分类号 TP311.5 学号 16060115

UDC 004.41 密级 公 开

工程硕士学位论文

面向性能优化的配置故障诊断技术研究

硕士生姓名 李云峰

学 科 专 业 软件工程

研 究 方 向 软件工程方法与技术

指 导 教 师 李姗姗 副教授

协助指导教师

国防科技大学研究生院

二〇一八年十月

论文书脊

（此页只是书脊样式，学位论文不需要印刷本页。）

面向性能优化的配置故障诊断技术研究

国防科技大学研究生院

Research on Configuration Fault Diagnosis Technology for Performance Optimization

Candidate：Yunfeng Li

Supervisor：ShanShan Li

Associate Supervisor：

A dissertation

Submitted in partial fulfillment of the requirements

for the degree of Master of Engineering

in Discipline Title

Graduate School of National University of Defense Technology

Changsha，Hunan，P.R.China

March，2018

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：面向性能优化的配置故障诊断技术研究

学位论文作者签名：李云峰　　日期：2018年10月24日

学位论文版权使用授权书

本人完全了解国防科技大学有关保留、使用学位论文的规定。本人授权国防科技大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：面向性能优化的配置故障诊断技术研究

学位论文作者签名：李云峰　　日期：2018年10月24日

作者指导教师签名：　　日期：2018年10月24日

目 录

表 目 录

图 目 录

摘 要

随着软件系统规模日益增加，软件中非功能属性配置项数量也不断增更多。软件配置相关的软件性能瓶颈已经成为软件性能下降的重要原因之一，理解配置项如何影响软件性能对检测配置相关的性能瓶颈至关重要。检测配置相关的性能瓶颈可以帮助开发人员更好的理解配置对软件运行状态的影响，更快的定位问题所在以及修复软件性能问题，提高效率。

已有工作主要分为两类：机器学习和性能分析方法。机器学习方法构建软件性能和配置项的模型；然而，构建准确度高的性能模型会产生不可接受的代价，并且这种模型受许多外部因素（例如，系统环境和工作负载）的影响。传统的性能分析方法通过代码插桩以检测每个方法的运行时间，不可避免地引入更多的开销。本课题从资源的角度提出配置与软件性能之间的新观点，配置项通过影响资源使用来影响性能，即配置相关的性能问题是由于资源相关的配置项引起的。本课题研究了 75 个与配置相关的性能问题以验证本课提出的假设，设计并实现了一个工具 CBHunter，基于资源依赖分析以检测软件配置相关的性能瓶颈，即通过对配置项影响代码块打分以量化该配置项对性能的影响程度，然后利用性能分析的方法检测软件配置相关的性能瓶颈。本课题的主要工作有以下几个方面：

1、从 Apache 的 bug report 系统筛选 75 个配置相关的性能问题，同时筛选 50 个资源相关的配置项。分析软件源码验证了本课题提出的配置通过影响软件资源的使用来影响软件性能。同时，本课题统计导致 75 个配置相关的性能问题的资源类型，经统计发现，CPU、内存和 I/O 是导致性能问题较多的资源类型。

2、从配置项通过影响资源使用来影响性能的观点出发，CBHunter 通过资源依赖分析的方法自动对配置项打分排序。CBHunter 首先对配置项切片，获取配置项影响的所有语句，并根据不同类型语句的基础分，同时针对处于循环或者分支中的语句，利用 in-house 测试的方法计算分支执行的概率和循环执行的次数，计算每个配置项的得分并对它们排序。实验结果表明，在排名前 10%的配置项中导致性能问题的配置项占比达到 80%。

3、CBHunter 实现了检测性能瓶颈的自动框架。CBHunter 对排名前 10%的配置项影响的语句插桩，通过运行测试程序，以检测软件由配置导致的性能瓶颈。本课题将 CBHunter 应用于 6 个性能问题，实验结果表明，CBHunter 检测出 5 个配置相关的性能瓶颈，与 JAVA 的标准性能分析工具 HPROF 对比，CBHunter 检测配置相关的性能瓶颈能力是 HPROF 的 5.9 倍。

基于资源依赖分析和动态插桩分析，CBHunter 能够检测影响软件性能的配置项，有效降低配置空间维度；其次，减少动态分析的插桩数量，降低测试的运行开

销，提高开发人员诊断性能问题的效率。

主题词：配置相关的性能瓶颈；资源依赖分析；性能分析；

ABSTRACT

As software systems grow in size, the number of non-functional attribute configuration options in software continues to increase. Configuration-related software performance bottlenecks are one of the important reasons for software performance degradation. Understanding how configuration options affect software performance is critical to detecting configuration-related performance bottlenecks. Detecting configuration-related performance bottlenecks can help developers better understand the impact of configuration on software, speed up problem location, and fix software performance issues to improve efficiency.

There are two main categories of work: machine learning and profiling. Machine learning methods build models of software performance and configuration options; however, building high-accuracy performance models can result in unacceptable costs, and this model is affected by many external factors (such as system environment and workload). Traditional profiling tools instrument source code to detect the runtime of each method, inevitably introducing more overhead. This paper proposes a new perspective between configuration options and software performance from the perspective of resources. Configuration options affect performance by affecting resource usage. That is, configuration-related performance problems are caused by resource-related configuration items. This paper studied 75 configuration-related performance problems to verify the hypothesis proposed. We designed and implemented a tool CBHunter based on resource dependency analysis. CBHunter detect software configuration-related performance bottlenecks, that is, by analyzing the configuration options to affect the code segments. To quantify the impact of the configuration option on performance, and then use the profiling to detect performance bottlenecks related to software configuration options. The main work of this topic has the following aspects:

1. We select 75 configuration-related performance issues from Apache's bug report system and 50 resource-related configuration options. By analyzing software source code to verify that the configuration options affects software performance by affecting the usage of software resources. According to statistics, CPU, memory and I/O are resource types that cause more performance problems.

2. From the point of view that configuration options affect performance by affecting resource usage, CBHunter automatically sorts configuration options by resource dependency analysis. Through configuration option slicing, CBHunter can get all the statements affected by the configuration option. And according to the basis of different types of statements, CBHunter use the in-house testing method to calculate the probability of branch execution and loop execution. The score for each configuration option is calculated and CBHunter sort all configuration options. The experimental results show

that the performance-related configuration options in the top 10% of the configuration options account for 80%.

3. CBHunter implements an automated framework for performance analysis. CBHunter instrument the statements affecting the top 10% of configuration options. By running a software test program, the performance bottleneck caused by the configuration of the software has been determined. This paper applies CBHunter to six performance problems. The experimental results show that CBHunter detects 5 configuration-related performance bottlenecks and compares it with JAVA's standard performance analysis tool HPROF. The ability of detecting configuration-related performance bottleneck of CBHunter is 5.9 times that the ability of HPROF. .

Based on static configuration options scoring analysis and dynamic profiling analysis, CBHunter can detect configuration options that affect software performance. CBHunter effectively reduce the number of configuration options that need dynamic analysis. Second, CBHunter reduces the number of instrumentations and reduce overhead, and improve the efficiency of diagnosing performance issues.

Key Words： Configuration-related performance bottleneck； Resource dependence analysis；Profiling；

第一章　绪论

随着软件系统不断发展，许多大规模软件系统和分布式软件系统已经广泛应用于通信、飞行、航天、制造和云计算等各个领域，在现代社会拥有着十分重要的地位。软件发展推动着社会进步，但是随着软件自身规模越来越庞大、程序逻辑越来越复杂、软件安全性、可靠性、性能要求不断提高，以及软件出现问题难以修复制约着软件本身的发展。其中软件配置相关的性能问题已经造成软件效率低下，浪费时间和计算资源的结果。因此，针对软件配置相关的性能问题的研究也逐渐成为研究主题。本章 1.1 节阐述本课题的研究来源和研究背景，并且简要概述目前针对配置相关的性能问题的已有工作，1.2 节概述本课题的研究内容及本课题主要贡献，1.3 节介绍本文的组织结构。

1.1 课题研究背景

本节介绍本课题的研究来源和研究背景。

1.1.1 课题来源

本课题来源于国家重点研发计划（可持续演化的智能化软件理论、方法和技术）。

1.1.2 课题背景

随着社会的不断进步，软件系统已经在各个领域得到广泛应用，在现代社会中扮演着举足轻重的角色，发挥了重要的作用。随着软件系统的不断发展，人们对软件的可靠性，安全性，性能要求越来越高，导致软件规模不断增大，软件复杂度不断提升。例如，Hadoop 分布式开源软件的 2.8.0 版本，源码文件数量超过 8000，代码总行数接近千万[1]。同时，软件系统提供更多更加灵活的配置项以使用户根据需求配置软件[2]。例如，Hadoop Common 软件中共有个 200 多个配置项，HDFS 中有 400 多个配置项。非功能属性的配置项所占的比例日益增加[2]，这些配置项与性能，安全性，可靠性密切相关。

提高软件性能是软件演化和维护最重要的任务之一，配置也成为引发软件性能问题的主要原因之一[3]。在对 148 家企业的调查中，92％的企业认为提高软件性是最重要的任务之一[4]。近年来，软件配置不当导致的软件性能问题造成了巨大的商业损失[5-7]。理解大规模软件系统的源码和配置空间十分困难，并且软件系统的复杂度日益增加，从而产生的软件性能问题导致的软件崩溃，使得不同领域损

失接近 20%[8]。针对软件性能问题，软件工程师通常利用随机性能测试以揭露软件性能瓶颈，以帮助工程师增强软件性能[9]。开发人员和测试人员需要性能管理工具来自动检测软件性能问题，以便在保持软件维护成本较低的同时实现更好的软件性能[9]。同时，为了获得不同配置下的软件性能，开发人员测试不同配置组合的软件性能，以获得性能最优的配置组合。检测配置相关的性能瓶颈对于由配置引发的软件性能十分重要，特别是在大规模分布式软件中。

但是诊断配置相关的软件性能瓶颈面临巨大的挑战。主要有以下三个原因：（1）配置空间巨大，不同配置项之间复杂的约束关系。软件系统拥有着上百个配置项，这些配置项之间具有复杂的约束关系，这种约束关系导致部分配置项只有在特定的条件下才能被触发，如图 1.1 所示，只有当 if 条件语句中的配置项 reloadGroupsInBackGround 为真时，reloadGroupsThreadCount 才会起作用。（2）没有详细文档描述配置如何影响软件性能，文档的缺失使得软件性能问题更加困难。软件配置文件中只简单描述该配置项的功能及作用，没有描述该配置项会影响软件性能。HDFS-12511 的 bug report 中明确写出希望在配置文件中为配置项增加标签，如图 1.2 所示。（3）当性能问题发生时，软件没有很好的反馈。软件性能问题不同于软件功能 bug，当软件出现性能问题时，通常不会记录日志，导致很难定位问题所在，只能依靠开发人员修复性能问题。

```
/*java/org/apache/hadoop/security.java*/
GroupCacheLoader() {
     if (reloadGroupsInBackground) {
          ThreadFactory threadFactory = ...;
          ThreadPoolExecutor parentExecutor =
new ThreadPoolExecutor(
               reloadGroupsThreadCount,...);
          ...
       }
}
```

图 1.1 配置之间约束关系

```
/*Add tags to config:*/

<property>
  <name>ozone.ksm.handler.count.key</name>
  <value>200</value>
  <tag>OZONE,PERFORMANCE,KSM</tag>
  <description>
   The number of RPC handler threads for each KSM
service endpoint.
  </description>
 </property>
```

图 1.2 HDFS-12511 增加配置项标签

软件性能分析（profiling）是一种重要性能分析技术，揭露软件性能瓶颈[9]。软件工程师通常使用性能分析工具，向目标软件中插入指令以获得函数调用次数，内存使用情况和执行时间。在进行性能分析时，软件工程师首先利用性能问题工具检测目标软件，然后根据函数调用次数等信息，确定该软件的性能瓶颈。性能分析工具广泛用于软件开发生命周期的不同阶段，以分析软件性能瓶颈。但是性能分析工具需要对软件大量插桩，以检测运行时间，执行次数等信息，加重性能问题，掩盖真实的性能瓶颈。因此，如果能够有效减少插桩的数量，那么将会大大提高诊断性能问题的效率。

1.1.3 研究现状

目前研究软件配置与性能的主要方向有预测最优配置，检测性能瓶颈，诊断性能问题和配置故障诊断四大类。

性能瓶颈检测主要关注寻找软件的性能瓶颈，利用插桩技术，记录软件执行路径、函数执行次数，以及函数的执行时间，以确定软件的性能瓶颈。主要工作包括：构建软件有限状态机的 ProCrawl[17]，HOLMES[16]关注有错误的执行路径，利用集群聚类方法检测性能瓶颈的 AutoAnalyzer[18]，以及基于遗传搜索算法的 GA-Prof[9]，基于机器学习方法的 FOREPOST[20]等。

预测最优配置主要是构建不同配置组合与软件性能的模型，得出最优性能的配置组合。首先测试不同配置组合的软件性能（响应时间，网络延迟，吞吐量等），并利用机器学习的方法构建模型学习规则，描述配置和软件性能之间的关系。同时结合不同的采样算法，有效的减小训练样本的开销，并得到更加精确配置组合。构建配置与软件性能模型主要包括 Norbert Siegmund、Jianmei Guo、Sven Apel 等人[10-15]提出的基于分类回归树（CART），迭代学习等学习算法和随机采样，成对采样，基于频率的采样所发。

基于程序分析的方法重点关注系统中与性能相关的语句，通过数据流和控制流等信息流的分析，较为准确的获取影响软件性能的代码块或执行路径，实现软件性能问题的诊断和软件性能优化。主要工作有 CARAMEL[21]、Toddler[22]、LDoctor[23]、X-ray[24]等。CARAMEL、Toddler、LDoctor 利用静态或动态分析的方法，重点关注软件中循环结构，因为循环结构更有可能到软件出现性能问题[23]。X-ray 采用动态程序插桩的方法检测代码块的执行时间，并比较不同执行路径的开销，从而定位配置导致性能下降点。

针对软件配置错误导致的软件功能异常和软件性能异常进行研究分析，主要工作是提高配置错误诊断过程的自动化程度[25-28]。基于软件测试的方法测试配置不同的值是否会触发软件 bug；静态分析的方法分析软件的源码，以获取配置项

的约束，同时将这种约束和配置项联系起来，避免配置时出错；基于动态分析的方法发现软件配置输入参数与软件功能异常和软件性能异常之间的关系（causality），实现配置错误诊断和修复。

1.2 课题研究内容和创新点

随着软件规模的不断增大，配置引起已经成为导致软件性能下降的主要原因之一。检测软件中的性能瓶颈，提高诊断软件性能问题的效率。本课题针对软件配置相关的性能瓶颈难以检测，维护人员解决性能问题效率低下的问题，本课题通过对大量软件配置项的调研，提出配置相关的性能问题与资源相关的配置项相关，设计并实现了一种基于资源依赖分析的配置相关的软件性能瓶颈检测工具 CBHunter。本课题的主要贡献包括以下几个方面：

1、调研分析了 7 款分布式开源软件的 75 个配置相关的性能问题和 50 个资源相关的配置项，包括 ZooKeeper，Hadoop Common，HDFS，HBase，Cassndra，MapReduce 和 Yarn。通过研究分析证明了本课题提出的配置相关的性能问题与资源相关的配置项有关联的观点。并将配置影响资源使用分为间接影响和直接影响，单配置项影响和多配置项影响。同时，本课题将配置相关的性能问题根据资源类型分为 5 类，并发现最有可能导致软件性能问题的 3 类资源是 CPU，MEM 和 IO；

2、基于调研分析验证的观点，本课题提出一种负载不敏感的资源依赖分析方法，可以有效地降低配置空降维度。将配置影响的语句对软件资源的影响作为该配置项对资源的影响。首先利用程序切片技术获得配置项影响的语句，然后量化配置项影响的语句对软件资源使用的影响，最后累加这些语句的量化值，将该值作为配置项对资源影响的量化值。实验评估证明，本课题提出的方法能够有效检测与资源相关的配置项，并且前 10%的配置项中导致软件性能问题的配置项占比达到 80%。

3、本课题实现了检测配置相关的性能瓶颈的工具 CBHunter，并且本课题验证了 CBHunter 检测性能瓶颈的有效性。本课题将 CBHunter 应用于 6 个软件性能问题，实验结果表明 CBHunter 能够检测到 5 个性能问题的性能瓶颈，诊断配置相关的性能问题能力是 HPROF 的 5.9 倍。并且动态分析的平均额外开销小于 5%，静态分析的平均运行时间为 8 分钟。

1.3 论文结构

本问共有 7 章，介绍每章主要内容：

第 1 章是绪论，介绍课题来源及背景，表明本课题的研究意义。然后提出配置影响软件资源使用的观点，对本课题所采用的方法做简单介绍，总结本课题研究内容、挑战及如何应对，最后介绍文章结构。

第 2 章是本课题的相关研究工作，本课题调研了大量软件性能与配置之间的工作，并根据研究方法分为四类，并逐个简明介绍每种方法。

第 3 章是配置影响资源使用的调研，主要介绍本课题如何调研软件性能问题，得出配置影响资源使用的结论。

第 4 章介绍 CBHunter 的主要框架，包含 CBHunter 的总体结构及具体模块的目标。

第 5 章具体描述 CBHunter 的设计和实现，介绍静态打分排序和动态插桩分析方法。

第 6 章是实验与评估，主要包括对 CBHunter 静态打分排序的评估和 CBHunter 检测性能瓶颈的评估。

第 7 章总结全文，并展望未来工作的研究方向。

第二章　相关文献综述

目前针对配置引起的软件性能问题已有大量的研究工作发表在国际顶级水平学术会议上。主要包括德国魏玛大学的 Norbert Siegmund 团队，德国帕绍大学的 Sven Apel 团队等，下面主要按照研究方向进行介绍。

2.1　性能瓶颈检测

性能瓶颈检测主要关注寻找软件的性能瓶颈，主要思想是向软件源码中插桩，记录软件执行路径或者函数执行时间，通过分析执行路径和执行时间以确定软件的性能瓶颈。

GAProf[9]首次将遗传算法应用于软件性能瓶颈的检测，主要思想是使用遗传算法作为启发式搜索来获得输入参数的组合，以最大化适应度函数，该适应度函数表示使用这些输入值表示应用程序的已执行时间，获得最终种群的软件执行路径和不同方法的执行时间，得到运行时间长的路径（Good Traces）和运行时间短的路径（Bad Trace），并构建函数与时间关系的矩阵，然后利用 ICA（独立成分分析方法）方法剔除运行时间短的函数，即获得软件性能瓶颈。

ProCrawl[17]关注性能瓶颈与理想执行路径的关系，基于 AUT 的实际执行路径生成测试用例，使用函数最小化方法来搜索影响理想执行路径的输入变量的值。ProCrawl 是一种全自动工具，挖掘企业 Web 应用程序显式行为模型以使系统测试和维护更加高效。ProCrawl 以 Web 应用程序的 URL，用户的登录凭据，Web 应用程序所需的各个部分和启动事件作为输入，生成有限状态自动机（FSA），该有限状态机中节点表示 Web 应用程序的抽象单个状态，所有节点按照 ProCrawl 检测到的顺序编号，节点之间状态转换表示由用户更改 Web 状态的操作。ProCrawl 产生的模型也可以作为软件演化过程中软件行为变化的验证。

HOLMES[16]关注可能包含 bug report 的软件部分，并基于路径得分的方式来预测可能失败的路径。HOLMES 研究使用更丰富的程序配置文件（如路径配置文件）对错误隔离的有效性的影响。 HOLMES 对程序仅从插桩获取软件执行路径的详细信息，记录软件运行失败与运行成功的路径，对运行失败的部分路径计算得分并排序，获得导致软件运行失败的路径以知道对错误隔离。并提供了版本自适应的 HOLMES 工具，该工具使用迭代的错误定向分析方法以降低时间和空间开销。HOLMES 使用 SIR 基准套件中的程序和大型的实际应用程序来评估 HOLMES。结果表明，路径配置文件可以通过提供有关发生错误的上下文的更多信息来帮助更准确地隔离错误。此外，错误导向的分析可以有效地隔离低开销的错误，为稀疏随

机抽样提供可扩展且准确的替代方案。但是该方法没有有效的定位到软件运行失败点，可以有效的避免错误发生，不能诊断错误。

AutoAnalyzer[18]关注性能瓶颈是否存在并定位性能瓶颈，使用集群算法鉴别存在性能瓶颈，然后使用搜索算法定位性能瓶颈。AutoAnalyzer 使用源到源转换，自动将检测代码插入到并行程序的源代码中，并将整个程序划分为代码块，每个代码块是带有一个代码入口和一个代码出口且从头到尾执行的代码段。对于 SPMD 样式的并行程序，如果在负责管理例程的主进程中排除代码段，则每个进程或线程应具有类似的行为。同时，如果代码段占用程序运行时间的小部分，改进代码段的性能将对程序的整体性能贡献很小。基于上述发现，该方法关注两种类型的性能瓶颈：导致进程或线程行为不相似的瓶颈，称之为不相似瓶颈，以及导致代码段行为差异的瓶颈。从四个层次结构（应用程序，并行接口，操作系统和硬件）收集代码段的性能数据之后，AutoAnalyzer 提出了一系列创新方法来搜索差异代码段，并揭示其性能优化的根本原因。

StackMine[19]针对现实大规模软件系统中执行路径复杂和巨大的特点，应用统计调试的方法，进行软件性能问题的调试与诊断。该方法提出一种代价模型聚类机制以减少性能分析师的调查工作。首先，在追踪（系统事件流，包括调用栈信息）栈上应用代价模式挖掘算法（例如，子序列挖掘），然后基于一组新的相似性度量对挖掘的代价模式应用聚类，这些度量反映程序执行轨迹的特定领域特征。

FOREPOST[20]在检测软件性能瓶颈上提出了一种新颖的解决方案，利用机器学习的方法，构建软件性能和导致大量计算的输入之间的模型，从而自动的发现软件中的性能瓶颈。FOREPOST 是一个自适应的，反馈导向的学习测试系统。首先，FOREPOST 监控 AUT 执行路径；然后，统计路径信息并通过 ICA（独立成分分析）计算函数权重并推荐给用户，同时利用机器学习技术学习规则，以发现更多性能问题。

2.2 最优配置预测

基于机器学习的方法主要是在不同配置组合获得软件的性能指标，并利用黑盒的机器学习方法构建配置与软件系统性能（响应时间，网络延迟，吞吐量等）模型，描述配置和软件性能之间的关系，并结合不同的采样算法，有效的减小训练样本的开销，并得到更加精确配置组合。基于机器学习的方法主要包括 Norbert Siegmund、Jianmei Guo、Sven Apel 等人[10-14]提出的使用不同的采样算法和机器学习算法。

Norbert Siegmund[10]等人针对软件性能优化的问题，提出将配置项和软件性能构建表达式以描述配置项与软件性能之间的关系。首次将 bool 类型的配置项和

数值类型的配置项结合起来，并针对两种类型的配置项采用不同的采样算法。针对bool 类型配置项采用传统的 Option-Wise Sampling (OW)单个配置项打开采样方法，Negative Option-Wise Sampling (nOW)单个配置项关闭采样方法， Pair-Wise Sampling (PW)一对配置项采样方法；使用等间距循环的皮克布莱曼采样方法对数值类型配置项采样，通过有效减少了配置空间采样的数量。使用梯度线性回归算法构建软件性能与配置项的关系模型，结果表明平均错误率 1%。该方法综合考虑两种配置项的组合，增加了性能预测的准确度并减少了采样的开销。

Jianmei Guo[11]针对小规模采样导致软件性能预测不准确的问题。提出了一种基于随机采样的迭代预测软件系统性能方法，该方法可以有效的减少采样数量，并得到较好的软件性能。该方法利用通过随机采样的方法获得初始样本以预测软件性能，通过两个迭代循环以预测软件性能，第一个迭代基于配置是否具有相同 feature 选择的决定规则，第二个迭代构建基于 CART 的显式性能模型，结合两种迭代产生更加精确、性能更优的配置组合。该模型表示 feature 选择与性能之间的相关性，该模型不需要考虑 feature 之间的关系，从而有效减少了软件配置空间的复杂度，并且该方法可以自动进行迭代的模型构建。

针对不同采样方法对性能预测结果的影响，Atri Sarkar、Jianmei Guo[12]等人研究渐进采样（Progressive Sampling）和投影采样（Projective Sampling）两种采样方法对可配置软件系统的性能预测的影响。该方法改进基于效用的性能模型作为预测模型。针对渐进采样方法，该方法提出两种不同的停止采样策略，分别是基于学习曲线斜率（Gradient-Based）和开销最小化（Cost Minimization）。针对投影采样方法，提出四种学习曲线的投影函数，分别是对数函数（Logarithmic）、平方根函数（Weiss and Tian）、幂次函数（Power Law）和指数函数（Exponential）。同时研究 T-way 采样方法，并提出配置选择频率特征作为启发式的方法，采用两种方法进行初始采样。在抽样成本和预测精度方面，投影抽样优于渐进抽样，但投影抽样依赖于适当的初始样本和恰当的投影函数，在四种常见的投射函数中，指数函数最适合准确和稳健地拟合学习曲线。

通过机器学习方法构建模型，并得到性能最优的配置组合。Vivek Nair[13]等人研究精度较差的模型是否也可以得到最优的配置。该方法提出一个新的观点，利用排序的方法对得到的配置组合进行排序，排名靠前的配置组合就认为该模型可以得到最好的性能配置组合。该方法对比渐进采样方法、投影采样方法和基于排序的方法，分析实际性能最好的配置组合与模型得到的配置组合的排序差，从而说明即使精确度较低的性能预测模型仍然可以使用。

针对不同配置项之间会产生关联的问题，Sergiy S. Kolesnikov[14]等人研究如何自动的判定配置项之间的组合关系，并提出一种黑盒自动检测与性能相关的

feature 交互以提高预测准确性的方法。该方法聚合每一个非功能属性的 feature 的影响以计算配置项与软件性能之间的关系，首先判定配置项之间是否互相影响，然后利用测试的方法衡量对软件性能的影响程度。并提出三种启发式规则，分别是两个配置项（一阶）相互影响 Pair-Wise Interactions (PW)，高阶相互影响 Composition of Higher-Order Interactions (HO)，热点功能 Hot-Spot Features (HS)，应用三种启发式规则可以有效的减少采样的输量以及测量样本的性能时间，进一步降低构建模型的开销。

2.3 性能问题诊断

基于程序分析的方法重点关注系统中与性能相关的此操作语句，通过数据流和控制流等信息流的分析，较为准确的获取影响软件性能的代码块或执行路径，实现软件性能问题的诊断和软件性能优化。主要工作有 CARAMEL[21]、Toddler[22]、LDoctor[23]、X-ray[24]。

CARAMEL[21]针对循环性能问题，循环中每条语句是否产生无效或低效计算。CARAMEL 采用静态分析方法，检测并修复可能被开发人员采用非侵入式修复的性能问题。CARAMEL 检测到的每个性能错误都与循环和条件相关联。当循环执行期间条件变为真时，循环执行的所有剩余计算都将被浪费。开发人员通常会修复此类性能错误，因为这些错误会浪费循环中的计算并具有非侵入式修复：当某些条件动态变为真时，只需突破循环即可。CARAMEL 静态检测给定程序的性能问题，并为开发人员提供每个性能问题的修复代码。CARAMEL 检测每次迭代无结果（Every No-Result）、循环后无结果（Late No-Result）、循环后无用结果（Late Useless-Result）和循环前无用结果（Early Useless-Result）四种无效或低效计算的性能问题。

Toddler[22]关注寻找软件嵌套循环中重复无效或低效的计算。Toddler 是一种动态性能缺陷检测工具，Toddler 检测重复计算，并且在循环迭代中计算非常相似的代码，并报告执行该循环的测试用例。该方法提出并证明以下两种观点，（1）嵌套循环包含许多严重的性能问题（超过 50%），如果在嵌套循环之外执行效率低下的代码，则需要效率非常低下（例如，慢速 I/O）代码，才能对软件整体性能产生影响；（2）无效或低效计算通常反映在循环迭代中的重复和部分相似的内存访问，如果一组指令重复访问类似的内存值，那么这些指令可能会计算出类似的结果。基于以上观点，Toddler 检测嵌套循环的外层循环冗余（Redundancy in Outer Loops）、内层循环冗余（Redundancy in Outer Loops）、外层循环无效（Inefficient Outer Loops）和内层循环无效（Inefficient Inner Loops）四种性能问题。

LDoctor[23]采用静态和动态分析结合的方法，分析循环结构中的性能问题，并首次提出跨迭代次数的解决方法。首先，LDoctor 对无效循环分为两大类：无结果

循环（resultless）和冗余循环（redundancy），又将无结果循环分为无任何结果（0*）、最后一次迭代产生结果（0*1？）、每次迭代可能产生结果（[0|1]*）和每次迭代都产生结果（1*）4 小类；将冗余循环分为跨迭代冗余（Cross-iteration Redundancy）和跨循环冗余（Cross-loop Redundancy）；然后，LDoctor 根据对无效循环分类，利用静态分析和动态分析混合的方法检测循环性能问题；最后，通过随机采样的方法进一步降低 LDoctor 动态分析产生的额外开销。该方法集合 CARAMEL 和 Toddler 两种工具的优势，利用静态和动态分析混合方法，不仅降低开销，而且增加了诊断的精确度，同时针对每一种类型的性能问题均提出修复意见。

X-ray[24]采用动态分析的方法，基于性能摘要的方法，首先为每个事件确定对应的性能开销（例如，网络延迟，代码块的运行时间，内存占比，IO 使用率等），然后基于动态信息流分析找到不同路径下性能差异的关联配置项，并根据相应概率计算单个配置参数导致性能差异的概率，则概率最大的配置项即为可能的错误配置项。同时，

SmartConf[29]采用控制理论的方法，重点关注软件运行期间动态调控软件，使得软件运行期间满足用户的性能要求。首先，SmartConf 需要用户设定性能指标，以及需要开发人员指定哪些配置与性能相关联；然后，SmartConf 修改配置的入口以满足动态调整的要求，初始化用户配置文件（包括与性能相关的配置项，是否是硬指标）、软件配置文件；最后，基于动态控制理论，遵循慢增快减的规则，SmartConf 根据软件系统当前的稳定程度动态调整配置值，同时设立虚拟目标以确保当前性能不会超过用户要求。该方法首次将控制论应用于调整软件配置满足用户性能要求，但是 SmartConf 需要开发人员参与并设定性能相关的配置项，并且需要修改软件源代码。

2.4 配置故障诊断

针对软件配置故障诊断问题，主要基于程序动态和程序静态分析技术的有 SigConf[25]，ConfAid[26]等工具，基于统计学习方法的配置诊断技术，主要包括 CODE[27]，EnCore[28]，以及基于对比和重放技术的诊断方法。这些配置故障诊断技术针对软件配置错误导致的软件异常进行自动化的分析和修复。同时，除此之外，还有 ConfErr[43]、SPEX[44]等利用软件测试技术，测试正确值和不正确值对软件系统的影响，从而评估系统的健壮性。

Sigconf[25]采用一种与以往状态对比的方法。通过从已知的有软件配置错误导致的软件状态中检测当前软件是否与该状态相同，将获得的信息存储在配置错误信息库当中，并根据软件当前状态与信息库中信息匹配，从而判断当前软件系统中是否存在配置错误。SigConf 是一种分析软件配置错误的工具。首先，SigConf 运

行测试用例时，记录软件的执行路径记录以及对应的因果关系；然后，根据路径和因果关系生成签名（signature）；最后，将生成的签名与错误信息库中的签名对比，判断软件系统当前是否存在配置错误。Sinconf 只记录执行路径，而不是记录所有配置项，或者配置指令。所以，Sinconf 在开销上具有显著优势。但是，由于没有记录所有配置项，使得该工具的精确度较差。

ConfAid 诊断配置项值的错误。在工具的准确度上比 SigConf 高很多。ConfAid 分析软件配置项和源代码，通过向字节码中插桩，记录由软件数据流和控制流导致的因果关系。当软件执行时，如果发现某个配置项改变了软件当前的状态，使软件系统产生异常，那么 ConfAid 将该配置项标记于错误配置项。该方法虽然可以获得较高的精确度，但是由于插桩导致的在线开销较大，且记录因果关系，会占用更多的空间。6

CODE 针对 Windows 下的注册表系统发生的事件进行机器学习，从而推断出正确的事件序列。基于学习出的模式，CODE 能够检测出配置事件当中可能隐藏的错误。EnCore 通过大量学习，从配置文件中学习配置项规则，并根据得到的规则，检测软件配置故障。

ConfErr 和 Spex 都是基于测试的方法，以评估软件系统的配置错误发生时的反应能力。ConfErr 将由人导致的错误分为 3 大类，分别是基于技巧、基于规则和基于知识。并依据 3 种类型，模拟人配置软件过程可能产生的错误，暴露软件的错误。但是 ConfErr 没有考虑配置项的约束关系。Spex 则在 ConfErr 的基础之上，引入软件配置项约束关系，将配置项分类并产生对应的约束条件，同时，根据不同类型配置项的不同约束，生成违反约束的配置组合。通过运行测试用例，评估软件对这类错误的反应能力

第三章　配置与资源关系研究

目前已有许多经验性调研工作研究软件系统的性能问题，配置与性能之间的关系，自动调整软件的配置以适应用户设定的性能目标[2，9，29，32]。已有工作表明，59%的软件性能问题与配置相关[2]。前人的工作着重于配置与软件性能之间的关系[14-20]（构建配置与软件性能模型），没有分析配置影响软件的性能原因。为了深入理解软件性能与配置之间的关系，本课题研究 7 个分布式开源软件，从软件配置文件中选取 50 个资源相关的配置项作为研究对象，从 Apache 的 bug report 系统[33]中筛选出 75 个与配置相关的性能问题，通过分析源码和每个性能问题的 bug report 以及 patch 研究配置如何影响软件的性能。本课题发现配置通过影响软件资源的使用进一步影响软件性能。为了理解配置、资源与性能之间的关系，本课题主要关注以下两个问题。

问题一：配置项如何影响软件的性能？软件的性能可能会受到很多因素的影响，本课题通过分析软件中与配置相关的源码以理解配置如何影响软件性能。

问题二：哪几种类型的资源是造成软件性能问题的主要因素？软件性能受到多种资源的影响，该问题主要分析哪几种资源最有可能导致软件性能问题。

3.1　数据来源

本课题研究 7 个分布式开源软件，分别是 ZooKeeper，Hadoop-Common，HDFS，HBASE，MapReduce，Yarn 和 Cassandra。这些分布式软件是高度可配置软件的典型代表，更容易产生软件性能问题，且每一种软件的发展历程均在十年以上[34]。7 个分布式软件在现实的生产生活中被广泛使用，并且具有良好的软件维护系统、bug 报告系统和优秀的软件文档，可以帮助本课题有效的分析软件与性能之间的关系。

表 3.1 软件性能问题数量

软件	描述	性能问题数量
ZooKeeper	分布式应用程序协调服务	9
HDFS	分布式文件系统	27
HBase	分布式开源数据库	9
Hadoop Common	Hadoop 公共内容	7
Cassandra	分布式 NoSQL 数据库系统	12
MapReduce	并行计算框架	9
Yarn	Hadoop 资源管理器	2

3.1.1 筛选性能问题

本课题从 Apache 的 bug report 系统中筛选出 75 个配置相关的性能问题，该系统提供 bug 的信息，修改历史，补丁等信息，对分析问题具有很大的帮助。首先，指定需要调研的软件；然后，通过搜索与性能关键字搜索当前软件中的 bug[29]，例如 performance，slow，speed，regression，degradation 等，同时本课题搜索与配置相关的关键字.xml，config，configuration 等[29]获取相关的 bug；最后，通过阅读每个 bug 的 report，分析与之相关的 comments 和补丁，判断该 bug 是否是与配置相关的性能 bug。本课题按照上述的方法迭代进行，直到找到 75 个与配置相关的性能问题，如表 3.1 所示。

3.1.2 筛选性能相关的配置项

为了进一步分析配置如何影响软件性能，本课题从 7 个软件中通过切片分析等方法筛选出 50 个性能相关的配置项。首先，利用本课题 4.1 节的切片技术对配置项进行切片；然后，对获得的所有语句进行分析与检查，判断这些语句中是否包含与性能相关的语句，例如 socket（），String（）等；最后，结合配置文件中配置项描述及源码分析判断该配置项是否与性能相关。

3.1.3 研究方法

针对问题一，本课题分析 50 个性能相关的配置项和 75 个导致软件性问题的配置项，由于 75 个性能问题中包含 11 个重复配置项，因此配置项分析数量是 114。针对问题二，本课题分析 75 个性能问题是由哪些资源引起的，并将 75 个性能问题根据资源进行分类，已确定哪种资源更有可能导致软件性能下降。

3.2 研究发现

3.2.1 配置影响软件资源使用从而影响性能

通过研究 114 个配置项，本课题发现配置影响软件资源的使用，进一步影响软件性能，并且这些配置项始终与资源相关的函数相关（例如，socket（），String（））。本课题根据配置项数量分为两大类，（1）单个配置影响资源使用；（2）多个配置同时影响资源使用。通过分类本课题发现，单个配置项影响资源使用占比达到 88%，两个或更多配置项影响资源使用占比为 12%。例如，如图 3.1 所示，bufferSize 的值由配置选项 io.file.buffer.size 的值确定，该配置项控制文件 I/O 缓冲区的大小，配置项的值越大，缓冲区越大，那么占用的内存越多。如图 3.2 所示，end = start +

self.getInitLimit（）* self.getTickTime（），其中 getInitLimit 获取配置选项 initLimit 的值，getTickTime 获取配置选项 tickTime 的值，start 是系统时间的初始值。当 electionFinished 设置为 false 时，程序循环等待直到 cur>end。如果这两个配置项的值乘积很大，那么软件运行时间将会增大。这两个配置选项会影响循环次数，并进一步占用更多 CPU 资源。

```
/*hadoop/io/compress/DefaultCodec.java*/
public CompressionOutputStream createOutputStream(OutputStream
out, ..)
    throws IOException {
    return new CompressorStream(...,
    conf.getInt("io.file.buffer.size", 4*1024));
}

/*hadoop/io/compress/CompressorStream.java*/
public CompressorStream(OutputStream out, Compressor
compressor, int bufferSize) {
    super(out);
    if (out==null||compressor==null) {
        throw new NullPointerException();
    } else if (bufferSize<=0) {
        throw new IllegalArgumentException("Illegal
bufferSize");
    }
        this.compressor=compressor;
        buffer = new byte[bufferSize];
}
```

图 3.1 io.file.buffer.size 设置 buffer 大小

```
/*zookeeper/server/quorum/leader.java*/
long start=System.currentTimeMillis();
long cur=start;
long end=start+self.getInitLimit()*self.getTickTime();
while(!electionFinished&&cur<end){
    electingFollowers.wait(end-cur);
    cur=System.currentTimeMillis();
}
```

图 3.2 initLimit 和 tickTime 影响 CPU

```
/*java/org/apache/hadoop/security.java*/
GroupCacheLoader() {
    if (reloadGroupsInBackground) {
        ThreadFactory threadFactory = ...;
        ThreadPoolExecutor parentExecutor =  new
ThreadPoolExecutor(
        reloadGroupsThreadCount,...);
    …;
  }
}
```

图 3.3 配置项间接影响资源使用

根据配置项影响资源的方式，本课题将 114 个配置项分为对资源的直接影响和间接影响。调研发现，73%的配置选项直接影响资源的使用。如图 3.1 所示，该配置项通过直接设置缓冲区大小直接影响内存。27%的配置选项间接影响资源的

使用。例如，在图 3.3 中，reloadGroupsInBackground 是配置项 hadoop.security.groups.cache.background.reload（配置项 A）的入口，reloadGroupsThreadCount 是 groups.cache.background.reload.threads（配置项 B）的入口。配置项 A 通过控制配置项 B 间接影响资源的使用，配置项 B 控制并发后台用户组缓存条目刷新的数量。如果 A 为真且 B 较大时，才会影响更多资源并导致性能问题。

根据以上分析，本课题发现配置项通过影响资源的使用，从而影响软件性能。如果某种类型的资源受某些配置项的高度控制，则更有可能由于配置不当而导致软件性能下降。因此，本课题得出结论，大多数与资源相关的配置项更可能导致软件性能问题。

3.2.2 导致软件性能问题的资源

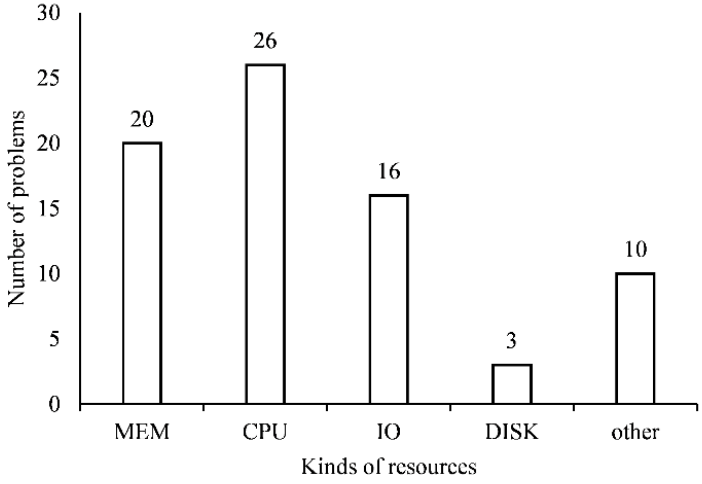

图 3.4 不同资源导致性能问题的数量

为回答问题二，本课题阅读性能问题的 bug report，并分析源代码以研究导致产生性能问题的根本原因。前人工作对配置项所述领域进行分类[2]，本课题根据导致性能问题的不同资源类型，将 75 个性能问题分为 5 类（如图 3.4 所示）：（1）与内存相关的性能问题，（2）与 CPU 相关的性能问题，（3）与 IO 相关的性能问题，（4）与磁盘相关的性能问题，以及（5）其他性能问题。对于与内存相关的性能问题，性能下降大多数由缓冲区或缓存不足引起。与内存相关的性能问题占 27%。例如，MAPREDUCE-6551，如果有许多小文件（甚至有些文件大小小于 1M），这将导致 MapReduce 创建许多 map 任务。一般来说，map 任务会使用默认配置 MRJobConfig #MAP MEMORY MB 来设置其资源容量的内存以处理其数据。即使目标文件很小，但是 map 任务占用高内存资源的问题，导致软件占用过高的内存。对于与 CPU 相关的性能问题，软件性能下降一般是由低效或无效的计算引起的。

这些低效或无效的计算导致 35%的性能问题。例如，Hadoop-14216，解析 XML 文件速度慢引起性能降低，导致 CPU 资源的浪费并增加了软件运行时间。解决该性能问题，可以重用或适当更改 XML 解析器（STAX），从而可以提高 XML 解析速度。对于与 IO 相关的性能问题，软件性能下降一般与读取或写入操作有关。IO 相关的性能问题的比例为 21%。对于与磁盘相关的性能问题，通常是由磁盘大小引起的，仅占所有性能问题的 4%。

根据本课题研究，发现所有性能问题都表现为资源紧缺，而大多数性能问题与 CPU（35%），内存（27%）和 IO（21%）的使用有关。磁盘大小（4%）和其他资源（网络、端口等 13%）导致少量性能问题。

3.3 总结

本课题分析 75 个性能问题以及 50 个与性能相关的配置项，验证本课题提出的配置通过影响软件资源从而影响软件性能观点。同时，根据配置项影响资源使用的数量分为单个配置项影响软件资源和两个及多个配置项影响软件子资源；根据配置项影响软件资源的方式分为配置项直接影响软件性能和配置项间接影响软件性能。本课题将性能问题分为 CPU 相关，内存相关，I/O 相关，硬盘相关及其他五种，并通过每种类型性能问题的数量，发现最有可能导致软件性能下降的 CPU、内存和 I/O。

第四章　CBHunter 体系结构

本课题第三章发现，软件性能瓶颈受到某些类型资源（内存、CPU、IO 等）的限制，并且性能问题很可能与资源相关的配置项相关联。基于上述发现，本课题设计并实现了 CBHunter，一种基于资源依赖性分析检测与配置相关的软件性能瓶颈自动化工具。

CBHunter 的目标是检测由配置导致的软件性能瓶颈，并将导致软件性能下降的配置项报告给开发人员，以提高开发人员诊断软件性能问题的效率。CBHunter 包括 3 个主要阶段：（1）资源竞争检测阶段，分析由哪种资源导致了软件性能降低；（2）资源依赖分析阶段，利用切片技术、in-house 测试方法，挖掘与资源密切相关的配置项，CBHunter 量化配置项影响代码对资源影响的程度，作为该配置项对得分，根据得分对配置项排序，选择与资源最相关的配置项作为可疑配置项（很有可能导致软件性问题的配置项）；（3）性能瓶颈检测阶段，CBHunter 插桩代码段以监视可疑配置项影响的代码块不同资源的变化量，根据变化量向用户报告软件性能瓶颈以及对应配置项。

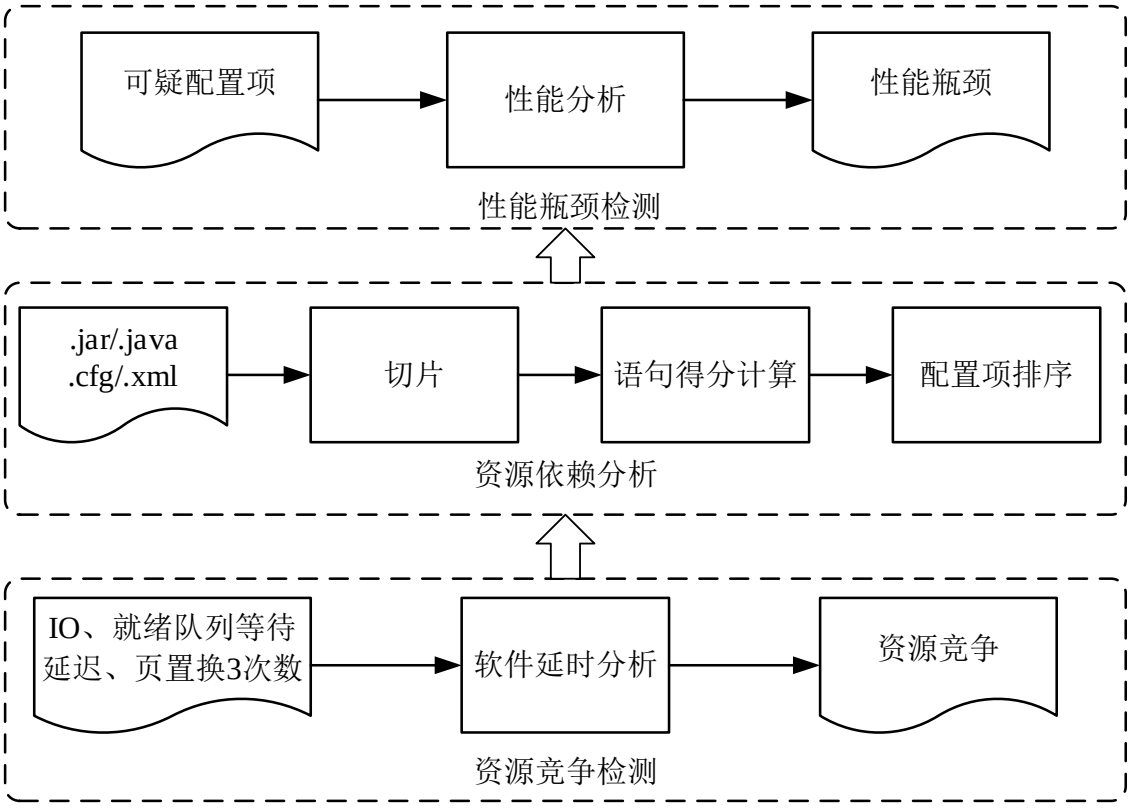

图 4.1 CBHunter 架构

4.1 资源竞争检测

资源竞争检测阶段，CBHunter 主要检测是哪种资源导致了软件性能下降。通过获得不同资源请求延迟和资源的总使用量是否大设定的阈值，判断哪种资源发生竞争，导致软件性能下降。资源竞争检测可以有效的检测哪种资源发生了竞争，提高开发人员诊断性能问题的效率，并且为下一步资源依赖分析提供输入。

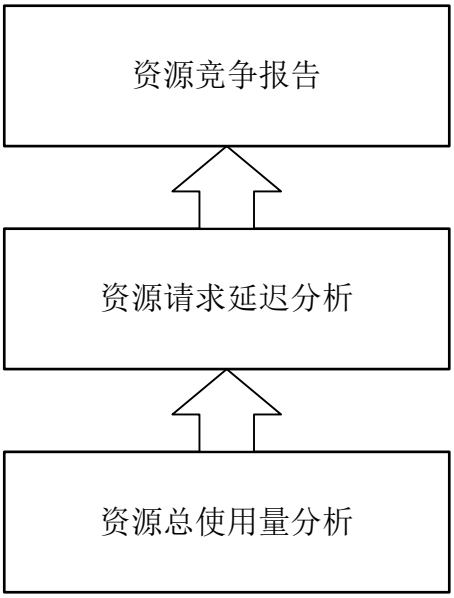

图 4.2 资源竞争检测

如图 4.2 所示，资源竞争检测首先检测当前系统中总资源的使用量。例如，利用 top 命令监控 CPU 的资源使用情况，当该资源使用总量大于预设值时，即进行下一步资源延迟分析。然后进行资源延迟分析，本课题检测 CPU，内存，IO，网络四种资源的延迟，分别以就绪队列等待延迟、内存页置换、IO 延迟三种资源延迟以判断软件系统哪种资源产生了竞争。最后，如果检测到当前系统产生了资源竞争，那么则将该资源报告给开发者。第 5.1 节详细描述 CBHunter 如何检测资源竞争。

4.2 资源依赖分析

资源依赖分析阶段，CBHunter 以产生竞争的资源为输入，对所有配置项打分并排序，以挖掘与该资源最相关的配置项（可疑配置项），并将这些配置项作为动态插桩分析的输入。资源依赖分析可以有效的降低配置空间维度，尤其针对配置项数量庞大的软件，分析每个配置项对性资源的影响对于开发人员来说是不可实现的。减少需要分析的配置项数量对于软件系统维护、软件配置故障诊断和软件配置

相关的性能问题诊断具有重要意义。

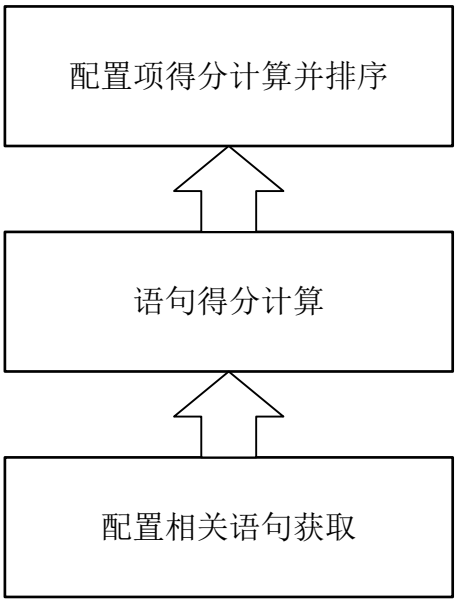

图 4.3 资源依赖分析

图 4.3 展示资源依赖分析阶段的结构图。CBHunter 将配置项相关的语句对软件资源的影响作为该配置项对软件资源的影响。因此，CBHunter 首先需要获取配置项相关的语句，CBHunter 以软件源码和配置文件为输入，基于函数名分析以确定配置项在软件源码中的读入函数，再利用静态切片工具 WALA[37]，获取配置项影响的代码块。然后，CBHunter 需要计算每一条语句的得分。根据预先定义的不同类型语句的得分，通过 in-house 测试的方法确定语句的执行次数，计算每条语句的得分。最后，CBHunter 根据每条语句的得分以累加的方式计算当前配置项的得分，把该得分作为配置项对软件资源影响的量化值，并对所有配置项排序，以获取可疑配置项，可疑配置项作为动态插桩分析的输入。第 5.2 节详细描述 CBHunter 资源依赖分析的详细过程。

4.3 性能瓶颈检测

性能瓶颈检测阶段，CBHunter 利用插桩方法检测软件检能瓶颈，并将软件瓶颈和对应的配置项报告给用户。检测软件性能瓶颈对软件维护人员诊断性能问题至关重要，能够极大的提高维护人员的工作效率，加快诊断性能问题的速度，提升用户的体验。

如图 4.4 所示，CBHunter 在性能瓶颈检测阶段以资源依赖分析的结果为输入；然后，例中静态软件插桩的方法向软件中插桩，以获取函数执行对资源影响的大小；再运行软件测试程序，获得函数执行前后的不同资源差值（例如，CPU 对应函数

执行时间，内存对应内存变化差值）等；最后，根据差值大小对所有得到结果的函数排序，并将排名靠前的作为性能瓶颈和对应配置项报告给用户。

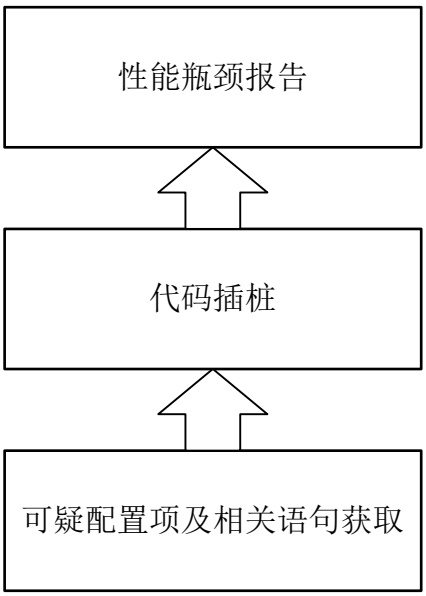

图 4.4 性能瓶颈检测

第五章　设计与实现

本节详细介绍了 CBHunter 的设计和实现。本课题目标是设计一个自动检测与配置相关的性能瓶颈工具，并将性能瓶颈和对应的配置项报告给用户。

5.1　资源竞争检测

本课题首先需要检测是由哪种资源竞争导致的软件性能性能下降。本课题借鉴前人[35]工作利用请求延时检测 CPU、MEM、IO、NET 是否产生了资源竞争。

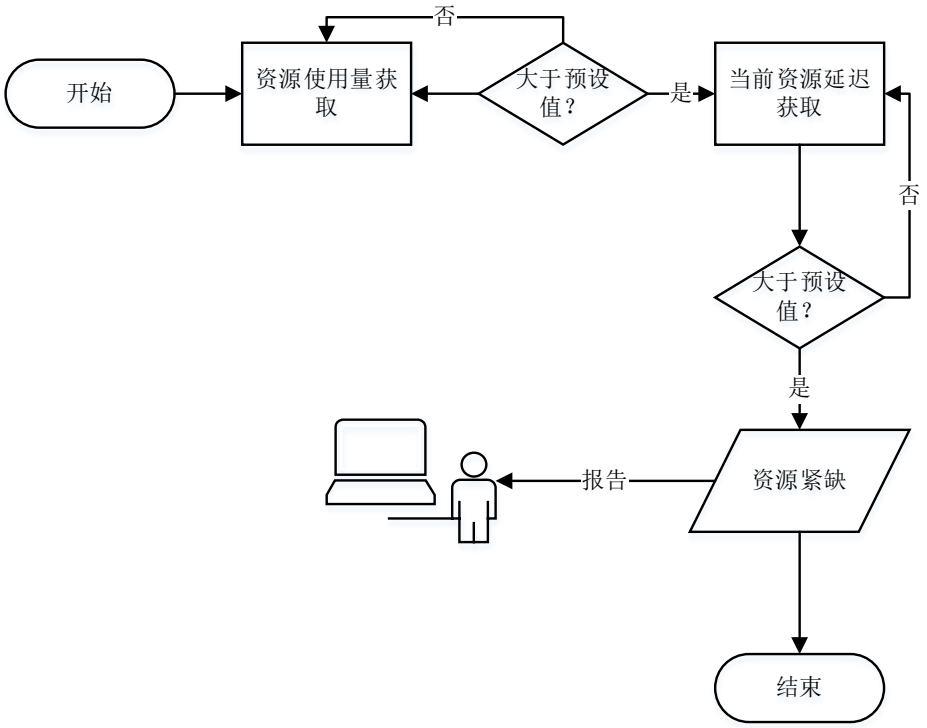

图 5.1 资源竞争检测算法

图 5.1 展示资源竞争检测的具体流程。本课题根据软件资源请求延时和资源总使用量是否超过临界值（计算不同负载下的软件响应延时的突变点，利用突变点处的资源请求延时作为判断资源竞争的条件），分析是哪种资源导致的软件性能降低。首先，确定当前系统总资源使用率是否大于预设值，将大于阈值的资源作为备选资源；然后，获得资源请求延时，如果资源请求延迟大于预设值，那么则认为该资源产生了竞争。例如，当前运行环境总内存使用率超过预设值，而此时软件系统中频繁进行页置换，就可以推断出当前运行环境中出现内存资源竞争。

5.1.1 资源总量分析

本课题使用资源检测工具 top、netstat、vmstat 检测当前资源的使用程度，这些工具可以实时的检测当前资源的使用程度。如图 5.2 所示，通过 top 命令，本课题可以获得内存和 CPU 的使用占比，以及每个进程的占比，但是，top 不能检测 NET 和 IO 的资源使用程度。因此，本课题使用 netstat 工具检测 NET 资源使用情况，使用 vmStat 检测磁盘的使用情况。netstat 是一款命令行工具，可用于列出系统上所有的网络套接字连接情况，包括 tcp，udp 以及 unix 套接字，另外它还能列出处于监听状态（即等待接入请求）的套接字。图 5.3 列出当前运行环境中的 TCP 连接，可以通过 Loacl Adress 查看当前网络资源占用较高的进程。如图 5.4 所示，vmStat 命令报告进程数、虚拟内存使用和空闲内存、磁盘与内存的交换页数、IO 发送的块数、系统终端和以及 CPU 活动的统计信息。

以上工具可疑实时检测软件系统当前资源的使用程度，帮助本课题判断当前资源使用总量是否大于预设值。

```
top - 19:03:09 up  9:29,  2 users,  load average: 0.15, 0.07, 0.06
Tasks: 320 total,   2 running, 318 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.3 us,  0.7 sy,  0.0 ni, 97.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   4041000 total,  2235512 used,  1805488 free,    83196 buffers
KiB Swap:  1046524 total,    12508 used,  1034016 free.  1491976 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1181 root      20   0  313284  51956  12668 S   3.3  1.3   7:31.58 Xorg
33663 liujun    20   0  679384  25444  14408 S   1.7  0.6   0:00.75 gnome-term+
 2078 liujun    20   0  583224  24296  12016 S   0.3  0.6   0:27.43 unity-pane+
 2247 liujun    20   0 1080148  92628  41572 S   0.3  2.3   3:48.48 compiz
```

图 5.2 CPU 和内存资源检测

```
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp6       0      0 ip6-localhost:ipp       ip6-localhost:37703     ESTABLISHED
tcp6       0      0 ip6-localhost:37702     ip6-localhost:ipp       TIME_WAIT
tcp6       0      0 ip6-localhost:37703     ip6-localhost:ipp       ESTABLISHED
```

图 5.3 NET 资源检测

```
 procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
  r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
  2  0      0 3048944  96384 374432    0    0   556    40  136  327  3  2 94  0  0
```

图 5.4 IO 资源检测

5.1.2 资源请求延时分析

本课题使用软件资源请求延时判断是否发生竞争资源，当软件需要但没有立即得到需要的资源时，则发生了资源的竞争。本课题将软件资源请求延时分为 3 类，分别是：(1)CPU 资源-就绪队列等待延时，(2)NET 和 IO 资源-IO 请求延时，(3)MEM 资源-页置换 3 次数。操作系统将进程的状态分为：创建状态、运行状态、就绪状态、阻塞状态和终止状态，当进程处于就绪状态时，如果当前 CPU 资源空闲，那么该进程就会被调度执行，而如果当前 CPU 资源紧缺，那么该进程就会一直处于就绪状态，直到该进程被调度执行才会变更为运行状态。所以，进程状态从就绪态变为运行态间隔的时间，称为就绪队列等待延时。IO 请求延时是指进程从发起 IO 请求到请求完成所间隔的时间。页置换 3 次数是指内存与硬盘之间置换三块的次数。

以上影响软件延时因素通常和资源的关系如表 5.1 所示。这些因素可以从 Linux 内核获取，并且本课题将软件响应时间突变点时的资源请求延迟作为判断是否产生资源冲突的阈值。

表 5.1 资源与影响软件延时因素

系统资源	延时因素
CPU	就绪队列等待延时
MEM	页置换次数
IO	IO 请求延时
NET	IO 请求延时

5.2 资源依赖分析

5.1 节检测软件性能下降是由哪种资源导致的，但是调整所有配置项对提高软件性能来说是不可实现的。因此，本课题挖掘与软件资源相关的配置项，根据配置项对资源使用的影响程度对配置项排序，并选择与资源最相关的配置项作为产生性能下降的根本原因。为实现该目标，本课题量化被配置项影响的代码块对资源的影响，并将该量化值作为配置项对资源的影响程度。

本课题将资源依赖分析分为三个阶段，如图 5.5 所示。首先，本课题利用语句间静态切片来获取配置项影响的语句；其次，基于 in-house 测试的方法计算语句的得分；最后，根据配置项对软件性能的影响程度对配置选项排序，并将排名靠前的配置项作为可疑配置项（即更加可能导致软件性能降低的配置项）。in-house 测试方法不依赖于实际工作负载，静态切片也对软件工作负载不敏感，这些保证配置项排序对实际工作负载不敏感，从而可以不必获取真实的工作负载，分析软件中与资

源相关的配置项。

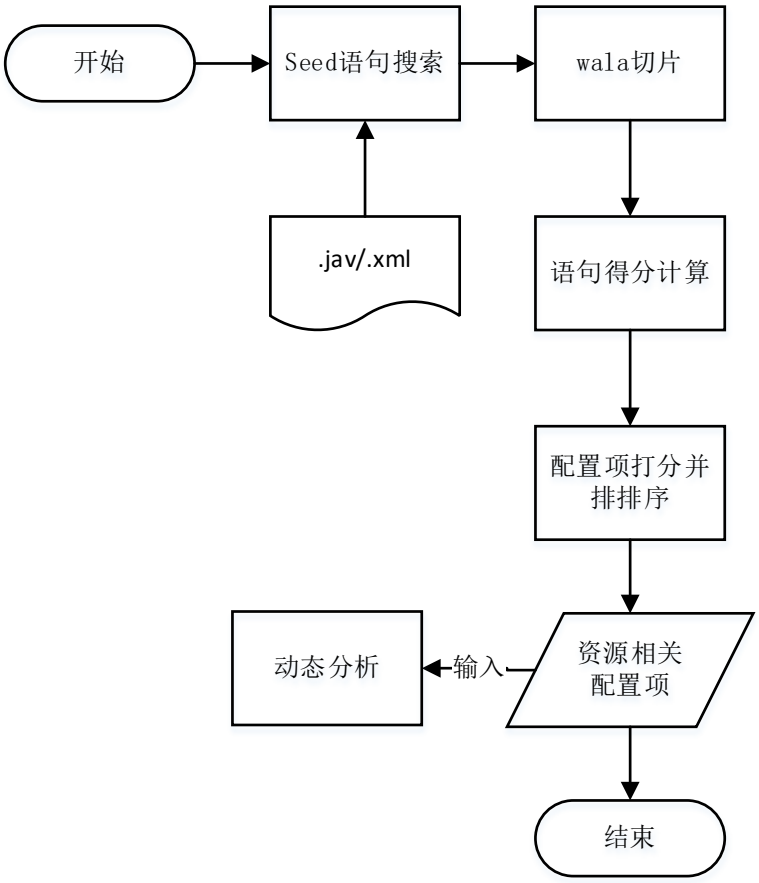

图 5.5 资源依赖分析流程

5.2.1 配置项切片分析

为量化配置项对资源的影响程度，本课题首先需要获取配置项所影响的代码块。本课题调研已有的应用于 Java 语言的软件切片技术 SOOT[36]和 WALA[37]，综合两种工具使用的难易程度，结果的准确性，使用广泛程度等因素，选择 WALA 工具作为切片分析工作。

WALA 是 IBM 公司于 2006 年推出的开源工具，该工具支持 Java 和 Javascript 语言，支持过程件数据流分析，基于上下文敏感的切片，指针分析与调用图构造等，为开发人员提供针对 Java 程序强大的工具。目前 WALA 仍在不断更新，并且引入瘦切片方法等新特性，瘦切片可以有效的减少切片分析的语句，同时保留必要的信息。本课题使用 WALA 的 1.3.8 版本，并利用其提供的上下文敏感正向切片方法对配置项切片以获取配置项影响的代码块。

但是该工具需要指定 seed 语句（即配置项在软件中读入语句）作为切片的起始，因此本课题需要首先分析所研究的软件如何读取配置项。本课题分析配置项在

源码中的读入及使用方法，发现这些软件通过专门的类读取配置文件（ZooKeeper 中的 SeverConfi.java 和 QuorumPeerConfig.java）。90％的配置选项具有 get 和 set 方法，并且配置项 get 方法遵循良好模式。

表 5.2 配置读入函数 3 种模式

配置读入模式	实例
get + key	getinitLimit()
get + types（key，…） types: int, double, float, long, lay, boolean, enum, time, pattern, range, string, password, socket, instance, file, mode	conf.getInt(dfsthroughput.buffer.size, …)
get + （Property，Props，Value，null）	getProperty（zookeeper.preAllocSize）

```
/*zookeeper/server/quorum/Leader.java*/
public void waitForNewLeaderAck(...){
    …;
    long start = System.currentTimeMillis();
    long cur = start;
    long end = start + self.getInitLimit() *
self.getTickTime();
    while (!quorumFormed && cur < end) {
        newLeaderProposal.qvAcksetPairs.wait(end -
cur);
        cur = System.currentTimeMillis();
    }
    …;
}
```

图 5.6 配置项读入函数 get+配置项键

本课题总结配置项读入函数获取配置项值的 3 种模式：（1）get + 配置项键，（2）get + 配置项类型（配置项键），（3）get + （Property，Props，Value，null）（配置项键）。同时，本课题根据分析软件源码及配置项值总结了 16 种配置项类型，如表 5.6 所示。如图 5.6 所示，函数 getInitLimit（）返回配置项 initLimit 的值，函数 getTickTime（）返回配置项 tickTime 的值，这两个配置项的读入函数遵循模式 1；如图 5.7 所示，conf.getInt(dfsthroughput.buffer.size, …) 函数返回 dfsthroughput.buffer.size 的值，该读入函数遵循模式 2，dfsthroughput.buffer.size 的值为 4*1024，为数值类型，即 get + 数值类型（Int）（dfsthroughput.buffer.size，…）；

如图 5.8 所示，函数 getProperty（zookeeper.preAllocSize）返回配置项 preAllocSize 的值，该读入函数遵循模式 3，即 get+Property（preAllocSize）。

```
/*org\apache\hadoop\hdfs\BenchmarkThroughput.java*/
public int run(String[] args) {
    ...;
    BUFFER_SIZE = conf.getInt(
            "dfsthroughput.buffer.size",
            4 * 1024);
    ...;
}

private Path writeLocalFile {
    ...;
    byte[] data = new byte[BUFFER_SIZE];
    ...;

}
```

图 5.7 配置项读入函数 get+配置项类型（配置项键）

通过总结以上 3 种模式，本课题即可通过分析函数名和函数的参数，通过字符串匹配的方式获得配置项的读入函数。利用 WALA 构建的函数调用图获取读入函数的调用函数，构建读入函数和调用函数对（<Caller，Call>），并且，将该对转化成 WALA 所需 seed 语句即可。但是并不是所有的读入函数可以被准确识别，因此，本课题保留了人工添加配置项入口的接口。

```
/*zookeeper\server\persistence\FileTxnLog.java*/
String size =
System.getProperty("zookeeper.preAllocSize");

public static long padLogFile(FileOutputStream
f,long currentSize,
        long preAllocSize) throws IOException{
    long position = f.getChannel().position();
    if (position + 4096 >= currentSize) {
        currentSize = currentSize + preAllocSize;
        fill.position(0);
        f.getChannel().write(fill, currentSize-
fill.remaining());
    }
    return currentSize;
}
```

图 5.8 配置项读入函数 get +（Property，Props，Value，null）

根据以上基于函数名的分析，即可获得 WALA 切片的 seed 语句，然后利用 WALA 工具的正向切片技术方法对该语句切片，即可获得配置项影响的语句。WALA 工具将所有.jar 包分为原始包（Primordial）、应用包（Application）和扩展包（Extension）三类，其中应用包是目标软件（即需要分析的软件）。该切片方法可以获得所有包中与 seed 语句数据依赖和控制依赖的语句，大大增加了计算配置项得分的复杂度。因此，为了减少其他包中语句导致的配置项得分不准确的影响，本课题仅计算切片结果中在应用包中的语句。同时为了降低计算开销，加快计算速度，本课题将WALA的切片方法的数据依赖和控制依赖设置为NO_EXCEPTIONS，并且采用上下文敏感正向切片的方法获取配置项影响的语句，具体配置如表 5.3 所示。

表 5.3 WALA 配置

配置	值
ReflectionOptions	NONE
CallGraph	0-1
切片数据依赖	NO_EXCEPTIONS
切片控制依赖	NO_EXCEPTIONS
切片方法	上下文敏感的正向切片

5.2.2 配置项打分

本课题利用 WALA 对配置项切片，获取配置项影响的语句。本课题需要量化配置项影响的语句对软件资源的影响，并将该影响量化值作为配置项对资源影响的得分。为了量化配置项对资源的影响，本课题累加该配置项影响的语句作为该配置项得分。因此，本课题需要计算每条语句的得分，不同的语句会影响不同资源的使用；例如，malloc 影响内存使用，socket 影响网络使用，等等。但是由于配置项切片的结果涉及到成百上千条语句，预定义每条语句的得分不可实现。因此，本课题直接利用 WALA 将配置项影响的语句转换为指令，以统一度量单位，计算每条指令的得分再进行累加即可，避免了由于大量函数语句带来预定义的困难。

根据指令的得分，本课题通过指令得分累加的方法，可以对所有配置项打分排序。首先，利用 WALA 对得到的语句转化为指令，得到配置影响语句的指令集合；然后，判断指令集合中的每条语句是否是调用指令，如果是调用指令，那么则将该函数增加到指令集合中；再计算每条指令的得分，将在第 5.2.3 节中详细介绍如何计算每条指令的得分；最后，累加每条指令的得分以获得配置项的得分；具体流程如图 5.9 所示。

图 5.10 展示配置项影响的代码块计算得分示例。该代码块中包含整数声明 int

sum=0，for（）循环，if（）条件判断，输出及函数调用 addAll()，并且代码块所有语句转换为 25 条指令（利用 WALA 转换得到的指令数），然后根据每一条指令的基本得分计算该指令的得分（第 5.3 节中详细介绍），再通过累加的方法获得该代码块的得分。

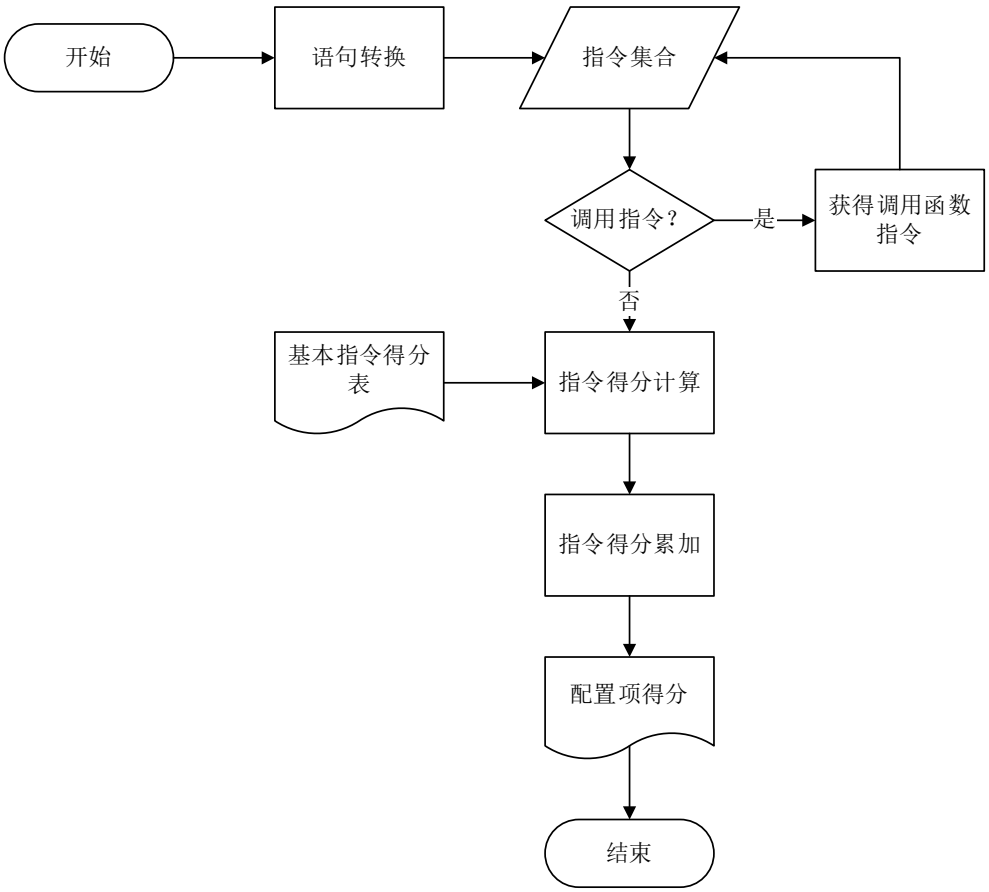

图 5.9 配置得分计算流程图

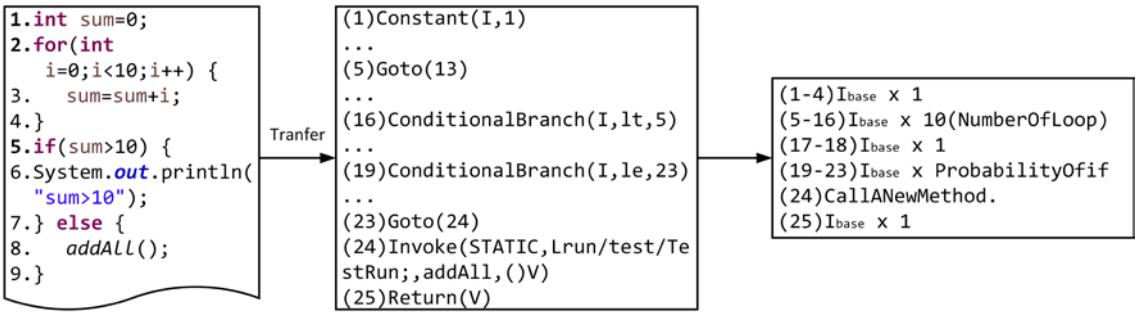

图 5.10 配置项影响代码块打分示例

本课题根据公式（1）对每个配置选项进行打分，其中 Score 是配置项的得分，I_m 是配置项影响的指令第 m 条指令的得分。然后根据配置项得分，再对所有配置项进行排名，并将排名靠前的配置项作为可疑配置项（即更加可能导致软件性能降

低的配置项）。

$$(1)\ \text{Score} = \sum_{m=1}^{n} I_m$$

5.2.3 指令得分计算

由于预定义每条语句得分不可实现，本课题将语句转换为汇编级的指令，以降低计算指令得分的难度。本节将详细介绍如何计算有语句转换的每一条指令。

1）基本指令得分预定义

首先，本课题需要量化每条类型的指令对软件资源的影响程度，即量化这些指令执行一次对软件资源使用的影响。本课题将这种执行一次的指令成为基本指令。通过阅读不同处理器的 datasheet 文档[38，39]，同时根据 Java 不同指令对栈的影响[40]，量化每条不同指令执行一次对软件资源的影响程度，即指令得分。表 5.4 展示基本指令得分的结果，这些指令是对实际运行指令的总结得到的不同类型指令的分数。

表 5.4 基本指令得分

指令类型	CPU	MEM	IO	NET
StoreInstruction	2	1	1	1
LoadInstruction	2	1	1	1
ArrayLoadInstruction	2	10	10	1
ArrayStoreInstruction	2	10	10	1
LoadIndirectInstruction	1	1	1	1
StoreIndirectInstruction	1	1	1	1
ConversionInstruction	3	1	1	1
NewInstruction	4	2	1	1
GetInstruction	2	1	1	1
SwapInstruction	5	5	1	1
PopInstruction	2	5	1	1
DupInstruction	2	5	1	1
GotoInstruction	5	1	1	1
ThrowInstruction	10	1	1	10
DIVInstruction	10	2	1	1
MULInstruction	5	2	1	1
REMInstruction	15	2	1	1
ADDInstruction	5	2	1	1
SUBInstruction	5	2	1	1

ANDInstruction	3	2	1	1
ORInstruction	3	2	1	1
XORInstruction	3	2	1	1
MonitorInstruction	2	2	1	20
SHLInstruction	1	2	1	1
SHRInstruction	1	2	1	1
USHRInstruction	1	2	1	1
NEGInstruction	1	2	1	1
CMPInstruction	3	2	1	1
CMPLInstruction	3	2	1	1
CMPGInstruction	3	2	1	1
EQInstruction	5	2	1	1
NEInstruction	5	2	1	1
LTInstruction	5	2	1	1
GEInstruction	5	2	1	1
GTInstruction	5	2	1	1
LEInstruction	5	2	1	1
SwitchInstruction	3	2	1	1
TypeTestInstruction	10	2	1	1
ArrayLengthInstruction	5	2	1	1
ThrowInstruction	5	2	1	1
ConstantInstruction	5	2	1	1
ReturnInstruction	5	1	1	1
InstanceofInstruction	5	2	1	1
InvokeInstruction	x	x	x	x

如表 5.4 所示，共有 44 种不同类型的指令，每种指令对资源的量化影响如表中所示。本课题用 I_{base} 表示基本指令的得分。StroeInstruction 对 CPU 的影响得分为 2，该指令对内存影响的得分为 2，对 IO 和 NET 的影响为 1。DIV 指令比 ADD 指令执行速度慢，DIV 指令对 CPU 的影响远大于 ADD 指令的影响。表 5.4 中的量化影响是根据 datasheet 文档以及对每条指令作用的总结得到的。虽然采用这样的方法不可能获得非常准确的值，但是本课题提出的静态配置项打分排序的方法只是估算配置项对资源的影响，即使不精确对本课题的方法仍然适用。如果可以获取更加精确的量化影响值，那么大幅度提升资源依赖分析的准确度。

2）循环与分支指令得分计算

上一节介绍如何计算配置项的得分，通过累加指令的得分得到配置项的得分。本节介绍详细介绍如何计算每条指令的得分。

对于基本指令，利用每种基本指令的得分，通过累加的方法即可。但是在真实的软件中，不可能所有指令都执行一次。例如图 5.10 中，位于循环体中的语句执行次数为 10，如果仍使用累加的方法，那么会降低资源依赖分析的精确度。因此，本课题需要准确的得出每条指令的执行次数，才能进一步增加资源依赖分析的准确度。一般情况下，软件通过递归和循环的方式多次执行某个函数或某些指令，并且处在分支中的指令执行次数可能小于 1，递归调用可以通过 Invoke 指令转换为基本指令，并且记录该函数的执行次数即可。

所以本课题必须考虑处于以下两种控制结构的指令：（1）循环（for、while）和（2）分支（if、switch）。两种控制结构的执行次数对配置项的得分有显着影响，如图 5.10 所示，语句 sum = sum + i 位于 for 循环中，如果只计算一次，那么该代码块的得分会减少 9 次加法带来的影响，又比如调用函数 addAll()，如果不考虑函数执行概率，那么该代码块的得分会被增加或减少。

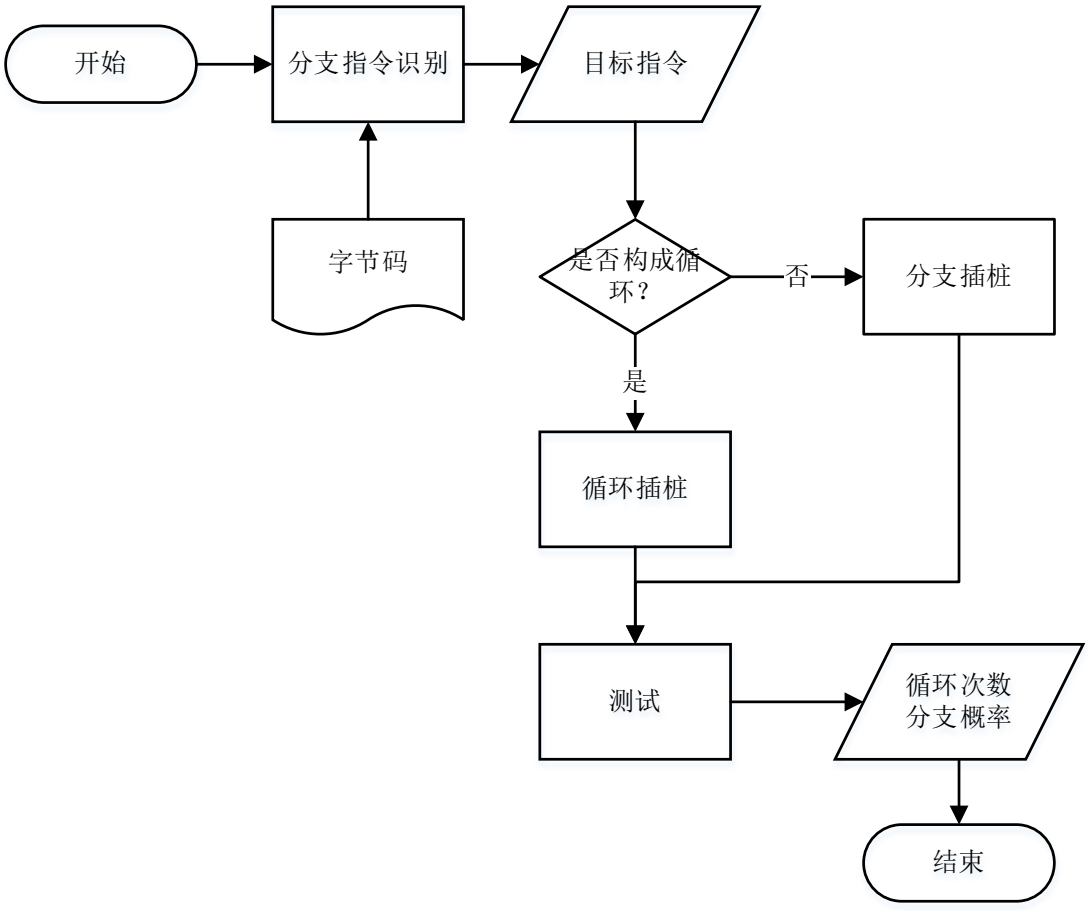

图 5.11 循环次数及分支概率计算流程图

为了获得相对准确的分数，本课题提出一种基于 in-house 测试的方法以计算循环的执行次数和分支的执行概率。如图 5.11 所示，首先，本课题分析 java 程序的字节码，通过识别不同指令以判断循环结构和分支结构。然后，利用静态插桩的

方法向字节码中插入计数器统计循环执行次数，向分支结构中插入标记，记录分支执行路径；最后，根据得到的结果，计算每个循环执行次数的平均值，计算每个分支的概率。特别的，如果在软件运行期间，某个循环或分支从没有被执行，那么则认为该循环次数为 0，分支概率为 0。

表 5.5 循环结构指令识别方式

序号	类型	示例
1	跳转→分支	Goto() … ConditionalBranch()
2	分支→跳转	ConditionalBranch() … Goto()

针对循环结构，首先识别字节码中的循环结构。主要考虑 while 和 for 两种结构，转换为指令通过分支语句和无条件跳转指令实现。本课题总结了 2 种识别循环结构的指令方式，如表 5.5 所示。如图 5.12 所示，共有 25 条指令，其中循环执行的指令是第 5-15 条指令，可以通过识别第 4 条指令 Goto 和第 15 条指令本课题 ConditionalBranch 以识别循环结构。然后，在每个循环中插入一个计数器 Counter（类名+函数名+ConditionalBranch 指令数，循环前声明该计数器，循环每执行一次，Counter++，循环结束输出循环执行次数）用于记录循环执行的次数，如果包含嵌套循环，则对每一层循环均插入计数器。再运行基准测试以尽可能多的遍历所有路径，因为如果测试程序覆盖更多的路径，那么就会得出每一条路径上循环的执行次数；相反，如果部分路径没有被覆盖，那么这些路径上的循环结构执行次数就为 0，降低配置项打分的准确度。根据输出结果，计算每个循环的平均值，并将该循环的平均值作为循环的基础值。同时，为了减少插桩的数量，本课题通过静态分析方法获取部分循环执行的次数。图 5.12 中所示，第 15 条分支指令 ConditionalBranch 是 lt 类型，第 14 条指令是常数指令，13 条指令时载入指令，通过简单的数据流分析，可以得到该循环执行次数为 10。利用该方法可以快速得到较循环次数为常数的循环结构。

针对分支语句，计算每条分支的执行概率。本课题重点关注以下 3 种结构：（1）if else，（2）switch case，（3）循环结构。循环结构已经通过插桩的方式对循环执行次数进行计数，如果循环体执行次数为 0，那么该循环的概率为 0；如果循环体执行次数大于 0，那么该循环的执行概率为 1。针对 if else 和 switch case 结构，本课题仍然采用插桩的方式，向字节码中插入不同标记记录分支执行情况。针对 if else 结构，首先需要识别字节码中的分支跳转指令。如图 5.12 所示第 18 条指令 ConditionalBranch 是 if 的跳转指令，为区分与循环跳转指令的区别。本课题进

一步分析发现，跳转指令的目标指令不是 Goto 指令，且目标指令大于跳转指令。然后，利用静态插桩方法，向字节码中插入标记（类名+函数名+指令行数+ConditionalBranch/Goto 指令数），记录软件不同执行路径。最后，根据输出结果，计算每个分支语句的执行概率。针对 switch case 结构，本课题采用同样的方法项字节码中插入标记（类名+函数名+switch 指令数+Goto 指令数）以计算每个 case 分支概率。图 5.13 展示针对循环和分支结构的插桩位置，其中红色表示循环的计数器 Counter，蓝色 Branch 表示分支标记。

```
0、Constant(I,1)
1、LocalStore(I,0)
2、Constant(I,0)
3、LocalStore(I,1)
4、Goto(13)
5、LocalLoad(I,0)
6、LocalLoad(I,1)
7、BinaryOp(I,add)
8、LocalStore(I,0)
9、LocalLoad(I,1)
10、Constant(I,1)
11、BinaryOp(I,add)
12、LocalStore(I,1)
13、LocalLoad(I,1)
14、Constant(I,10)
15、ConditionalBranch(I,lt,5)
16、LocalLoad(I,0)
17、Constant(I,10)
18、ConditionalBranch(I,le,23)
19、Get(Ljava/io/PrintStream;,STATIC,Ljava/lang/System;,out)
20、Constant(Ljava/lang/String;,"sum>10")
21、Invoke(VIRTUAL,Ljava/io/PrintStream;,println,…)
22、Goto(24)
23、Invoke(STATIC,Lrun/test/TestRun;,addAll,()V
24、Return(V)
```

图 5.12 指令段示例

一旦获得了循环的执行次数和分支的执行概率，就可以计算位于循环结构和分支结构中的指令得分。按照公式（2）中计算每条指令的得分。其中，I_{base}表示基本指令的分数；B_i表示第 i 层分支的执行概率，$\prod_{i=1}^{n} B_i$表示 n 层分支概率的乘积，即处在最内层分支中的指令的执行概率；L_j表示第 j 层循环结构的循环执行次数，

$\prod_{j=1}^{m} L_j$表示 m 层循环执行次数的乘积，即处在最内层循环结构中的指令的执行次数。

$$(2) I_{score} = I_{base} \times \prod_{i=0}^{n} B_i \times \prod_{j=0}^{m} L_j$$

```
…
Counter
4、Goto(13)
Counter++
…
15、ConditionalBranch(I,lt,5)
CounterOut
…
18、ConditionalBranch(I,le,23)
Branch
…
22、Goto(24)
Branch
…
```

图 5.13 插桩后指令段

结合公式（1）和公式（2），首先计算每条指令分数，就可以计算配置项的得分，得到配置项对软件性能的影响程度量化值。将图 5.10 和图 5.12 结合，演示如何计算代码块的分数。首先，分析指令是否处在循环或分支结构中，计算出指令所在的循环和分支结构的层数。根据先前的讨论，可以得出第 5 到 15 条指令处在循环之中不在 if 分支之中，且循环次数为 10，所以 n=1，B_1=1，m=0，L_0=0，根据图二公式可以得出，第 5 到 15 条指令的得分为 I_{base}×10，其中 I_{base} 表示该类型的基本指令对资源的影响程度。并且，第 18 到 21 条指令在 if 分支中，第 23 条指令在 else 中，同时，第 18 条分支跳转指令判断 sum 值是否大于 10，如果大于 10，则执行第 18 到 22 条指令，否则直接跳转到第 23 条指令。运行时，sum 值大于 10，那么则执行第 18 到 22 条指令。针对第 18 到 22 条指令，n=0，B_0=0，m=1，L_0=1，所以第 18 到 22 条指令的得分为 I_{base}；针对第 23 条指令，n=0，B_0=0，m=0，L_0=0，所以第 23 条指令得分为 0；其余指令执行一次，故得分为 I_{base}。再根据公式一，累加不同类型指令的得分，计算该代码块的值。

5.2 性能瓶颈检测

本课题设计并实现了一种轻量化的插桩工具以检测软件配置相关的性能瓶颈。本课题在静态分析阶段通过资源依赖分析寻找与资源相关的配置项，降低配置空间维度，减少动态分析的插桩数量。

CBHunter 利用 Javassist 插桩[42]以检测软件性能瓶颈，具体流程如图 5.14 所示。首先，CBHunter 根据静态分析结果选取排名靠前的配置项作为可疑配置项（可能导致软件产生性能问题的配置项）。然后，根据静态切片的结果，CBHunter 向可疑配置项影响的函数中插桩，以记录函数的时间和运行次数。本课题仍然只考虑处在应用包中的函数，不考虑处在原始包和扩展包中的语句，因为本课题关注的是软件配置原因导致的软件性能下降，并且 JDK 的库函数可以通过基本的测试方法获得该函数性能。再根据运行时间、内存消耗、网络使用的差值等对这些函数排序；最后，将性能瓶颈和对应的配置项报告给用户。

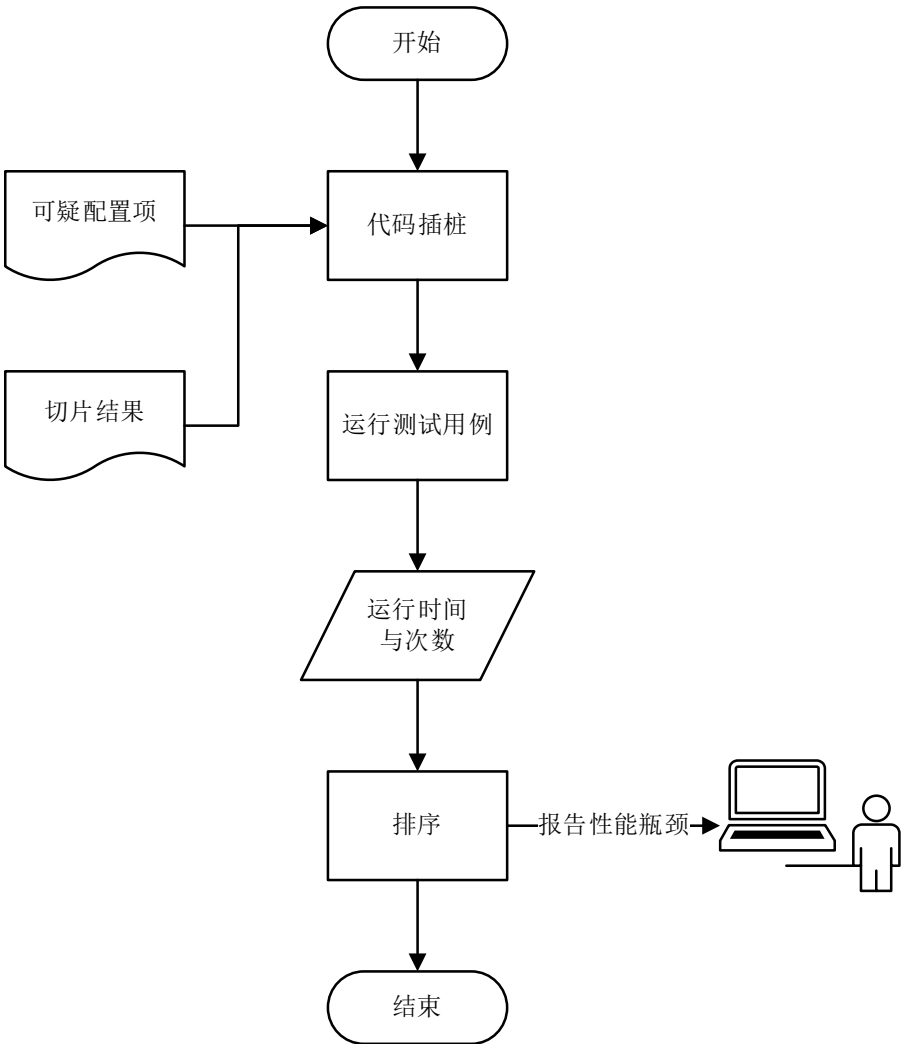

图 5.14 性能瓶颈检测流程

CBHunter 通过分析字节码的方法，向目标软件中切片得到的函数插入拦截器，以获得该函数的执行时间。如图 5.15 所示获取 getNameSpace 的执行时间。首先，将函数 getNameSpace 重新命名为 getNameSpace$imp，然后插入新的函数 getNameSpace（拦截器），该函数中调用 getNameSpace$imp，并将该值存储在变

量 localNameSpace 中，在该调用语句前后获取系统当前时间，以毫秒为单位，并将所用时间存储到 arrayOfByte 中，并调用 DS.out（arrayOfByte）函数将该函数执行时间发送到指定接受线程。通过这样的方法，就可以获取函数的执行的时间，同时，接受信息的线程统计函数的执行次数，并根据执行时间和执行次数计算该函数所用总用时。基于函数总用时，对函数及对应的配置项排序，并将排名靠前的函数及配置项报给开发人员。

```
public NameSpace getNameSpace$impl() {
     return this.ns;
}

public NameSpace getNameSpace(){
     long l = System.currentTimeMillis();
     NameSpace localNameSpace = getNameSpace$impl();
     byte[] arrayOfByte = "getNameSpace$impltook" +
(System.currentTimeMillis() - l) + "ms".getBytes();
     DS.out(arrayOfByte);
     return localNameSpace;
}
```

图 5.15 动态分析插桩示例

第六章 实验与评估

本章主要评估 CBHunter 的有效性、静态分析的准确率和对软件性能产生的额外开销。6.1 节介绍本实验的评估方法，实验环境等；6.2 节评估 CBHunter 检测性能瓶颈有效性；6.3 节评估资源依赖分析的有效性和准确性；6.4 节评估 CBHunter 插桩对软件产生的额外开销；6.5 节对 CBHunter 进行讨论，阐述 CBHunter 的缺点不足及部分实验结果的讨论。

6.1 评测方法

6.1.1 性能问题选取

为了验证 CBHunter 的有效性，本课题将 CBHunter 应用于 6 个新的与配置相关的性能问题，这些性能问题与在第三章中调研分析的性能问题不同。如表 6.1 所示，这 6 个性能问题分别是 ZooKeeper-1583、HDFS-3738、HDFS-11412、MapReduce-6551、Hadoop-13263 以及 Hadoop-8021，并且将性能问题根据资源进行了分类，同时表明导致该性能问题的配置项。这六个性能问题涵盖 CPU，MEM 和 IO 性能问题。CBHunter 的目标是检测配置选项影响的性能瓶颈，一旦检测性能瓶颈，就可以提高诊断软件性能问题的效率。

表 6.1 性能问题

性能问题	领域	配置项	静态分析
ZooKeeper-1583	CPU	maxClientCnxns	Y
HDFS-3738	CPU	dfs.client.socket.timeout	N
HDFS-11412	MEM	dfs.namenode.maintenance.replication.min	Y
MapReduce-6551	MEM	mapred.job.map.memory.mb	Y
Hadoop-13263	MEM	hadoop.security.groups.cache.background.reload	Y
Hadoop-8021	I/O	mapred.output.compres mapred.output.compression.codec	Y

6.1.2 in-house 测试输入集

CBHunter 在静态分析阶段需要计算指令得分，因此本课题运行测试程序以计算每个循环和分支的概率。本课题选择针对 4 款软件选择广泛使用且非常流行的基准测试：TestDFSIO，mrbench，WordCount 和 zk-smoketest[41]。如表 6.2 所示，TestDFSIO 可以测试读取和写入 HDFS 文件效率。本课题选择 10 到 100 个文件，每个文件大小从 10M 到 100M 进行写入和读取，也可以测试清除 HDFS 上的所有数据。mrbench 是 MapReduce 基准测试，本课题运行 10 到 100 次的 MapReduce 任务。WordCount 是 Hadoop 最常用的测试程序，本课题选择文本文件作为输入。zk-smoketest 是 ZooKeeper 的标准测试集。这 4 种测试集虽然单独针对某一种软件，但是当运行 Hadoop 分布式情况时，所有软件会同时运行，即每种测试都会产生 4 款软件的结果。

表 6.2 4 种测试基准测试

软件	测试程序	测试输入及工具配置	
HDFS	TestDFSIO	读文件（10-100M，10-100）	写文件（10-100M，10-100）
MapReduce	mrbench	10-100 次	
Hadoop	WordCount	1M，1-100 次	
ZooKeeper	zk-smoketest	端口：2181，超时：5000，节点大小：25，节点数量：10000	

6.1.3 工具配置

CBHunter 需要检测软件的资源使用率以确定是哪种资源了软件性能的下降，本课题选择 top，netstat，vmstat 检测软件 CPU，内存，网络和磁盘 IO，并且使用 3 种工具的默认配置。

为了验证 CBHunter 的有效性，本课题将 CBHunter 与 Java 标准的性能分析工具 HPROF 比较。HPROF 是 JDK 自带一个简单的性能分析工具，它是一个动态链接库文件，监控 CPU 的使用率、内存堆栈分配情况等。为了减少干扰，本课题使用 HPROF 默认配置。

6.1.3 实验环境

本课题利用 PC 对配置项排序，该 PC 拥有一个 4 核 Intel i5-4590 CPU 和 8GB RAM。受限于硬件环境，本课题使用 2 个节点验证 CBHunter 的有效性，每台虚拟机配置 2 核 CPU 和 2G 内存，共组成 2 个节点的网络。

6.2 CBHunter 有效性评估

本节评估 CBHunter 的有效性，首先检测真实的性能问题，判断 CBHunter 是否可以检测到导致性能下降的软件性能瓶颈；然后，利用相反的方法，本课题先获得 CBHunter 的结果，搜索配置项和性能瓶颈函数，是否能发现真正的由该配置导致的性能问题。

6.2.1 真实性能问题检测

如第五章所述，CBHunter 按降序对插桩的函数进行排序，排名越高表明该函数成为性能瓶颈的可能性越大，然后将排序靠前的函数和对应的配置项报高给用户。

表 6.3 性能瓶颈检测

性能问题	CBHunter	HPROF
HDFS-11412	8	114
Hadoop-13263	14	204
ZooKeeper-1583	20	176
Hadoop-8021	33	283
MapReduce-6551	50	317
HDFS-3738	-	40

为了评估 CBHunter 检测瓶颈的有效性，本课题根据软件 bug report 系统获取如何触发 6 个配置相关的性能问题的信息。通过阅读 bug report 的内容，patch 以及 commit 等信息，运行测试程序触发了 6 个由配置导致的性能问题。例如 ZooKeeper-1583，当配置项 maxClientCnxns 的值小于真实连接的用户数量，就会产生拒绝接连的结果，ZooKeeper 服务器产生用户数量过多的异常。

本课题将 CBHunter 应用于这 6 个与配置相关的性能问题，并将结果与 Java 标准性能分析工具 HPROF 比较。结果如表 6.3 所示，数字表示性能瓶颈的排名，较小的数字代表排名中较高的位置。以 HDFS-11412 为例，导致该性能问题的真实瓶颈在 CBHunter 排名中排名第 8，HPROF 排名第 114。与 HPROF 的排名相比，CBHunter 中 HDFS-11412 的瓶颈排名提高了 114/8 倍。

前人工作表明[2]，大多数（59%）性能问题都与配置项有关，而 41%是的性能问题与性能无关。与配置相关的性能问题的根本原因（如果不是所有时间）通常位于受配置选项影响函数中。但是，表 6.3 显示 HPROF 排名列表中只有少数方法受配置项的影响。也就是说，大多数性能问题的根本原因在于受少数配置项影响的函数。考虑到这一点，CBHunter 仅对受配置选项影响的方法进行排名。在大多数情况下，此策略可以显着提高诊断性能问题的有效性。对于与配置相关的性能问题，如表 6.3 所示，与 HPROF 平均排名相比，CBHunter 的瓶颈排名可提高 10 倍；对于非配置相关的性能问题，CBHunter 无法检测。因此，与 HPROF 的能力相比，

CBHunter 检测软件性能瓶颈的能力为 10×59%+0×41%=5.9 倍。

6.2.2 导致性能问题配置项检测

为了进一步说明 CBHunter 检测性能瓶颈的有效性，本课题将第三章调研的结果与 CBHunter 产生结果的比对。如果 CBHunter 能够发现导致了性能问题的配置项，那么说明 CBHunter 可以检测到性能问题的配置项；并且检测结果所占的比例越高，那么说明 CBHunter 检测性能瓶颈的能力强大。

因此，本课题将 CBHunter 应用于 ZooKeeper，HDFS，Hadoop Common 以及 MapReduce。本课题分别选取四款软件最后得到性能瓶颈的前 5、20、10、10 个配置项作为导致性能下降的瓶颈。以图 6.1 中 ZooKeeper 的结果为例，根据测试的结果，得到前 5 个配置项，并且每个配置项对应的函数瓶颈如图 6.1 中所示。

1、peerType：run, ...
2、maxClientCnxns：doaccpet, ...
3、maxSessionTimeout：readFrom,...
4、purgeInterval：initializeAndRun,...
5、snapRetainCount：initializeAndRun, ...

图 6.1 ZooKeeper 中与 CPU 相关性能瓶颈

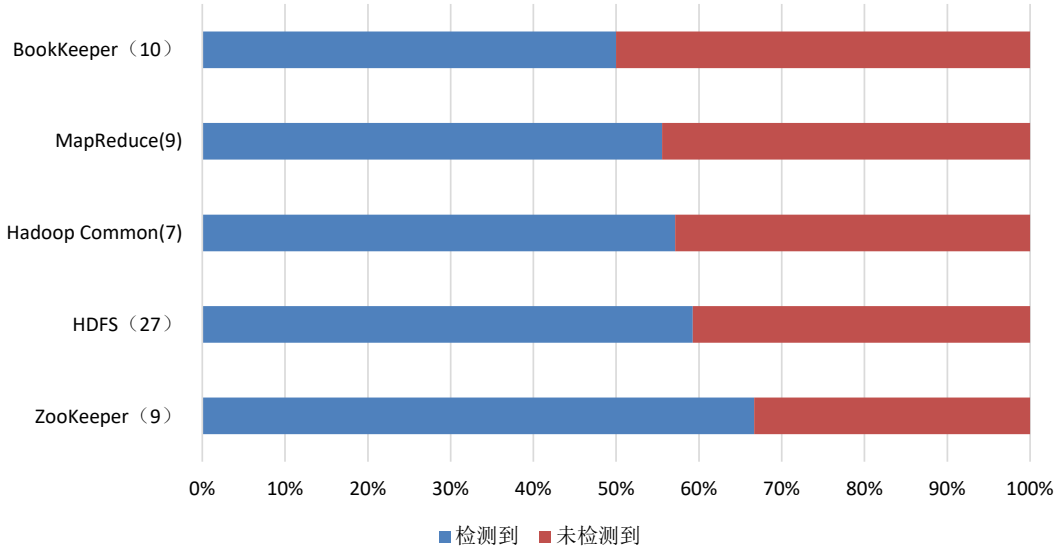

图 6.2 检测到（蓝色）和为、未检测到（红色）导致性能问题的配置项数量

然后，本课题根据图 6.1 中的配置项的排序，将得到的 5 个配置项与第三章中调研的性能问题比对（4 款软件分别由 9、27、7、9 个性能问题），如果该配置项导致了性能问题，那么则认为 CBHunter 可疑检测到导致性能问题的配置项，图 6.2 展示 CBHunter 可以检测到导致性能问题的配置项的数量和未能检测到导致性能问

题配置项的数量（红色表示该配置导致了软件性能问题，蓝色表示搜索到与该配置项相关的 bug 数量）。图 6.2 表明，CBHunter 可以检测到超过一半的配置项，在最好情况下，CBHunter 可以检测到 6/9 个性能问题，4 款软件的平均值达到了 60%。

为了进一步验证 CBHunter 有效性，本课题以同样的方法应用于 BookKeeper，从 bug report 系统中选取 10 个与配置相关的性能问题，并统计 CBHunter 可以检测到多少导致了性能问题的配置项。CBHunter 可以检测到 5 个性能问题的配置项（占比为 5/10），不能检测到 5 个配置项。根据图 6.2 及 BookKeeper 的结果，可以得出结论，CBHunter 可以有效检测到导致软件性能问题的配置项，同时能够降低配置空间的维度，增加分析结果的准确度，降低软件维护人员修复配置相关的性能问题的成本。

6.3 资源依赖分析有效性评估

本课题利用资源依赖分析降低配置空间维度，减少需要分析的配置项数量，因此，评估资源依赖分析的有效性至关重要。本节评估 CBHunter 是否能够将真正导致性能下降的配置项作为可疑配置项，并评估选取不同参数（α）对资源依赖分析的影响。

6.3.1 可疑配置项选取

本课题通过资源依赖分析的方法对配置项排序，并选取排名高的配置项作为可疑配置项。然而，选取多少个配置项或多少百分比（α）作为可疑配置性是 CBHunter 的一个非常重要的参数，α 会影响 CBHunter 检测性能问题的有效性。如果该参数比较小，那么资源依赖分析可能会遗漏部分导致性能下降的配置项，如果该参数比较高，那么就会产生噪音配置项，增加动态分析需要的插桩数量，增加运行开销，并且很有可能掩盖真正的性能瓶颈。

为了衡量 α 对 CBHunter 有效性的影响，本课题检测资源依赖分析能够检测到多少导致软件性能下降的配置项。因此，本课题将 α 设置为 5%，10%，15%和 20%（如果超过 20%，那么产生的配置项数量过多，引入更多的噪音配置项），且至少为 10。本课题根据第三章的调研结果，将 CBHunter 资源依赖分析应用于 4 款软件，根据不同的资源，应用表 5.2 中的指令得分，计算配置项得分。并统计 CBHunter 可以检测到多少导致性能问题的配置项。图 6.3 显示该分析结果。在 52 个与配置相关的性能问题中，如果将 α 设置为 5%，则可疑检测到 23/52 个导致性能问题的配置项，为 10%时可以检测到 40/52 个，为 20%时可以检测到 45 个，仅比 10%时多 5 个。即使该参数为 20%时，可以获得更好的结果，但是引入了更多的噪音。为了减少噪音配置项，本课题在资源依赖分析中选择前 10%配置项作为可疑配置

项。

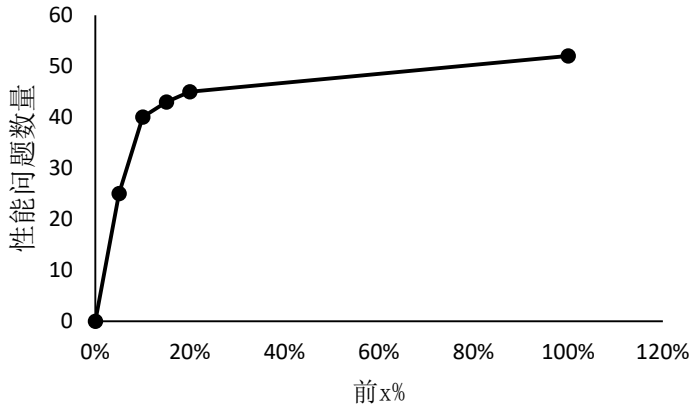

图 6.3 前 x%配置项检测性能问题配置项数量

6.3.2 资源依赖分析有效性评估

6.3.1 节中说明本课题如何选择 α 值，为了进一步说明 CBHunter 资源依赖分析的有效性，本课题将判断 α 值为 10%时，资源依赖分析得到导致性能问题的配置项数量。本而课题利用上一节中的方法评估并应用于 4 款软件，结果如图 6.4 所示，资源依赖分析的准确度均在 70%以上，大约 25%导致性能问题的配置项不能被选为可疑配置项（6.5 节中详细讨论原因）。从图中可以明确得到 CBHunter 资源依赖分析可以有效的得到可疑配置项，降低配置空间的维度。

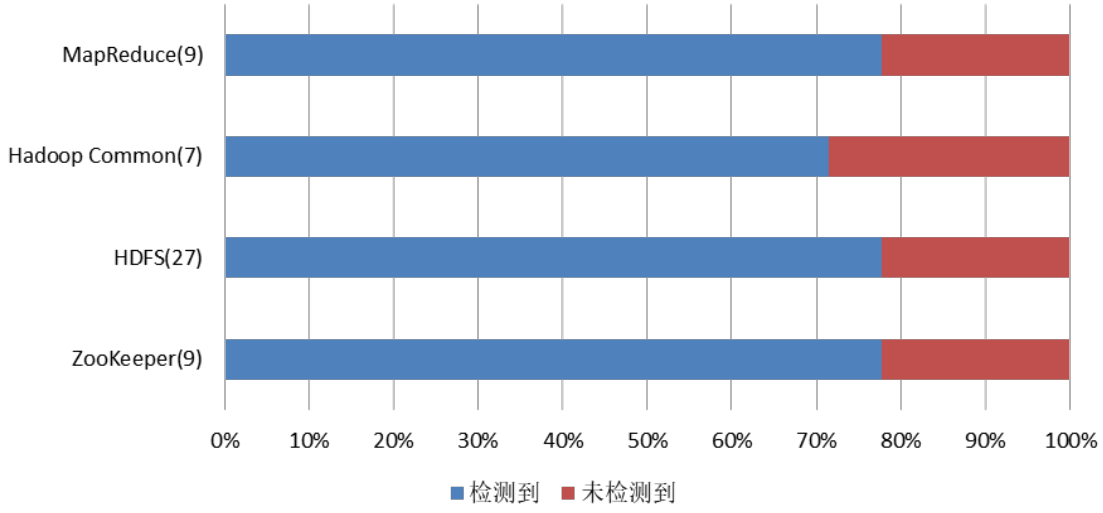

图 6.4 4 款软件资源依赖分析结果

6.4 CBHunter 开销评估

由于动态分析插桩，会给软件运行带来额外开销，因此，本课题评估 CBHunter

的插桩数量及由于插桩带来的额外开销。6.4.1 节评估 CBHunter 与 HPROF 的插桩数量；6.4.2 节评估插桩导致的额外开销。

6.4.1 插桩数量评估

为了评估 CBHunter 的开销，本课题统计 ZooKeeper、Hadoop、MapReduce 和 HDFS 共 4 款软件，应用 CBHunter 和 HPROF 针对 4 种资源需要的桩数量.HPROF 对每个函数插桩，因此只需要统计类的数量和函数的数量即可。表 6.4 展示结果，CBHunter 的需要插桩类数量平均为 140，函数数量平均为 663。相较于 HPROF，CBHunter 向软件中插桩的类和函数数量更少。

表 6.4 插桩数量对比

评测软件	CBHunter		HPROF	
	类	函数	类	函数
ZooKeeper	128	491	472	3756
Hadoop	145	561	1928	13576
MapReduce	126	459	806	5652
HDFS	159	1142	1819	16798
平均	140	663	1256	9945

通过表 6.4 可以发现，HPROF 针对每一个函数插桩必定会给软件带来更大的开销。本课题提出的 CBHunter 针对很少的类和函数，可以达到相同的效果。虽然，软件运行期间并不会运行到所有的函数，但是本课题相信 CBHunter 必定会获得更小的开销。与此同时，本课题发现，源代码中代码行数小于 5 且不包含函数调用的函数产生性能问题的概率较小，如果能够剔除这些影响较小的函数，那么 CBHunter 的插桩数量会进一步减少。

6.4.2 额外开销评估

虽然 CBHuner 的插桩数量少于 HPROF 的插桩数量。但是为了更加准确的评估插桩对软件产生的额外开销，本课题计算软件实际运行产生的额外开销（获取函数执行时间）。本课题在计算 4 种测试程序下的额外开销，分别是 WordCount、DFSIO-Write、DFSIO-Read 和 ZKTest（本课题实现的脚本）的运行时间。首先运行不带任何插桩的原始代码，作为参照；然后分别将 CBHunter 和 HPROF 应用于目标软件；最后基于 3 种情况的运行时间计算额外开销比例。对于 WordCount，本课题将.txt 文本（每个文本文件大小为 1MB）作为输入，并根据使用实验次数增加输入的大小。对于 DFSIO-Write 和 DFSIO-Read，本课题利用示例程序测试 HDFS 的读写速度，10 个，20 个，30 个和 40 个文件（每个文件 10，50，100，150MB）。

ZKTest 中，本课题执行相同的动作 100，200，300，400 次，以评估软件开销。所有测试程序运行 20 次，然后将平均时间作为最后的结果。如图 6.5 展示 WordCount、DFSIO-Write、DFSIO-Read 的结果，其中 black 表示 CBHunter，orange 表示 HPROF，blue 表示无插桩的程序。

表 6.5 CBHunter 和 HPROF 额外开销比例

程序	CBHunter		HPROF	
	平均	最坏	平均	最坏
WordCount	4.7%	9.0%	119.0%	153.4%
DFSIO-Write	4.7%	10.0%	23.0%	59.1%
DFSIO-Read	5.1%	8.7%	32.1%	69.6%
ZKTest	2.2%	3.8%	-	-

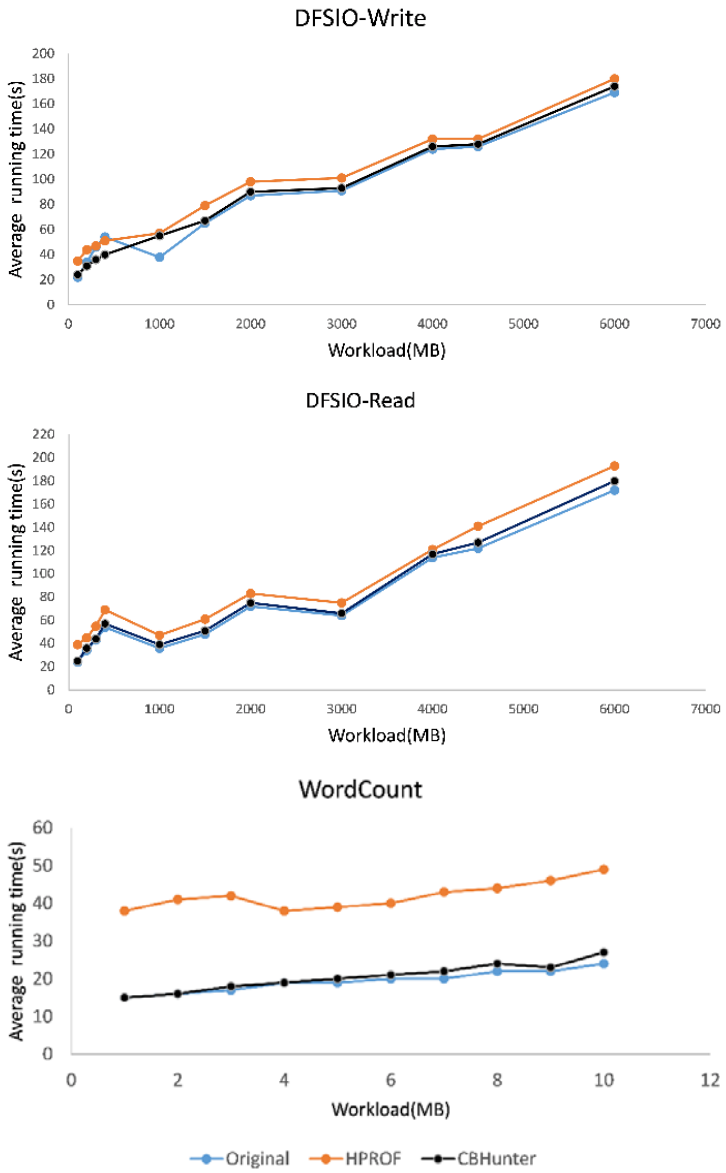

图 6.5 插桩导致的额外开销额外开销

为了更加清楚计算额外开销，本课题计算 CBHunter 和 HPROF 额外开销的比例。如表 6.5 所示，运行 WordCount，CBHunter 平均只有 4.7%的开销，但 HPROF 的运行时间是无插桩情况下的两倍多，过高的额外开销对于维护人员来说是难以接受的。但是，在 DFSIO-Write 中，CBHunter 平均只有 4.7%的开销，最差的情况是 10%，而 HPROF 平均为 23.0%。DFSIO-Read 与 DFSIO-Write 相同，CBHunter 平均引入 5.1%的开销，最差的情况是 8.7%，而平均值为 32.1%。总之，在这 3 种不同测试程序下，具有不同工作负载的情况下，CBHunter 总是优于 HPROF，与 HPROF 的开销相比，CHunter 的平均开销为 5%。同时，本课题应用 ZKTest 测试 ZooKeeper 时发现，平均开销仅有 2.2%，最坏情况下开销也只有 3.8%，这样的开销可以应用于软件在线分析，而且不会给用户带来较差的体验。

表 6.6 CBHunter 的资源依赖分析运行时间

软件	平均运行时间（s）			
	CPU	MEM	IO	NET
ZooKeeper	148	167	142	145
Hadoop Common	319	312	311	317
MapReduce	267	295	241	230
HDFS	1157	449	466	453
Average	472.75	305.75	290	286.25

同时，本课题测量资源依赖分析的运行时间。如果运行时间过长，对于开发人员仍然不可接受的。本课题使用一台笔记本为 ZooKeeper、Hadoop Common、MapReduce 和 HDFS 运行 CBHunter 的资源依赖分析，获得每个软件的运行时间。结果如表 6.6 所示，平均运行时间为 8 分钟。总的来说，这样的离线分析时间对于软件维护人员是可以接受的，因为资源依赖分析只需要运行一次。

6.5 原因分析

本课题分析 CBHunter 为什么不能检测到所有导致性能问题的配置项。CBHunter 有效性主要受到以下两类因素的影响：

1、无法构建软件真实运行环境及输入。测试程序并不会触发所有配置项，某些路径仅在某些输入条件下有效。但是本课题不能获取真实的工作负载及输入参数。例如，当 Hadoop 中输入为多个小文件时，会创建多个 mr 任务，每个任务占用固定的内存，就会导致内存不足，产生性能问题。但是，这种性能问题受到实际硬件环境的影响，创建文件的多少，以及固定内存的值都会影响性能问题的检测。同时，某些参数受到其他参数的约束，必须满足一定条件才会触发该参数其作用，如图 1.1 所示，只有当 if 条件语句

中的配置项 reloadGroupsInBackGround 为真时，reloadGroupsThreadCount 才会起作用。本课题只运行了测试程序以获得当前的性能瓶颈，如果可以运行每一种测试程序，就可能获得更加准确的结果，但是这种方法带来的开销不可接受。

2、其次，CBHunter 检测性能瓶颈受到资源依赖分析有效性影响。资源依赖分析对所有配置项进行排序，但是不能准确计算每个配置项的得分，主要有以下原因：

（1）in-house 测试用例不能触发所有路径，不同路径会影响指令计算的得分，从而影响配置项得分。虽然本课题使用不同负载，不同输入对软件进行测试，以获得软件循环次数和分析概率，但是并不能保证能够触发所有路径，导致本课题不能准确的计算每个配置项的得分。例如，zookeeper 中 doaccept 函数会判断当前软件用户连接数是否超过最大连接数配置项值，只有当用户数超过该值时，才会产生异常，如果未达到该用户数量，就不能产生异常，如果未能触发该路径，那么就不能准确的最大连接数配置项的得分；

（2）预定义的指令分数不够精确。指令分数是计算配置项得分的基础，虽然本课题参考了不同芯片的 datasheet，阅读 java 字节码的相关文档，但是仍然不能准确量化不同指令对资源的影响程度。例如，DIV 指令周期数明显大于 ADD 指令周期数，但是，本课题不能准确度定义 DIV 是 ADD 的多少倍，只能设置为一个估算值。

（3）配置项读入函数识别准确率较低。虽然本课题通过分析源码总结了配置项读入函数的 3 种模式，但是仍然会遗漏部分配置项。CBHunte 利用反汇编技术分析字节码以获得配置项的读入函数，平均可以检测到 80%配置项的读入函数，CBHunter 仍会遗漏大约 20%的配置项，这一部分配置项仍有可能导致软件性能下降，从而降低了资源依赖分析的准确性。字节码在编译软件源码时，会丢失配置项读入的信息。例如，get 函数只返回配置项的值，而不带有配置项的键，导致部分配置项没有办法通过分析字节码识别配置项的读入函数。本课题下一步将会结合软件源码和字节码进一步提高配置项读入函数识别准确率，从而提高资源依赖分析有效性；

（4）切片结果可能存在误差。WALA 是通过构建系统依赖图（SDG）的方法来获得切片结果，这种静态切片方法相比于动态切片方法会引入更多的噪音语句。与此同时，受限于个人电脑性能，不能够将切片方法设置为 FULL，这会遗漏部分配置项影响的语句，降低资源依赖分析的准确性。例如，本课题将反射设置未 NONE，虽然这种设置可以有效的降低计算开

销，但是不能得到反射影响到的语句，数据依赖和控制依赖均未考虑异常，即不能够获取异常处理语句。

6.6 讨论

本节主要讨论实验评估不足，CBHunter 检测性能瓶颈的能力受到哪些因素的影响，CBHunter 自身的缺陷以及教训总结及建议。

6.6.1 实验评估不足

受限于复现分布式软件系统中性能问题的困难，本课题将 CBHunter 应用于 6 个配置相关的性能问题。实验表明，CBHunter 可以检测到 5/6 的性能瓶颈，并且产生的额外开销对。但是仅有 6 个配置相关性能的性能问题作为评估不能说明 CBHunter 检测性能瓶颈的能力。CBHunter 仅检测配置相关的性能瓶颈，因此，本课题验证运行测试用例产生的性能瓶颈是否在 Apache 的 bug report 系统中搜索到对应的 bug，本而课题认为这种方法是有效的。即使没有复现该性能问题，但是仍然可以使维护人员提高诊断性能问题的效率。

本课题评估 CBHunter 在 3 个测试程序的额外开销，并且每个测试程序的工作负载及输入参数随着实验次数变化。并且，本课题运行每个测试程序 20 次，将平均值作为最后的开销，以避免由于外在因素（系统环境变化等）导致带来误差。但是，这些测试程序不能覆盖全部路径，不能够完全说明插桩带来的额外开销不会增加。因此，本课题同时评估插桩的数量，说明额外开销是可接受的。

6.6.2 CBHunter 的缺陷

CBHunter 是一种轻量级的性能分析工具。首先，CBHunter 只针对配置相关的性能问题，如果由于其他问题导致的软件性能问题，那么 CBHunter 将不能检测到导致性能问题的性能瓶颈；其次，CBHunter 需要目标软件的字节码和源文件，如果只有字节码或源文件，且源文件不可编译，那么 CBHunter 将无法工作。最后，CBHunter 处理不良的编码风格上的能力较差，CBHunter 基于函数名分析的方法识别配置项的读入函数，编码风格较差则导致读入函数识别准确率较低，影响资源依赖分析的结果。

6.6.3 教训总结及建议

本小节讨论本课题研究过程中遇到的问题，以及解决过程的经验和资源相关的配置项相关建议。

添加配置项标签：研究分析过程中，本课题发现，部分用户在 bug report 中提出增加配置项标签的意见，即添加该配置项属于什么类型，与什么领域相关。例如该配置项是整数类型，与软件安全相关等。添加标签可以更加快速的理解该配置项与哪些相关。

完善配置文件中配置项与性能的描述：添加配置项改变会如何影响软件性能，可以帮助用户快速的优化软件性能，而不需要修改软件源码。虽然这种方式给软甲开发和测试人员增加了很大的负担，但是这对软件的维护任务有着极大的帮助。例如，Hadoop 的配置文件中，只描述了配置项的功能，而没有与性能相关的描述。

配置项修改说明：软件不同版本，添加新的配置项或修改旧的配置项时，在配置文件中说明该版本修改了哪些配置项，增加了哪些配置项。这样可以有效的帮助用户在使用新版本的软件中快速了解新增的特性及如何配置。例如，HDFS-9313 中指出需要更新配置文件中的配置项。

配置读入的一致性：在研究分析软件配置项读入函数时，本课题发现，并不是所有的配置项都通过特定的方式读入，即使已经提供了特定的类，部分代码仍然使用变量的形式进行操作。

结 束 语

本章回顾本课题的主要贡献，总结面向性能优化的配置故障诊断技术研究，分析了目前工作成果的成果及不足。针对本工作存在的不足并结合目前配置相关的性能故障诊断发展趋势，展望课题研究的下一步工作。

工作总结

随着软件规模的不断增大，软件中非功能属性配置项的不断增加，配置引起已经成为导致软件性能下降的主要原因之一。检测软件性能瓶颈可以加快诊断软件性能问题。本课题针对配置相关的性能问题，检测软件中与配置相关的性能瓶颈。通过对大量软件配置项的调研，提出配置相关的性能问题与资源相关的配置项相关，设计并实现了一种基于资源依赖分析的配置相关的软件性能瓶颈检测工具CBHunter。本课题的主要贡献包括以下几个方面：

1. 调研分析了 7 款分布式开源软件的 75 个配置相关的性能问题和 50 个资源相关的配置项，包括 ZooKeeper，Hadoop Common，HDFS，HBase，Cassndra，MapReduce 和 Yarn。通过研究分析证明了本课题提出的配置相关的性能问题与资源相关的配置项有关联的观点，同时本课题将配置相关的性能问题分为 5 类，并发现最有可能导致软件性能问题的 3 类资源是 CPU，MEM 和 IO；
2. 基于调研分析验证的观点，本课题提出一种负载不敏感的资源依赖分析方法以降低配置空降维度，对配置排序以挖掘与资源相关的配置项。依据程序切片技术获得配置项影响的语句，并量化这些语句对软件资源使用的影响作为该配置项对资源的影响。通过实验评估发现，本课题提出的方法能够有效检测与资源相关的配置项，并且前 10%的配置项中导致性能问题的配置项占比达到 80%。
3. 本课题实现了检测配置相关的性能瓶颈的工具 CBHunter，并且本课题验证了 CBHunter 的检测性能瓶颈的有效性。本课题将 CBHunter 应用于 6 个软件性能问题，实验结果表明 CBHunter 能够检测到 5 个性能问题的性能瓶颈，并且动态分析的平均额外开销小于 5%。

相比于前人工作，本课题利用资源依赖分析，降低配置空间维度，挖掘于资源相关的配置项，减少需要分析的配置项数量，并通过插桩的方法检测性能瓶颈。CBHunter 可以实现更小的额外开销，并且能够快速的检测到由配置导致的软件性能瓶颈，有效提高维护人员诊断性能问题的效率。

研究展望

针对本课题针对配置相关的性能问题的不足以及结合当今研究的发展趋势，本课题计划在以下方面继续研究：

1. 增加对配置相关的性能问题调研的数量，进一步证明本课题提出的配置相关的性能问题与资源相关的配置项有关联的观点。
2. 针对资源分类问题，目前只考虑了硬件资源，没有考虑线程同步，电量消耗等方面，接下来的工作需要进一步细化资源的分类。
3. 针对指令得分方面，进一步研究更加精确的指令得分。指令得分更精确，那么资源依赖分析的结果就会更加精确。
4. 测试程序较少，导致不能够获取软件运行的全部路径。下一步计划运行更多的测试程序以获得更加精确的结果。
5. CBHunter 检测性能瓶颈的能力评估上，还明显不足，接下来的工作是进一步评估 CBHunter 的能力，检测软件性能瓶颈。
6. 检测软件配置相关的性能瓶颈只是诊断配置相关的性能问题的关键一步。接下来本课题将要继续朝着诊断配置相关的性能问题的目标，研究如何从软件性能瓶颈中修复软件性能问题。

致　谢

时光飞逝，转眼间两年半紧张而又充实的研究生生活即将画上句号。在这两年半的学习期间，我得到了很多老师、同学和朋友的关怀和帮忙。在学位论文即将完成之际，我要向所有期间给予我支持、帮忙和鼓励的人表示我最诚挚的谢意。

首先，我要感谢我的指导老师李姗姗对我的教导。从论文的选题、构思、撰写到最终的定稿，李姗姗老师都给了我悉心的指导和热情的帮忙，使我的毕业论文能够顺利的完成。李姗姗老师对工作的认真负责、对学术的钻研精神和严谨的学风，都是值得我终生学习的。

其次，感谢计算机学院的全体领导和老师，由于他们的悉心教导，我学到了专业的计算机知识，掌握了扎实的专业技能。同时，也感谢论文指导组老师刘晓东老师，贾周阳师兄，徐尔茨师兄对论文的修改提出了宝贵意见，使我的论文更加完善。

最后，感谢我的家人在此期间给予我的包容、关爱和鼓励，以及所有陪我一路走来的同学和朋友，正是由于他们的支持和照顾，我才能安心学习，并顺利完成我的学业。

毕业在即，在今后的工作和生活中，我会铭记师长们的教诲，继续不懈努力和追求，来报答所有支持和帮忙过我的人!

参考文献

[1] http://hadoop.apache.org/core/

[2] Han X, Yu T. An Empirical Study on Performance Bugs for Highly Configurable Software Systems[C]// ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, 2016:23.

[3] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. J. of Softw. Maint. and Evo. R. P., 13(1):3–30, 2001

[4] C. Schwaber, C. Mines, and L. Hogan. Performance-driven software development: How it shops can more efficiently meet performance requirements. Forrester Research, 2006.

[5]G. E. Morris, ``Lessons from the Colorado benefits management system disaster" 2004, www.ad-mktreview. com/public html/air/ai200411.html.

[6] T. Richardson, ``1901 census site still down after six months," 2002, http://www.theregister.co.uk/2002/07/03/1901 census site still down/.

[7] D. Mituzas, ``Embarrassment", 2009, http://dom.as/2009/06/26/embarrassment/

[8] Y. Group. Enterprise application management survey. 2005

[9] Shen D, Luo Q, Poshyvanyk D, et al. Automating performance bottleneck detection using search-based application profiling[C]// International Symposium on Software Testing and Analysis. ACM, 2015:270-281.

[10] Nair V, Menzies T, Siegmund N, et al. Using bad learners to find good configurations[J]. 2017:257-267.

[11] Sarkar A, Guo J, Siegmund N, et al. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T)[C]// Ieee/acm International Conference on Automated Software Engineering. IEEE Computer Society, 2015:342-352.

[12] Sarkar A, Guo J, Siegmund N, et al. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T)[C]// Ieee/acm International Conference on Automated Software Engineering. IEEE Computer Society, 2015:342-352.

[13] Siegmund N, Grebhahn A, Apel S. Performance-influence models for highly configurable systems[C]// Joint Meeting. 2015:284-294.

[14] Siegmund N, Kolesnikov S S, Apel S, et al. Predicting performance via automated feature-interaction detection[C]// International Conference on Software Engineering. IEEE, 2012:167-177.

[15] Grechanik M, Fu C, Xie Q. Automatically finding performance problems with feedback-directed learning software testing[J]. 2012, 14(19):156-166.

[16] Chilimbi T M, Liblit B, Mehra K, et al. HOLMES: Effective statistical debugging via efficient path profiling[C]// IEEE, International Conference on Software

Engineering. IEEE, 2009:34-44.

[17] Schur M, Roth A, Zeller A. Mining behavior models from enterprise web applications[C]// Joint Meeting on Foundations of Software Engineering. 2013:422-432.

[18] Liu X, Zhan J, Zhan K, et al. Automatic performance debugging of SPMD-style parallel programs[J]. Journal of Parallel & Distributed Computing, 2011, 71(7):925-937.

[19] Han S, Dang Y, Ge S, et al. Performance debugging in the large via mining millions of stack traces[C]// International Conference on Software Engineering. IEEE Press, 2012:145-155

[20] Grechanik M, Fu C, Xie Q. Automatically finding performance problems with feedback-directed learning software testing[J]. 2012, 14(19):156-166.

[21] A Nistor, PC Chang, C Radoi, S Lu. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. IEEE/ACM IEEE International Conference on Software Engineering. 2015 , 1 :902-912.

[22] A Nistor , L Song, D Marinov, S Lu. Toddler: Detecting performance problems via similar memory-access patterns. International Conference on Software Engineering , 2013 , 8104 :562-571

[23]S Lu, S Lu. Performance Diagnosis for Inefficient Loops. International Conference on Software Engineering , 2017 :370-380

[24] Attariyan M, Chow M, Flinn J. X-Ray: Automating root-cause diagnosis of performance anomalies in production software. In Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI). Hollywood, 2012.

[25] Attariyan M, Flinn J. Using Causality to Diagnose Configuration Bugs. [C]. In Usenix Technical Conference, Boston, Ma, Usa, June 22-27, 2008. Proceedings. 2008: 281–286.

[26] Attariyan M, Flinn J. Automating configuration troubleshooting with dynamic information flow analysis [C]. In Usenix Conference on Operating Systems Design and Implementation. 2010: 1–11.

[27] Yuan D, Xie Y, Panigrahy R, et al. Context-based online configuration-error detection [C]. In Usenix Conference on Usenix Technical Conference. 2011: 28–28.

[28] Zhang J， Renganarayana L， Zhang X， et al. EnCore: exploiting system environment and correlation information for misconfiguration detection[J]. Acm Sigarch Computer Architecture News， 2014， 42(1):687-700.

[29] Wang S, Li C, Sentosa W, et al. Understanding and Auto-Adjusting Performance-Related Configurations[J]. 2017.

[30] Su YY， Attariyan M， Flinn J. AutoBash: Improving configuration management with operating system causality analysis. In: Proc. of the 21st ACM Symp. on Operating Systems Principles (SOSP). 2007. [doi: 10.1145/1294261.1294284]

[31] Attariyan M， Flinn J. Using causality to diagnose configuration bugs. In: Proc.

of the 2008 USENIX Annual Technical Conf. (ATC). Boston， 2008

[32]Yan C, Cheung A, Yang J, et al. Understanding Database Performance Inefficiencies in Real-world Web Applications[C]// ACM on Conference on Information and Knowledge Management. ACM, 2017:1299-1308.

[33] https://issues.apache.org/

[34] http://hadoop.apache.org/

[35] Relax: Automatic Contention Detection and Resolution for Configuration related Performance Tuning

[36] https://en.wikipedia.org/wiki/Soot

[37] https://en.wikipedia.org/wiki/Wala

[38]http://pdf1.alldatasheet.com/datasheetpdf/view/174758/ATMEL/ATMEGA16\-16AU.html

[39] https://software.intel.com/en-us/articles/intel-sdm

[40] https://blog.csdn.net/web_code/article/details/12164733

[41] https://github.com/phunt/zk-smoketest

[42] http://www.javassist.org/

[43]Keller L, Upadhyaya P, Candea G. ConfErr: A tool for assessing resilience to human configuration errors [C]. In IEEE International Conference on Dependable Systems and Networks with Ftcs and DCC. 2008: 157–166.

[44] Xu T, Zhang J, Huang P, et al. Do not blame users for misconfigurations [C]. In Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 244– 259.

[45] JC Huang. Program Instrumentation and Software Testing. 《Computer 》, 1978, 11 (4) :25-32.

[46] 王克朝, 成坚, 王甜甜, 任向民.面向程序分析的插桩技术研究. 《计算机应用研究》, 2015 (2) :479-484.

[47] Zhang S, Ernst MD. Which configuration option should I change? In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE). 2014. [doi: 10.1145/2568225.2568251]

[48] Kavulya S P, Joshi K, Giandomenico F D, et al. Failure Diagnosis of Complex Systems[M]// Resilience Assessment and Evaluation of Computing Systems. Springer Berlin Heidelberg, 2012:239-261.

[49] Zhou B, Xia X, Lo D, et al. Build Predictor: More Accurate Missed Dependency Prediction in Build Configuration Files[C]// 2014 IEEE 38th Annual Computer Software and Applications Conference (COMPSAC). IEEE Computer Society, 2014:53-58.

[50] Xu T, Zhou Y. Systems Approaches to Tackling Configuration Errors[J]. Acm Computing Surveys, 2015, 47(4):1-41.

[51] 陈伟, 黄翔, 乔晓强,等. 软件配置错误诊断与修复技术研究[J]. 软件学报, 2015, 26(6):1285-1305.

[52] 章敏敏, 徐和平, 王晓洁,等. 谷歌 TensorFlow 机器学习框架及应用[J]. 微型机与应用, 2017, 36(10):58-60.

[53] Mitchell T M. Machine learning.[M]. China Machine Press ;McGraw-Hill Education (Asia), 2003.

[54] Abadi M, Agarwal A, Barham P, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems[J]. 2016.

[55] 李福亮, 杨家海, 吴建平,等. 互联网自动配置研究[J]. 软件学报, 2014, 25(1):118-134.

[56] 刘虎球, 马超, 白家驹. 面向驱动配置的自动日志插入方法研究[J]. 计算机学报, 2013, 36(10):1982-1992.

[57] Abadi M, Agarwal A, Barham P, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems[J]. 2016.

[58] Vitter J S. Random sampling with a reservoir[J]. Acm Transactions on Mathematical Software, 1985, 11(1):37-57.

[59] Hirzel A, Guisan A. Which is the optimal sampling strategy for habitat suitability modelling[J]. Ecological Modelling, 2002, 157(2):331-341.

[60] Tan W C, Chen I M, Tan H K. Automated identification of components in raster piping and instrumentation diagram with minimal pre-processing[C]// IEEE International Conference on Automation Science and Engineering. IEEE, 2016:1301-1306.

[61] AutoSLIDE: Automatic Source-Level Instrumentation and Debugging for HLS.

作者在学期间取得的学术成果

[1] 李云峰,李姗姗,刘晓东,王智明. 相关配置项检测方法[J]. 全国软件与应用学术会议,2018

附录 A　附录 A 题目

附录是学位论文主体的补充，并不是必需的。