文章编号:1671-4598(2022)02-0017-07　DOI:10.16526/j.cnki.11-4762/tp.2022.02.003　中图分类号:TP311　文献标识码:A

大规模C++工程单元测试性能优化研究

刘堂臣,王雅文,宫云战
(北京邮电大学 网络与交换技术国家重点实验室,北京　100876)

摘要: 为了解决自动化单元测试工具在测试大规模C++工程时经常发生内存溢出故障且耗时较长这一问题,在测试流程中引入了缓存优化技术,并提出了一种面向不同测试方式的缓存优化方法;当用户直接对整个工程进行测试时,系统将采用缓存预取的方式,通过设计的缓存预取模型,在缓存出现读缺失之前为其提供数据块;当用户对单个文件进行测试时,系统将采用改进的GDSF替换算法进行缓存替换;实验表明,该方法能够有效地避免此类单元测试工具发生内存溢出故障并减少了测试的时间,使其支持的被测工程规模由5 000行左右增加至十几万行,大大提升了系统的性能。

关键词: 单元测试;大规模工程;缓存替换算法;缓存预取;性能优化

Research on Unit Testing Performance Optimization of Large-scale C++ Project

LIU Tangchen, WANG Yawen, GONG Yunzhan
(State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing　100876, China)

Abstract: In order to solve the problem that the automatic unit test tool often occurs memory overflow failure and spends a long time in testing large-scale C++ projects, the cache optimization technology is introduced in the test process, and a cache optimization method for different testing methods is proposed. When the user directly tests the entire project, the system will adopt the cache prefetch method. Through the designed cache prefetch model, it will provide data blocks before the cache reads and misses; When the user tests a single file, the system will use the improved GDSF replacement algorithm for cache replacement. Experimental results show that this method can effectively avoid memory overflow failures for such unit test tools and reduces the test time. The scale of the tested project is supported, it increases from about 5, 000 lines to more than 100, 000 lines, and the performance of the system is greatly improved.

Keywords: unit test; large-scale project; cache replacement algorithm; cache prefetch; performance optimization

0　引言

随着互联网技术的飞速发展,软件的规模迅速膨胀,软件测试的困难与代价也变得越来越大。软件测试的基础是单元测试,它可以在软件的初期就发现很

收稿日期:2021-11-18;　**修回日期:**2021-12-15。
基金项目:国家自然科学基金项目(U1736110)。
作者简介:刘堂臣(1996-),男,湖北襄阳人,硕士,主要从事软件测试、程序分析方向的研究。
宫云战(1962-),男,山东威海人,教授,博导,主要从事软件测试、软件可靠性方向的研究。
通讯作者:王雅文(1983-),女,陕西凤翔人,副教授,博导,主要从事软件测试、程序分析方向的研究。
引用格式:刘堂臣,王雅文,宫云战.大规模C++工程单元测试性能优化研究[J].计算机测量与控制,2022,30(2):17-23.

多故障，并且修改它们的成本也很低[1]。单元测试的效果会在很大程度上影响到后阶段的测试，会进一步影响到产品的开发时间、费用和质量[2]。

作者所在实验室一直从事着软件测试的研究工作，现有的代码测试系统（CTS－CPP，code testing system for c plus plus）是一款基于JAVA语言编写的测试C＋＋工程的自动化单元测试工具，它可以对被测工程进行以函数为单元的模块划分，进而对各单元进行自动测试并生成测试用例，依据语句、分支和修订条件判定覆盖准则统计出覆盖率。CTS－CPP可以测试单个文件和小规模的C＋＋工程，但是在测试大规模C＋＋工程（含有多个C＋＋文件的工程，一般超过20个总代码行数超过5 000行）时总是会出现内存溢出故障。经分析，CTS－CPP在测试C＋＋工程时会为每一个CPP文件生成一个分析文件类对象（下文中简称被测对象），当生成的被测对象过多时将会消耗掉虚拟机所有内存从而导致内存溢出故障。如果将生成的被测对象全部序列化到磁盘中，需要用到的时候再进行反序列化操作，这样是可以解决内存溢出的故障，但是对每一个CPP文件都会对应一对I/O操作，频繁的I/O操作将会大大增加系统的测试时间[2]。

为了解决自动化单元测试工具中存在的以上问题，在测试流程中引入了缓存优化[3-7]技术，通过将部分被测对象存储到内存中以提升访问的速度。为了保证缓存的高命中率和高效性，提出了一种面向不同测试方式的缓存优化方法。当用户直接对整个工程进行测试时，采用缓存预取的方法增加缓存的命中率；当用户对单个文件进行测试时，采用改进的GDSF算法（GDSF－UT，GDSF for unit test）进行缓存的替换。实验表明，面向不同测试方式的缓存优化方法能够有效避免内存溢出故障，提升缓存的命中率，减少测试所需的时间。

本文第1节对国内外系统性能优化的相关工作进行概述；第2节介绍缓存预取模型设计；第3节设计并实现GDSF－UT缓存替换算法；第4节介绍方法应用后的实验结果分析；第5节总结全文。

1 相关工作

随着新技术的不断发展，软件规模也日益增大[8]，软件的性能优化研究已成为国内外许多学者的重点研究方向，其中通过缓存优化技术来提升系统整体的性能已经成为主流的优化方式之一。

目前国内外有很多学者都在从事缓存优化的研究工作。杨冬菊[9]等人为了保证在高并发、大用户流量的场景下身份认证系统能够稳定高效的运行，提出了基于缓存的分布式统一身份认证机制。它通过将热点数据预存到缓存中以提高响应的速度，并结合复杂多样的用户行为提出了基于Hybrid的多因素缓存替换算法。胡森森[10]等人通过总结超长指令字处理器发射宽度动态变化的特点，并利用其动态特征来驱动缓存的重构，从而达到合并处理器核或动态分离的目的。王永功[11]等人分析了信息中心网络的架构，指出多跳LRU缓存中“缓存退化”的问题，提出一种基于预过滤的O（1）复杂度的改进算法，极大的增强了内容分发效率。张艳[12]等人比较并分析传统缓存模型MRM和IRM的思想，基于字节代价以及相对流行度的概念，提出满足延迟时间、命中率和字节命中率等多种性能指标要求的缓存优化模型，并给出相关算法。刘磊[13]等人在研究缓存技术的过程中，提出了一种最小驻留价值的缓存替换算法，该算法结合对象大小和访问频率进行驻留价值的计算，优先选取价值最小的对象进行替换，但是其忽略了缓存的使用时间和物理资源的使用情况。

综上所述，针对缓存的研究，一方面是对缓存内容和实现方法的研究，另一方面是对缓存替换算法的研究[14-15]。作者对单元测试工具的优化也将从以上两方面进行。

2 缓存预取模型的设计

当用户选择直接对整个工程进行测试时，为了尽量减少I/O等待的时间，本节设计了一种缓存预取的模型，它借助一个缓存队列将缓存预取模块和文件测试模块彻底解耦，使得原来只能串行执行的两个模块现在可以并行执行。

2.1 模型构成

结合生产者－消费者的思想，本文设计的缓存预取模型如图1所示。

该模型主要由以下6部分组成：

1）控制器主要负责各部件之间的任务调度和操作控制，并且负责与文件测试模块之间的交互，使各部件有序稳定的运行。

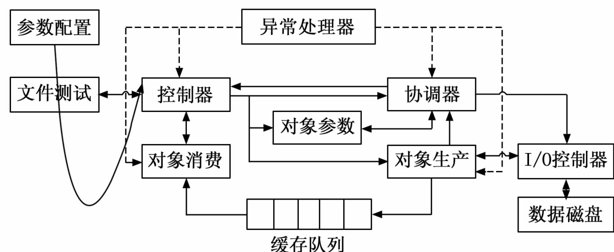

图 1　缓存预取模型的基本构成

2）协调器主要负责接收对象生产部件和对象消费部件发送过来的数据，通过数据协调双方的运行，并在参数对象中记录数据。

3）对象生产部件主要负责预先批量的从磁盘中读取对象放入缓存队列中，并记录读取对象的个数和所消耗的时间，将数据发送给协调器。

4）对象消费部件主要负责从缓存队列中读取对象交给控制器处理，并记录读取对象的个数和测试所花的时间。

5）缓存队列主要是用来存放预取出来的对象，其大小支持配置，避免对虚拟机内存的过度使用从而导致内存溢出故障。同时，它还支持同步阻塞，例如缓存队列为空时，消费方将一直处于阻塞状态，直到生产方调入下一批预取对象。

6）异常处理器主要负责对各部件运行过程中出现的异常进行捕捉处理。

2.2　预取机制

2.2.1　相关定义及符号表示

定义 1：预取对象的总个数：指在测试 C＋＋工程时，需要生成的被测对象的总个数，记为 CT。由于一个 CPP 文件对应一个被测对象，所以该参数等于被测工程中 CPP 文件的个数。

定义 2：缓存队列的大小：指缓存队列中最多能存放的被测对象的个数，记为 QS。此参数限制了虚拟机内存的使用量，避免发生内存溢出故障。

定义 3：生产总批次：指从磁盘中批量预取被测对象的总次数，记为 PT。为了尽量减少 I/O 的次数，约定当磁盘中未取被测对象个数大于等于 QS 时，按 QS 大小来预取。因此在数量上 PT 可以表示为：

$$PT = \left|\frac{CT}{QS}\right| \tag{1}$$

定义 4：每批生产被测对的数量：指每批从磁盘中预取到缓存队列中被测对象的个数，记为 QS_i，其中，i 表示生产批次（$1 \leqslant i \leqslant PT$）。在数量上，$QS_i$ 可以表示为：

$$QS_i = \begin{cases} QS & 1 \leqslant i \leqslant PT-1 \\ CT-(PT-1)QS & i = PT \end{cases} \tag{2}$$

定义 5：每次消费的被测对象个数：指每次从缓存队列中取出的被测对象的个数，记为 CO。由于每次都是从缓存队列中取出一个被测对象用于测试，测试完毕后再取下一个，所以该参数的大小恒为 1，可以表示为：

$$CO = 1 \tag{3}$$

定义 6：生产一批被测对象的开始时间和结束时间：指每次批量从磁盘中读取被测对象的开始时间和结束时间，分别记为 TPS_i 和 TPE_i，其中，i 表示生产批次（$1 \leqslant i \leqslant PT$）。

定义 7：生产一批被测对象所消耗的时间：指从磁盘预取一批被测对象到缓存队列中所消耗的时间，记为 ΔTP_i，其中，i 表示生产批次（$1 \leqslant i \leqslant PT$）。在数量上，该参数等于每批生产的结束时间与每批生产的开始时间的差值，可以表示为：

$$\Delta TP_i = TPE_i - TPS_i \tag{4}$$

定义 8：消费一个被测对象的开始时间和结束时间：指从缓存队列中取出一个被测对象的时间和被测对象测试完毕的时间，分别记为 $TCS_{i,j}$ 和 $TCE_{i,j}$，其中，i 表示生产批次（$1 \leqslant i \leqslant PT$），$j$ 表示缓存队列中的第 j 个元素（$1 \leqslant j \leqslant QS_i$）。

定义 9：消费一个被测对象所需的时间：指消费方从缓存队列中取出被测对象到测试完毕所消耗的时间，记为 $\Delta TC_{i,j}$，其中，i 表示生产批次（$1 \leqslant i \leqslant PT$），$j$ 表示缓存队列中的第 j 个元素（$1 \leqslant j \leqslant QS_i$）。在数量上，该参数等于消费一个被测对象的结束时间与开始时间的差值，可以表示为：

$$\Delta TC_{i,j} = TCE_{i,j} - TCS_{i,j} \tag{5}$$

定义 10：消费一批被测对象所需的时间：指从缓存队列中取出第一个被测对象开始到同一批被测对象全部测试完毕所消耗的总时间，记为 ΔTC_i，其中，i 表示生产批次（$1 \leqslant i \leqslant PT$）。如图 2 所示，$\Delta TC_{i,1}$ 表示消费第 i 批第 1 个被测对象所需的时间，$\Delta TC_{i,2}$ 表示消费第 i 批第 2 个被测对象所需的时间，以此类推，$\Delta TC_{i,j}$（$1 \leqslant j \leqslant QS_i$）表示消费第 i 批第 j 个被测

对象所需的时间，该参数表示它们的总和。

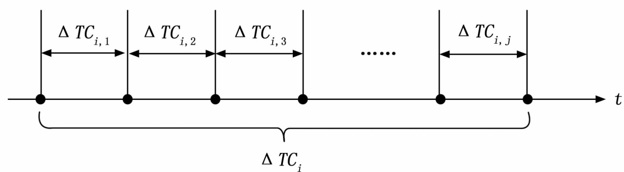

图2 批量消费与单个消费所需时间关系示意图

在数量上可以表示为：

$$\Delta TC_i = \sum_{j=1}^{QS_i} \Delta TC_{i,j} \tag{6}$$

2.2.2 等待时间

缓存预取模型设计的核心目的就是尽量减少等待I/O的时间，为了达到这一目的，模型采取了两点关键的设计：1）将原来串行执行的缓存预取模块和文件测试模块通过一个缓存队列实现解耦，使得两模块在一定程度上可以并行执行；2）采用批量预取的方式，减少I/O的次数。

消费方在消费一批被测对象之前需要先等待生产方生产完毕，这一等待时间即为CPU等待I/O的时间，记为ΔTW_i,其中,i表示生产批次（$1 \leqslant i \leqslant PT$）。如图3所示,$\Delta TP_1$表示生产第1批被测对象所消耗的时间，此时，缓存队列还为空，消费方需要等待ΔTW_1的时间，然后生产方和消费方开始并行执行，ΔTP_2表示生产第2批被测对象所消耗的时间,ΔTC_1表示消费第1批被测对象所需要的时间，它们的差值即为ΔTW_2,如果$\Delta TC_1 \geqslant \Delta TP_2$,说明消费第1批比生产第2批花费的时间更长,此时消费完第1批第2批已经生产完毕,因此ΔTW_2等于0。以此类推,ΔTP_i表示生产第i批被测对象所消耗的时间；ΔTC_{i-1}表示消费第$i-1$批被测对象所需要的时间，它们的差值即为ΔTW_i,如果$\Delta TC_{i-1} \geqslant \Delta TP_i$,说明消费第$i-1$批比生产第$i$批花费的时间更长，此时消费完第$i-1$批第$i$批已经生产完毕，因此$\Delta TW_i$等于0。

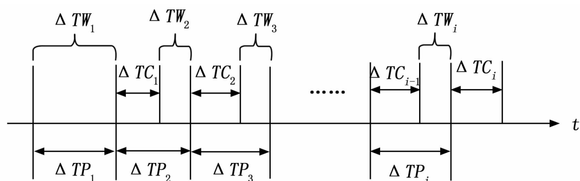

图3 生产时间消费时间等待时间关系示意图

ΔTW_i在数量上可以表示为：

$$\Delta TW_i = \begin{cases} \Delta TP_i & i = 1 \\ \Delta TP_i - \Delta TC_{i-1} & \text{条件 2} \\ 0 & \text{条件 3} \end{cases} \tag{7}$$

条件2：$2 \leqslant i \leqslant PT \&\& \Delta TP_i > \Delta TC_{i-1}$；

条件3：$2 \leqslant i \leqslant PT \&\& \Delta TP_i \leqslant \Delta TC_{i-1}$；

整个测试流程等待I/O的总时间即为各批次等待I/O时间之和，记为ΔTW。在数量上，可以表示为：

$$\Delta TW = \sum_{i=1}^{PT} \Delta TW_i \tag{8}$$

2.2.3 预取流程

当单元测试工具启动对一个工程进行单元测试时，缓存预取模型的运行流程如下：

1）通过参数配置模块配置缓存队列的大小，通过控制器将配置的参数写入到参数对象中；

2）消费方和生产方同时启动，此时缓存队列为空，消费方进入阻塞状态，生产方开始预取数据，记录开始时间和结束时间，通过公式（4）计算出生产当前批被测对象所消耗的时间ΔTP_1；

3）消费方阻塞ΔTW_1(数值上等于ΔTP_1）后被唤醒,与生产方开始并行执行；

4）消费方从缓存队列中一个一个取出被测对象并进行单元测试，记录每一个被测对象消费的开始时间和结束时间，通过公式（5）计算出消费当前被测对象所需的时间，通过公式（6）进一步计算出消费当前批被测对象所需的时间ΔTC_{i-1}($2 \leqslant i \leqslant PT$)；

5）生产方与步骤4）并行执行，预取下一批的被测对象，记录预取的开始时间和结束时间，通过公式（4）计算出生产下一批被测对象所消耗的时间ΔTP_i($2 \leqslant i \leqslant PT$)；

6）根据公式（7）计算出当前批的等待时间ΔTW_i；

7）重复执行步骤4）～6)，直到$i = PT$；

8）根据公式（8）计算出整个流程中等待I/O的时间。

3 缓存替换算法的设计

当用户选择对工程下的单个文件进行测试时，如果选中文件的被测对象恰好在缓存中，将会少一次I/O操作，从而节省测试的时间。所以一个高命中率的缓存替换算法就显得尤为重要。本节在目前广泛被

使用的 GDSF 缓存替换算法的基础上，进一步考虑了单元测试的空间局部性特征和测试结果等因素的影响，提出了 GDSF－UT 缓存替换算法。

3.1　GDSF 算法

GDSF 算法[16-18]是缓存替换算法中考虑影响因素相对比较全面的一种算法，相对传统考虑影响因素比较单一的算法具有较高的命中率，因此也被广泛的使用。它综合考虑了对象的大小、对象访问频率、时间和引入对象到缓存所需的代价的影响[19]，通过这些影响因素计算缓存中对象的权重，当缓存空间满时，替换出缓存中权重最小的对象[20]。计算公式为：

$$W(O) = L + F_{\text{freq}}(O) \cdot \frac{Cost(O)}{Size(O)} \tag{9}$$

式中，O 为缓存对象；$W(O)$ 代表缓存对象的权重；L 为膨胀因子，初始值为 0，每次需要缓存替换时，L 被重新赋值为当前缓存中最小的权重值；$F_{\text{freq}}(O)$ 为缓存对象的访问频次，初始值为 1，之后每命中一次其值加 1；$Size(O)$ 为缓存对象的大小；$Cost(O)$ 为将对象从磁盘取回到缓存所需的代价。

虽然 GDSF 算法考虑了比较多的影响因素，且容易实现，计算开销也比较小，但是用在单元测试工具中就缺乏对测试结果、测试行为特征等因素的考虑，接下来将在此算法的基础上设计面向单元测试工具的缓存替换算法 GDSF－UT。

3.2　GDSF－UT 算法

3.2.1　相关概念

设 $\boldsymbol{S}=\{O_1, O_2, O_3, \cdots, O_m, \cdots, O_n\}$ 表示被测对象的集合，其下标 $1\cdots n$ 表示在文件系统中的相对位置。

定义 1：对象距离：指两个被测对象在文件系统中相对位置的差值，记为 D_{mn}，其中，m、n 分别表示两个被测对象在文件系统中的相对位置。在数值上，该参数等于 m 与 n 差值的绝对值，可以表示为：

$$D_{mn} = |m - n| \tag{10}$$

定义 2：分支覆盖率：指程序中被测试执行的判定和分支数占判定和分支总数的比率，记为 BCR。

定义 3：语句覆盖率：指程序中被测试执行的语句数占可执行的语句总数的比率，记为 SCR。

定义 4：修订条件/判定覆盖率：简称 MC/DC，它是一种特殊的分支覆盖率，它不但会使用分支覆盖率报告复杂条件下的 TRUE 和 FALSE 输出，同时也会报告复杂条件下的全部分支条件输出，记为 $MDCR$。

3.2.2　算法描述

用户在对单个被测对象进行测试时的行为特征包括：所测试的对象具有很强的空间局部性，即一段时间内连续多次的测试行为通常只发生在相邻的一些对象中，也就是对象距离越小，被选中测试的概率就越大；测试结果的好坏也将会影响再次被选中测试的概率（测试结果的好坏由定义 2、3、4 给出的覆盖率来评判），测试结果越好，再次被选中测试的概率就越小，测试结果越差，再次被选中测试的概率就越大。此部分的影响因素用 P 来表示，计算方式如下：

$$P(O_i, O_j) = \frac{1}{e^{(BCR(O_i)+SCR(O_i)+MDCR(O_i))}} \cdot \frac{1}{1+D_{ij}} \tag{11}$$

式中，O_i 为缓存权重计算对象，O_j 为访问对象。

结合用户测试行为特征的影响，改进后的算法 GDSF－UT 的权重计算公式为：

$$W(O_i, O_j) = L + F_{\text{freq}}(O_i) \cdot \frac{Cost(O_i)}{Size(O_i)} \cdot P(O_i, O_j) \tag{12}$$

GDSF－UT 算法在原有算法基础上进一步考虑了用户测试行为特征的影响，由此进一步优化了缓存权重的计算公式，可以看出改进后的算法在单元测试工具中具有较强的适应性。

3.2.3　算法流程

GDSF－UT 算法的请求处理流程如图 4 所示。

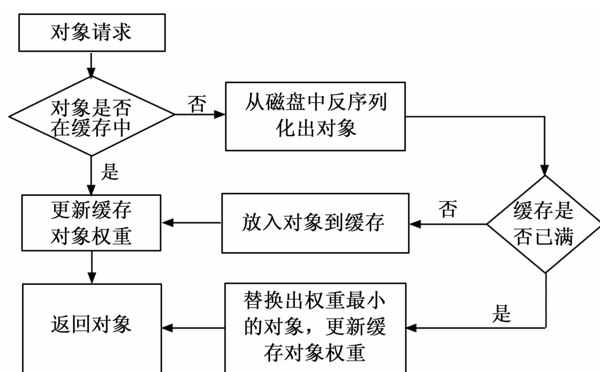

图 4　GDSF－UT 算法的请求处理流程图

由图 4 可以看出，当用户发出对象请求时，首先判断缓存中是否有该对象。如果有，更新缓存对象的权重并返回对象；如果没有，则需要从磁盘中反序列化出对象，然后判断缓存是否已满。如果已满，替换出权重最小的对象并更新缓存对象的权重，然后返

回对象；如果缓存没有满，则直接将对象放入缓存并更新缓存对象的权重，然后返回对象。

4　实验及实验结果分析

为了验证缓存预取模型和 GDSF－UT 替换算法的有效性，本文选取了 5 个不同规模的开源 C＋＋工程在 CTS－CPP 中进行测试实验，它们的工程属性如表 1 所示。

表 1　C＋＋代码工程的属性列表

工程名	源文件个数/个	头文件个数/个	文件总数/个	代码行数/行
mapreduce	51	243	294	3 661
cppast	79	397	476	13 659
libdynd	239	1 513	1 752	29 812
symengine	197	1 085	1 282	63 675
magnum	627	4 965	5 592	164 472

4.1　实验环境

本文实验中，CTS－CPP 运行在 lenovo 台式机上，CPU 型号为 Intel（R）Core（TM）i5－1038NG7，CPU 频率为 3.80 GHz，物理内存为 8 GB，操作系统为 Windows 10 专业版，编译器为 VC14，开发平台为 Eclipse，开发语言为 Java，虚拟机最大内存设置为 512 M。

4.2　实验结果及分析

4.2.1　缓存预取模型对性能的影响

为了评估缓存预取模型对 CTS－CPP 性能的影响，本文对 5 个不同规模的测试工程进行了工程级别的测试，分别记录了优化前、每次从磁盘读取和采用预取机制 3 种情况下对整个工程进行模块划分所需的时间，同时也统计出了每次从磁盘读取和采用预取机制 2 种情况下 CPU 等待 I/O 的总时间，具体实验结果如表 2 所示，表中的 OOM 代表发生内存溢出故障。为了避免缓存队列大小对实验结果的影响，本文实验将其设置为 60。

从表 2 可以发现：当被测工程规模较小，即内存中能存放下所有被测对象，优化前的方案模块划分的速度是最快的；当被测工程规模越来越大，优化前的方案将会发生内存溢出故障，采用每次从磁盘读取和预取机制可以有效避免内存溢出故障，但模块划分的时间也越来越长，采用预取机制相比每次从磁盘读取平均节约 15%左右的时间；CPU 等待 I/O 的总时间

表 2　缓存预取模型对性能的影响评估

工程名	对整个工程进行模块划分所需时间/ms			节省时间百分比/%	预取次数	CPU 等待 I/O 的总时间/ms	
	优化前	每次从磁盘读取	采用预取机制			每次从磁盘读取	采用预取机制
mapreduce	2 626	4 408	3 658	17	1	1 762	1 012
cppast	OOM	16 446	14 107	14	2	3 954	1 815
libdynd	OOM	47 641	40 467	15	4	8 365	2 371
symengine	OOM	116 835	107 403	8	4	9 850	2 418
magnum	OOM	251 387	230 586	8	11	2 1745	2 257

方面，每次从磁盘读取等待的总时间随着工程规模增大而变长，而采用预取机制的等待总时间稳定在第一批被测对象预取完成所需的时间附近，因为由实验结果发现，测试完上一批所需的时间远大于下一批的预取时间，因此测试完上一批后不需要等待 I/O，即 $\Delta TW_i=0$（$2\leqslant i\leqslant PT$），随着被测工程规模的增大，两种方案等待 I/O 总时间的差值就越大，采用预取机制节省的时间就越多。

4.2.2　GDSF－UT 替换算法对性能的影响

为了评估 GDSF－UT 替换算法对 CTS－CPP 性能的影响，本文对 5 个不同规模的测试工程进行了文件级别的测试，分别记录了优化前、每次从磁盘读取、采用 GDSF 算法和采用 GDSF－UT 算法 4 种情况下抽样测试单个文件的平均响应时间，同时也统计出了采用 GDSF 算法和采用 GDSF－UT 算法 2 种情况下的缓存命中率，具体实验结果如表 3 所示。

表 3　GDSF－UT 替换算法对性能的影响评估

工程名	抽样测试单个文件的平均覆盖率/%			抽样测试单个文件的平均响应时间/ms				缓存命中率/%	
	语句覆盖	分支覆盖	MC/DC 覆盖	优化前	每次从磁盘读取	采用 GDSF 算法	采用 GDSF－UT 算法	采用 GDSF 算法	采用 GDSF－UT 算法
map－reduce	88	80	75	49	86	51	51	100	100
cppast	87	77	72	OOM	206	147	139	79	83
libd－ynd	90	86	80	OOM	199	171	163	41	50
sym－engine	73	67	58	OOM	596	551	537	44	49
mag－num	85	75	70	OOM	401	381	373	23	27

从表 3 可以发现：当被测工程规模较小，即内存中能存放下所有被测对象，此时的缓存命中率都为 100%，优化前、采用 GDSF 算法和采用 GDSF－UT 算法测试单个文件的平均响应时间几乎相等；针对同一被测工程，在缓存放不下所有被测对象前提下，每次从磁盘读取、采用 GDSF 算法和采用 GDSF－UT 算法测试单个文件的平均响应时间依次减小；随着被测工程规模越来越大，由于缓存的大小一定，所以缓存命中率越来越低，但 GDSF－UT 算法的命中率比 GDSF 算法的命中率平均高出 4～5 个百分点。

5　结束语

针对单元测试工具无法测试大规模 C＋＋工程且耗时较长这一问题，在测试流程中引入了缓存优化技术，并提出了一种面向不同测试方式的缓存优化方法。对于工程级别的测试，本文提出了一种缓存预取模型；对于文件级别的测试，本文提出了改进的 GDSF 缓存替换算法。实验结果表明，该方法使得被测工程规模从 5 000 多行扩大至十几万行，大大提升了单元测试工具的性能，使其在软件规模迅速膨胀的今天具有更强的适用性。同时，该方法提升了单元测试工具的响应速度，缩短了测试所需要的时间，为单元测试工作节省了人力成本和时间成本，具有较高的应用价值。在下一步研究工作中，由于被测工程规模较大，生成的测试用例数量将会急剧增加，要执行所有的测试用例几乎是不可能的，因此，接下来将重点研究测试用例约简方法。

参考文献：

[1] 宫云战，赵瑞莲，张　威，等．软件测试教程［M］．北京：机械工业出版社，2008.

[2] 徐教显，王雅文．基于缓存估算模型的代码测试系统性能优化方法［J］．软件，2013，34（12）：10－13.

[3] 张秋平，孙　胜，刘　敏，等．面向多边缘设备协作的任务卸载和服务缓存在线联合优化机制［J］．计算机研究与发展，2021，58（6）：1318－1339.

[4] 李　源，何友全．WebGIS 中带图业务数据的缓存和预取机制研究［J］．计算机测量与控制，2016，24（5）：209－212.

[5] 翟　月．面向数据卸载的 5G 分布式缓存优化研究［J］．佳木斯大学学报（自然科学版），2020，38（6）：127－129，148.

[6] 赵永乐，吴坤芳．结合流行度选择的集群网络高速缓存优化仿真［J］．计算机仿真，2020，37（9）：262－265.

[7] 唐　榜，吴　珏，杨福军，等．基于高斯混合模型的 Web 代理服务器缓存替换策略［J］．计算机测量与控制，2021，29（2）：166－170，175.

[8] 朱　彬．企业级应用软件架构模式的研究和应用［D］．沈阳：沈阳工业大学，2004.

[9] 杨冬菊，冯　凯．基于缓存的分布式统一身份认证优化机制研究［J］．计算机科学，2018，45（3）：302－306，312.

[10] 胡森森，苏加福．自适应运行时可重构缓存优化［J］．计算机工程与应用，2018，54（4）：25－30，89.

[11] 王永功，李振宇，武庆华，等．信息中心网络内缓存替换算法性能分析与优化［J］．计算机研究与发展，2015，52（9）：2046－2055.

[12] 张　艳，石　磊，卫　琳．Web 缓存优化模型研究［J］．计算机工程，2009，35（8）：85－87，90.

[13] 刘　磊，熊小鹏．最小驻留价值缓存替换算法［J］．计算机应用，2013，33（4）：1018－1022.

[14] 王　波，胡军台，肖承仟，等．基于通告转移机制的 CCN 网络缓存替换策略［J］．计算机应用与软件，2020，37（6）：148－153.

[15] 刘翠梅，杨　璇，贾刚勇，等．一种代价感知的细粒度闪存缓冲区替换算法［J］．小型微型计算机系统，2019，40（5）：972－977.

[16] CHAO W. Web cache intelligent replacement strategy combined with GDSF and SVM network re－accessed probability prediction［J］. Journal of Ambient Intelligence and Humanized Computing，2020，11（2）：581－587.

[17] 黎慧源，易国洪，代　瑜，等．贪婪双尺寸频率算法的优化与改进［J］．武汉工程大学学报，2018，40（6）：685－690.

[18] SONGWATTANA A，THEERAMUNKONG T，VINH P C. A learning－based approach for web cache management［J］. Mobile Networks and Applications，2014，19（2）：258－271.

[19] 杜建海，吕江花，高世伟，等．面向航天器综合测试系统的 Web 缓存替换策略［J］．北京航空航天大学学报，2018，44（8）：1609－1619.

[20] 曹　旻，刘文中．混合架构下多请求模式的缓存替换模型研究［J］．计算机科学，2015，42（6）：175－180.