程并行符号执行期间，协调线程根据用户设定的结束条件(超时、内存资源用尽或完成所有路径探索)，在结束条件满足时通知工作线程结束符号执行.

PKLEE 在执行涉及外部环境的函数调用时，会

使用当前工作线程的 errno 来访问内存建模中的对象，因此每一个工作线程在初始化时需要对类似 errno 的线程局部变量在全局变量映射中进行注册. 值得注意的是，由于 KLEE 与外部环境交互的重要函数 *runProtectedCall* 是不可重入的，PKLEE 目前在处理 *runProtectedCall* 时保证只能有一个工作线程，面对与外部环境交互频繁的真实程序，PKLEE 的可扩展性会受到一定影响.

PKLEE 中还对工作窃取算法做了额外的优化.已有并行符号执行工作中的负载均衡算法往往只考虑了路径数量，忽略了不同路径的求解难度. 本文注意到，KLEE 中的 nurs:qc 搜索算法在搜索过程中考虑了每一条待搜索路径的上一次约束求解成本，并选择每一次求解成本最低的路径，提升路径吞吐量. 基于这一思路启发，本文在工作窃取算法中加入对于当前工作线程中不同路径求解难度的衡量，将每一次求解成本较高的路径优先分配给其他工作线程，减少当前线程的求解开销. 从任务并行的角度来看，将一个约束求解难度更高的路径分配给其他路径，在并行求解下，最终总执行时间不会更长；从利用缓存的角度来看，其他线程在求解了更复杂的路径约束之后，很可能通过更新约束优化信息来加速后续的约束求解.

在完成每一次约束求解之后，工作线程都会将其求解过的约束放入全局反例缓存中. 一方面，在不同工作线程中共享反例缓存带来了一定的同步开销，但这部分开销与约束求解开销相比较小. 另一方面，并行化进行符号执行也一定程度上提升了缓存中可行解的数量，从而可以进一步减少约束求解的开销.

5 实验评估

本节对本文基于 KLEE 实现的 PKLEE 进行了 2 个方面的实验分析. 一方面，考虑了在诸多小规模合成程序上，KLEE 与 PKLEE 完成所有可执行路径探索所需时间的对比，即在穷尽路径情况下的并行加速比；另一方面，对比了在若干个大规模真实程序上，相同时间内 PKLEE 的有效工作负载吞吐量的提升. 本节还分析了在以上 2 种场景下工作窃取算法、共享反例缓存优化是否开启对于整体性能的影响. 最后，对比了 PKLEE 与现有工作的并行加速比.

5.1 实验设置

对 PKLEE 的实验评估试图回答 4 个研究问题：

问题 1. PKLEE 的可扩展性如何？（5.3 节）

问题 2. 本文提出的工作窃取算法是否可以缓解不同工作线程的负载不均衡问题？（5.4 节）

问题 3. 本文提出的共享反例缓存是否能在并行执行场景下减少约束求解次数？（5.5 节）

问题 4. PKLEE 与相关工作对比表现如何？（5.6 节）

本节开展的所有实验均基于 Ubuntu 18.04，使用的 CPU 为 AMD® Ryzen 7 4800h(8 核)，8 GB 内存. 编译器为 clang 9，KLEE 以及 PKLEE 使用的求解器为 z3-4.8.8. KLEE 默认使用的约束求解器 STP 在本文的实验中无法正常地多实例并行运行，因此我们选择了更新的 z3 求解器.

5.2 实验程序选取

为了测试 PKLEE 在不同规模程序上的可扩展性、动态负载均衡效果等性能指标，本文选取了小规模、大规模 2 类程序，并与 KLEE 进行对比.

对于小规模程序，本文选取了 LLSC 在文献 [17] 中给出的测试基准样例，其中包括 3 个人工合成的大量路径的程序、一些常见的较少涉及外部库调用以及 KLEE 自身的环境建模的小型程序（归并排序、背包问题等）. 为了保证并行化执行能够在较短时间内能被执行，且避免 KLEE 与系统建模的频繁交互影响并行执行的性能，挑选的测试程序满足较少涉及外部库（例如 malloc 等）的调用、总体指令数量和路径数量较少的特点. 表 1 中列出了测试程序名称以及其对应的执行指令数量、全部可探索的路径数量. 对于每一个小规模测试程序，对比 KLEE 与 PKLEE 探索全部路径耗时的区别.

Table 1 List of Small-scale Benchmark

表 1 小规模合成测试程序列表

测试程序	执行指令数量	探索路径数量
multipath_1 024_kle	29 733	1 024
multipath_65 536_klee	2 687 035	65 536
multipath_1 048 576_klee	51 380 299	1 048 576
knapsack	391 134	1 666
mergesort	92 685	720
regexp	473 590	1 058

对于大规模程序，本文选取了 GNU Coreutils 中的若干测试程序. KLEE 在文献 [3] 中使用 GNU Coreutils 程序集进行性能的测试，并面向 GNU Coreutils 等程序设计了一套比较完善的环境建模. 本文测试使用的版本为 GNU Coreutils 6.11. 如表 2 所示，本文将 GNU Coretuils 中程序根据行数从高到低排序，并按

大小随机抽取 6 个程序(200 行以上)进行测试. 对于每一个测试程序, 设定符号执行时间在 10 min 内, 并对比 KLEE 与 PKLEE 执行的指令数量的区别. 此处之所以选择 KLEE 执行的指令数量而非完成的路径数量作为有效工作负载的度量单位, 是由于 KLEE 面对较大规模的程序在较短时间内很难完成较多路径的探索(可能存在大量未探索完毕的路径), 相比之下执行的指令数量这一指标则更为直观, 已有的工作 Cloud9[5] 在测试其可扩展性时也使用了该指标, 说明了该指标具备可信度.

Table 2 List of Large-scale Benchmark

表 2 大规模真实测试程序列表

测试程序	代码行数
echo	276
runcon	284
uname	380
who	819
join	1 108
stty	1908

在 PKLEE 路径搜索策略的选择上, 从对比公平的角度出发, KLEE 以及 PKLEE 均采用深度优先搜索算法进行路径选择. 采用深度优先搜索算法的一个潜在好处是, 保证了 KLEE 每一次执行次序的相对固定, 相比之下, KLEE 中的诸多启发式算法在不同的执行过程中可能会选择差距较大的执行路径, 需要进行多次实验.

5.3 可扩展性

图 4 直观反映了在小规模程序上, 以 PKLEE 的单个工作线程为基准, 在增大工作线程数量的情况下, 探索程序中所有可行路径的耗时. 图 4 中展示了在 multipath_1048576_klee 样例上, PKLEE 在不同线程数量下与 KLEE 耗时的对比. 从耗时较长的 knapsack 以及 multipath_1048576_klee 2 个样例可以明显看出, 随着工作线程数量的增加, 总体执行时间显著减少, 并且在 6 个线程左右时仍然保持线性的减少趋势. 然而, 在 8 个工作线程时, 探索全部路径的耗时保持恒定. 本文认为, 一方面, 这可能是由于本文使用的实验机器最多支持 8 个物理线程, 此时已经达到了最大负载, 导致每一工作线程的执行效率有所下降; 此外, 从 multipath_1048576_klee 样例上 KLEE 与 PKLEE 完全探索耗时的对比也可以发现, PKLEE 目前的实现中存在底层诸多数据结构的同步开销, 影响了在单机上进一步扩展.

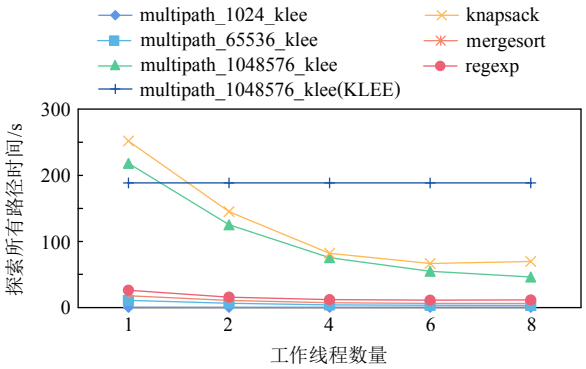


Fig. 4 Total time taken to explore paths exhaustively under small-scale benchmark

图 4 小规模测试程序探索完整路径耗时

在路径数量较少的情况下, 例如 multipath_1024_klee 样例, 从图 4 中可以看出, 当程序中的整体执行路径数量较少时, 无法从并行化符号执行中获得收益, 甚至在增大工作线程的情况下会获得负收益. 相比之下, 在 multipath_1048576_klee 样例上, PKLEE 凸显了其收益. PKLEE 在所用的测试样例程序上整体可以达到 3.0x 的加速, 其中在 2 个耗时最长的 knapsack 和 multipath_1048576_klee 上可以达到 4.0x 的加速.

图 5 绘制出了随着工作线程增长, 在不同大规模测试程序上, 指定时间内完成的有效工作的数量. 图 5 中同时展示了在 who 样例上, PKLEE 在不同线程数量下与 KLEE 执行指令数量的对比.

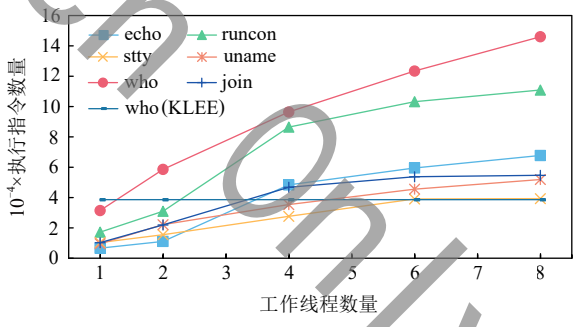


Fig. 5 Total number of executed instructions under large-scale benchmark

图 5 大规模测试程序执行指令数量

不难看出, 随着工作线程数量的增加, 所有工作线程共计执行完成的指令数量基本呈线性增长的趋势, 并且增长的速度大致相同. 图 5 说明了 PKLEE 在 8 核并行执行现实程序的场景下依然具备一定线性可扩展性, 能够充分利用不断增加的计算资源.

本文也对比了在大规模程序上, 未经修改的 KLEE 与单个工作线程、8 个工作线程的有效工作负载. 如表 3 所示.

Table 3 Comparison of the Number of Executed Instructions Between KLEE and PKLEE

表 3 KLEE 与 PKLEE 执行指令数量对比

测试程序	KLEE/万条	单工作线程/万条	8 个工作线程/万条	加速比
echo	22 178	6 637	67 744	3.0x
runcon	58 619	17 218	110 904	1.9x
uname	10 817	9 555	51 911	4.8x
who	38 624	31 465	146 016	3.7x
join	19 585	10 271	54 681	2.8x
stty	13 412	10 353	39 304	2.9x

通过加速比一栏可以看出，单个工作线程与 KLEE 相比，在相同的执行时间内，执行的指令数量更少，这部分开销可以视作并行执行中同步不同工作线程的加锁、解锁等操作的开销. 在 8 个工作线程的情况下，所有线程执行的工作数量最终平均是 KLEE 的 3.2 倍，是单个工作线程的 5.8 倍.

由表 3 以及图 5 中的实验数据可以得出结论，尽管并行化给单个线程执行带来了一定开销，但是实际执行过程中不同线程均摊了一部分的同步开销，总体工作负载依然随工作线程的数量增加而提升. 从表 3 中数据也可以发现，规模最大的 join, stty, who 等 3 个测试程序获得了较高的工作负载提升比例，相比之下 PKLEE 在规模较小的 runcon 上对工作负载的提升有限，这一定程度上说明并行化符号执行的方法随着程序规模的增大可以获得更多的潜在收益. 此外，单个工作线程执行过程中的较大执行开销也可以看出，目前的实现中依然存在较大的优化空间.

对于问题 1，经过小规模与大规模测试程序两个实验的评估，可以看出 PKLEE 具备一定的可扩展性，在 8 个工作线程并行执行的情况下依然能够保证较好的可扩展性. 在小规模程序上进行的穷尽所有路径探索实验中，平均可以获得 3 倍的运行速度提升，且最多可以获得 4 倍以上的提速；在大规模的现实程序上，与 KLEE 相比，在相同工作时间内，PKLEE 可以将有效工作负载提升为原先的 3 倍左右. 总体看来，无论是在涉及较少库函数、外部环境建模的小规模程序上，还是在真实的大规模程序上，基于多线程并行的符号执行方法都能充分利用多核的计算能力，在没有做过多深入底层优化的前提下，有效地提升了符号执行的可扩展性.

此外，在大、小规模程序上的实验数据表明，目前的 PKLEE 依然存在一定的额外开销. 这部分开销不仅来自底层的数据结构改为线程安全所带来的额外开销，也来自并行符号执行中同步多个工作线程的开销. 同时，由于 KLEE 对外部环境的建模方式，目前实现中仅允许最多一个工作线程进行与外部环境的交互（例如修改 errno）. 在现实场景中的大规模程序上，势必需要对该机制进行进一步的优化.

5.4 工作窃取算法效果

表 4 中对比了小规模程序上，8 个工作线程下，是否开启工作窃取情况下的全部路径探索耗时以及 KLEE 的探索路径耗时.

Table 4 Time Consumption of Work-stealing Algorithms Under Small-scale Benchmark

表 4 小规模程序上工作窃取算法的时耗 s

测试程序	KLEE 耗时	8 个工作线程耗时	8 个工作线程耗时（无工作窃取）
multipath_1 024_kle	0.31	0.45	0.46
multipath_65 536_klee	9.57	3.2	6.78
multipath_1 048 576_klee	188.39	46.21	121.42
knapsack	277.75	69.68	246.15
mergesort	18.36	6.21	12.28
regexp	29.59	11.56	23.57

从表 4 实验数据可以看出，在 8 个工作线程没有开启工作窃取的前提下，执行总体时间与 KLEE 相比，仅有较小的提升. 在没有开启工作窃取的情况下，不同工作线程的全部工作负载就是其被初始化时所分配的路径以及执行过程中衍生出的路径. 在最坏情况下，如果没有工作窃取算法，系统整体的执行时间会变成所有工作线程中最慢的一个. 同时未列出的各个工作线程执行指令数量在开启负载均衡后基本相同.

在表 5 中可以看出，在大规模程序上，工作窃取算法开启对于完成执行指令数量的影响.

在 10 min 的执行时间内，本文观察到 echo,

Table 5 Effect of Work-stealing Algorithms on the Number of Instructions Under Large-scale Benchmark

表 5 大规模程序上工作窃取算法对指令数量影响

测试程序	开启工作窃取/万条	关闭工作窃取/万条
echo	67 744	64 510
runcon	110 904	1 100 630
uname	51 911	49 430
who	146 016	144 290
join	54 681	54 598
stty	39 304	37 566

uname 等样例上存在工作线程提前完成自身工作负载的情况, 这些样例对应的工作负载减少了 5% 左右. 其余样例上没有发现工作负载产生明显变化, 这是由于测试用的程序执行流较为复杂, 故而在一定时间内几乎每一个工作线程的工作负载都处于饱和状态, 故而此时的工作窃取算法是否开启对整体的性能没有较大影响. 该实验也证明了在工作负载饱和的情况下, 工作窃取算法的开启不会对整体的执行性能产生负面影响.

对于问题 2, 通过 2 个实验, 本文发现工作窃取算法在程序中整体可执行路径较少的情况下, 可以有效缓解由于静态切分 SET 形状不同带来的负载不均衡问题, 并且在程序规模较大的情况下工作窃取算法也几乎不会引起额外的开销. 小规模程序的测试实际上也可以视作大规模程序在执行后期的场景, 即许多路径会因为探索策略、系统开销等原因提前终结, 不同工作线程之间的工作负载差异极大, 印证了本文第 3.1 节的假设.

5.5 共享约束求解信息效果

表 6 中给出了小规模程序上, 在开启与不开启共享反例缓存优化情况下 8 个工作线程的反例缓存命中情况对比.

Table 6 Effect of Shared Cex Cache on the Number of Hits Under Small-scale Benchmark

表 6 小规模程序上共享反例缓存对命中次数的影响

测试程序	开启时反例缓存未命中次数	未开启时反例缓存未命中次数	减少比例/%
multipath_1 024_kle	32	78	59
multipath_65 536_klee	80	122	34
multipath_1 048 576_klee	112	156	28
knapsack	3 239	3 575	10
mergesort	720	783	8
regexp	1 311	2 134	38

从表 6 中数据可以看出, 在小规模程序上, 共享反例缓存可以降低反例缓存的未命中次数, 平均减少了 30% 的反例缓存未命中次数. 换而言之, 在并行执行场景下, 共享反例缓存减少了 30% 的底层约束求解. 在符号执行以约束求解为主的工作负载上, 共享反例缓存利用多个工作线程同时生成大量缓存的特点, 有效提升了约束求解的效率. 相比之下, 传统的基于多进程的并行执行方案中无法有效功共享诸如反例缓存这样成熟的约束求解优化技术.

表 7 中给出了大规模程序上, 在开启与不开启共享反例缓存优化情况下 8 个工作线程的反例缓存命中情况对比.

Table 7 Effect of Shared Cex Cache on the Number of Hits Under Large-scale Benchmark

表 7 大规模程序上共享反例缓存对命中次数的影响

测试程序	开启时反例缓存未命中次数	关闭时反例缓存未命中次数	减少比例/%
echo	981	1818	46
runcon	1 454	2 953	51
uname	5 300	6 490	18
who	2 975	5 953	50
join	22 265	29 240	23
stty	17 924	19 260	5

平均而言, 共享反例缓存在测试程序上降低了反例缓存的未命中次数达 32%, 充分说明在大型实际程序的场景中, 通过共享约束求解信息也可以对整体约束求解效率带来一定幅度的提升. 在执行指令数量最多的两个样例程序上, 共享反例缓存降低了整体的反例缓存未命中次数近 50%. 共享反例缓存在大规模工作负载上可以获得更高的收益, 因为不同的工作线程的执行树之间往往存在一定的相似性.

对于问题 3, 本节的实验也充分证明了, 在大、小规模程序的场景下, 共享反例缓存在多个工作线程的情况下可以有效减少底层约束求解器被调用的次数. 在待执行路径较多情况下, 多个工作线程可以同时增加大量约束求解信息, 实验中最多可以减少 50% 的底层约束求解调用, 大大提升了约束求解的效率, 并且同步的额外开销较小. 反例缓存只是 KLEE 约束求解优化技术中的其中一种, 本文的尝试也说明了在并行化执行场景下, 如何充分利用已有的约束求解优化技术值得进一步探讨.

5.6 与已有工作对比

由于相关的并行化符号执行工具[5,10,11,16]并未开源或长期缺乏维护而无法正常运行, 因此无法通过在相同实验环境下复现已有方法进行完整、精确的性能对比. 然而, 为了在一定程度上理解 PKLEE 和相关并行化工作的优劣, 本节选取 Ranged Analysis[16] 实验中的公开测试集, 在其上对 PKLEE 进行实验, 收集加速比数据, 然后和 Ranged Analysis[16] 论文中报告的加速比数据进行直接对比.

Ranged Analysis 中的实验测试使用 6 个工作节点并且开启负载均衡, 给出了在 GNU Coreutils 程序上的加速比. 由于 Ranged Analysis 工作未给出源代码

并且没有给出测试时使用的具体优化、搜索方法以及参数，将 PKLEE 与该工作实验结果直接对比存在一定不可靠性，本文直接使用与真实程序测试中相同的测试参数及搜索方法，不进行任何的优化设置，尽可能保证对比的公平性.

Ranged Analysis 中通过其提出的技术保证了多个工作节点的符号执行与单个节点的符号执行遍历的路径完全相同，并比较两者遍历完相同路径所需的时间. 为了简单起见，本文依然采用与真实程序测试中的执行的指令数量来计算加速比. 由于不能保证 PKLEE 多工作线程执行过程中的路径与单个工作线程相同，为规避偏见(bias)影响，仅保留多线程探索路径数量较多的测试程序，表 8 展示了具体的加速比数据.

Table 8 Comparison of Speedup Between Ranged Analysis and PKLEE

表 8 Ranged Analysis 与 PKLEE 的加速比对比

测试程序	PKLEE 加速比	Ranged Analysis 加速比
tr	4.5x	1.2x
dirname	3.5x	1.1x
factor	1.3x	1.1x
dircolors	2.7x	4.2x
pr	2.5x	4.4x
cut	5.8x	4.6x
echo	2.7x	4.3x
fold	9.8x	9.7x
chgrp	2.7x	7.7x
chown	3.0x	8.8x
readlink	3.3x	9.5x

通过表 8 可以发现，仅仅通过多线程方法并行化符号执行过程，PKLEE 在许多 Ranged Analysis 表现不佳的测试样例上依然可以达到 3x~4x 的加速比. PKLEE 在 Ranged Analysis 的样例加速比基本持平. 其中，在测试程序 fold 上，PKLEE 达到了超过工作线程数的并行加速比. 这是由于 PKLEE 每个工作节点在探索 fold 的局部符号执行树时，由于子树规模小，工作节点能充分利用已有的约束求解优化，进而获得了超过工作节点数的加速比.

值得注意的是，PKLEE 目前无法达到 Ranged Analysis 表现最好的几个测试程序上的加速比. 由于 Ranged Analysis 使用了范围分析的技术，对符号执行算法本身进行了优化，因此可以在某些测试程序上达到大于 6x 的加速比，而 PKLEE 对此没有进行优化. 此外，PKLEE 由于 KLEE 自身设计的缺陷，同一时刻最多只有一个线程可以同外部库交互，这也导致了 PKLEE 在与外部库交互多的程序上表现不佳.

6 讨　论

6.1 线程同步开销

PKLEE 为了在 KLEE 的基础上实现多线程并行符号执行，引入了一定的线程同步开销. 一部分同步开销来自于底层数据结构的线程安全化，例如 KLEE 中使用的智能指针类 ref; 另一部分同步开销来自不同线程执行过程中对关键数据结构的同步保护，例如 KLEE 中的不同线程的工作队列.

为了削减底层数据结构带来的额外开销，本文使用更加轻量级的同步原语改写现有的数据结构，同时降低数据结构中的锁粒度，以减少不同线程的竞争，尽可能减少频繁使用底层数据结构带来的开销.

为了减少多线程执行过程中对于关键数据结构的竞争，本文使用更具备扩展性的数据结构改写 PKLEE 中的关键模块. 例如，使用无锁队列改写每个工作线程的工作队列.

6.2 KLEE 中其他约束求解优化技术的可扩展性

KLEE 中除了本文着重介绍的反例缓存外，还有诸多成熟的优化技术，其中多数优化技术都使用了缓存来提升效率.

基于求解结果缓存的优化直接缓存已有的约束求解结果，当遇到完全相同的约束求解信息时可以直接复用已有的解. 基于求解结果缓存的优化同样可以通过本文提出的多线程共享方式进一步被优化.

KLEE 中在进行约束求解之前，会对需要求解的约束进行一定的化简，例如对数组访问中的索引值可能的取值范围进行限定. KLEE 在执行此类优化过程中同样会缓存已经简化过的约束. 简化约束过程中产生的缓存信息在不同线程上可能存在冗余，也可以考虑使用共享的方式进行优化.

值得注意的是，目前 KLEE 中所有的缓存都没有设置缓存大小上限.随着符号执行的进行，优化信息的缓存将会占据越来越多的内存，同时查询缓存的时间也将越来越长[15]. 在未来的研究中，本文计划使用 LRU 等算法识别出当前被多个线程频繁命中的优化缓存信息，及时淘汰无用的信息，以减少内存开销以及每次扫描缓存的时间开销.

7 结 论

本文从现有并行符号执行引擎基于多进程模型通信开销大、无法有效利用现有的约束求解优化技术的问题出发,设计并实现了一种基于单进程多线程并行模型的并行符号执行引擎(PKLEE). 本文基于KLEE实现的原型PKLEE,不仅使用了基于多线程的无需中心线程的负载均衡方法,还通过不同线程共享彼此的约束求解信息来减少约束求解. 经过实验分析,在小规模程序的完整路径探索时间上,PKLEE在8个工作线程并行化下加速比为2.5x~4.0x;在大规模程序上的有效工作负载吞吐量这一指标上,PKLEE在8个工作线程下最终有效执行的指令数量可以达到KLEE的2.0x~4.8x.

本文工作依然存在一些不足,如线程同步的开销没有进一步削减和针对KLEE内部的数据结构进行适合并行化的改造. 未来工作中,本文将进一步分析PKLEE当前的瓶颈,并对其加以优化以达到更高的加速比;此外,本文也计划探索如何在多线程并行场景下,更有效地利用不同约束求解优化技术对已有的一些符号执行技术[23-25]在并行化场景下进行设计.

作者贡献声明: 周彭提出了算法思路,实现工具并撰写论文;左志强提出指导意见并修改论文.

参 考 文 献

[1] Baldoni R, Coppa E, D'elia D C, et al. A survey of symbolic execution techniques[J]. ACM Computing Surveys, 2018, 51(3): 1−39

[2] Elkarablieh B, Godefroid P, Levin M Y. Precise pointer reasoning for dynamic test generation[C] //Proc of the 18th Int Symp on Software Testing and Analysis. New York: ACM, 2009: 129−140

[3] Cadar C, Dunbar D, Engler D R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs[C] //Proc of the 8th USENIX Conf on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2008, 8: 209−224

[4] Felmetsger V, Cavedon L, Kruegel C, et al. Toward automated detection of logic vulnerabilities in web applications[C] //Proc of the 19th USENIX Security Symp. Berkeley, CA: USENIX Association, 2010: 143−160

[5] Bucur S, Ureche V, Zamfir C, et al. Parallel symbolic execution for automated real-world software testing[C] //Proc of the 6th Conf on Computer Systems. New York: ACM, 2011: 183−198

[6] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C] //Proc of Int Symp on Code Generation and Optimization, CGO. Los Alamitos, CA: IEEE Computer Society, 2004: 75−86

[7] Monniaux D. A survey of satisfiability modulo theory[C] //Proc of Int Workshop on Computer Algebra in Scientific Computing. Cham, Switzerland: Springer, 2016: 401−425

[8] Moura L, Bjørner N. Z3: An efficient SMT solver[C] //Proc of Int Conf on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2008: 337−340

[9] Dong Shiyu, Olivo O, Zhang Lingming, et al. Studying the influence of standard compiler optimizations on symbolic execution[C] //Proc of 2015 IEEE 26th Int Symp on Software Reliability Engineering (ISSRE). Piscataway, NJ: IEEE, 2015: 205−215

[10] Siddiqui J H, Khurshid S. ParSym: Parallel symbolic execution[C] // Proc of the 2010 2nd Int Conf on Software Technology and Engineering. Los Alamitos, CA: IEEE Computer Society, 2010, 1: 405−409

[11] Staats M, Păsăreanu C. Parallel symbolic execution for structural test generation[C] //Proc of the 19th Int Symp on Software Testing and Analysis. New York: ACM, 2010: 183−194

[12] King A. Distributed parallel symbolic execution[D]. Kansas: Kansas State University, 2009

[13] Rakadjiev E, Shimosawa T, Mine H, et al. Parallel SMT solving and concurrent symbolic execution[C] //Proc of 2015 IEEE Trustcom/BigDataSE/ISPA. Piscataway, NJ: IEEE, 2015, 3: 17−26

[14] Singh S, Khurshid S. Parallel chopped symbolic execution[C] //Proc of Int Conf on Formal Engineering Methods. Cham, Switzerland: Springer, 2020: 107−125

[15] Karna A K, Du Jinbo, Shen Haibo, et al. Tuning parallel symbolic execution engine for better performance[J]. Frontiers of Computer Science, 2018, 12(1): 86−100

[16] Siddiqui J H, Khurshid S. Scaling symbolic execution using ranged analysis[J]. ACM Sigplan Notices, 2012, 47(10): 523−536

[17] Wei Guannan, Tan Shangyin, Bračevac O, et al. LLSC: A parallel symbolic execution compiler for LLVM IR[C] //Proc of the 29th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2021: 1495−1499

[18] Bugrara S, Engler D. Redundant state detection for dynamic symbolic execution[C] //Proc of the 2013 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2013: 199−211

[19] Yi Qiuping, Yang Zijiang, Guo Shengjian, et al. Postconditioned symbolic execution[C] //Proc of the 2015 IEEE 8th Int Conf on Software Testing, Verification and Validation. Piscataway, NJ: IEEE, 2015: 1−10

[20] Jaffar J, Murali V, Navas J A, et al. TRACER: A symbolic execution tool for verification[C] //Proc of the Int Conf on Computer Aided Verification. Berlin: Springer, 2012: 758−766

[21] Krishnamoorthy S, Hsiao M S, Lingappan L. Tackling the path

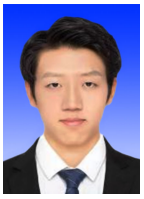
explosion problem in symbolic execution-driven test generation for programs[C] //Proc of the 19th IEEE Asian Test Symp. Los Alamitos, CA: IEEE Computer Society, 2010: 59−64

[22] Li You, Su Zhendong, Wang Linzhang, et al. Steering symbolic execution to less traveled paths[J]. ACM SigPlan Notices, 2013, 48(10): 19−32

[23] Wang Haijun, Liu Ting, Guan Xiaohong, et al. Dependence guided symbolic execution[J]. IEEE Transactions on Software Engineering, 2016, 43(3): 252−271

[24] Santelices R, Harrold M J. Exploiting program dependencies for scalable multiple-path symbolic execution[C] //Proc of the 19th Int Symp on Software Testing and Analysis. New York: ACM, 2010: 195−206

[25] Liu Jingde, Chen Zhenbang, Wang Ji. Solving cost prediction based search in symbolic execution[J]. Journal of Computer Research and Development, 2016, 53(5): 1086−1094



Zhou Peng, born in 2000. Master candidate. His main research interests include system software and symbolic execution.

周　彭, 2000 年生. 硕士研究生. 主要研究方向为系统软件, 符号执行.

Zuo Zhiqiang, born in 1986. PhD, associate professor. Senior member of CCF. His main research interests include system software, compilers and program analysis.

左志强, 1986 年生. 博士, 副教授. CCF 高级会员. 主要研究领域为系统软件、编译技术、程序分析.