软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(9):3435−3456 [doi: 10.13328/j.cnki.jos.007602] [CSTR: 32375.14.jos.007602]

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

基于按需切片计算的并行化程序分析框架*

李俊锋, 苑旭东, 魏晨雨, 厉剑豪, 张迎周

(南京邮电大学 计算机学院、软件学院、网络空间安全学院, 江苏 南京 210023)

通信作者: 张迎周, E-mail: zhangyz@njupt.edu.cn

摘　要: 传统程序分析方法在处理大规模软件时, 常需显式构建依赖图 (如系统依赖图、程序依赖图), 因程序实体与依赖关系数量随规模呈指数级增长而面临“构图困难”与冗余计算的性能瓶颈. 为应对此挑战, 提出一种基于按需切片计算的并行化程序分析框架. 该框架融合了符号化切片与高阶函数摘要技术, 其核心思想是: (1) 通过将函数调用的影响抽象为可复用的高阶函数式摘要, 通过动态函数求值替代显式依赖图存储与遍历, 从根本上规避了构图瓶颈; (2) 设计了一种针对高阶函数摘要的按需实例化机制, 基于用户指定的切片准则 (如关键变量), 动态地、惰性地实例化摘要, 仅计算与分析目标直接相关的依赖, 从而大幅削减了计算冗余. 该框架通过子分析摘要的独立性和惰性实例化特性, 实现了任务的细粒度划分, 展现出良好的并行潜力. 实验结果表明, 该框架下的按需切片计算较符号化切片工具 (SymPas) 在分析时间上减少了 65.3%, 内存峰值下降了 68.2%; 其并行化版本 (6 线程) 时间进一步减少 93.8%, 并行效率接近 90%. 为验证该框架在大型程序分析中的实用性与有效性, 将该框架应用于访问越界缺陷检测, 成功识别了 132 个真阳性实例, 为解决大规模软件分析的性能瓶颈提供了新的形式化解决思路.

关键词: 程序分析; 程序切片; 程序分析框架

中图法分类号: TP311

中文引用格式: 李俊锋, 苑旭东, 魏晨雨, 厉剑豪, 张迎周. 基于按需切片计算的并行化程序分析框架. 软件学报, 2026, 37(9): 3435–3456. http://www.jos.org.cn/1000-9825/7602.htm

英文引用格式: Li JF, Yuan XD, Wei CY, Li JH, Zhang YZ. Parallel Program Analysis Framework Based on On-demand Slicing Calculation. Ruan Jian Xue Bao/Journal of Software, 2026, 37(9): 3435–3456 (in Chinese). http://www.jos.org.cn/1000-9825/7602.htm

Parallel Program Analysis Framework Based on On-demand Slicing Calculation

LI Jun-Feng, YUAN Xu-Dong, WEI Chen-Yu, LI Jian-Hao, ZHANG Ying-Zhou

(School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China)

Abstract: Traditional program analysis methods, when dealing with large-scale software, often require explicit construction of dependency graphs (such as system dependency graphs and program dependency graphs). However, as the number of program entities and dependency relationships grows exponentially with the scale of software, these methods face performance bottlenecks of “graph construction difficulty” and redundant computation. To address this challenge, this study proposes a parallel program analysis framework based on on-demand slicing computation. The framework integrates symbolic slicing and higher-order function summary techniques, and its core ideas are as follows. (1) By abstracting the impact of function calls into reusable higher-order function summaries and replacing explicit dependency graph storage and traversal with dynamic function evaluation, the graph construction bottleneck is fundamentally avoided. (2) An on-demand instantiation mechanism for higher-order function summaries is designed, which dynamically and lazily instantiates summaries based on user-specified slicing criteria (e.g., key variables), computing only dependencies directly relevant to the analysis target, thereby significantly reducing computational redundancy. Through the independence of sub-analysis summaries and the lazy instantiation feature, this framework enables fine-grained task partitioning, demonstrating good parallelization potential. Experimental results show that the on-demand slicing computation under this framework reduces analysis time by 65.3% and peak memory usage by 68.2% compared to the

* 本文由“形式化方法与应用”专题特约编辑安杰副研究员、顾斌研究员、李建文教授推荐.
收稿时间: 2025-09-06; 修改时间: 2025-10-28; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-24
CNKI 网络首发时间: 2026-05-14

symbolic slicing tool SymPas. Its parallel version (6 threads) further reduces analysis time by 93.8%, with parallel efficiency approaching 90%. To verify the practicality and effectiveness of the framework in large-scale program analysis, it is applied to out-of-bounds access defect detection, successfully identifying 132 true positive instances, providing a new formal solution for addressing performance bottlenecks in large-scale software analysis.

Key words: program analysis; program slicing; program analysis framework

在软件工程领域, 以静态分析、模型检验和约束求解为代表的形式化方法是保障软件可靠性与安全性的关键技术手段. 其中, 高精度的程序分析 (如路径敏感分析、过程间分析) 对计算资源的需求极高. 随着软件系统规模的持续扩展, 现有的分析技术在处理大规模程序时面临严重的效率瓶颈, 显著限制了其在实际工程场景中的应用. 以程序切片为例, 传统切片方法在计算程序依赖关系时, 通常需要先构建显式的依赖图结构, 如系统依赖图 (system dependency graph, SDG) 或程序依赖图 (program dependency graph, PDG) 等[1]. 在构建这些图之后, 通过在图上进行可达性分析来获得与兴趣点相关的程序切片. 然而, 随着程序规模的扩大, 程序中包含的变量或语句之间的依赖关系数量呈指数级增长, 导致图结构中的节点与边数急剧上升, 从而引发“构图困难”问题. 该问题在大型程序中尤为突出, 常导致基于图结构的分析框架在构图阶段即因资源消耗过大而失效, 或在后续的分析过程中出现显著的时间与空间开销, 严重影响其实用性.

针对当前程序分析面临的性能与扩展性瓶颈, 我们团队在前期研究中围绕两个关键方向开展了探索工作: (1) 符号化程序切片 SymPas: 通过将程序切片的结果符号化, 使其依赖关系可以被参数化, 从而实现切片的惰性计算. (2) 高阶函数摘要: 将函数的摘要形式化为一个高阶函数, 通过函数应用替代摘要合并, 以更高的精度处理过程间上下文. 这两种分析方法均着眼于避免显式构图, 从而为解决大规模程序分析中的构图困难问题提供了可行的解决思路. 本研究在此基础上, 进一步整合并拓展了相关技术, 旨在构建一个适用于并行环境下的高效程序分析框架.

尽管基于高阶函数摘要的切片方法在分析效率上相较于传统依赖图方法已有显著提升, 但在精度控制与并行性方面仍存在进一步优化空间. 如图 1 所示, 示例程序中的 main 函数调用函数 *func* 计算变量 %7 和 %8 的和并将结果存储至 %9, 同时函数 *func* 对变量 %7 执行加一操作. 将 main 函数中的 %7 作为切片准则, 则在该准则下函数 *func* 仅需考虑第 1 个参数相关的操作, 但在目前基于函数摘要的切片方法当中, 对函数摘要的使用通常是整体的, 从而带来了冗余计算. 此外, 图 1 示例程序中的调用链 main→*func* 带来了严格的分析次序约束, 导致并行分析算法因任务依赖而丧失并行性, 效率低下甚至退化为串行执行.

由上述例子可知本团队前期工作主要聚焦于串行分析的精度和表达能力, 容易产生冗余分析且难以有效利用现代多核处理器的并行计算能力. 因此, 迫切需要构建一种理论上更适合并行计算、同时具备更加轻量化特征的形式化分析框架.

针对以上需求, 本文提出了一种基于按需切片计算的并行化程序分析框架. 该框架以我们前期的符号化程序切片与高阶函数摘要技术为基础, 融合两者优势并将其推广至并行计算环境中. 通过引入高阶函数摘要机制, 框架能够在过程间分析中对依赖信息进行压缩表达, 从而有效规避传统依赖图构建中面临的内存爆炸问题. 通过将高阶函数摘要局部化地、按需地应用于副作用相关的子摘要, 本框架减少了冗余计算, 实现了精确的按需切片. 更重要的是, 高阶函数摘要之间以及不同子分析摘要之间的独立性, 为程序分析的并行化提供了坚实的理论基础. 本文通过多组对比实验, 系统性地验证了该框架中按需切片计算方法的有效性, 及其在大型程序分析任务中所展现出的时间和空间性能优势. 同时, 通过将并行化分析框架应用于访问越界检测任务, 进一步验证了框架在程序分析领域中的实用性.

本文第 1 节介绍现有程序分析中程序切片与并行分析的相关方法和研究现状. 第 2 节介绍基于按需切片计算的并行化程序分析框架. 第 3 节详细介绍框架中的核心模块以及技术实现. 第 4 节通过对比实验验证所提按需切片计算与现有切片算法的效率优越性体现, 并通过将本文所提框架应用于访问越界检测来验证本框架的可行性. 最后总结全文.

```
1  define dso_local void @func(ptr noundef %0, ptr noundef %1, ptr noundef %2){
2    %3 = load i32, ptr %0, align 4
3    %4 = load i32, ptr %1, align 4
4    %5 = add nsw i32 %3, %4
5    store i32 %5, ptr %2, align 4
6    %6 = add nsw i32 %3, 1
7    store i32 %6, ptr %0, align 4
8    ret void
9  }
10
11
12 Define dso_local i32 @main(){
13   %7 = alloca i32, align 4
14   %8 = alloca i32, align 4
15   %9 = alloca i32, align 4
16   store i32 2, ptr %7, align 4
17   store i32 3, ptr %8, align 4
18   call void @func(ptr noundef %7, ptr noundef %8, ptr noundef %9)
19   %10 = load i32, ptr %9, align 4
20   ret i32 %10
21 }
```

图 1　高阶函数摘要程序切片分析示例

1　相关工作

1.1　程序切片相关工作

程序切片技术是程序工程领域中的一种基础且重要的分析技术, 由 Weiser[2]提出. 在程序分析中, 开发者们通常需要获得程序中更丰富的语法语义信息, 程序的语义信息不仅包含代码的字面逻辑, 更隐含在语句间的控制流、数据流及调用关系中. 通过构建程序依赖图 (PDG)、系统依赖图 (SDG) 等图结构, 可以将这些隐含的依赖语义信息转化为图中的边, 使隐含的语义信息可视化.

现有的大多研究都聚焦于图的扩展与优化[1]. Jia 等人[3]基于程序依赖图提出了一种动态切片方法, 通过结合输入的具体执行信息生成更精确的切片. Chalupa[4]提出一种基于依赖图的新型切片器 DG, 结合了数据依赖和控制依赖分析. Joern 工具将抽象语法树、控制流图等叠加起来, 形成代码属性图[5], 提升分析通用性. Galindo 等人[6]提出了异常敏感的系统依赖图 ES-SDG, 通过条件控制依赖的引入确保切片完整性. Galindo 等人[7]还提出对象流依赖和对象引用依赖并扩展了 JSysDG 的表示能力. Halder 等人[8]提出了数据库导向的程序依赖图 DOPDG, 并在此基础上进行抽象域的提升和细化, 最终通过处理依赖条件图来生成抽象切片.

基于图结构的程序切片算法一经提出受到广泛关注, 但基于图的缺陷同样显著, 在最初的构造图结构到后续对图结构进行遍历的过程中, 不可避免地要面临构建图开销大、查询复杂度较高等难题. 在处理大型程序中, 基于图的切片算法会因为难以构造庞大的图而无法分析程序. 基于上述问题, 本团队的前期工作提出了 SymPas[9]这一符号化切片工具, 将指令依赖表作为依赖结构, 避免图的构建. 同时在过程间分析时, 提出了符号化过程摘要, 通过将参数符号化来使切片结果被多次调用复用, 大量减少计算量. SymPas 在保持切片精度的同时, 显著降低了计算和存储开销, 具有高效性以及可扩展性.

近年来, 程序切片研究开始尝试借助语言模型从源代码中学习潜在的依赖关系, Yadavall 等人[10]提出新型预测程序切片方法 NS-Slicer, 通过大模型学习变量与语句之间的上下文依赖关系, 从而在无需显式构造 PDG 的前提下实现对切片路径的隐式建模.

除了上述关注变量具体值的切片算法, 还有学者提出了基于抽象解释的程序切片. Mastroeni 等人[11]提出了一种基于抽象解释的抽象程序切片. 其通过分析变量对目标属性的语义依赖来更精确地提取与特定计算相关的程序语句, 这种抽象切片通过弱化传统的语法依赖, 仅保留影响目标属性的语句, 显著减少了冗余代码.

1.2 并行分析相关工作

在实际应用中, 效率问题始终是制约静态分析技术落地的瓶颈. 传统图结构方法的高开销、大模型的高推理资源需求均限制了大规模代码分析的效率. 因此, 研究者们探索了多种并行化分析的路径. 现有的并行化静态分析路径主要可以分为以下 3 类: 任务级并行、分布式数据并行和硬件加速并行.

在任务级并行中, Méndez-Lojo 等人[12]通过将指针分析中的图约束求解过程并行化, 实现了任务级多线程加速. 这类方法主要将程序分析视为一个大的迭代求解过程, 其并行性主要体现在求解阶段. 例如, Datalog 求解器的研究[13–15]中将分析问题转化为逻辑规则集, 自动对规则的评估过程进行粗粒度的并行化. 然而, 这些方法本质上仍然是全量分析, 需要求解整个程序状态空间, 可能产生较大的计算冗余.

在分布式数据并行中, 以 BigSpa[16]为代表的分布式数据并行方法通过 MapReduce 等模型将程序分析任务映射到大规模集群. 其并行粒度通常较粗, 例如将程序按模块或组件划分. BigSpa 将程序模块映射为组件级可独立分析的子任务, 这种方法在处理超大规模、结构松散的代码库时展现出良好的可扩展性. 然而其粗粒度的任务划分可能导致严重的负载不均, 且节点通信与调度开销较大, 导致不适合处理依赖关系紧密的细粒度分析.

在硬件加速并行的方法中, 以 GPU-base 方法[17,18]为代表的多核并行方法利用 GPU 数以千计的核心来加速数据流分析中的矩阵运算或图遍历. 其并行粒度极细, 可达单个约束或图节点. 但其瓶颈在于程序切片这类分析具有不规则的内存访问模式和复杂的控制流依赖, 难以高效映射到 GPU 的架构上, 限制了其通用性与适用性.

本文提出的框架则采用一种不同的方式. 本框架融合高阶函数摘要技术, 从根本上规避了传统图结构方法构图的高开销. 其并行粒度是中等的, 以高阶函数摘要或变量为单位划分任务, 既避免了分布式数据并行中常见的粗粒度负载不均问题, 也绕开了 GPU 加速面临的不规则依赖映射难题.

与前述 3 类方法相比, 本框架的核心优势在于按需性与惰性计算: 它通过避免构建全局依赖图, 仅根据切片准则按需实例化摘要, 生成和处理与分析目标直接相关的最小必要计算任务. 因此, 它从根本上减少了总计算量, 它特别适用于需要高精度、上下文敏感的程序切片场景, 而这正是前述依赖全量分析或粗粒度并行化方法难以高效处理的.

2 基于按需切片计算的并行化分析框架

随着软件规模的持续增长, 传统程序分析框架在处理大规模代码时面临构图困难与冗余计算的双重挑战. 因此, 本文提出一种基于按需切片计算的并行化分析框架来解决上述问题. 本节首先介绍将按需切片技术模块化的程序分析框架, 从整体框架、模块交互与工作流程两方面进行阐述, 关于核心模块设计以及关键技术细节放在第 3 节详细介绍.

2.1 整体框架

框架整体流程见图 2. 框架采用模块化设计, 主要包含 4 个核心模块: 需求分析模块、高阶函数摘要计算模块、并行按需切片计算模块以及后续分析模块.

需求分析模块作为框架的需求翻译器, 负责将用户分析需求转化为具体兴趣点 (如关键变量、指令等), 以指导后续切片, 其设计直接影响分析的准确性与效率. 高阶函数摘要计算模块通过符号化过程间依赖关系, 将函数调用抽象为可复用的摘要, 避免传统依赖图的高内存开销. 并行按需切片计算模块则基于兴趣点, 动态调度已生成的高阶函数摘要进行并行切片, 仅执行最小必要计算, 在避免冗余的同时提升分析速度. 后续分析模块接收切片生成的子程序, 结合原始需求进行针对性分析, 并输出最终结果. 其中, 高级函数摘要计算与并行按需切片是框架的核心, 具体实现详见第 3 节.

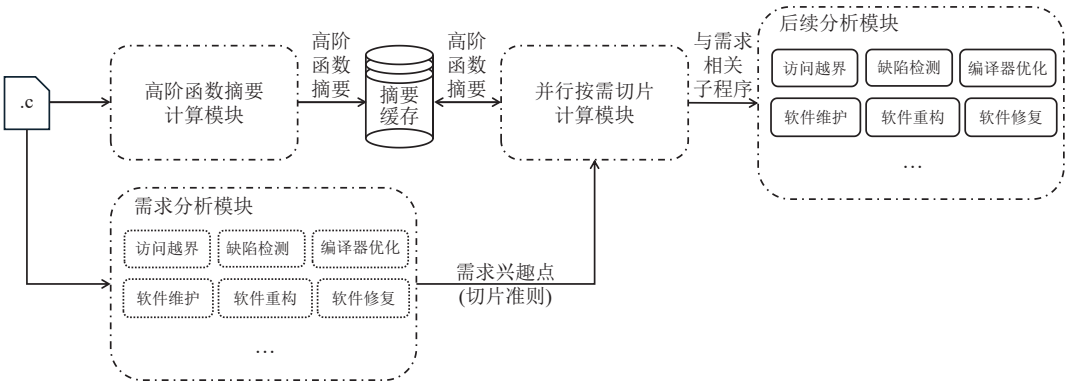

图 2 基于按需切片计算的并行化程序分析框架图

2.2 模块交互与工作流程

该框架构建了一条从需求到结果的高效分析流水线: 首先由需求分析模块将用户任务转化为具体的切片准则, 随后高阶函数摘要计算模块对源代码进行符号化处理, 生成可复用的高阶函数摘要并存入缓存, 在此过程中可利用可达性分析剔除死代码以避免冗余开销; 接着, 并行按需切片计算模块根据切片准则, 从缓存中调取相关摘要进行并行实例化, 生成仅包含核心依赖的子程序片段; 最后由后续分析模块对这些片段进行针对性分析并输出结果, 这种只关注兴趣点相关语句的机制极大降低了分析大规模程序所需的时间与空间消耗.

3 核心模块设计与技术实现

本框架的核心模块包括高阶函数摘要计算与并行按需切片计算模块. 在前期工作中, 我们将函数摘要形式化为高阶函数, 通过函数应用替代摘要合并, 将其称为高阶函数摘要, 并使用高阶函数摘要提出了一种依赖簇分析[19]. 高阶函数摘要将形参和全局变量的数据依赖作为摘要参数, 以匿名函数形式保存过程间依赖分析结果, 在调用点处通过直接实例化高阶函数摘要来快速获取过程间依赖关系, 无需像 SymPas 递归处理符号参数, 进一步降低了计算复杂度.

框架的核心思想如图 3 所示, 可概括为“摘要即模版, 切片即实例化”. 高阶函数摘要相当于函数的“动态模版”, 描述了其对输入和全局变量的影响, 避免重复处理同一个函数. 在调用时, 只需将实参填入模版完成实例化. 高阶函数摘要计算模块负责生成这个“动态模板”, 并行按需切片计算模块则执行模版实例化.

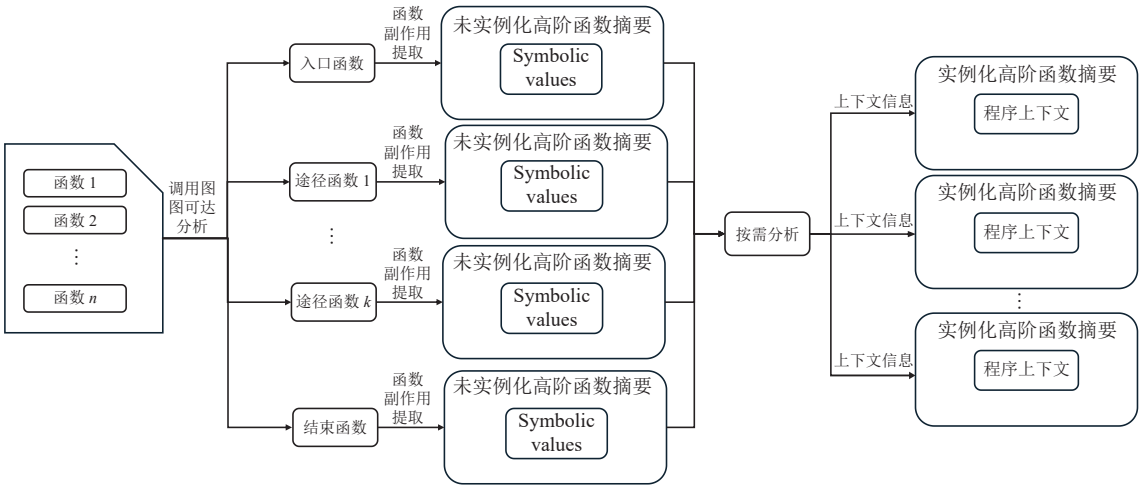

图 3 框架核心思想图

为了表示分析过程中产生的中间状态, 本文设计了一种切片分析摘要及子切片分析摘要. 切片分析摘要用于存储指令、变量、全局变量等程序元素的依赖信息, 为高阶函数摘要计算与切片实例化提供统一的操作结构. 同时, 参考符号化切片的思想, 实现分析摘要实例化过程中的惰性计算, 这一操作可以大大减少分析过程中的计算开销. 高阶函数摘要能够符号化地表示函数调用对切片计算结果产生的影响, 消除了计算切片摘要时各摘要之间的计算依赖关系, 使得方法在计算各函数的函数摘要时能够拥有较高的并行效率. 此外本文设计的子切片分析摘要也能够有效降低切片实例化的任务粒度.

3.1 高阶函数摘要计算模块

高阶函数摘要计算模块具体架构流程见图 4. 模块将接收的源程序使用 LLVM 编译为 LLVM IR, 对 LLVM IR 进行分析, 对其函数摘要计算进行任务分片, 之后将任务放入摘要计算任务池, 由任务池负责进行高阶函数摘要计算任务的分配. 进行并行化计算后, 将摘要存储至摘要缓存中. 下面介绍高阶函数摘要计算的算法实现.

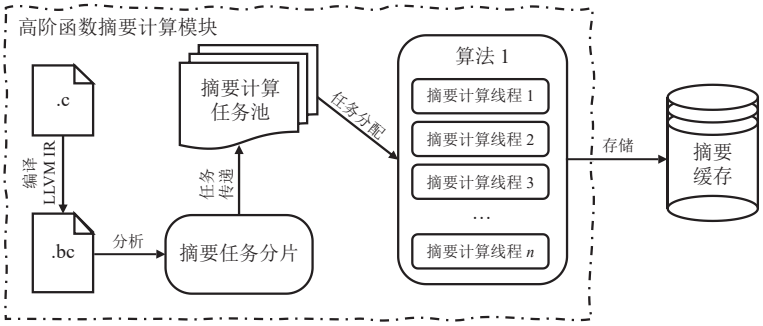

图 4 高阶函数摘要计算模块

首先介绍模块在分析过程中产生的中间摘要.

定义 1. 切片分析摘要. 在对目标程序切片分析计算的过程中, 对某个函数中的指令或是语句块进行分析时的环境称为切片分析摘要 (以下简称为分析摘要), 其类型记为 A. 分析摘要中允许通过符号值表示尚未确定的外部输入变量 (如函数参数和全局变量), 即对任意函数 f, 其外部输入变量集合 $InputValues_f=\{a_0,a_1,\ldots,a_n\}$, 在分析函数 f 时的任意形式化分析摘要 s 中, 有符号值集合 $SymbolValues_s=\{sv_i \mid 0\leqslant i\leqslant n, sv_i$存在于$s\}$ 且集合 $InputValues_f$ 与集合 $SymbolValues_s$ 存在双射 $\pi\subseteq InputValues_f\times SymbolValues_s$ 使得 $\pi(a_i)=sv_i$. 本文方法中, 分析摘要 A 由 I、V、P、G 和 R 这 5 个不同类型切片表以及一个调用点信息表 *callSiteMap* 构成, 即:

$$A:=<I,V,P,G,R,callSiteMap> \tag{1}$$

其中, 每个切片表中的元素为形如<*key*, *set*> 的二元组, *key* 为 LLVM 变量, 而 *set* 表示 *key* 的切片 (形式上是一组由 LLVM 指令构成的集合). 在这 5 个类型的切片表中, I 表示指令的切片表; V 表示变量的切片表; P 表示传入地址 (即指针) 的切片表; G 表示全局变量的切片表; R 表示返回值对应的切片表. 在本文当中, 对于任意分析摘要 s 以及任意 LLVM 指令 (变量) x, 从分析摘要 s 中相应的切片表中取出指令 (变量) x 对应的切片可以表示为 $I_s(x)$ (或 $V_s(x)$, $P_s(x)$, $G_s(x)$, $R_s(x)$), 在摘要计算的过程中这些切片的下标可以省略. 调用点信息表 *callSiteMap* 以元组为基本元素, 每个元组包含了调用点及其涉及的实参对应的切片. 调用点信息表的引入是为了在实例化分析摘要时能够准确还原函数调用的输入环境.

在对程序进行分析的过程中, 大部分情况下仅需对部分相关环境进行操作即可. 为此本文将一种符号化子切片分析摘要作为分析的基本单元.

定义 2. 子切片分析摘要. 构成分析摘要 s 的任意一个部分或是子集称为分析摘要 s 的子切片分析摘要 (以下简称子分析摘要) n_s, 其类型记为 N, 子分析摘要仍是一个分析摘要, 即 $N<:A$.

子分析摘要作为切片算法的基本分析单元, 能够有效表示程序中任意一点的部分摘要. 构成切片分析摘要 s 的 5 个切片表中的任意数量的元素组成的集合以及其中包含的调用点处的调用点信息都可以组成 s 的子分析摘

要 n_s (n_s 仍保留切片分析摘要的结构, 即 n_s 也是由 I、V、P、G、R 以及 *callSiteMap* 组成的). 根据定义 1 可知, 切片分析摘要本质上就是函数中各类元素的切片集合, 因此对切片的计算就是对切片分析摘要中的特定子分析摘要的计算.

在基于高阶函数思想的纯函数式编程语言当中, 函数作为一等公民可以作为参数或是返回值, 而摘要之间的相互依赖关系也可以表述为一个摘要是另一个摘要的参数, 因此在函数副作用以及分析摘要的基础上, 本文通过高阶函数思想设计了一种高阶函数摘要用于过程间分析.

定义 3. 高阶函数摘要形式. 对于任意函数 *func*, 其高阶函数摘要 SUM_{func} 的形式如下:

$$SUM_{func}(SUM_{f_1}, SUM_{f_2}, \ldots, SUM_{f_n}) : (A \to A) \tag{2}$$

高阶函数摘要作为一个函数其在接收一个分析摘要 (该摘要可以为空) 之后返回一个新的分析摘要, 该过程表示传入摘要经过高阶函数摘要对应的函数调用影响后产生了一个新的分析摘要. 当目标函数调用其他函数时, 被调用函数的高阶摘要会作为参数直接嵌入调用方的摘要定义中 (即函数作为参数), 整个过程可以视为对函数副作用的符号化.

假设存在函数 f_0 中的一程序点处的分析摘要为 s_0, 在该程序点处之后是一个调用语句 $call_1$, 该语句对函数 f_1 进行了调用, 函数 f_1 的高阶函数摘要为 SUM_{f_1}, 则在调用语句 $call_1$ 处的分析摘要 s_1 可以形式化地表示为:

$$s_1 = SUM_{f_1} s_0 \tag{3}$$

需要注意的是此处被调用函数 f_1 的摘要 SUM_{f_1} 的具体内容在分析 f_0 时是未知的, 因此此处无法通过 SUM_{f_1} 计算出 s_1 最终的结果. 本文将通过函数 f_1 的摘要 SUM_{f_1} 以及 s_0 计算 s_1 具体值的过程称为高阶函数摘要的应用, 将存在未应用高阶函数摘要或是符号值的分析摘要称为未实例化的分析摘要.

根据定义 1 与定义 2 对分析摘要以及子分析摘要的定义可知, s_0 可以包含多个子分析摘要, s_1 的形式化表示中的高阶函数摘要 SUM_{f_1} 作用于整个分析摘要 s_0. 从上文中对函数副作用的定义可知, 与函数副作用无关的子分析摘要不会受到函数调用的影响, 因此可以认为函数调用对分析环境的影响是局部的, 其具体影响的部分 (即子分析摘要) 可以根据被调用函数的副作用唯一确定. 此外, 由于部分应用的高阶函数摘要不再作用于整个分析摘要, 因此对函数的调用还需考虑函数输入相关的分析子摘要. 因此本文引入副作用提取函数以获得受到被调用函数副作用影响的子分析摘要.

定义 4. 副作用提取函数. 对于一个调用语句 $instCall_f$ 以及该语句调用的函数 f, 设在该语句处的分析摘要为 s_0, 则对该函数调用的副作用提取函数为:

$$\begin{aligned} &ExtractSideEffect : ((InstCall_f \to A) \to A) \\ &= \lambda instCall : InstCall_f.\lambda s : A.\{n \mid n \subseteq s_0, n_1 = SUM_f n,\ n_1 \neq n\} \end{aligned} \tag{4}$$

通过副作用提取函数可以得到受到被调用函数副作用影响的子分析摘要, 最终通过将高阶函数摘要应用到这些子分析摘要以实现对高阶函数摘要的部分应用.

最后, 本文引入摘要更新函数, 该函数用于通过一个分析摘要更新另一个分析摘要, 其定义如下.

定义 5. 摘要更新函数. 对于一个分析摘要 s_0 以及另一个分析摘要 s_1, 则通过 s_1 将 s_0 更新的摘要更新函数为:

$$Update : (A \to A \to A) = \lambda s_0 : A.\lambda s_1 : A.\{n \mid n \subseteq s_0, n \notin s_1\} \cup s_1 \tag{5}$$

根据定义 4、定义 5, 可以得到上文中函数 f_1 的高阶函数摘要对分析摘要 s_0 的局部应用表示:

$$s_1 = Updates_0 SUM_{f_1} ExtractSideEffect(instCall_{f_1}, s_0) \tag{6}$$

高阶函数摘要的局部应用使得摘要不再作用于整个分析摘要上, 在进行按需计算时避免了函数摘要的重复使用. 在本文方法中, 高阶函数摘要的应用均采用局部应用的方式将其应用到副作用对应的子分析摘要当中, 为了方便表示, 上文中 s_1 所表示的高阶函数摘要局部应用将缩写为:

$$s_1 = SUM_f^l s_0 \tag{7}$$

在切片分析摘要中, 传入地址切片表 P、全局变量切片表 G 与返回值切片表 R 涵盖了函数调用时可能对外产生影响的全部变量, 因此可以在调用点处将实参或是全局变量对应的切片 (即子分析摘要) 结合被调用函数的

切片分析摘要来还原函数调用产生的副作用, 即在接收一个分析摘要后产生调用影响后的一个新分析摘要. 因此计算函数的切片高阶函数摘要就是计算函数的最终切片分析摘要.

本文方法采用数据流迭代的方式, 对目标函数中类型为 A 的符号化分析摘要进行求解, 通过遍历函数的控制流图 (control flow graph, CFG), 对 A 中的 I (指令切片表)、V (变量切片表)、P (参数输入切片表)、G (全局变量切片表)、R (返回值切片表) 进行更新计算. 在更新计算的同时对调用点信息集进行更新.

指令切片表 I 作为本文切片方法的基础, 在沿着 CFG 对每条指令进行遍历分析时先进行计算与更新. 指令切片表记录了每条 LLVM 指令的指令切片, 指令切片由数据依赖与控制依赖两个部分组成, 它们的定义如下.

定义 6. 数据依赖. 一条指令的数据依赖可以通过指令参数对应的指令依赖以及变量依赖计算得到, 其定义如下:

$$DD:((Inst\to A)\to Set_v)=\lambda i.\lambda s.\bigcup\nolimits_{x\in UsedInst_i} I_s(x)\cup\bigcup\nolimits_{x\in UsedValue_i} V_s(x) \tag{8}$$

其中, $UsedInst_i$ 为指令 i 的参数集合 (该集合包含指令 i 自身), $UsedValue_i$ 为指令 i 指代内存变量的参数集合. 数据依赖表示该指令的结果可能由哪些指令、变量或是值直接决定. 由数据依赖的定义可知, 指令 i 的数据依赖由指令参数对应的指令切片以及指令参数中是内存变量的变量切片构成.

定义 7. 控制依赖. 一条指令的控制依赖可以通过目标为指令所在基本块的条件跳转指令计算得到, 其定义如下:

$$CD:((Inst\to A)\to Set_v)=\lambda i.\lambda s.\bigcup\nolimits_{x\in BlockPreBrValue_i} I_s(x) \tag{9}$$

其中, $BlockPreBrValue_i$ 表示目标是指令 i 所在基本块的条件跳转指令集合. 控制依赖表示该指令的执行与否受到哪些指令的影响. 由控制依赖的定义可知, 指令 i 的控制依赖由可以通过能够到达指令所在的基本块的条件跳转指令对应的指令依赖组成.

定义 8. 指令依赖. 一条指令的指令切片 $I(i)$ 可以通过指令 i 的数据依赖 DD_i 以及控制依赖 CD_i 指令依赖计算得到, 其定义如下:

$$I(i)=DD_i\cup CD_i \tag{10}$$

由指令依赖的定义可知, 一条指令的指令切片可以通过对其数据依赖以及控制依赖求并集得到. 一条指令的指令切片表示了该指令的结果可能由哪些指令或是变量决定, 即以该指令为切片准则得到的切片.

虽然 LLVM IR 在语法上是静态单赋值的, 但 LLVM IR 允许通过内存操作指令对同一个 LLVM 变量所对应的内存进行多次写入, 为了能够收集到这些内存操作带来的变量间依赖关系, 本文方法通过变量切片表记录这些内存变量的依赖集. 与指令切片表的计算更新不同, 变量切片表仅会在遇到内存操作指令时才会依据指令类型对变量切片表进行更新.

当遍历分析指令遇到 alloc、load 和 store 等指令时, 算法首先尝试对这些指令操作的内存变量的切片进行初始化. 即使得 $V(x)$={}. 当遇到 store 等内存写入指令时, 为被写入的内存变量及其别名的变量切片进行更新. 若内存写入的目标为全局变量, 还需要对全局变量切片表进行更新.

定义 9. 变量切片更新函数. 内存变量 v 的变量切片更新计算方法如下:

$$ValueUpdateByStore:((Value_{\mathrm{llvm}}\to Inst\to A)\to Set_v)=\lambda v.\lambda i.\lambda s.V_s(v)\cup I_s(i) \tag{11}$$

变量切片更新函数通过将当前内存写入指令的指令切片并入变量切片的方式实现对目标变量切片表的更新. 分析摘要中的 P、G、R 统称为副作用切片表, 该表表示函数被调用时对调用点处的分析摘要产生的影响. 其中输入参数切片表 P 在 I、V 到达不动点后通过函数参数以及对应的变量切片结果进行计算; 全局变量切片表在遍历时根据内存操作指令进行更新; 返回值切片表则是在遇到函数返回指令时进行更新.

在进行高阶函数摘要计算之前, 框架需要获取程序中的别名信息以精确处理指针操作. 本框架依赖于一个标准的别名分析模块, 该模块实现了一种上下文不敏感、域不敏感的 Andersen 风格指针分析. 在实践中, 我们利用了 LLVM 生态中成熟的指针分析 Pass:llvm/Analysis/CFLAndersAliasAnalysis 来完成此任务. 需要强调的是, 在为单个函数生成高阶函数摘要时, 别名查询是过程内的. 然而本框架通过高阶函数摘要的符号化和实例化机制, 使得过程间的指针依赖关系得以在调用点被正确地重建和传播, 从而在整体上达到了过程间分析的效果.

切片的高阶函数摘要计算过程如算法 1 所示.

算法 1. 切片的高阶函数摘要计算.

输入: 待分析的函数 *func*;
输出: 符号化分析摘要 *s*.

1. **Function** *CalculateSliceSummary*(*func*)
2. 将 s 中 5 个切片表 i_s, v_s, p_s, g_s, r_s 以及 $callSiteMap_s$ 初始化为空
3. **For each** $arg \in args_{func}$ **do**
4. $v_s(arg) \leftarrow \{arg\}$
5. **End for**
6. **While** s_{out} 被修改 **do**
7. **For each** $node \in CFG_{func}$ **do**
8. **For each** $inst \in node$ **do**
9. $i_s(inst) \leftarrow i_s(inst) \cup CD(inst, s) \cup DD(inst, s)$ //公式 (10)
10. **Switch** *inst* **do**
11. **Case** *inst* 是内存操作指令:
12. 初始化 $v_s(inst)$
13. **If** *inst* 是内存写入指令
14. **Then**
15. **For each** *alias* ∈ 写入变量及其全部别名 **do**
16. $v_s(alias) \leftarrow v_s(inst) \cup ValueUpdateByStore(alias, inst, s)$ //公式 (11)
17. **End for**
18. **If** 写入目标是全局变量, g' 为写入的全局变量
19. **Then** $g_s(g') \leftarrow g_s(g') \cup v_s(g')$
20. **End if**
21. **End if**
22. **Case** *inst* 是函数调用指令, *f* 为被调用函数:
23. $callSiteMap_s[inst] \leftarrow \{n \mid n \in s, n.key \in inst.args\}$
24. **For each** $<key, value> \in ExtractSideEffect(inst, s)$ **Do**
25. 初始化 $v_s(key)$
26. $v_s(x) \leftarrow SUM_f(v_s(key))$
27. **End for**
28. **Case** *inst* 是 ret 指令且存在返回值
29. $r_s(inst) \leftarrow i_s(inst) \cup v_s(inst)$
30. **End switch**
31. **End for**
32. **End for**
33. **End while**
34. **For each** $arg \in args_{func}$ **Do**
35. $p_s(arg) \leftarrow v_s(arg)$
36. **End for**
37. **Return** *s*
38. **End function**

在算法 1 中, 遍历函数 *func* 的控制流图中所有指令, 动态更新切片表, 当切片表 *s* 不再变化后, 将符号化分析摘要 *s* 返回. 此时的符号化分析摘要代表了函数 *func* 对程序上下文的影响, 其中包含了函数 *func* 的指令、变量、参数、全局变量以及返回值的依赖信息.

按照上述定义以及算法 1 的思想, 对图 1 中的 *func* 函数进行一个完整的代码走查以贯穿上述内容. 首先初始化一个空的分析摘要 *S_func* 并初始化参数的变量切片表: V_{S_func}(%0)={%0}、V_{S_func}(%1)={%1}、V_{S_func}(%2)={%2}. 之后循环遍历 *func* 函数中的每一条指令, 针对不同指令对切片表进行不同的处理, 比如指令“%3=load i32, ptr %0”, 则更新 V_{S_func}(%3) 依赖于 %0, 指令“store i32 %5, ptr %2”, 这是一个副作用, 调用 *ValueUpdateBystore*, 更新 %2 的变量切片, 并将 V_{S_func}(%2) 与 V_{S_func}(%5) 合并, 使 V_{S_func}(%2) 依赖于{%0, %1, %2}. 如此循环分析直到 *S_func* 不再发生变化. 最终更新副作用切片表 P_{S_func}(%0)=V_{S_func}(%0)、P_{S_func}(%1)=V_{S_func}(%1)、P_{S_func}(%2)=V_{S_func}(%2). 返回最终的 *S_func*, 即 *func* 的高阶函数摘要 SUM_{func}. 通过这种方式, 函数 *func* 的内部逻辑被符号化且惰性地传递到了 main 函数的上下文中, 之后再次遇到 *func* 的调用时便不需要再次完整分析 *func* 函数, 极大提高了分析效率.

3.2 并行按需切片计算模块

并行按需切片计算模块具体架构流程见图 5. 并行按需切片计算模块是整个框架的核心模块, 在此模块中可以有效体现按需与并行. 在按需切片计算阶段, 算法需要依据给定的切片准则实例化相应的分析摘要, 从而得到最终的切片结果. 在实例化分析摘要的过程中, 算法将分析摘要中尚未应用的高阶函数摘要与其对应的调用点信息相结合以实现对函数调用副作用的还原. 在去除符号化输入参数部分, 算法需要根据分析摘要所在函数的调用情况收集调用点信息, 之后将实际值替换到分析摘要.

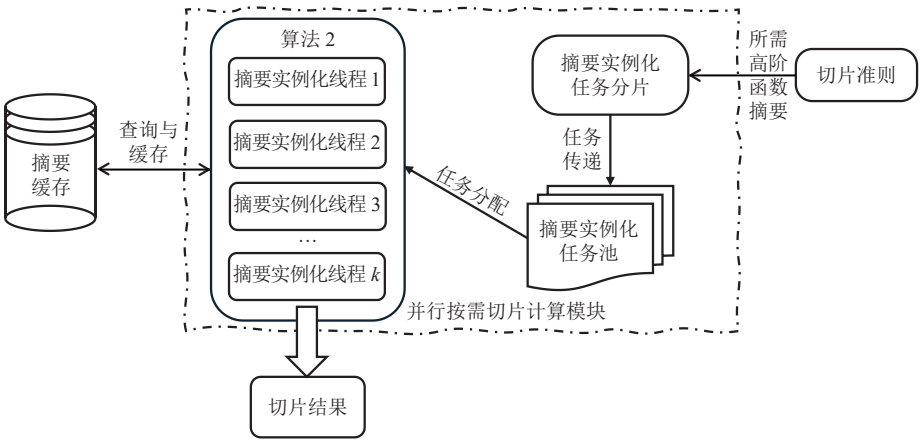

图 5 并行化按需切片计算模块

在本文提出的程序分析框架中, 切片准则以 LLVM 变量形式定义, 并与切片表基本元素中的键 (*key*) 保持结构一致. 基于这一关联性, 可通过以下公式直接从分析摘要中提取与给定 LLVM 变量对应的程序切片.

定义 10. 切片提取计算. 对任意切片分析摘要 *s*, I_s、V_s 分别是切片分析摘要的指令切片表和变量切片表, 则通过切片准则或是 LLVM 变量 *key* 提取其对应切片 $slice_{key}$ 的方法如下:

$$slice_{key} = s(key) = \begin{cases} V_s(key), & V_s(key) \neq \varnothing \\ I_s(key), & V_s(key) = \varnothing \text{ and } I_s(key) \neq \varnothing \\ \varnothing, & V_s(key) = \varnothing \text{ and } I_s(key) = \varnothing \end{cases} \tag{12}$$

这里需要注意, 由算法 1 可知当在分析摘要 *s* 时变量 *key* 的变量切片表由它的指令切片表中单调增计算出来, 指令切片表是计算得来的变量切片表的一个子集. 所以当变量切片表不为空时, *key* 的切片可以由变量切片表代替而非变量切片表和指令切片表的并集, 从而避免因计算并集带来的冗余计算.

在介绍子分析摘要的去符号化以及实例化之前, 首先介绍切片子分析摘要的并集操作∪.

定义 11. 切片分析摘要的并集操作. 切片分析摘要 s_1 与 s_2 的并集 $s_1 \cup s_2$ 计算方式为:

$$s_1 \cup s_2 =< I_{s_1} \cup I_{s_2}, V_{s_1} \cup V_{s_2}, P_{s_1} \cup P_{s_2}, G_{s_1} \cup G_{s_2}, R_{s_1} \cup R_{s_2}, callSiteMap_{s_1} \cup callSiteMap_{s_2} > \tag{13}$$

在将实际参数代入到符号值的过程中, 需要通过符号值获取调用语句中与该符号值对应的实际参数, 为此本文引入实参提取函数以实现该转换.

定义 12. 实参提取函数. 对任意函数 f 有其外部输入变量集合 $InputValues_f = \{a_0, a_1, \ldots, a_n\}$, 函数 f 的高阶函数摘要中有与输入变量集合对应的符号值集合 $SymbolValues_s = \{sv_i \mid 0 \leqslant i \leqslant n, sv_i \in s\}$ 且存在双射 $\pi(a_i) = sv_i$. 在对函数 f 调用的任意调用点 p 处有与函数 f 的输入变量集合对应的实际参数与全局变量列表 $ActualValues_p = \{act_1, act_2, \ldots, act_n\}$ 且存在双射 $\Omega(a_i) = act_i$, 则实参提取函数为:

$$getArgBySym(sv_i, p) = act_i \tag{14}$$

由于最终的切片结果获取以及高阶函数摘实例化过程中都需要通过带入实参的方式消除子分析摘要中的符号值, 因此本文切片引入符号带入函数以实现对子分析摘要的去符号化.

定义 13. 切片分析摘要的符号带入. 对于一个存在参数输入符号值的切片分析摘要 s_0, s_1 为对 s_0 所在函数的调用点处的切片分析摘要, 则切片的符号带入定义如下:

$$Substitution : ((A \to A) \to A) = \lambda s_0.\lambda s_1.s_0 \cup \bigcup\nolimits_{x \in SymbolValues_{s_0}} (getArgBySym(x, s_1) \wedge s_1) \tag{15}$$

在符号带入函数对子分析摘要的去符号化基础上, 本文引入对高阶函数摘要的应用函数实现对子分析摘要的实例化.

定义 14. 切片的高阶函数摘要应用. 对于一个待实例化的切片分析摘要 s_1, 其中存在未实例化的高阶函数摘要 SUM_f 且有 $s_1 = SUM_f^l s_0$, SUM_f 对应的切片分析摘要为 s_f, s_f 已完成实例化 (即 s_f 中不存在未实例化的高阶函数摘要), 该调用对应的调用点信息分析摘要为 s_p. 则 s_1 通过摘要 SUM_f 的实例化后得到的分析摘要 s_2 的计算方式如下:

$$InstantiateSummary : ((A \to A \to A) \to A) = \lambda s_1.\lambda s_p.\lambda s_f.s_1 \cup \bigcup\nolimits_{x \in SymbolValues_{s_1}} Subtitution(x, Subtitution(s_f, s_p)) \tag{16}$$

结合前文中对符号值带入的定义以及切片摘要的应用, 最终可以通过算法 2 中进行按需切片计算.

算法 2. 按需切片计算算法.

输入: 作为切片准则的 LLVM 变量 v, 高阶函数摘要查询集合 *summaryMap*, 调用点信息集合 *callSiteDataMap*;
输出: 切片准则 v 对应的切片 $slice_v$.

```
1. Function DemandSlice(v, summaryMap, callSiteDataMap)
2.   f ← v 所属于的函数
3.   SUM_f ← summaryMap[f]
4.   n_v ← SUM_f.s(v)
5.   callSiteDataWorkList ← callSiteDataMap[f]
6.   While callSiteDataWorkList 不为空
7.     callSiteData ← callSiteDataWorkList.pop()
8.     n_v←Substitution(n_v, callSiteData.data)
9.     callSiteDataWorkList.push(callSiteData.caller)
10.  End while
11.  While n_v 中存在未实例化的高阶函数摘要 SUM_x, 调用语句为 call Do
12.    SUM_x ← summaryMap[x]
13.    If SUM_x.s 未完全实例化
```

14. **Then**
15. **For each** $n \in SUM_x.s$&&n 未实例化
16. $n \leftarrow DemandSlice(n.key, summaryMap, callSiteDataMap)$
17. $SUM_x.s \leftarrow n$
18. **End for**
19. $s_x \leftarrow SUM_x.s$
20. **Else**
21. $s_x \leftarrow SUM_x.s$
22. **End If**
23. $n_v \leftarrow InstantiateSummary(n_v, s_x, SUM_f.s.callSiteMap[call])$ //公式 (16)
24. **End while**
25. $slice_v \leftarrow n_v.set$
26. **Return** $slice_v$
27. **End function**

在算法 2 中, 按需切片函数首先根据传入的切片准则获取其所在的函数、函数摘要以及对应该准则的子分析摘要, 根据 *callSiteDataMap* 找到所在函数的调用点集合, 每个调用点的数据结构保存了调用点处与调用有关的依赖集, 对调用点集合中的调用点进行符号带入, 并将已经处理过的调用点移出调用点集合, 通过以上步骤实现前向分析. 之后结合调用点信息与分析摘要中尚未应用的高阶函数摘要实例化分析摘要, 完成后向分析, 待分析摘要中不存在未应用的高阶函数摘要后, 将该分析摘要作为切片结果返回. 除此之外, 在完成实例化时也会对已经实例化的摘要进行缓存, 当后续摘要实例化时会先查询缓存寻找已经实例化的摘要, 若存在则直接使用, 并且在操作过程中通过读写锁来确保各线程之间的同步问题. 通过此种方式, 避免了在不同变量实例化任务间存在可重用的实例化摘要而重复分析导致的冗余分析问题.

同样对图 1 中的程序进行分析来贯穿上述内容. 将 %10 作为切片准则来进行演示. 算法从切片准则 %10 开始, *DemandSlice* 被调用, *v* 为 %10, *func* 为 main 函数. 为计算 %10 的切片, 算法分析指令"%10 = load i32, ptr %9", 发现 %10 依赖于 %9. 之后需要计算 %9 的切片, 向上追溯发现 %9 在 *func* 中被修改, 为精确计算 *func* 对 %9 的影响, 调用上一模块生成的高阶函数摘要 SUM_{func}. 之后进行摘要实例化, 在前向分析中, 算法分析 *call* 指令, 获取调用点信息: *func* 的 %0、%1、%2 分别与 main 的 %7、%8、%9 对应, 在后向分析中, SUM_{func} 中 %2 的摘要 (依赖 %0 与 %1) 被实例化为 %9 依赖 %7 与 %8, %0 的摘要 (初始值加 1) 被实例化为 %7 依赖其调用前的值. 通过这个过程, %9 的依赖关系成功从 *func* 函数内部传递到了 main 函数中. 根据实例化后的摘要, 算法继续在 main 函数找向上追溯 %7 和 %8 的定义并将指令加入最终的切片结果中.

3.3 按需分析、并行化分析、算法正确性说明

3.3.1 按需分析说明

该分析框架实现按需分析主要体现在以下机制: (1) 切片准则的驱动性. 分析过程由用户明确指定的切片准则直接驱动. 框架的需求分析模块首先将用户的分析需求转化为具体的切片准则, 后续所有计算仅围绕该准则展开. (2) 高阶函数摘要的局部应用. 高阶函数摘要通过副作用提取函数仅作用于受函数调用影响的子分析摘要, 而非全局摘要. 副作用提取函数可定位函数调用对分析摘要的具体影响范围, 并仅对该部分应用摘要. (3) 惰性实例化与动态计算. 高阶函数摘要在生成时是符号化的, 仅描述了函数副作用的"模版". 只有当分析过程追溯到与切片准则相关的依赖路径, 且该路径需要某个函数调用的具体影响时, 框架才会动态、及时地实例化对应的高阶函数摘要和符号值.

假设分析图 1 中 main 函数的返回值 %10. 在一个非按需的、全面的分析中, 分析器会计算并记录下 *func*

中“%6 = add nsw i32 %3, 1”以及“store i32 %6, ptr %0, align 4”两行代码, 这两行代码修改了 *func* 的第 1 个参数 %0, 即 main 函数中的 %7. 在按需分析中, 因为最终目标 %10 的计算路径 (%10→%9→%2→%5→%3+%4→%0+%1→%7+%8) 与 *func* 函数内部对 %7 的修改无关, 因此分析器会忽略上述两行代码的影响. 本框架的“按需”特性正体现于此, 仅在绝对必要时才去计算和实例化依赖关系. 假设 *func* 在另一行还有一次调用, 因为 %10 只与第 18 行的 *func* 调用相关, 因此在整个切片计算过程中, 只有第 18 行中 *func* 的高阶函数摘要 SUM_{func} 会被惰性地实例化, 而 *func* 的第 2 次调用会始终保持未计算状态. 它通过切片准则反向追踪, 仅探索与最终目标直接相关的代码路径, 忽略所有不相关的计算, 从而显著减少计算冗余.

3.3.2 并行化分析说明

该分析框架实现并行化分析主要体现在以下机制: (1) 子分析摘要的独立性. 子分析摘要是切片分析摘要的子集, 仅包含与特定变量或指令相关的局部依赖信息. 由于不同子分析摘要之间无依赖关系, 在并行计算中, 不同的子分析摘要可分配到不同线程同步处理, 避免了全局摘要的竞争. (2) 高阶函数摘要的局部应用. 高阶函数摘要通过副作用提取函数仅作用于受函数调用影响的子分析摘要, 而非整个分析摘要. 这种局部应用特性使得不同函数调用点的摘要实例化操作相互独立, 为并行化提供了任务拆分的基础. (3) 实例化计算的细粒度任务. 算法的实例化阶段以变量为粒度进行处理. 每个变量的实例化任务可作为独立任务分配到不同线程, 实现细粒度并行. (4) 调用点信息的并行处理. 算法通过工作列表管理调用点信息, 逐个处理调用点的符号带入操作. 由于各调用点的处理仅依赖其自身的实参信息, 无跨调用点的依赖, 因此不同调用点的处理可并行执行.

同样以图 1 程序为示例, 在高阶函数摘要计算阶段, 计算 main 函数的摘要和计算 *func* 函数的摘要是两个独立的过程, 因为计算一个函数的摘要只依赖于函数代码本身, 不依赖于其他函数摘要的计算, 因此计算各个函数摘要的任务可以完全并行化. 在并行化按需切片阶段, 依旧假设分析 %10, 分析发现 %10 依赖 %9, 而 %9 的值最终依赖于 %7 和 %8. 当需要计算 %7 和 %8 的依赖来源时, 由于子分析摘要的独立性与实例化计算的细粒度任务, 可以将分析 %7 和分析 %8 的依赖任务分配给两个线程. 计算结束后再将结果合并, 得知 %9 的最终依赖.

3.3.3 算法正确性说明

本节阐述本框架所提按需切片计算方法的正确性. 这里的正确性指, 针对任意给定的切片准则, 本框架计算所得的程序切片与采用传统详尽数据流分析 (如基于完整系统依赖图遍历) 所得到的切片是语义等价的.

本框架设计的高阶函数摘要是对函数副作用的精确形式化表示, 其生成过程基于对控制流图的完全遍历和不动点计算, 保证了不丢失任何过程内依赖信息. 与传统方法预先计算并存储所有依赖不同, 本框架的核心在于动态按需遍历: 将依赖计算推迟到实际应用时, 仅针对切片准则相关的依赖路径进行计算. 通过副作用提取函数和符号替换, 将调用上下文精确应用于摘要. 此过程等价于对完整依赖图的动态按需遍历, 因此结果与先构造完整图再遍历一致. 综上, 本框架的按需与惰性特性是一种保守性的剪枝策略, 它通过避免计算与当前目标无关的冗余依赖来提升效率, 但不会省略任何影响最终切片结果的必要计算, 保证了计算结果的准确性和一致性.

4 实验分析

为了验证本文提出的基于按需切片计算的并行化程序分析框架的有效性, 本文选取了来自开源 GNU/Linux 的真实程序组成测试集并对组成的测试集进行测试, 本文实验在搭载 Intel Core i7-13700H 处理器 (主频 2.40 GHz)、内存大小为 32 GB (DDR5 32 GB 4800 MT/s) 的计算机平台上进行. 实验将重点观察本文提出方法的分析性能与并行性能, 为此本文将围绕以下 3 个问题来说明本文方法的有效性.

RQ1: 相较于现有的切片方法, 本文中提到的按需切片计算方法的性能如何?

RQ2: 本框架中使用到的并行化按需切片计算方法的分析效率如何?

RQ3: 本框架在实际场景中的实用性和有效性效果如何?

为了方便表述, 本文将按需切片计算方法简称为 ODS, 将并行按需切片计算方法简称为 ODSP.

4.1 按需切片计算方法效率分析

为了验证本文提出的方法相较于现有方法的性能提升, 本实验仍选取来自开源 GNU/Linux 的真实程序组成

测试集. 为了能更好地展示差异性, 本实验选取的测试样例规模数量级基本一致 (IR 指令数在 20 万–50 万之间), 测试集的具体信息如表 1 所示. 在本实验中, 本文将 ODS 以及 ODSP 将与符号化切片工具 SymPas 进行比较. SymPas 是一种基于 LLVM 实现的符号化切片工具, 该工具通过不动点算法以及符号化函数摘要实现了程序切片, 由于其在切片计算时不构建过程间分析图, 规避了构图开销, 因此该工具相较于传统基于图的算法 (如 IFDS 算法) 在程序切片问题上具有较高的性能优势.

表 1 切片算法测试集

测试项目	IR指令数	准则数	函数个数	调用点数	循环数	CGSCC
objdump	504 214	70 813	3 822	44 425	2 117	31
nm	359 026	47 368	2 758	27 482	1 559	19
strings	356 856	46 720	2 709	27 059	1 556	19
size	355 484	46 626	2 716	27 022	1 546	19
cxxfilt	354 807	46 424	2 697	26 867	1 547	19
xg++	313 714	63 610	4 872	45 551	1 408	7
xgcc	313 058	63 558	4 872	45 499	1 408	7
strip	263 221	23 937	2 052	68 146	2 062	19
readelf	203 995	29 436	1 449	22 739	829	9

为了系统地评估本文提出的 ODS 和 ODSP 工具相对于基准工具 SymPas 的表现, 我们对一系列测试案例进行了实验. 在计算结果一致的情况下利用各工具内部集成的性能监控功能来获取计算时间 (time) 和内存使用峰值 (peak memory usage, PMU) 等关键指标. 此外还计算了本文方法相对于基准工具 SymPas 的平均对比值.

平均对比值的计算公式如下:

$$Avg.Comparison = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{X_i - Base_i}{Base_i}\times 100\%\right) \tag{17}$$

其中, N 表示测试集中测试样例的个数, X_i 表示第 i 个样例在目标方法下的指标值, $Base_i$ 表示第 i 个样例在基准方法下的指标值.

本文实验结果如表 2 与图 6 所示, 其中 ODSP-6T 表示 ODSP 工具 6 线程配置下的分析结果. 实验结果以 SymPas 工具为基准, 展示工具之间差异. 如表 2 所示, 在分析结果一致的情况下, ODS 相较于 SymPas 其分析时间其运行时间平均减少了 65.3%, 内存使用峰值平均下降了 68.2%. ODSP-6T 相较于 SymPas 其分析时间平均减少了 93.8%, 内存使用峰值与 ODS 基本一致.

表 2 切片算法实验结果

测试项目	SymPas		ODS		ODSP-6T	
	Time (s)	PMU (MB)	Time (s)	PMU (MB)	Time (s)	PMU (MB)
objdump	2210.6	3400	770.2	1060	131.7	1075
nm	1222.4	2050	467.2	770	83.7	773
strings	1185.1	1968	469.9	768	83.2	771
size	1184.6	1962	457.6	761	80.9	769
cxxfilt	1199.4	1967	474.0	769	82.7	772
xg++	1201.7	2096	254.1	600	46.1	602
xgcc	1350.1	2102	254.0	570	46.0	573
strip	956.1	1203	392.0	340	71.9	344
readelf	1000.7	2224	400.0	363	74.0	367
Avg.Comparison (%)	—	—	−65.3	−68.2	−93.8	−67.9

由图 6 可知, ODS 和 ODSP 相较于 SymPas 展现了显著的效率优势. ODS 通过将高阶函数摘要实现按需应用, 解决了 SymPas 在非高密度切片任务中存在的额外计算开销问题. ODS 只对必须的指令依赖进行实例化, 实现了以最小计算量完成切片, 有效避免了全量计算和冗余计算. 在内存使用方面, ODS 同样占优. 能够分离过程内和过

程间分析, 使函数摘要中不包含冗余的过程间信息, 从而占用更小的空间. 作为一个按需分析方法, ODS 的计算量可以实现最优化, 且对内存的需求较为固定, 不受线程数配置的影响.

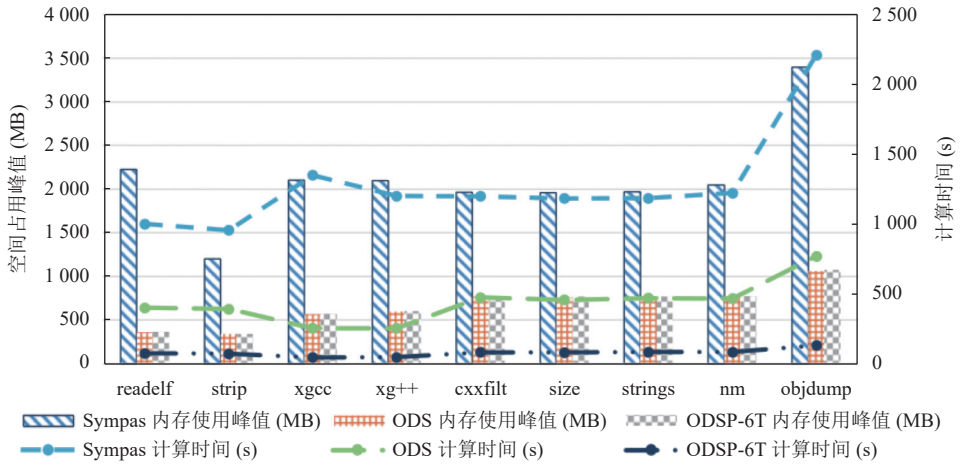

图 6 切片性能对比组合图

实验结果清晰地表明, 无论是分析时间还是内存峰值, ODS 和利用多核并行加速的 ODSP-6T 相较于 SymPas 均表现出优势. ODS 通过按需应用避免全量计算, ODSP 则在此基础上进一步利用并行加速, 两者性能的递进关系验证了框架设计的有效性, 使其非常适用于大规模程序的分析任务.

4.2 并行化按需切片计算方法分析效果

为了验证并行分析效果 (RQ2), 实验选取了 14 个来自开源 GNU/Linux 的真实程序作为测试集. 本实验所采用的测试集的具体信息如表 3 所示. 测试集涵盖 IR 指令数从 20 000 到 500 000 的项目, 能够有效评估算法在不同规模程序上的性能. 测试中收集了各样例的规模、分析任务量及结构复杂度 (调用点数、循环数、CGSCC), 这些指标可充分反应样例特征以支撑后续分析.

表 3 并行分析测试集

测试项目	IR指令数	准则数	函数个数	调用点数	循环数	CGSCC
objdump	504 214	70 813	3 822	44 425	2 117	31
nm	359 026	47 368	2 758	27 482	1 559	19
strings	356 856	46 720	2 709	27 059	1 556	19
size	355 484	46 626	2 716	27 022	1 546	19
cxxfilt	354 807	46 424	2 697	26 867	1 547	19
xg++	313 714	63 610	4 872	45 551	1 408	7
tar	94 746	16 009	1 162	12 022	638	2
m4	82 496	11 545	723	7 892	554	4
find	47 665	9 173	652	5 917	212	3
date	25 499	3 426	128	2 651	127	0
gzip	22 493	2 498	162	1 904	213	1
touch	22 130	3 108	166	2 299	84	0
df	21 970	3 422	234	2 427	124	0
stat	20 852	3 208	194	2 289	79	0

为了展示本文提出的并行分析算法的并行效率情况, 本实验使用 ODSP 工具对测试集进行测试. 实验通过随机抽取的形式抽取各测试样例中的 LLVM 变量作为切片准则, 最后通过 ODSP 工具计算结果. 本文实现的工具均基于 LLVM PASS, 需要通过 LLVM 项目的 opt 工具加载并运行, 此工具需要接收 bc 文件作为输入. 在实验的过

程中, 需先将待测程序通过 LLVM 转换为 bc 文件, 之后通过 opt 对转换后的 bc 文件运行分析. 与此同时, 为了使计算结果更准确, 在实验过程中尽量避免了不必要进程的运行, 并且进行重复实验, 每组测试用例运行 5 次, 取其平均值, 从而减少误差.

本实验通过 ODSP 内置的运行时统计模块收集 ODSP 在不同并行数配置 (并行数分别为 1、2、4、6) 下的摘要计算时间 (summary computation time, SCT)、实例化计算时间 (instantiation computation time, ICT)、摘要计算线程间最大时间差 (SCT max thread time deviation, SMTTD) 以及实例化计算线程间最大时间差 (ICT max thread time deviation, IMTTD)、各计算阶段的加速比 (speedup ratio, *SR*) 以及并行效率 (parallel efficiency, *PE*).

加速比 *SR* 的计算公式如下:

$$SR(M)=\frac{T^{i}_{\text{single}}}{T^{i}_{\text{M-Parallel}}} \tag{18}$$

其中, T^{i}_{single} 表示第 i 个项目单线程下的分析运行时间 $T^{i}_{\text{M-Parallel}}$ 表示第 i 个项目在 M 线程下的分析运行时间. 加速比为单线程耗时与并行耗时的比值, 能够衡量并行化带来的性能增益.

并行效率 *PE* 的计算公式如下:

$$PE(M)=\frac{SR(M)}{M}\times 100\% \tag{19}$$

并行效率是衡量并行计算系统资源利用率的指标, 用于量化并行化带来的性能提升是否充分利用了计算资源. 实验结果如表 4 与表 5 所示. 为了突出本方法在多个线程的表现中的有效性来自方法的并行效率而不仅是线程的堆叠, 将表 4 中能体现此优势的较优数值, 采用加粗显示.

表 4　并行分析时间使用表

测试项目	SCT(s)				ICT(s)			
	1-Thread	2-Thread	4-Thread	6-Thread	1-Thread	2-Thread	4-Thread	6-Thread
objdump	202.5	115.2	60.2	34.6	567.7	295.4	149.4	97.1
nm	175.2	92.5	51.5	30.9	292.0	182.1	95.2	52.8
strings	177.3	98.5	51.3	31.3	292.6	179.5	87.3	51.9
size	172.6	98.4	50.7	29.3	285.0	170.2	91.5	51.6
cxxfilt	178.8	99.5	50.5	30.5	295.2	189.4	92.9	52.1
xg++	84.6	47.8	25.7	16.0	169.5	90.1	46.0	30.1
tar	9.9	6.5	3.8	3.3	28.5	15.2	7.8	5.2
m4	28.0	14.9	9.0	8.7	5.8	3.9	1.8	1.0
find	16.5	9.0	6.8	5.8	30.5	15.8	8.1	5.4
date	25.5	26.1	25.4	25.5	15.4	7.9	3.9	3.2
gzip	25.3	25.3	25.3	25.3	1.4	0.7	0.4	0.3
touch	25.6	25.4	25.7	25.6	21.5	11.5	5.9	4.4
df	25.5	25.6	25.4	25.5	3.1	1.6	0.8	0.6
stat	25.4	25.7	25.5	25.4	1.0	0.6	0.3	0.2
Avg. *SR*	—	1.49	2.40	**3.44**	—	1.78	3.50	**5.37**
Avg. *PE* (%)	—	**74.9**	60.0	57.4	—	89.2	87.7	**89.6**

根据表 4、表 5 中的数据可知, 在诸如 objdump、nm 以及 strings 等较为大型的测试样例中, 摘要的计算时间随着线程数的提高大幅下降, 其并行效率接近 90%, 具有良好的并行性. 在诸如 df、stat 等小型样例中摘要的计算速度随着线程数的提高并没有获得较大提升, 并且这些小型测试样例的摘要计算线程间最大时间差占摘要计算时间的比例较高. 这是因为基于不动点的过程内分析其分析时间并不与程序规模直接相关, 而是与程序的具体语义相关, 导致对小型程序的摘要计算存在单函数分析瓶颈.

这一现象在对应的折线图 (图 7) 中更加明显. 摘要计算的并行效率在处理小型项目如 stat、df 等时明显偏低. 这背后的原因正如上述所说, 实例化计算以变量为单位, 任务可以被细粒度地分派给不同线程. 而摘要计算以函数为单位, 其性能瓶颈在于基于不动点算法的过程内分析. 在小型项目中, 少数几个包含复杂控制流的函数可能会占

据大部分计算时间, 成为单线程瓶颈, 导致其他核心闲置, 从而限制了并行效率的提升. 而在大型程序中, 由于整体分析时间较长, 分析这些复杂结构或是函数所占用的时间相较于整体较小, 故其并行效率较为稳定且具有良好的表现. 与摘要的并行计算不同, 图 8 所示的实例化并行计算在各类规模下的测试样例中表现较为稳定, 其并行分析效率基本接近 90%, 这是因为实例化并行计算的粒度为变量级别, 算法能够高效利用各线程实现对程序的分析.

表 5　并行分析线程时间差结果表 (s)

测试项目	SMTTD			IMTTD		
	2-Thread	4-Thread	6-Thread	2-Thread	4-Thread	6-Thread
objdump	10.5	9.0	9.1	2.0	2.1	2.6
nm	5.4	6.7	6.0	1.2	1.6	1.8
strings	3.4	3.0	3.2	2.0	1.9	1.6
size	1.5	2.0	1.9	1.1	1.3	1.2
cxxfilt	1.1	1.5	2.1	0.8	0.6	0.7
xg++	0.9	1.1	1.9	0.5	1.0	1.2
tar	0.5	0.4	0.4	1.2	1.5	1.3
m4	0.6	0.5	0.4	0.2	0.1	0.1
find	3.5	2.9	3.4	1.9	2.1	2.1
date	22.1	22.1	22.9	1.0	1.1	1.5
gzip	22.4	22.7	22.5	0.1	0.1	0.1
touch	21.9	21.8	21.9	1.0	0.8	0.7
df	22.4	22.5	22.4	0.2	0.1	0.1
stat	21.5	21.4	21.7	0.1	0.2	0.1

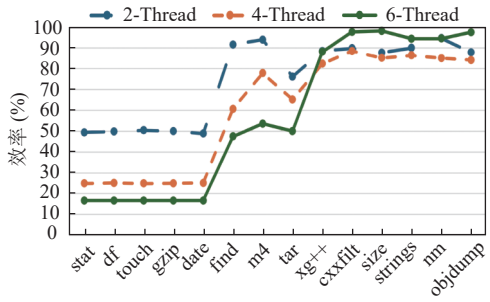

图 7　摘要计算并行效率折线图

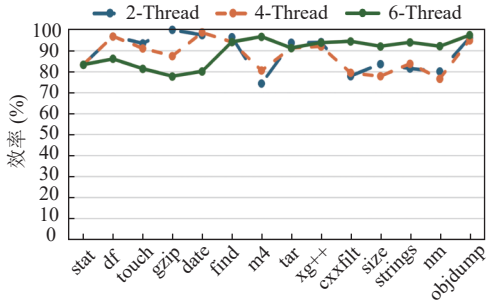

图 8　实例化计算并行效率折线图

综上所述, 本文提出的并行化方法具有良好的并行效率, 并且能够应用于现实中的真实程序当中, 在软件分析中的部分领域中起着重要作用.

4.3　框架实用性与有效性验证

为了验证本框架的实用性与有效性, 实验中将本框架应用到访问越界检测. 本文的越界检测基于按需切片并行化计算方法, 主要通过以下步骤实现: 首先, 识别程序中的访问越界点, 定义越界条件 (访问偏移≥目标空间大小); 其次, 利用定制化的越界语义按需切片, 收集访问偏移和目标空间大小的相关信息, 构建包含路径约束和越界条件的目标约束; 最后, 通过并行按需切片技术沿函数调用图迭代更新约束, 结合约束求解器验证越界条件是否可满足, 从而判定是否存在访问越界. 通过上述思路, 基于本框架的访问越界检测流程见图 9.

实验中依照常见弱点枚举 (common weakness enumeration, CWE) 中对越界检测相关漏洞的定义与分类从公开数据集中收集相关测试样例. 本实验从属于访问越界的两类 CWE 分类: 访问越界读取 (CWE-125 out-of-bounds read)[20]与访问越界写入 (CWE-787 out-of-bounds write)[21]中选取部分样本类, 并收集与这些样本类相关的访问越界错误测试样例. 本实验构建的测试集信息如表 6 所示.

本实验将使用框架实现的越界检测工具 OOBDP 与 Cppcheck[22]、TscanCode 和 Clang static analyzer (CSA)[23]

这 3 种工具进行比较. Cppcheck 是一款开源的静态分析工具, 它通过双向数据流分析实现了对 C/C++程序的缺陷检测, 该工具能够有效检测单个文件以及项目整体中的访问越界缺陷. TscanCode 是由腾讯静态分析团队开发的一款免费的 C/C++静态分析工具, 其分析不依赖编译器, 具有较大的吞吐量, 因此该工具能够有效检测大型项目中的访问越界缺陷. CSA 作为 Clang 工具链的一部分, 可用于 C、C++、Objective-C 代码的静态分析, 其支持采用约束求解器进行分析, 能够有效检测项目中的访问越界问题. 以上 3 种工具是目前较为常用的静态分析工具, 并且这些工具目前均处于发展和维护阶段, 其分析效果受到业界认可.

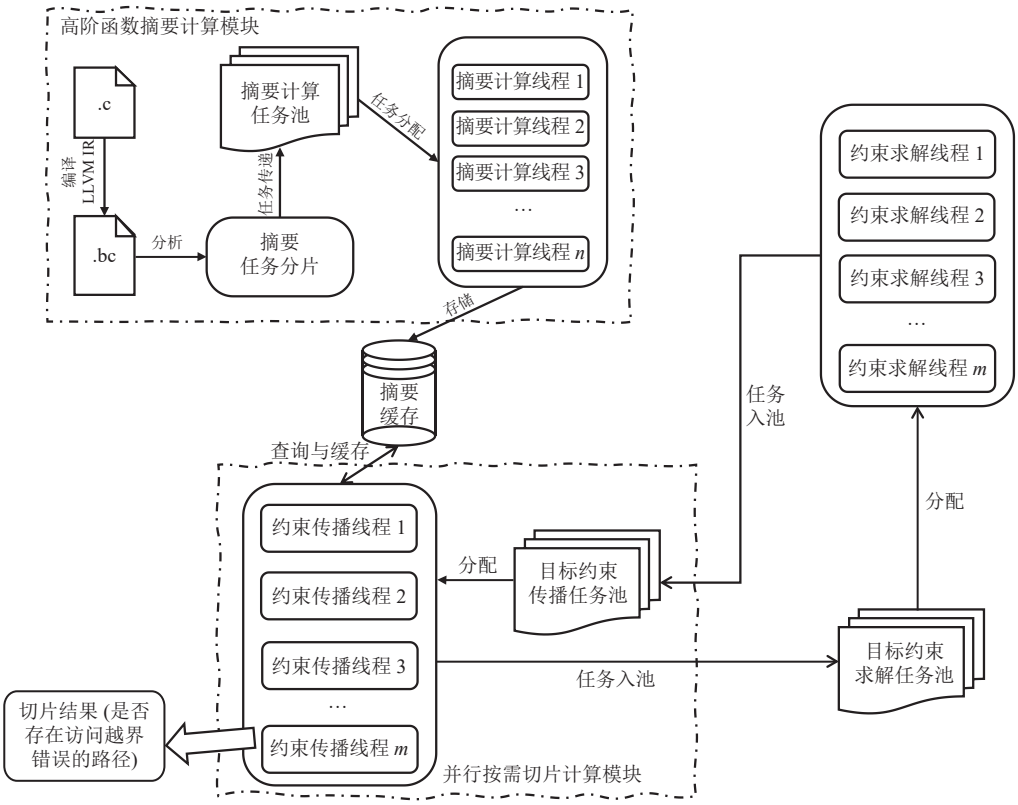

图 9　访问越界检测应用

表 6　越界检测精度对比实验集

源测试集	CWE类别	总缺陷数
Juliet C/C++	CWE-121	112
	CWE-122	58
	CWE-124	36
	CWE-126	22
	CWE-127	30
IARPA STONESOUP Phase 1—Memory Corruption for C	CWE-121	15
	CWE-122	15
	CWE-124	22
	CWE-126	15
	CWE-127	16

本实验主要收集测试工具在测试集上的 3 类实验数据: 真阳性 TP (true positive)、假阳性 FP (false positive) 和假阴性 FN (false negative). 在本实验中, TP 表示工具正确检测出访问越界的数量, FP 表示工具将正确代码误报

为访问越界或是将访问越界问题误报为其他问题的数量, FN 则表示工具未能检测到的访问越界数量. 实验的具体结果如表 7.

表 7 越界检测精度实验结果

缺陷测试集类型	Cppcheck			TscanCode			CSA			OOBDP		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Juliet CWE-121	15	1	97	11	2	101	34	0	78	42	0	70
Juliet CWE-122	20	0	38	1	0	57	15	0	43	27	0	31
Juliet CWE-124	11	0	25	10	1	26	10	0	26	18	0	18
Juliet CWE-126	0	0	22	0	0	22	0	0	22	9	0	13
Juliet CWE-127	10	0	20	10	3	20	6	0	24	15	0	15
STONESOUP CWE-121	3	1	12	3	1	12	3	0	12	5	2	10
STONESOUP CWE-122	0	0	15	0	0	15	0	0	15	3	2	12
STONESOUP CWE-124	0	1	22	0	2	22	0	0	22	3	3	19
STONESOUP CWE-126	0	0	15	0	0	15	0	0	15	4	2	11
STONESOUP CWE-127	13	0	3	0	1	16	0	0	16	6	1	10
Total	72	3	269	35	10	306	68	0	273	132	10	209

根据表 7 对实验结果的统计, OOBDP 在真阳性数量上表现出优越性, 共检测到 132 个缺陷实例, 高于所有对比工具. 然而在误报数量方面, OOBDP 检测出 10 个误报, 与 TscanCode 持平, 但高于 Cppcheck (3 个) 和 CSA (0 个). 对误报分析发现其产生主要是因为本框架所采用的别名分析存在精度限制. 本框架采用一种主流的 Andersen 风格的指针分析来实现别名查询. 该方法虽然高效, 但在面对复杂的指针算术、函数指针或多级指针嵌套的场景时, 可能会产生过于保守的指向关系, 导致后续的切片计算和约束求解可能沿着一条在实际执行中不可达的虚假路径进行, 从而报告一个虚假的缺陷. 例如, 一个数组的大小可能在一个分支被正确定义, 但在另一条虚假的代码路径上被错误的赋了一个较小的值. 由于别名分析的错误, 约束求解器会同时考虑这两条路径, 并错误地认为存在一个偏移量大于数组大小的可满足解, 从而报告一个假阳性.

为了验证工具 OOBDP 的分析性能, 本实验选取了来自 GNU/Linux 的真实程序以及来自 CoREBench 的真实程序组成了数据集. CoREBench 是一个包含 70 个真实且复杂的缺陷集合, 这些缺陷来自 4 个开源项目 (Make、Grep、Findutils 和 Coreutils). 本实验所采用的测试信息如表 8 所示.

表 8 越界检测并行化分析测试集

测试项目	IR指令数	函数个数	调用点数	循环数	CGSCC
tar	94 746	1 162	12 022	638	2
wget	93 963	837	10 648	487	3
m4	82 496	723	7 892	554	4
make	55 277	418	6 376	545	4
nano	51 202	489	6 831	294	2
find	47 665	652	5 917	212	3
grep	38 920	453	4 003	309	3

实验仍选用上文中的 3 个缺陷检测工具作为对比, 收集各工具在分析测试时的分析测试样例所需的计算时间 Time、内存使用峰值 PMU 以及各指标相较于本文方法的平均提升率. 对于 Cppcheck、TscanCode 以及 CSA 工具, 本实验通过 GNU time 获取各工具的执行时间与内存使用峰值. 实验结果如表 9 所示.

从实验中可以看出, TscanCode 使用的时间以及空间最少, CSA 次之, 而 Cppcheck 与 OOBDP 需要的时间与空间资源较多. TscanCode 通过规则匹配实现对程序的缺陷检测, 其分析过程无需构建程序图或者路径树等复杂结构, 因此仅需要较少的资源即可完成分析. CSA 在分析时虽然涉及约束求解, 但是其仅将其用于可达性分析, 这使得 CSA 在精度有限的情况下不需要太多资源完成分析. Cppcheck 与 OOBDP 的分析精度较高, 但这两个方法依赖基于图算法或是约束求解等需要大量资源的分析方法, 故在整体性能方面表现一般.

表 9 各工具分析性能实验结果

测试项目	Cppcheck		TscanCode		CSA		OOBDP-6T	
	Time (s)	PMU (MB)	Time (s)	PMU (MB)	Time (s)	PMU (MB)	Time (s)	PMU (MB)
tar	371.2	130.4	29.8	27.3	48.8	176.6	789.6	1, 023.5
wget	353.6	145.2	34.2	49.1	47.2	176.2	715.1	954.0
m4	246.0	239.5	17.1	11.9	36.1	170.7	535.2	741.1
make	213.7	202.1	13.3	17.4	18	124.3	499.8	702.3
nano	155.1	189.7	8.6	31.0	19.2	131.2	328.0	415.7
find	180.2	290.9	68.1	196.7	18.7	125.8	315.2	376.5
grep	166.5	177.4	65.3	238.8	15.3	112.0	347.2	368.2

4.4 威胁性分析

本文选取部分测试集与来自现实世界的真实程序项目对本文方法进行实验与测试, 尽管所提出方法在实验中展现了较好的精度与分析性能, 但我们仍需谨慎讨论可能影响结论有效性的潜在威胁, 本节将通过以下 4 个方面讨论可能影响结论的有效性.

(1) 实验数据集对工业级项目的分析不足

实验采用来自 GNU/Linux 的真实程序测试样例, 其最大规模约为 50 万个 IR 指令, 但规模与一些工业级项目 (如系统内核、工控系统) 仍有差距, 这可能导致本文方法在这些超大规模上的泛化性受限. 针对此问题, 本文还选取了部分大型真实程序样例进行了补充实验, 实验数据见表 10, 结果表明本文提出方法在诸如 WireShark (约 190 万个 IR 指令) 和 FFmpeg (约 500 万个 IR 指令) 等大型项目上仍然有效, 算法能够在 20 min 与 75 min 左右通过 6 个线程分别计算出 26 万和 75 万个切片准则对应的切片结果.

表 10 大型真实程序样例补充实验数据

测试项目	函数个数	调用点数	基本块数	指令个数	摘要计算时间 (ms)	摘要实例化时间 (ms)	总耗时 (ms)	最大空间占用 (MB)
WireShark	13 360	1 005 653	280 636	1 925 076	157 970	1 046 663	1 204 633	4 971
FFmpeg	20 297	1 652 388	392 838	5 201 023	1 575 827	2 966 412	4 542 239	12 320

(2) 切片准则选取代表性有限

针对同一测试样例, 在不同的切片准则数量需求下, 切片的最终效果会不同. 本文选取的切片准则占测试样例的 15%, 所产生的效果较好. 需要强调的是, 15% 这一比例是为构建一个标准化的中等密度分折负载, 以便在可控条件下对比不同框架性能而设定的实验参数, 而并非本框架的内在要求. 在实际应用中, 切片准则由具体任务来驱动, 其密度是变化的. 为验证切片准则的选取对最终效果的影响, 进行了补充实验. 实验结果显示当测试样例中选取的切片准则密度不高 (10%–60%) 的情况下, 本文按需切片计算方法有着良好的效果. 在测试样例中选取的切片准测密度较高 (80% 以上) 时, 因为 SymPas 不需要额外考虑调用点信息, 所以 SymPas 的表现略优于本文方法.

(3) 过程内分析中的不动点算法瓶颈

本文提出的切片计算方法在进行过程内分析时采用了不动点计算方法, 此类方法对于一些特殊结构 (如多层嵌套循环) 分析性能较差, 这使得本文方法用于计算小规模测试样例以及部分中型样例时出现了函数摘要计算 (过程内分析) 上的瓶颈. 由于本文方法的改进点主要涉及过程间分析, 且该改进不依赖于特定的过程内分析, 因此我们将在往后的研究中采用非不动点算法来替代目前的过程内分析方法, 以解决目前过程内分析瓶颈.

(4) 端到端应用到大型项目上的验证局限性

本文当前未在大型项目上进行端到端的越界检测, 主要考虑到端到端分析中包含的约束求解等步骤在大型项目中会成为新的性能瓶颈, 这会干扰本文提出的并行分析框架本身的性能评估. 因此我们的实验设计将精度验证 (在标准测试集上进行) 与框架核心性能验证 (在大型项目上进行) 分离开来, 以获取更清晰、无干扰的结论. 尽管如此, 在大型项目上开展完整的端到端缺陷检测是验证本框架最终实用价值的关键一步, 并将其列为未来的重点工作.

5 总结与未来展望

5.1 总 结

本文针对大规模程序分析中显式依赖图构建导致的内存爆炸、冗余计算及现有并行化方法效率不足等问题, 提出了一种基于按需切片计算的并行化程序分析框架. 框架采用模块化设计, 包含需求分析、高阶函数摘要计算、并行按需切片计算及后续分析这 4 大核心模块: 需求分析模块将用户需求转化为具体切片准则, 驱动计算聚焦目标; 高阶函数摘要模块通过符号化抽象函数调用影响, 避免显式图构建的高内存开销; 并行按需切片模块基于切片准则动态实例化子摘要, 利用子摘要独立性、惰性实例化及细粒度任务拆分实现高效并行; 后续分析模块结合切片结果完成针对性输出. 实验表明, 框架中的按需切片计算方法较传统工具时间减少 65.3%、内存峰值下降 68.2%, 并行版本 (6 线程) 时间减少 93.8%, 并行效率近 90%. 应用于访问越界检测时真阳性达 132 个, 在现实项目中精度表现优秀, 为大规模程序分析提供了可扩展的解决方案.

5.2 未来工作展望

在未来的研究工作中, 由于本框架在设计上具有良好的语言无关性, 所有分析均构建于通用的中间表示 LLVM IR 之上, 使得本框架的适用范围并不局限于 C/C++. 理论上, 任何能够通过相应编译器前端转换为 LLVM IR 的编程语言 (如 Rust、Go 等) 均有可能受益于本框架提供的高效、并行的按需切片分析能力. 但根据不同语言的不同特性, 后期仍需进行进一步的适配与深度研究. 将本框架应用于 Rust 等现代系统级语言的缺陷分析与安全审计是未来一项极具价值的研究工作.

关于高阶函数摘要实例化后的可重用问题, 本文在设计上仅是初步考虑了这一情况, 并未进行深入探讨, 在未来的工作中计划将实例化摘要缓存设计得更加高效, 以便进一步减少冗余计算, 提升在具有重复调用模式的程序上的分析效率.

此外, 将在大型项目上开展端到端的缺陷检测验证. 当前研究聚焦并行分析框架的核心性能, 为避免约束求解等其他环节的性能开销对本框架的评估, 未在大型项目中集成全流程步骤进行端到端分析. 未来计划在超大规模代码库 (如系统内核、工业软件) 中实施完整的端到端缺陷检测, 这不仅能验证本框架在实际工程场景中的综合性能, 也将暴露出与下游工具 (如约束求解器) 集成时可能出现的新瓶颈, 从而全面评估框架的实用价值.

References

[1] Silva J. A vocabulary of program slicing-based techniques. ACM Computing Surveys, 2012, 44(3): 12. [doi: 10.1145/2187671.2187674]

[2] Weiser M. Program slicing. IEEE Trans. on Software Engineering, 1984, SE-10(4): 352–357. [doi: 10.1109/TSE.1984.5010248]

[3] Jia LM, Chen ZY, Jin YH. A dynamic slice algorithm based on program dependence diagram. In: Proc. of the 2nd Int'l Conf. on Consumer Electronics, Communications and Networks (CECNet). Yichang: IEEE, 2012. 2273–2276. [doi: 10.1109/cecnet.2012.6201848]

[4] Chalupa M. DG: A program analysis library. Software Impacts, 2020, 6: 100038. [doi: 10.1016/j.simpa.2020.100038]

[5] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2014. 590–604. [https://doi.org/10.1109/SP.2014.44]

[6] Galindo C, Pérez S, Silva J. Exception-sensitive program slicing. Journal of Logical and Algebraic Methods in Programming, 2023, 130: 100832. [doi: 10.1016/j.jlamp.2022.100832]

[7] Galindo C, Pérez S, Silva J. Program slicing of Java programs. Journal of Logical and Algebraic Methods in Programming, 2023, 130: 100826. [doi: 10.1016/j.jlamp.2022.100826]

[8] Halder R, Cortesi A. Abstract program slicing of database query languages. In: Proc. of the 28th Annual ACM Symp. on Applied Computing. Coimbra: ACM, 2013. 838–845. [doi: 10.1145/2480362.2480524]

[9] Zhang YZ. SymPas: Symbolic program slicing. Journal of Computer Science and Technology, 2021, 36(2): 397–418. [doi: 10.1007/s11390-020-9754-4]

[10] Yadavally A, Li Y, Wang SH, Nguyen TN. A learning-based approach to static program slicing. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA1): 83–109. [doi: 10.1145/3649814]

[11] Mastroeni I, Zanardini D. Abstract program slicing: An abstract interpretation-based approach to program slicing. ACM Trans. on Computational Logic, 2017, 18(1): 7. [doi: 10.1145/3029052]

[12] Méndez-Lojo M, Mathew A, Pingali K. Parallel inclusion-based points-to analysis. In: Proc. of the 2010 ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications. Reno/Tahoe: ACM, 2010. 428–443. [doi: 10.1145/1869459.1869495]

[13] Jordan H, Subotić P, Zhao D, Scholz B. Specializing parallel data structures for Datalog. Concurrency and Computation: Practice and Experience, 2022, 34(2): e5643. [doi: 10.1002/cpe.5643]

[14] Abeysinghe S, Xhebraj A, Rompf T. Flan: An expressive and efficient Datalog compiler for program analysis. Proc. of the ACM on Programming Languages, 2024, 8(POPL): 2577–2609. [doi: 10.1145/3632928]

[15] Sun YH, Kumar S, Gilray T, Micinski K. Column-oriented Datalog on the GPU. In: Proc. of the 39th AAAI Conf. on Artificial Intelligence. Philadelphia: AAAI, 2025. 15177–15185. [doi: 10.1609/aaai.v39i14.33665]

[16] Gu R, Zuo ZQ, Jiang X, Yin H, Wang ZK, Wang LZ, Li XD, Huang YH. Towards efficient large-scale interprocedural program static analysis on distributed data-parallel computation. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(4): 867–883. [doi: 10.1109/TPDS.2020.3036190]

[17] Su Y, Ye D, Xue JL. Parallel pointer analysis with CFL-reachability. In: Proc. of the 43rd Int'l Conf. on Parallel Processing. Minneapolis: IEEE, 2014. 451–460. [doi: 10.1109/ICPP.2014.54]

[18] Su Y, Ye D, Xue JL, Liao XK. An efficient GPU implementation of inclusion-based pointer analysis. IEEE Trans. on Parallel and Distributed Systems, 2016, 27(2): 353–366. [doi: 10.1109/TPDS.2015.2397933]

[19] Yang JY, Zhang YZ, Li JF, Ma R, Wang QS, Xue YC. A dependence clusters detection method based on higher-order function summary. Acta Electronica Sinica, 2024, 52(4): 1337–1348 (in Chinese with English abstract). [doi: 10.12263/DZXB.20230862]

[20] CWE. CWE-125: Out-of-bounds read. 2025. https://cwe.mitre.org/data/definitions/125.html

[21] CWE. CWE-787: Out-of-bounds write. 2025. https://cwe.mitre.org/data/definitions/787.html

[22] Cppcheck. Net. Cppcheck: A tool for static C/C++ code analysis. 2025. http://cppcheck.net/

[23] CSA: Clang static analyzer. 2025. https://clang-analyzer.llvm.org/

附中文参考文献

[19] 杨嘉毅, 张迎周, 李俊锋, 马锐, 汪全盛, 薛渝川. 一种基于高阶函数摘要的依赖簇检测方法. 电子学报, 2024, 52(4): 1337–1348. [doi: 10.12263/DZXB.20230862]

作者简介

李俊锋, 硕士, 主要研究领域为程序分析, 程序切片.

苑旭东, 硕士生, 主要研究领域为程序分析, 程序切片.

魏晨雨, 硕士生, 主要研究领域为程序分析, 缺陷定位.

厉剑豪, 硕士生, CCF 学生会员, 主要研究领域为程序分析.

张迎周, 博士, 教授, CCF 高级会员, 主要研究领域为程序分析, 程序切片.