软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(5):1243−1254 [doi: 10.13328/j.cnki.jos.005950]

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

静态程序分析并行化研究进展*

陆申明, 左志强, 王林章

(计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 左志强, E-mail: zqzuo@nju.edu.cn; 王林章, E-mail: lzwang@nju.edu.cn

摘 要: 静态程序分析发展至今,已在多个方面取得了长足的进步,应用于软件开发的众多方面.但对现代大规模复杂软件系统(如千万行代码规模的 Linux 操作系统、分布式大数据处理系统 Hadoop 等)进行高精度的静态分析,因其极大规模数据量的计算,仍有一定难度.精度、效率和可扩展性相互制约,是静态分析技术在工业界应用的主要障碍.对此,近年来,随着多核、众核架构的兴起,研究人员提出了静态分析的各种并行化解决方案.首先梳理了静态分析的发展历程,然后针对当前静态分析面临的挑战,在分析了传统算法优化研究的不足后,对利用硬件资源进行并行优化的方法进行了充分讨论,包括单机的 CPU 并行、分布式和 GPU 实现这 3 个方面;在此基础上,对一些使用较为广泛的支持并行的静态分析工具进行了评估;最后,对未来如何从算法和算力角度对静态分析进行并行优化作了讨论和展望.

关键词: 静态程序分析;软件质量保障;并行计算;分布式处理;GPU 加速

中图法分类号: TP311

中文引用格式: 陆申明,左志强,王林章.静态程序分析并行化研究进展.软件学报,2020,31(5):1243−1254. http://www.jos.org.cn/1000-9825/5950.htm

英文引用格式: Lu SM, Zuo ZQ, Wang LZ. Progress in parallelization of static program analysis. Ruan Jian Xue Bao/Journal of Software, 2020,31(5):1243−1254 (in Chinese). http://www.jos.org.cn/1000-9825/5950.htm

Progress in Parallelization of Static Program Analysis

LU Shen-Ming, ZUO Zhi-Qiang, WANG Lin-Zhang

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: After years of research, static program analysis has made great progress in many aspects. However, performing sophisticated program analysis over large-scale software systems (such as Linux kernel, Hadoop, etc.) is still challenging due to its high complexity. To address the poor scalability of static analysis, with the rise of multi-core computation architectures, researchers have proposed various parallel static analysis techniques. This paper first introduces the basic concepts of static analysis, the key techniques, and the challenges. Then the traditional optimization approaches are discussed followed by the studies in parallelizing static analysis in three categories—CPU parallelism, distributed and GPU implementation, and the representative parallel static analysis tools. Finally, the potential research trend in parallelizing static analysis is described.

Key words: static program analysis; software quality assurance; parallel computing; distributed processing; GPU acceleration

随着现代软件规模日趋庞大、功能日益复杂,其中的缺陷和漏洞数量也急剧增加.如何精确而又快速地发现这些缺陷和漏洞,提高软件质量,是学术界和工业界共同关注的重要问题.

* 基金项目: 国家自然科学基金(61802168); 江苏省自然科学基金(BK20191247)

Foundation item: National Natural Science Foundation of China (61802168); Natural Science Foundation of Jiangsu Province (BK20191247)

本文由“系统软件构造与验证技术”专题特约编辑赵永望副教授、刘杨教授、王戟教授推荐.

收稿时间: 2019-09-01; 修改时间: 2019-10-24; 采用时间: 2019-12-24; jos 在线出版时间: 2020-04-07

目前,软件缺陷(和漏洞)的检测方法主要分为两大类:静态分析和动态测试.不同于动态测试需要实际运行被测程序,静态分析是指在不运行代码的情况下对程序代码进行系统的分析.不同于动态测试受限于测试用例的完备性,静态分析可以比较全面地考虑可能的执行路径,能发现更多的缺陷,是当前被学术界和工业界普遍采用的技术.

在实际应用中,静态分析的结果往往包含大量的误报,导致大量的人工确认工作,耗时且容易出错.为提高分析的精度,往往需要执行高复杂度的静态分析,例如上下文敏感的过程间分析、流敏感/路径敏感的分析等.然而,高复杂度的分析必然导致极大的计算量和内存消耗,例如在过程调用上下文敏感的分析中,上下文数量会随目标程序的大小成指数级增长,一个中等规模的软件就可能包含上百万个不同的调用上下文[1].其次,在开源软件生态下,软件系统的规模急剧增长,百万行、千万行甚至更大的代码规模越来越常见[2].

对此,多年来,人们在静态分析优化方面做了大量的研究工作,取得了不少的进步.传统的优化研究主要关注对顺序算法的优化,但随着摩尔定律的失效,单核计算效率的提升不足,顺序算法分析效率相对低下.而随着多核、众核架构的兴起,并行化计算成为有效解决计算效率瓶颈的关键,各种并行化的分析算法被提出,更进一步地提高了分析的效率.

本文主要通过调研静态程序分析并行化方面的最新研究进展以及具有代表性的静态分析工具,简要综述静态分析并行化方面的研究动态,同时根据调研内容,讨论在静态分析并行化方面的未来可能的研究方向.本文第 1 节介绍静态分析的基本概念和关键技术.第 2 节简要讨论当前静态分析面临的挑战.传统算法优化研究及其不足将在第 3 节详细讨论.第 4 节和第 5 节重点介绍并行/分布式静态分析研究以及并行静态分析工具.第 6 节讨论未来可以开展的研究.第 7 节对本文进行总结.

1 静态程序分析简介

程序分析指的是对计算机程序进行人工或自动化的分析,以确认或发现其特性,比如性能、正确性、安全性等,是否符合预期[3].程序分析的历史几乎与程序的历史一样长,自从有了程序就有了程序分析.最初的分析主要是人工进行的,但人工分析往往需要消耗大量的时间和精力,因此,后来人们越来越多地关注自动分析.其中,编译技术的发展大大带动了程序的自动分析技术,目前的许多分析技术都可以在编译技术中找到基本雏形[4].

以分析过程“是否需要运行被测程序”为准,可将程序分析划分为两大类:静态分析和动态测试.静态分析是指在不运行代码的方式下进行的分析,动态测试则通过运行具体程序并获取程序的输出或内部状态等信息来验证或发现程序性质.一般来说,动态测试由于获取了具体的运行信息,因此报出的缺陷更为准确,但只能对有限的测试用例进行检查;而静态分析可以比较全面地考虑可能的执行路径,能够发现更多的缺陷.本文主要关注于静态分析.除了缺陷检测以外,静态分析还可用于程序理解、修复以及测试用例生成等方面,是程序语言和软件工程领域的一个重要研究方向[5].

静态分析涉及的基础理论包括抽象解释(abstract interpretation)、约束求解等,而关键技术包括数据流分析、指针分析、符号执行(symbolic execution)等.下面将进行简要介绍.此外,人们在发展出众多静态分析技术的同时,也开发了许多高效的静态分析工具,例如被广泛使用的商业静态分析工具 Coverity[6]、Fortify[7]、Klocwork[8]等,而且这些工具往往综合运用多种分析技术,这也是当前程序分析发展的一个趋势——各种分析技术的集成与融合.图 1 是一些关键的静态分析技术和工具的时间线.

1) 抽象解释

程序的抽象解释[12,17]就是使用抽象对象域上的计算逼近程序指称的对象域(具体对象域)上的计算,使得程序抽象执行的结果能够反映出其真实运行的部分信息.抽象解释本质上是一种为在计算效率和计算精度之间取得均衡,以损失计算精度求得计算可行性,再通过迭代计算增强计算精度的抽象逼近方法.基于抽象解释的代表性程序分析工具包括 Frama-C Value Analysis[18]、Astrée[19]等.

2) 数据流分析

数据流分析试图获取不同程序点上相关数据的流动信息[20],它对程序控制流图(control flow graph,简称

CFG)中的每个节点(一条语句或一个基本块)建立数据流方程:$out_b=trans_b(in_b)$, $in_b = join_{p\in pred_b}(out_p)$.其中,$trans_b$是基本块 b 的转移函数(transfer function),它作用于块 b 的入口状态 in_b,产生其出口状态 out_b,运算符 $join$ 将块 b 的各个前驱节点 p 的出口状态 out_p 合并起来,产生其入口状态 in_b.然后沿着程序的执行路径向前(或向后)传播状态信息(求解这一系列方程),由此确定在某个程序点有关变量的使用情况或可能的取值,例如可达定义(reaching definition)和活跃变量(live variable)分析.

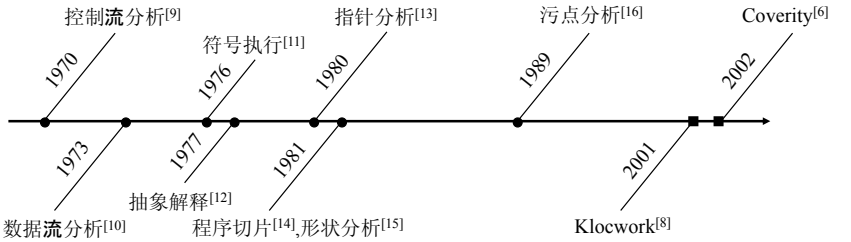

Fig.1 Timeline of key static analysis techniques and tools

图 1 关键的静态分析技术和工具时间线

3) 指针分析

指针分析[21−23]试图确定一个指针变量可能指向的内存对象的集合(指向分析)或不同指针变量是否指向同一内存区域(别名分析),其分析结果往往是其他程序分析的基础,例如安全分析[24]、缺陷检测[25]等,并且这些分析的效果严重依赖于指针分析的精度.

4) 符号执行

符号执行[11]通过使用抽象的符号代替具体值来模拟程序的执行.当遇到分支语句时,它会探索每一个分支,将分支条件加入到相应的路径约束中,若约束可解,则说明该路径是可达的.因其排除了不可达的路径,得到的结果更接近程序的实际执行情况.但由于需要探索的路径数随分支数呈指数级增长,存在路径爆炸问题,并且其效率受限于约束求解[26]的效率.

静态分析往往需要在分析精度和分析速度之间进行折中,而分析的精度很大程度上取决于分析时使用了多少控制流信息.根据考虑的控制流信息的不同,每一种静态分析方法又可分为是否是流敏感(flow-sensitive)、域敏感(field-sensitive)、路径敏感(path-sensitive)和上下文敏感(context-sensitive)的.流敏感的分析考虑程序语句的执行顺序.路径敏感的分析考虑各个分支之间的区别,记录每一条分支路径上的程序执行状态.上下文敏感指的是在过程间分析时,区分对同一过程的带不同上下文信息的不同调用.

2 当前静态分析面临的挑战和解决方案

作为软件技术研究领域的核心内容之一,静态分析经过多年的发展,已经在多个方面取得了长足的进步,但仍面临不少的挑战.比如,随着信息化时代的推进,软件已然成为现代社会的基础设施,随着软件功能以及应用范围的持续外延,通用功能软件的广泛复用,使得软件结构日趋复杂,软件规模不断增大,这对程序分析提出了更高或更新的要求.

1) 分析精度:理论上,根据 Rice 定理:不存在可以判定任何程序是否具有某种非平凡属性的通用算法[27],例如停机问题,且实际上对于一个有一定规模的程序,想要穷尽其所有可能的执行状态通常是不现实的.因此,我们往往不要求对程序进行完全精确的分析.这就可能导致误报(false positive)、漏报(false negative)等问题.为了降低误报率,提高分析的精度,我们需要考虑更多更完整的程序信息,例如前述的上下文敏感的过程间分析、流敏感/路径敏感的分析等.

2) 分析效率:但分析精度的提高,往往意味着更长的分析时间.程序分析作为软件开发生命周期的一个重要环节,其效率影响了整个软件开发的成本和进度,分析效率过低,会限制静态分析技术在工业界广泛使用.

3) 可扩展性:随着现代软件的功能越来越丰富、系统架构日趋复杂以及编程辅助工具代码生成能力的提升和开放源代码理念的被认可,百万行、千万行甚至更大规模的软件系统已成常态化存在[2].要对如此大规模软件系统进行(高精度)静态分析(高密集计算以及高内存消耗),必然对分析系统的可扩展性提出更高的要求.

4) 通用性:一方面,随着(主要是基于 Web 的)软件系统更多地是由用多种语言编写的多个部分构建而成[28],多语言分析变得更加重要;另一方面,由于程序性质的多样性等,目前程序分析的内容十分丰富,且相互之间的界限也不甚清晰,有些分析过程需要另外某个或某些分析的支持,有些分析针对具体的性质展开,而有些分析则可以支持多种性质的分析.

5) 新型软件形态:新型的软件形态不断出现,如面向区块链的智能合约、深度学习系统等,要针对这类软件进行静态分析,需要将已有的程序分析技术针对其特点进行适配,甚至提出全新的技术.例如,近年来,机器学习技术在业界得到了广泛的应用,尤其是深度学习技术和工具在语音识别、自动驾驶等领域取得了较大突破.然而,由于多层传播的复杂网络结构、黑盒形式的用户接口等特性,现有的程序分析技术难以直接应用,需要一系列新的特定的程序分析方案用以发现深度学习系统的潜在风险与程序错误,提升深度学习系统的质量[29].由于篇幅限制,本文不对此进行详述.

面对上述挑战,人们做了大量的研究工作,提出了各种解决方案,例如简单分析、基于摘要的分析、并行化分析等,见表 1.

Table 1 Current challenges and corresponding solutions of static analysis

表 1 静态分析面临的挑战和解决方案

挑战	解决方案							
	简单分析		优化分析			并行化分析		
	基于模式匹配	过程内	基于摘要	需求驱动	采用简洁表示	CPU 并行	分布式	GPU 并行
分析精度	–	–	[30,31]	[32]	[1,33]	[34–37]	[38,39]	–
分析效率	[40,41]			[32,42,43]		[34–37,44–46]	[38,39,47,48]	[49–51]
可扩展性			–	–		[36,37,46]		–
通用性			[30]	–	–	[35,36,45]	[38,39,48]	–

其中,简单分析这类工作只计算少量的语义信息,精确度低,在一定条件下能够较快地检测大规模软件;优化分析尽可能在不损失精度的前提下提高分析的效率,其主要关注对顺序算法的优化,采用的手段较为多样;并行化分析包括单机的 CPU 并行、GPU 并行以及分布式的(本文主要关注静态分析并行化方面的研究).后面会结合具体工作做详细介绍.

3 传统静态分析算法优化研究及其不足

为了提高分析的效率,简单分析这类工作采用简单的较浅语义的分析,如基于模式匹配或过程内的分析.它们只计算少量的语义信息,算法复杂度低,计算量小,因而在一定条件下能够检测大规模软件.例如,Engler 等人使用简单程序分析来研究操作系统内核中的错误[40].然而,此类简单分析语义信息完整性差、精确度低,如静态缺陷检测工具 FindBugs[41],分析的误报率通常高达 30%~100%[52],如此高的误报率,极大地限制了静态分析的实用性[53,54].

为了尽可能地在不损失精度的前提下提高分析的效率,一些研究人员在静态分析算法优化方面做了大量的研究工作.传统的优化研究主要关注对顺序算法的优化,主要代表性工作如下.

1) 基于摘要的分析

在模块化编程中,一个程序模块往往需要调用其他程序模块(或组件、类、函数等).对主调模块的分析必然涉及到对被调模块的分析.与代码复用能提高软件生产效率类似,基于摘要的分析通过复用被调模块的分析结果提高分析主调模块的效率.Cousot 等人从理论上总结了模块化分析的几种基本方法[30],针对模块间的循环依赖(circular dependency)的存在与否分别给出相应的解决方案.Tang 等人提出了基于树-邻接语言可达性(tree-adjoining language reachability,简称 TAL-reachability)的摘要分析技术(具体对过程间的数据依赖分析)[31],以描

述包含回调函数的条件可达性(conditional reachability)关系.

2) 需求驱动的分析

对于某些具有极端资源限制的环境,例如即时编译器和交互式开发环境等,全程序的分析时空开销太大,也不一定有必要.例如,IDE 往往只需关注开发人员检查的那部分代码.在这种情况下,需求驱动的分析可以作为一种解决方案,它只为客户查询做必需的工作.例如,Reps 应用逻辑程序设计的 magic-sets 转换实现需求驱动的过程间数据流分析[32].Sridharan 等人提出的需求驱动的指向分析算法可以从用户关注的某一个程序点出发,逆向求解该程序点的指向关系[42].此外,Zheng 等人提出基于上下文无关语言可达性(context-free language reachability,简称 CFL-reachability)的需求驱动的别名分析[43],实现及时性别名分析.

3) 采用简洁表示的分析

Sparse Evaluation Graph(SEG)是一种能够有效地获取程序中的数据流信息的中间表示,它基于观察:对于任何给定的分析问题,程序中的部分语句可能与分析问题无关[55−57].例如,Hardekopf 等人将其用于提高流敏感的指针分析的可扩展性[33].基于克隆的上下文敏感分析为每一处调用生成被调函数的一份实例,从而可以简单地在扩展了的调用图上运行上下文不敏感的算法,但对有一定规模的软件,上百万的上下文数量令人望而却步.Whaley 等人注意到这些上下文之间有许多相似,而二元决策图(binary decision diagram,简称 BDD)能够有效地表示大量冗余数据,于是提出可以使用 BDD 来表示上下文敏感的调用图[1],以减少内存需求以及计算开销,使得上述分析成为可能.

上述优化分析通过各种特定的技术虽然在一定程度上提高了分析的效率,但其代码实现更为复杂,工程难度提高.而且对顺序分析算法的优化对分析效率和可扩展性的提升终究有限,它们仍不足以支持对现实中大规模软件系统的高精度分析.其不足主要包括:

1) 复杂算法导致程序分析实现的高工作量问题

以上传统优化研究往往引入相对复杂精妙的算法,实现这些复杂算法往往给开发者带来极大的开发与维护工作量,这极大地影响了其实用性.然而,大部分复杂性源于算法优化部分而不是基本算法的实现.例如,在广泛使用的 Java 指针分析算法[58]中,基本算法实现部分占比很小,而超过 3/4 的代码用于近似(approximation)分析功能的实现,以确保在用户给定的时间预算内可以返回近似结果.

2) 程序分析可扩展性仍难满足实际应用需要

以上优化算法在很大程度上提高了分析的效率,针对特定分析任务,一些成熟的优化算法已经可以成功分析一定规模的系统[1,33,58].然而随着现代软件系统规模的不断扩大,要对现代大规模软件进行高精度程序分析必然对运算与内存提出更高的要求.例如,Linux 操作系统超过千万行的代码量,如此大的分析空间必然产生高密集计算以及对内存的高消耗,进而导致程序分析的低可扩展性.例如,以上传统优化算法都严重依赖内存,一旦内存不足,分析无法正常完成.正如 Aiken 等人[59]提到,内存将是限制高可扩展程序分析的主要瓶颈,对内存的依赖极大制约着传统优化方法的可扩展性.

4 静态分析并行化研究

针对传统静态分析算法优化研究的不足,为了进一步提高分析的效率和可扩展性,充分利用多核架构下的计算资源,一些研究人员提出了各种并行化的静态分析算法.相较于传统顺序算法优化研究,程序分析并行化[34,35,44−46]试图充分利用已有的丰富硬件计算资源,通过设计特定的并行化分析算法,尽量降低在并行处理环境下数据竞争引发的额外开销,力求并行效率最大化,从而达到较高的分析效率.同时,一些研究面向高精度程序分析的低可扩展性问题提出最新的解决方案,通过对单机海量硬盘[36,37]以及大规模分布式集群[38,39,47]的资源利用,实现高可扩展性分析目标.

下面根据处理环境的不同,分单机的 CPU 并行、分布式以及 GPU 实现这 3 类对这些工作展开分析介绍.

1) CPU 并行

其实,前述模块摘要生成也是可以并行化的,只要对这些模块的分析可以独立进行.例如,Voung 等人提出了

一种基于相对锁集(relative lockset)的静态数据竞争检测算法[60].相对锁集描述了相对于函数入口点持有的锁的变化,它允许我们独立于调用上下文创建函数摘要,这使得我们可以进行模块化的、自底向上的分析(易于并行化).

但是,在这里我们更多地关注于那些专门的并行化研究,例如,Méndez-Lojo 等人提出了一种基于约束图改写(constraint graph rewriting)的并行指向分析算法[44].它将基于包含(inclusion-based)的指针分析表述为一个集合约束(set constraint)问题,通过对约束图应用一些改写规则进行约束的求解.而这是可较好地并行的(amorphous data-parallelism[61]).实验结果表明,该方法不仅是可行的,而且超过了相应的高度优化的顺序实现,并且这是指向分析的第 1 个并行实现.

Rodriguez 等人提出了一种基于参与者模型(actor model)的并行(IFDS[62])数据流分析算法[34].它为 CFG 中的每个节点构建一个参与者,消息沿着边传送.因为参与者会缓冲所有收到的消息直到它们被处理,所以我们不再需要一个集中式工作表.但这样分析完成与否就不能用工作表是否为空来判定.对此,该算法使用一个全局计数器来监测未被处理的消息数.实验结果表明,使用 8 个核,IFDS-A(用 Scala 实现)的分析速度是相应的基线顺序实现的 3.35 倍.

上述两项研究工作都只是针对某一种(类)特定的静态分析问题进行特定的处理.为得到更普遍的方法,Su 等人提出了一种基于 CFL 可达性的并行指针分析算法[35].CFL 可达性最初由 Yannakakis 在数据库检索中提出[63],后被 Reps 应用到静态分析领域[64].除了指针分析,像数据流分析、程序切片、形状分析等都可以转化为 CFL 可达性问题.

另外,Albarghouthi 等人提出了一个使用 map-reduce 策略[65]并行化自顶向下分析的通用框架[45].在自顶向下的分析中,对过程 P 的查询会导致对 P 调用的过程的子查询.而这样一棵查询树可用 map-reduce 策略探索——map 阶段并行运行一组处于“ready”状态的查询,这可能会导致新的子查询,reduce 阶段管理查询之间的相互依赖关系,如垃圾收集那些不再被父查询需要的查询等,两者交替进行.实验结果表明,相比于顺序分析,该方式平均加速 3.71x,最大加速 7.4x(8 个核).

以上这些研究工作的计算都依赖于内存,一旦内存不足,算法就无法完成计算,系统的可扩展性受限.对此,Wang 等人提出了 Graspan[36],一个单机的基于海量硬盘的并行静态分析系统.Graspan 的图处理是一种核外计算(out-of-core computation)——如果程序图太大无法全部读入内存,则每次只(从硬盘)调入部分数据参加计算.通过对外存的利用,Graspan 支持对大规模系统软件(如 Linux 内核)的复杂代码分析(如上下文敏感的过程间分析).

在 Graspan 的基础上,Zuo 等人提出了 Grapple[37],一个用于大规模软件系统中状态相关缺陷检测的图系统.它通过上下文敏感和路径敏感的别名分析和数据流分析(可转化为带约束的 CFL 可达性问题)跟踪程序中指定类型的对象的流,对其进行有限状态特征检查,以寻找一类与状态相关的缺陷.Grapple 的后端是在 Graspan 的基础上加上路径约束,而约束的求解采用的是 Z3[26].

符号执行可被用于生成高覆盖率的测试用例.但对大多数现实程序而言,符号执行需要探索的路径数非常大,存在可扩展性问题.对此,Staats 等人提出了一种称为简单静态分区(simple static partitioning,简称 SSP)的技术来并行化符号执行[46].该技术用一组前置条件对符号执行树进行划分,前置条件通过先一次“shawdow”符号执行来收集计算.SSP 的并行实例之间需要很少的通信,且适用于多种架构(从多核机器到云).

2) 分布式

可单机的处理能力毕竟有限,为充分满足高可扩展性要求,有研究人员进行了分布式环境下静态分析系统的构建.例如,Google 认为,要扩展到大规模工业代码库,分析必须是可分片的(shardable),且能作为仅具部分信息的分布式计算的一部分运行.他们提出了 Tricorder[48],一个基于云的程序分析平台,不过只适用于简单的过程内分析.

Garbervetsky 等人提出了一个基于参与者模型(同文献[34])的分布式静态分析框架,并给出了调用图分析的实现[38].在设计分析时,我们可以选择为程序中的每个方法构建一个参与者,应当尽量将相关的参与者放在同一台机器上,以及考虑好节点失效如何恢复(按需重新计算状态)等问题.他们展示了处理大约 1 000 万行代码的

输入的可扩展能力,并指出通信开销是主要性能瓶颈.

BigSpa[39]是 Zuo 等人近期提出的一个基于大数据开发平台 Spark 优化设计实现的分布式静态分析系统,其通过高效的 join-process-filter 模型,可在短时间内(不高于 30min)实现对千万行规模代码的高精度分析.与 Graspan、Grapple 等类似,BigSpa 也是将静态分析问题转化为图可达性问题处理.

Cloud9[47]是 Bucur 等人提出的一个基于 KLEE[66]的分布式符号执行平台,它包含多个工作节点和一个负载均衡器(load balancer,简称 LB).工作节点运行独立的符号执行引擎,探索执行树的一部分,并将其进度发送给 LB.LB 分析上报信息,在必要时指导工作节点平衡彼此的负载.

3) GPU 实现

前述 CPU 并行算法,因 CPU 仅包含几个专为串行处理而优化的核心,效率提升有限.近年来,GPU 凭借其突出的大规模并行计算能力(由数以千计的更小的、专为并行处理而设计的核心组成)已被广泛应用于大数据分析等领域.并且也已经有利用 GPU 加速静态分析的研究工作.例如,Méndez-Lojo 等人(继文献[44])做了 Andersen 指向分析的 GPU 实现[49].考虑到 GPU 架构的一些特性(很影响性能):主要是 SIMT(single instruction multiple threads)执行模式和合并访存(memory coalescing)[67],该工作选用稀疏位向量(sparse bit vector)来表示 GPU 上的图.位向量的一个元素是 32 个字,第 i 个字被分配给一个线程束中的第 i 个线程(线程束是 GPU 的基本执行单元,一个线程束 32 个线程),这样对一个元素的全局内存读写就合并为一次请求,而且集合操作导致的线程束内线程之间的分歧也很小.实验结果表明,该实现当在 14 SMGPU 上执行时,与相应的顺序 CPU 实现相比,平均速度提高了 7 倍,且优于在 16 个 CPU 核上的并行实现.

之后,Su 等人又实现了 CPU-GPU 异构计算的 Andersen 分析系统[50].它根据 CPU 端和 GPU 端处理不同类型的任务(改写规则)的适合程度动态地分配负载.两端相同地初始化约束图,然后不断地各自应用分到的改写规则,交换新加的边,直至不动点.实验结果表明,该 CPU-GPU 解决方案(平均)比仅用 CPU 的解决方案快 50.6%,比仅用 GPU 的解决方案快 78.5%.在他们的后续研究中,给出了 Andersen 分析的一个更为高效的 GPU 实现[51].Andersen 算法对图进行了大量修改,这些修改是高度不规则的,且无法静态预测,导致工作负载难以均衡.为了有效地在 GPU 上并行化 Andersen 分析,他们引入了一种失衡感知(imbalance-aware)的工作负载划分方案,最初是以线程束为中心的方式,但后来当检测到负载不均衡时切换到基于任务池的模型.实验结果表明,该实现的分析速度(在 NVIDIA Tesla K20c GPU 上)比之前的平均提高了 46%.

5 并行静态分析工具

学术界和工业界在发展出众多静态分析技术的同时,也开发了许多高效的静态分析工具.而随着多核时代的到来,越来越多的静态分析工具开始支持在多核平台上的并行分析(或特为此研发).表 2 列出了一些使用较为广泛(商业)/比较有代表性(学术研究)的支持并行的静态分析工具及其特征.

Table 2 Parallel static analysis tools and their features

表 2 支持并行的静态分析工具及其特征

工具	并行类型	检测类型	精度	检测语言	商业/学术研究
Fortify	单机 CPU	缺陷和安全漏洞	过程间	Java、C#、C、C++、Swift、PHP 等	商业
Parasoft	分布式	缺陷和安全漏洞	过程间	C、C++、Java、.NET	商业
Coverity	单机 CPU	缺陷和安全漏洞	路径、流和部分上下文敏感	C、C++、Java、C#、JavaScript 等	商业
CodeSonar	单机 CPU	缺陷和安全漏洞	路径和上下文敏感	C、C++、Java	商业
Clang	单机 CPU	缺陷	路径、流和上下文敏感	C、C++、Objective-C	学术研究
Klocwork	单机 CPU	缺陷和安全漏洞	过程间	C、C++、C#、Java	商业
Saturn	分布式	缺陷检测和验证	路径和上下文敏感	C	学术研究
Graspan	单机 CPU	缺陷	上下文敏感	理论上任何语言	学术研究
Grapple	单机 CPU	状态相关缺陷	路径和上下文敏感	理论上任何语言	学术研究
BigSpa	分布式	缺陷	上下文敏感	理论上任何语言	学术研究
GpuSpa	单机 CPU	缺陷	上下文敏感	理论上任何语言	学术研究

注:并行类型是指工具支持的并行方式,检测类型是指能够检测的软件质量问题,精度是指分析带有的敏感类型,“过程间”是指分析是不敏感的

下面对这些工具进行介绍.

1) Saturn

Saturn 项目[68]旨在探索用于缺陷查找和验证的高可扩展和精确的程序分析技术.Saturn 主要基于以下 3 个思想.

- Saturn 是基于摘要的:对函数 *f* 的分析是对 *f* 的行为的总结.在 *f* 的调用点,仅使用 *f* 的摘要.
- Saturn 也是基于约束的:分析被表示为一个描述一个程序点的状态如何与相邻程序点的状态相关的约束系统.
- Saturn 中的程序分析以逻辑编程语言表达,并带有一些扩展以支持约束和函数摘要.

结合起来,这些思想使 Saturn 能够简洁地表示精确的分析,同时还能扩展到非常大的程序,例如整个 Linux 内核和其他大型开源项目,并且已经在这些项目中发现了数千个以前未知的错误.

当前版本的 Saturn 提供了并行实现,允许同时分析多个函数,通常使用 25 个~80 个处理器的集群,具体取决于程序大小和分析的计算强度.Saturn 目前只用于 C 程序,当然这些想法可以应用于其他语言.

2) Coverity

Coverity[6]是 Synopsys 的软件开发产品品牌,主要包括静态代码分析工具和动态代码分析服务,其帮助工程师和安全团队在用 C、C++、Java、C＃、JavaScript 等编写的源代码中发现缺陷和安全漏洞.

静态分析用于缺陷检测的一个重要但经常被忽视的目标,是能够快速地向程序员显示缺陷.但通过简单地添加更多资源(CPU,内存)或基于缺陷检测功能将分析拆分成多个子分析来减少分析时间的方式的改进有限.对此,Coverity 提出了一种基于工作单元(work unit)的并行和增量静态分析技术[69].它结合了自顶向下、自底向上和全局规范推断.工作单元包含要分析的函数的源代码和抽象语法树、被调用者行为摘要和调用上下文摘要、分析选项等.这样设计使其能够被独立、快速和确定地分析.工作单元由分析主进程创建和消耗,主进程协调多个分析过程,并增量计算.同时,多个分析工作进程使用抽象解释来计算工作单元结果.

3) Klocwork

除了发现编程错误之外,Klocwork[8]还允许用户检测代码中的安全漏洞.该工具能够与许多常见的 IDE(如 Eclipse、Visual Studio 和 Intellij IDEA)很好地集成.它还可以进行逐行检查并提供立即解决缺陷的功能.

Klocwork 支持在多核和多处理器机器上运行并行分析以提高性能.Klocwork 默认启用并行分析.它根据可用处理器核心的数量将编译作业的数量设置为适当的级别.如果没有多核或多处理器机器,则运行顺序分析.

4) Fortify

Fortify[7]是一款惠普的宣称可让开发人员构建无错误且安全的代码的静态分析工具.它能够支持 25 种语言,包括 Java、C#、C、C++、Swift、PHP 等.在扫描代码时,它会对发现的问题进行排序,并确保最先修复最关键的问题.从 4.00 版本开始,Fortify 支持并行处理.

5) Parasoft

Parasoft[70]与其他静态分析工具相比,能够支持多种类型的静态分析技术,如基于模式的、基于流的、多变量分析等.该工具的另一个好处是它提供自动缺陷预防功能.Parasoft 支持集群上的并行分析.

6) CodeSonar

CodeSonar[71]是一款 Grammatech 的静态分析工具,它不仅可以帮助用户找到编程错误,而且还有助于找出领域相关的编码错误.它还允许自定义检查点以及可根据需求配置内置检查.CodeSonar 可以对 10M+行代码执行全程序分析,而且分析可并行运行,以充分利用多核环境.

7) Clang Static Analyzer

Clang 静态分析器[72]是一个用于检测 C、C++和 Objective-C 程序中的缺陷的源代码分析工具.当前,它既可以作为独立工具运行(从命令行调用),也可以在 Xcode 中运行.该分析器是 100%开源的,是 Clang 项目的一部分.它也支持并行分析.

8) Graspan 系列(Grapple、BigSpa、GpuSpa)

这 4 个静态分析工具是我们研究团队在静态分析并行化方面的一系列的研究成果.我们的目标是构建高可扩展的大规模高精度静态分析系统.其中,Graspan、Grapple 和 BigSpa 前面已经给出介绍.GpuSpa[73]是一个基于 GPU 的核外静态分析引擎.它在 GPU 上进行并行的基于 CFL 可达性的静态分析计算,并利用硬盘暂存中间结果,其代码在 Github 上开源.BigSpa 和 GpuSpa 都支持上下文敏感的过程间分析.

6 未来可以开展的研究

传统的静态分析算法优化研究主要关注对顺序算法的优化,但随着摩尔定律的失效,单核计算效率的提升不足,顺序算法分析效率相对低下.而随着多核、众核架构的兴起,并行化计算成为有效解决计算效率瓶颈的关键,人们越来越多地关注于静态分析的并行化解决方案.下面从 3 个方面讨论未来可以开展的研究.

1) 针对特定的静态分析的并行化研究

程序的类型不同,应用环境不同,关注的性质不同,需要的静态分析技术也多种多样.不同的分析技术之间可能差异较大.静态分析并行化不仅要考虑运行平台,而且要考虑要进行的分析的特点,比如其各部分任务或数据的独立性如何,还要考虑如何划分、均衡工作负载等问题.对不同的分析技术,有效的并行化策略可能大不相同.因此,针对各种特定的静态分析技术的高效的并行化处理还有很多可做的工作.

2) 通用静态分析并行化研究

另一方面,探索通用的并行化方法也很有意义.例如前述的 Garbervetsky 等人提出的基于参与者模型的分布式静态分析框架.参与者模型是一种并行计算模型.参与者被视为并行计算的基本单元:当一个参与者接收到一个消息时,它可以做出一些决策,创建更多的参与者,传送更多的消息等.在静态分析中,参与者可以是函数、类、基本块等,而传送的消息等则取决于分析的类型.研究通用的并行化方法对各种特定的分析的适用程度及可能的优化对后来的工作具有一定的指导意义.

3) 基于硬件平台的静态分析加速研究

并行化研究兴起的根本原因还是底层硬件平台的发展.多核、众核以及分布式架构的日益普及,机器算力的大幅提升,极大地推动了软件技术的发展,也为程序分析技术注入了新的生机与活力.例如,对于硬件加速这方面,除了前述的 GPU 实现,还可以考虑半定制化的 FPGA(field-programmable gate array,现场可编程门阵列)芯片.FPGA 具有天然的并行性,并且延迟低、功耗小,但开发难度较大,目前还没有看到利用 FPGA 加速静态分析的工作.此外,还会有更多新的硬件技术不断出现,它们也可能被很好地用于静态分析.

7 总 结

静态分析作为软件缺陷检测的关键技术手段,一直是学术界、工业界的研究和实践热点.近年来,由于软件的功能越来越丰富,以及通用功能软件的广泛复用,使得软件结构日趋复杂,代码规模不断扩大,百万行、千万行,甚至更大规模的软件系统已不少见,这也为静态分析带来了新的挑战.

本文在该背景下,调研了学术界和工业界面对此挑战,在提升静态分析的精度、效率和可扩展性方面所取得的最新代表性研究成果,并从单机的 CPU 并行、GPU 并行以及分布式等主要方向对这些利用硬件资源进行并行优化的方法进行了充分的分析和讨论.最后,还从算法和算力角度对未来可以开展的研究进行了展望.

References:

[1] Whaley J, Lam MS. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proc. of the PLDI. 2004. 131−144.

[2] Codebases: Millions of lines of code. 2015. https://informationisbeautiful.net/visualizations/million-lines-of-code/

[3] Nielson F, Nielson HR, Hankin C. Principles of Program Analysis. Springer-Verlag, 1999.

[4] Mei H, Wang QX, Zhang L, Wang J. Software analysis: A road map. Chinese Journal of Computers, 2009,32(9):1697−1710 (in Chinese with English abstract).

[5] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019,30(1):80−109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[6] Coverity. 2019. http://www.coverity.com/

[7] Fortify. 2019. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

[8] Klocwork. 2019. https://www.klocwork.com/products-services/klocwork

[9] Allen FE. Control flow analysis. ACM SIGPLAN Notices, 1970,5(7):1−19.

[10] Kildall GA. A unified approach to global program optimization. In: Proc. of the POPL. 1973. 194−206.

[11] King JC. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385−394.

[12] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the POPL. 1977. 238−252.

[13] Weihl WE. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In: Proc. of the POPL. 1980. 83−94.

[14] Weiser M. Program slicing. In: Proc. of the ICSE. 1981. 439−449.

[15] Muchnick SS, Jones ND. Program Flow Analysis: Theory and Applications. Englewood Cliffs: Prentice-Hall, 1981.

[16] Taint checking. 2019. https://en.wikipedia.org/wiki/Taint_checking

[17] Cousot P, Cousot R. Systematic design of program analysis frameworks. In: Proc. of the POPL. 1979. 269−282.

[18] Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-C: A software analysis perspective. Formal Aspects of Computing, 2015,27(3):573−609.

[19] Astrée. 2019. https://www.absint.com/astree/index.htm

[20] Aho AV, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[21] Andersen LO. Program analysis and specialization of the C programming language [Ph.D. Thesis]. DIKU: University of Copenhagen, 1994.

[22] Hind M. Pointer analysis: Haven't we solved this problem yet? In: Proc. of the PASTE. 2001. 54−61.

[23] Smaragdakis Y, Balatsouras G. Pointer analysis. Foundations and Trends in Programming Languages, 2015,2(1):1−69.

[24] Fink SJ, Yahav E, Dor N, Ramalingam G, Geay E. Effective typestate verification in the presence of aliasing. ACM Trans. on Software Engineering and Methodology, 2008,17(2):1−34.

[25] Li L, Cifuentes C, Keynes N. Practical and effective symbolic analysis for buffer overflow detection. In: Proc. of the FSE. 2010. 317−326.

[26] De Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the TACAS. 2008. 337−340.

[27] Rice HG. Classes of recursively enumerable sets and their decision problems. Trans. of the American Mathematical Society, 1953, 74(2):358−366.

[28] Binkley D. Source code analysis: A road map. In: Proc. of the Future of Software Enginering. 2007. 104−119.

[29] Masuda S, Ono K, Yasue T, Hosokawa N. A survey of software quality for machine learning applications. In: Proc. of the ICSTW. 2018. 279−284.

[30] Cousot P, Cousot R. Modular static program analysis. In: Proc. of the CC. 2002. 159−179.

[31] Tang H, Wang X, Zhang L, Xie B, Zhang L, Mei H. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In: Proc. of the POPL. 2015. 83−95.

[32] Reps T. Solving demand versions of interprocedural analysis problems. In: Proc. of the CC. 1994. 389−403.

[33] Hardekopf B, Lin C. Semi-sparse flow-sensitive pointer analysis. In: Proc. of the POPL. 2009. 226−238.

[34] Rodriguez J, Lhoták O. Actor-based parallel dataflow analysis. In: Proc. of the CC. 2011. 179−197.

[35] Su Y, Ye D, Xue J. Parallel pointer analysis with CFL-reachability. In: Proc. of the ICPP. 2014. 451−460.

[36] Wang K, Hussain A, Zuo Z, Xu G, Sani AA. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In: Proc. of the ASPLOS. 2017. 389−404.

[37] Zuo Z, Thorpe J, Wang Y, Pan Q, Lu S, Wang K, Xu G, Wang L, Li X. Grapple: A graph system for static finite-state property checking of large-scale systems code. In: Proc. of the EuroSys. 2019. 1−17.

[38] Garbervetsky D, Zoppi E, Livshits B. Toward full elasticity in distributed static analysis: The case of callgraph analysis. In: Proc. of the FSE. 2017. 442−453.

[39] Zuo Z, Gu R, Jiang X, Wang Z, Huang Y, Wang L, Li X. BigSpa: An efficient interprocedural static analysis engine in the cloud. In: Proc. of the IPDPS. 2019. 771−780.

[40] Engler D, Chen DY, Hallem S, Chou A, Chelf B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proc. of the SOSP. 2001. 57−72.

[41] FindBugs. 2019. http://findbugs.sourceforge.net/

[42] Sridharan M, Gopan D, Shan L, Bodík R. Demand-driven points-to analysis for Java. In: Proc. of the OOPSLA. 2005. 59−76.

[43] Zheng X, Rugina R. Demand-driven alias analysis for C. In: Proc. of the POPL. 2008. 197−208.

[44] Méndez-Lojo M, Mathew A, Pingali K. Parallel inclusion-based points-to analysis. In: Proc. of the OOPSLA. 2010.428−443.

[45] Albarghouthi A, Kumar R, Nori AV, Rajamani SK. Parallelizing top-down interprocedural analyses. In: Proc. of the PLDI. 2012. 217−228.

[46] Staats M, Păsăreanu CS. Parallel symbolic execution for structural test generation. In: Proc. of the ISSTA. 2010. 183−194.

[47] Bucur S, Ureche V, Zamfir C, Candea G. Parallel symbolic execution for automated real-world software testing. In: Proc. of the EuroSys. 2011. 183−198.

[48] Sadowski C, Van Gogh J, Jaspan C, Söderberg E, Winter C. Tricorder: Building a program analysis ecosystem. In: Proc. of the ICSE. 2015. 598−608.

[49] Méndez-Lojo M, Burtscher M, Pingali K. A GPU implementation of inclusion-based points-to analysis. In: Proc. of the PPoPP. 2012. 107−116.

[50] Su Y, Ye D, Xue J. Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems. In: Proc. of the HiPC. 2013. 149−158.

[51] Su Y, Ye D, Xue J, Liao XK. An efficient GPU implementation of inclusion-based pointer analysis. IEEE Trans. on Parallel and Distributed Systems, 2015,27(2):353−366.

[52] Kremenek T, Engler D. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In: Proc. of the SAS. 2003. 295−315.

[53] Johnson B, Song Y, Murphy-Hill E, Bowdidge R. Why don't software developers use static analysis tools to find bugs? In: Proc. of the ICSE. 2013. 672−681.

[54] Lewis C, Lin Z, Sadowski C, Zhu X, Ou R, Whitehead EJ. Does bug prediction support human developers? Findings from a Google case study. In: Proc. of the ICSE. 2013. 372−381.

[55] Ramalingam G. On sparse evaluation representations. Theoretical Computer Science, 2002,277(1-2):119−147.

[56] Choi JD, Cytron R, Ferrante J. Automatic construction of sparse data flow evaluation graphs. In: Proc. of the POPL. 1991. 55−66.

[57] Duesterwald E, Gupta R, Soffa ML. Reducing the cost of data flow analysis by congruence partitioning. In: Proc. of the CC. 1994. 357−373.

[58] Sridharan M, Bodík R. Refinement-based context-sensitive points-to analysis for Java. In: Proc. of the PLDI. 2006. 387−400.

[59] Aiken A, Bugrara S, Dillig I, Dillig T, Hackett B, Hawkins P. An overview of the Saturn project. In: Proc. of the PASTE. 2007. 43−48.

[60] Voung JW, Jhala R, Lerner S. Relay: Static race detection on millions of lines of code. In: Proc. of the FSE. 2007.205−214.

[61] Pingali K, Kulkarni M, Nguyen D, Burtscher M, Méndez-Lojo M, Prountzos D, Sui X, Zhong Z. Amorphous data-parallelism in irregular algorithms. Technical Report, TR-09-05, The University of Texas at Austin, 2009.

[62] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the POPL. 1995. 49−61.

[63] Yannakakis M. Graph-theoretic methods in database theory. In: Proc. of the PODS. 1990. 230−242.

[64] Reps T. Program analysis via graph reachability. In: Proc. of the ILPS. 1997. 5−19.

[65] Dean J, Ghemawat S. MapReduce: Simplifieddata processing on large clusters. In: Proc. of the OSDI. 2004. 137−150.

[66] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the OSDI. 2008. 209−224.

[67] CUDA C Programming Guide 10.0. NVIDIA, 2018.

[68] Xie Y, Aiken A. Saturn: A scalable framework for error detection using Boolean satisfiability. ACM Trans. on Programming Languages and Systems, 2007,29(3):1−16.

[69] McPeak S, Gros CH, Ramanathan MK. Scalable and incremental software bug detection. In: Proc. of the FSE. 2013. 554−564.

[70] Parasoft. 2019. https://www.parasoft.com/

[71] CodeSonar. 2019. https://www.grammatech.com/products/codesonar

[72] Clang Static Analyzer. 2019. https://clang-analyzer.llvm.org/

[73] Zuo ZQ, Lu SM, Wang LZ. GpuSpa: A scalable GPU-based out-of-core system for inter-procedural static analysis. 2019. https://github.com/scalablespa/GpuSpa

附中文参考文献:

[4] 梅宏,王千祥,张路,王戟.软件分析技术进展.计算机学报,2009,32(9):1697−1710.

[5] 张健,张超,玄跻峰,熊英飞,王千祥,梁彬,李炼,窦文生,陈振邦,陈立前,蔡彦.程序分析研究进展.软件学报,2019,30(1):80−109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

陆申明(1994−),男,浙江杭州人,硕士生,主要研究领域为程序分析.

左志强(1986−),男,博士,助理研究员,CCF专业会员,主要研究领域为系统软件,软件工程,程序语言,大数据系统.

王林章(1973−),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为模型驱动的软件测试与验证,安全测试,软件测试自动化.