动态资源感知的并行化模糊测试框架*

廉 美[1,2]，邹燕燕[1]，霍 玮[1]，邹 维[1]

(1. 中国科学院信息工程研究所 网络安全防护技术北京市重点实验室，北京 100093；2. 中国科学院大学，北京 100049)

摘 要：针对现有的并行模糊测试在测试效率、资源利用率以及异常处理上的局限性，围绕测试资源的生成、使用及容错三个方面提出了一种动态资源感知的系统化解决方案。针对测试环境在大规模和多场景两个维度快速搭建的需求，提出一种基于云平台的动态构建方法，加快测试环境部署，提高有效 fuzz 时间；针对并行模糊测试中资源利用率低的问题，提出一种多层次并行度动态调整的资源配置策略，优化整体测试资源配置并提高单机负载；针对大规模并行测试中节点易发生故障的问题，提出基于优先级调度的容错处理方法。最后，设计并实现了一个基于四级流水线并行处理结构的通用模糊测试框架。实验证明，该框架能够有效提高并行模糊测试的测试效率和资源利用率，实现系统的有效容错。

关键词：漏洞挖掘；并行模糊测试；资源感知；测试框架；云平台

中图分类号：TP302.7　　**文献标志码**：A　　**文章编号**：1001-3695(2017)01-0052-06

doi:10.3969/j.issn.1001-3695.2017.01.010

Dynamic resource awareness framework for parallel fuzzing

Lian Mei[1,2], Zou Yanyan[1], Huo Wei[1], Zou Wei[1]

(1. *Beijing Key Laboratory of Network Security & Proctection Technology, Institute of Information Engineering, Chinese Academy of Sciences, Beijing* 100093, *China*; 2. *University of Chinese Academy of Sciences, Beijing* 100049, *China*)

Abstract: It is not trivial to apply current parallel fuzzing techniques on large scale cluster directly, due to its limitation on testing efficiency, resource usage and system fault tolerance. This paper proposed a dynamic resource-aware approach, which could systematically tackle the problem from the generation, the usage and the fault tolerance of testing resources. To building the large scale testing cluster quickly, it described a cloud-based dynamic building approach to reduce the generation time. For the low utilization problem of testing resources, it proposed a multi-level dynamic scheduling police to improve the overall recourse usage and the single node workload. In order to make test continually, it proposed a priority-based fault tolerant method. Finally, it finished a general parallel fuzzing framework, which based on a four-stage pipeline structure. The experiment shows that the framework is very effective in respect to the parallel efficiency, resource usage, and fault tolerance.

Key words: vulnerability; parallel fuzzing test; resource awareness; testing framework; cloud infrastructure

0 引言

模糊测试(fuzz testing)[1]作为一种有效的漏洞挖掘技术，它通过反复操纵目标软件，为其提供大量非预期的输入，并监视软件执行异常来发现软件故障。现有的模糊测试主要采用两类技术，即基于生成的模糊测试技术和基于变异的模糊测试技术。这两类技术都是基于目标程序对输入验证的不完整性假设，采用相对随机的方法，在基于假设的输入状态空间中探索能够使目标程序发生异常的输入。由于这种基于假设的输入状态空间往往巨大，再加上这两类技术本身具有的随机特性，使得模糊测试的实际运行效果在很大程度上依赖于生成的测试用例数量。对于传统的单机模糊测试，要获得一个路径覆盖率或代码覆盖率较高的测试集，需要花费大量的测试时间，这就导致了单机模糊测试在单位时间内的异常触发率低的问题。对此，国内外的研究者通过引入并行技术来缩短模糊测试执行的总体时间，提高模糊测试的效率[2~21]。但现有的并行模糊测试在任务分发管理、调度与容错处理方面存在不足，导致测试效率、资源利用率不高，没有充分展现并行测试的优势。并且随着模糊测试技术在实际中的应用越来越广泛，测试目标的范围也在不断扩大，不同测试目标的运行环境又各不相同，这就产生了多系统或同一系统不同配置的多测试环境需求。

本文提出的基于云平台的动态资源感知的并行化模糊测试框架，实现了模糊测试环境的半自动构建，可提高有效 Fuzz 的时间，满足不同测试目标的多测试环境需求。在测试任务的并行处理方面，框架设计实现了四级流水线并行处理结构，能够有效提高对大规模测试任务的并行处理效率，并基于此结构实现了对多层次并行度动态调整的资源配置策略以及系统容错的支撑。本文介绍了并行模糊测试技术的研究现状，从测试资源的生成、使用及容错三个方面介绍本文提出的模糊测试并行化技术和框架总体实现。

收稿日期：2015-12-09；**修回日期**：2016-02-19　　**基金项目**：国家“863”计划资助项目(2015AA017202)

作者简介：廉美(1990-)，女，山东临沂人，硕士，主要研究方向为并行模糊测试、基于字体的漏洞挖掘与分析(lianmei@iie.ac.cn)；邹燕燕(1989-)，女，硕士，主要研究方向为软件脆弱性分析、Web 安全；霍玮(1981-)，男，副研究员，博士，主要研究方向为软件安全、脆弱性分析、移动安全、Web 安全等；邹维(1964-)，男，研究员，博导，主要研究方向为软件安全、脆弱性分析、风险分析等.

1　并行模糊测试发展现状

在并行模糊测试方面,国内外的研究主要集中在对特定算法或测试方法的并行化上。国内的研究中,针对白盒模糊测试中符号执行[22]面临的路径爆炸问题,吴世忠等人[3]提出基于路径簇规约的并行符号执行方法,Wu 等人[4]基于 S2E 平台[23]实现了一种分布式的符号执行方法;余啸[5]通过对动态符号执行中测试数据生成效率低的问题进行研究,提出并行的动态测试数据生成技术;为了提高针对特定目标的测试覆盖率,曹琰[6]提出了基于敏感点逼近的并行测试方法。国外的研究中,Boyapati 等人[7]提出了基于约束的复杂结构化测试用例生成算法,Misailovic 等人[8]在此基础上添加了生成并行扩展支持算法和执行并行扩展支持算法;Siddiqui 等人[9]在 Boyapati 研究的基础上,实现了复杂测试用例的并行生成过程。以上工作或是针对一种具体算法,或是针对一种测试方法,设计实现时考虑基于算法或测试方法本身的特征,其适用的测试目标范围和并行扩展性存在较大的局限。在并行模糊测试工具的研究上,梁洪亮等人[10]基于混合符号执行路径取反算法和复合化的用例生成方法,设计实现了并行化智能模糊测试系统——谛听。王连赢[11]基于静态分析的二进制代码脆弱性评估方法,实现了基于异步编程模型的分布式文件 Fuzzing 系统。微软的 Godefroid 等人[12,13]基于动态插桩和混合符号执行技术开发了面向文件解析类程序的并行化模糊测试工具 SAGE。Bucur 等人[14,15]提出了基于计算节点簇的可线性扩展的并行符号执行方法,实现了分布式测试系统 Cloud9。类似的支持并行的模糊测试工具还有开源框架 Sulley[16]、peach[17],商业工具如 Google 的 ClusterFuzz[18]、Codenomicon 的 Defensics 3.0[19]以及 peach 的企业版 PeachE[20]。此类工具主要用于发现特定测试目标中的漏洞,如 ClusterFuzz 被用于挖掘 chrome 中的漏洞,peach 则主要用于发现文件解析类软件中的漏洞。本文提出的框架主要面向灰盒和黑盒模糊测试,强调框架对不同模糊器和测试目标的通用性。

与本文工作比较相近的是 Xie[21]的工作,他提出基于网络资源将网格计算用于大规模并行模糊测试的想法,并实现了对大规模测试任务的分发、测试以及数据收集等整个测试流程。但该工作基于已有网络资源,必然存在目标软件环境需求和现有测试环境不一致的问题,所以该测试系统支持的目标软件类型较为单一。同时这种开放的网络环境会给测试带来很多不确定的干扰因素;针对测试过程中的异常,也没有给出合理的解决办法。

综合考虑现有模糊测试中存在的问题,本文提出了基于云平台的动态资源感知的并行化模糊测试框架:a)实现了测试环境的半自动构建,解决了随测试对象范围扩大带来的多测试环境需求问题;b)采用多层次并行度动态调整的资源配置策略,能够有效提高对测试资源的利用率;c)针对大规模并行中的常见节点出错,提出了基于优先级调度的容错处理过程。

2　模糊测试并行化技术

在模糊测试的实际应用场景中,随测试对象范围的不断扩大,测试环境的需求更加多样化,如何实现不同测试环境的快速构建、多测试环境下的基础资源共享和高效利用以及大规模并行节点间的有效容错,是实现大规模并行模糊测试的关键。本节从测试资源的生成、使用及容错三个方面介绍了本文提出的动态资源感知的模糊测试并行化技术(图 1)。

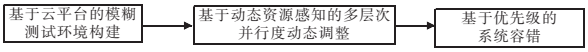

图 1　模糊测试并行化技术

2.1　基础环境构建

本文基于实验室内部私有云,实现了测试环境的半自动构建,能够有效满足大规模、多样化的测试环境快速搭建的需求,也为多测试环境下的基础资源共享提供了支撑。

首先将完成一次模糊测试所需环境划分为硬件和软件环境。硬件环境由完成模糊测试必需的物理资源组成,一般可由三元组 Eh = (Ncpu, Cmem, Cdisk) 描述,分别代表单台机器处理器核数、内存容量、硬盘容量。软件环境一般可由四元组 Es = (Tos, Tprog, Tfuzzer, Iconf) 描述,分别代表操作系统类型(不同的子版本表示不同的系统类型)、目标应用程序、模糊器类型、模糊器配置。整个模糊测试环境可由 Eh 和 Es 联合描述:EFuzz = (Eh, Es)。

针对大规模并行模糊测试环境的部署,本文采用先手工配置单机镜像模板后基于模板批量申请部署虚拟机的双阶段方式。第一阶段,支持两种镜像模板配置方式:a)在一台独立的装有 KVM[24]的物理机上配置镜像(qcow2 格式[25]);b)在已申请的云虚拟机中配置后导出作为新的镜像模板。前者适用于硬件配置或者操作系统类型前后有变化的情形,即 $Eh1 \neq Eh2 \vee Tos1 \neq Tos2$;后者适用于仅目标程序、模糊器或者模糊器配置前后有变化的情形,即 $Eh1 = Eh2 \wedge Tos1 = Tos2 \wedge (Tprog1 \neq Tprog2 \vee Tfuzzer1 \neq Tfuzzer2 \vee Iconf1 \neq Iconf2)$。该阶段可有效解决现有并行模糊测试中的多测试环境需求问题。第二阶段,基于云平台提供的基础测试资源,以第一阶段配置好的镜像为模板,调用平台接口实现大规模并行模糊测试节点的自动化部署。该资源获取方式使得不同环境需求的测试任务间共享资源成为可能。以上两阶段联合实现了模糊测试环境的半自动化构建。实践证明,相比纯手工的方法,这种方法大大节省了测试环境的构建时间和劳动力成本,也能够支持前面所提到的多测试环境的应用场景,满足简便快速的搭建需求。

2.2　多层次并行度动态调整

有多个独立测试任务并存的大规模并行化模糊测试应用场景中,如何合理调配基础测试资源,实现有限资源的最大化利用,是并行模糊测试的一个关键。在资源利用上,现有并行模糊测试主要存在两方面问题:a)单机负载过低;b)资源调配不合理,整机资源浪费严重。单机负载主要受单机环境配置、模糊器以及测试目标的影响。整机资源利用率则取决于基础测试环境和系统框架的动态扩展性。本文基于云平台构建模糊测试环境,通过资源释放、资源再申请的方式可实现多测试环境下的基础资源共享。基于这一前提,本文提出了多层次并行度动态调整的资源配置策略,从节点资源的有效利用和节点内部的并行扩展两个层面,提高了测试资源的利用率。

测试节点层面的并行扩展主要基于对空闲测试资源的实时检测和下文实现部分提到的四级流水线并行处理结构。前者用于获取可用虚拟机节点,后者可实现对该节点的任务分配、调度以及运行过程监测。本章重点介绍节点内部的并行调整。本文将 CPU 使用率作为单机负载的主要衡量因子,并用 VmLoad 表示单机负载。VmLoad 受测试目标、硬件环境以及机

内并行度的共同影响,该影响因素可用四元组 Ptr = 〈Target , Eh,Para,Stable〉描述。其中:Target 代表测试目标;Eh 表示硬件环境;Para 表示机内并行度;布尔变量 Stable 表示 Para 是否为当前测试环境下的最优值。本文通过维护一张由四元组 Ptr 构成的基础表 PTable,实现主控端对测试节点端的机内并行调控。主控端基于各个测试节点的负载反馈实现表中对应项的更新,测试节点端读取参数 Para 及状态 Stable,实现自身的并行控制。主控端算法如算法 1 所示。

算法 1 机内并行度动态调整算法

```
1  while (Running) do
2    Target,Eh = ReadConfig ( "target""eh" )
3    Re = PTable. search( Target,Eh)
4    if (Re) then
5         break
6    Stable = False
7    PTable. insert( Target,Eh,1 ,Stable)
8    while (not Stable) do
9      if( ExecutingPool = ∅) then
10       for each( fnode in ExecutingPool) do
11         vload = FetchVmInfo ( fnode. Vm. Id )
12         LoadList. put (vload)
13       end for
14     end if
15       if (LoadList≠∅) then
16       VmLoad = GetMeanLoad ( LoadList )
17       if (VmLoad≥LThresh ) then
18            Stable = True
19            PTable. update( Target,Eh,Stable)
20            break
21            else
22            Para = ReCalculatePara ( VmLoad,Para)
23            PTable. update (Target,Eh,Para)
24             end if
25         end if
26    end while
27  end while
```

算法 1 描述了在一个并行模糊测试任务执行过程中,主控端基于该任务所有测试节点的负载信息的均值,动态调整单机内并行参数,更新 PTable 表的过程。2 行,获取当前任务对应的测试环境信息:测试目标 Target 和硬件环境配置 Eh。3 行,根据 Target 和 Eh 信息,查询 Ptable 表,如果表中已存在该测试环境下的最优并行度,执行退出;反之,插入新表项,继续执行。8 ~ 26 行,基于负载信息的并行调整过程。ExecutingPool 为已调度资源池,存储正在运行的测试节点信息。9 ~ 14 行,获取 ExecutingPool 中节点的单机负载信息 vload,存储到表 LoadList 中。16 行,计算 LoadList 中所有负载信息的均值 VmLoad,代表当前测试环境下的单机负载。17 行,判断当前单机负载 VmLoad 是否达到阈值 LThresh,若达到阈值,说明当前的并行参数 Para 已是最优,更新 PTable 中的 Stable 状态,动态调整过程结束;反之,22 行,根据当前负载调整 Para 值。上述过程继续执行,直到 VmLoad 达到阈值。

2.3 系统容错

在大规模并行模糊测试系统的实际运行中,随着并行节点数的增加以及云平台网络性能的影响,测试节点无法连接等节点出错的情况普遍存在。同时受模糊器以及测试节点本身性能的制约,测试过程中也会出现进程阻塞、进程异常退出的情况。这种不同应用场景下的不同异常的发生加大了错误判断的难度,也降低了并行模糊测试系统的测试效率。容错作为保障系统可靠性的重要手段,已成为大规模并行测试系统必不可少的基础特性。

针对大规模并行模糊测试中的异常处理,本文提出了基于优先级调度的容错处理方法。通过监测并行模糊测试执行过程,根据运行出错情况对节点进行分类和容错,并赋予其不同的优先级,再结合高优先级优先选择、同优先级 FIFO[26] 的任务分配策略,最终实现了系统的有效容错。节点优先级的划分基于主控端对各运行节点的检测记录。一级节点池用于存储执行过程正常、性能稳定的测试节点;二级节点池用于存储测试过程不稳定,如出现进程阻塞、异常退出等情况的测试节点;异常节点池用于存储无法连通或失去响应的测试节点。

系统的容错处理算法如算法 2 所示。其中,ExecutingPool 为已调度资源池,存储已调度测试节点信息;ScheduleDic 是一个用来存储各测试节点执行进度的字典结构,关键字为节点 Id 号;D-value 是用于衡量节点的测试执行是否正常的阈值;AbnormalPool 为异常节点资源池,存储失去响应的测试节点信息;Task 为待分配的测试任务列表; ErrRecord 用于记录执行中出现进程异常的测试节点信息。

算法 2 系统容错算法

```
1  while (Running) do
2    if (ExecutingPool≠∅) then
3    for each (fnode in ExecutingPool) do
4      currentInfo: = ReadInfo( fnode. Vm. Id )
5      lastInfo: = ScheduleDic. getInfo( fnode. Vm. Id )
6      if (sub(currentInfo,lastInfo) < D-value) then
7        if NodeErr then
8          fnode. RoolBack( )
9          ExecutingPool. delete( fnode)
10          ScheduleDic. delete( fnode)
11          AbnormalPool. put( fnode. Vm )
12           Task. push( )
13         end if
14         if ProcessErr then
15           finishProcess( fnode. Vm. Ip)
16           ReDispatch( fnode. Vm. Ip)
17           ErrRecord. put( fnode. Vm)
18         end if
19       end if
20       end for
21     end if
22  end while
```

首先进行异常监测。4、5 行获取节点 fnode 的本次和前一次的测试执行进度,通过第 6 行计算最新测试执行进度,与 D-value 进行比较,以判断该节点是否在正常执行。7 ~ 18 行为容错处理过程,其中 7 ~ 13 行对上文中提到的节点异常进行处理。监测到节点异常后,将 fnode 对应的测试进度回滚到上一个容错点,将出错节点从 ExecutingPool 和 ScheduleDic 中删除,然后存储到异常节点资源池,其对应的测试任务放回 Task 列表,等待其他空闲节点执行。第 14 ~ 18 行为进程异常处理过程。主控端调用远程命令结束该节点上的所有与模糊测试相关的进程,然后重新调度任务,基于上一次执行进度继续执行。17 行记录进程异常,待执行完成,将其分配到二级资源池。

上述容错过程根据节点运行状态将测试节点划分为三类。本文采用基于优先级的任务分配策略,使得测试节点被分配与其性能相匹配的测试任务量,有利于测试节点间的负载均衡。任务分配算法如算法 3 所示。

算法 3 基于优先级的调度算法

```
1  while (Running) do
2    while (FirstPool≠∅ and Running) do
```

```
3        vm = FirstPool. get( )
4        task = Task. pop( )
5        ReadyPool. put( vm,task)
6      end while
7      while ( SecondaryPool≠∅ and Running) do
8        if ( FirstPool≠∅ ) then
9          break
10       end if
11       vm = SecondaryPool. get( )
12       task = Task. pop( )
13       ReadyPool. put( vm,task)
14     end while
15   end while
```

2 ~ 6 行,主控端优先选择一级节点池 FirstPool 中的测试节点为其分配任务。7 ~ 14 行,为二级节点任务分配过程。在该过程中,如果检测到一级节点池有新的测试节点加入,即 FirstPool 非空,则优先选择一级测试节点;反之选择 SecondaryPool 中的测试节点为其分配任务。在上述过程执行的同时,主控端运行一个单独的线程循环检测异常测试节点的可用性,并将检测可用的节点加入二级节点池。同级资源池中的测试节点调度遵循 FIFO 原则。

3　实现

本文提出的并行模糊测试框架(图 2)从总体上可分为两大模块,即支撑模块和主体模块。支撑模块封装了大量自定义的基本操作函数和数据库交互函数,通过接口调用的方式为主体模块提供服务。主体模块采用主从控制模式。主控端是整个框架的核心,它包含配置和流程控制两部分。配置模块提供模糊器的集成接口,可实现对各类模糊器的集成。流程控制模块基于四级流水线并行处理结构,控制整个框架的执行流程。从控端由大量并行测试节点构成,是模糊测试的具体执行单元。

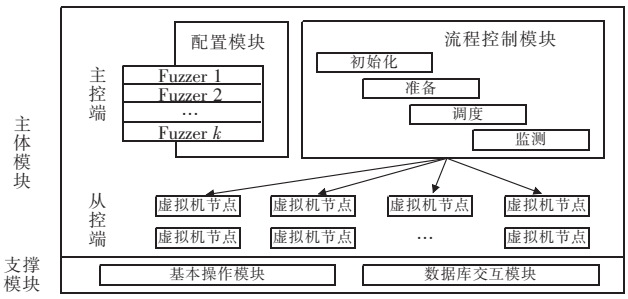

图 2　并行模糊测试框架

整个框架的运行可分为启动、执行和 crash 的管理三个部分。

启动过程完成模糊测试前的准备工作。主控端通过读取数据库获取本次测试的镜像 ID、模糊器类型以及相关测试文件路径等信息,调用配置模块所提供的模糊器参数设置接口对测试执行路径和相关命令进行设置。

执行过程基于主控端的流程控制模块实现。该模块将执行过程划分为初始化、准备、调度和监测四个阶段。初始化阶段完成测试节点的申请和测试任务的划分,并对测试过程中的并行度动态扩展提供支撑。准备阶段完成测试节点和测试任务的分配。调度阶段对待调度任务池中的测试任务进行并行调度。监测阶段则负责控制整个模糊测试的执行进度和异常监测处理。以上每个阶段由一个线程循环执行,各阶段基于共享资源池的阻塞等待机制实现同步,四个线程并行执行,实现了主控端的四级流水线并行处理结构。流程处理过程如图 3 所示。

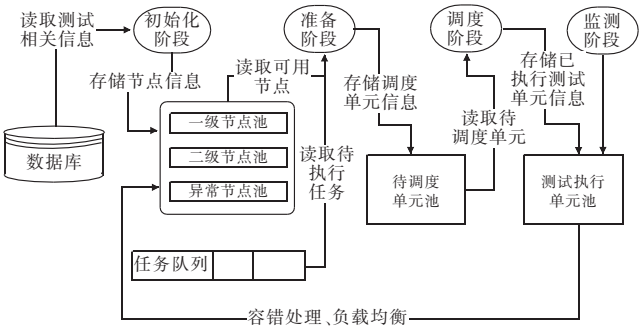

图 3　流水线并行处理流程

Crash 管理是并行模糊测试框架的重要组成部分。并行模糊测试会触发大量软件异常,这些异常信息通常存在着大量重复。例如在针对 IE11 的实验中,触发 crash 724 103 个,非重复的 crash 只有 420 个。所以,对 crash 进行筛选能大大减轻漏洞分析人员的工作量,具有十分重要的意义。本文提出的并行模糊测试框架在运行环境下对模糊测试的 crash 结果进行监控和管理,通过分析 crash 日志获取栈信息,对栈信息进行哈希[27]处理,利用测试节点端的局部去重和主控端的全局去重,实现了 crash 集的筛选。

4　实验测试

本章主要通过实验证明框架在提高资源利用率、系统容错以及提高测试效率等方面的有效性。实验所使用的基础测试环境由硬件环境 Eh 和操作系统类型 Tos 构成,主要划分为 C_1 和 C_2 两类。具体配置如表 1 所示。

表 1　基础测试环境

测试环境	处理器	内存/GB	硬盘/GB	操作系统
C_1	2 核	4	40	Win 7 SP1
C_2	1 核	2	30	Win XP SP3

4.1　资源的高效配置

本节分别从调配整机资源和提高单机负载两个层面开展实验,说明本文提出的多层次并行度动态调整的资源配置策略在提高资源利用率上的有效性。

4.1.1　节点并行扩展实验

本节基于表 2 中的测试任务,采用两种不同的资源配置方式开展对比实验,运行记录如图 4 所示。

表 2　测试任务描述

测试任务	测试环境	初始并行度/个	用例数/个
A	C_1	10	5 000
B	C_1	10	50 000

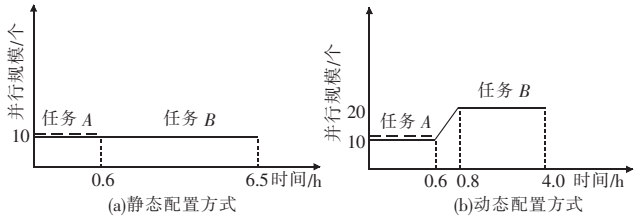

图 4　资源的静态配置和动态配置效果

根据图 4 可以看出,采用静态资源配置方式,任务 A、B 在整个运行过程中的并行规模固定不变,任务 A 的 10 个测试节点在 0.6 h 之后一直处于空闲状态;采用基于并行度动态扩展的资源配置方式,任务 B 能检测到任务 A 释放的空闲资源,实

现并行扩展。图 5 记录了 B 在并行扩展前后的测试用例生成变化。

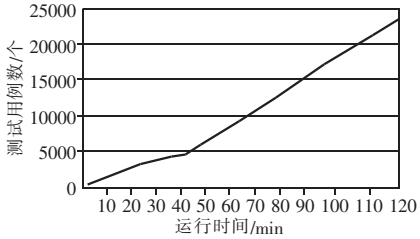

图 5 任务 B 并行扩展效果

基于图 5 中数据可以得出,在测试资源一定的情况下:

a)采用基于并行度动态扩展所实现的动态资源分配方式,能够实现对空闲节点的再分配,提高对测试资源的利用率。

b)在测试执行过程中,动态扩展测试并行度能提高单位时间内的测试用例生成数量,加快测试任务的执行速度,进而缩短测试的总体执行时间。

4.1.2 单机负载实验

如表 3 所示,F_1 和 F_2 分别基于 10 个并行节点,通过控制不同的机内并行度来开展对比实验。实验结果如图 7 所示,其对应内存和 CPU 占用情况如图 6 所示。

表 3 单机负载实验场景

节点	测试环境	节点并行度	机内并行度
F_1	C_1	10	1
F_2	C_1	10	6

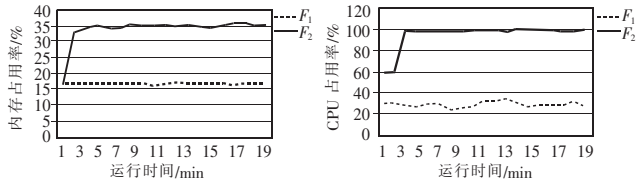

图 6 内存及 CPU 占用情况

根据图 6 中数据可以看出:

a)机内并行扩展能有效提高单机资源利用率。

b)通过并行提高单机负载,可以加快测试用例的生成速度,有助于提高模糊测试的效率。

4.2 系统容错

本节通过对比实验来说明本框架的容错处理算法的有效性。实验场景描述如表 4 所示。

表 4 容错对比实验场景

任务	测试目标	测试环境	并行度	模拟出错	是否容错
A	FontEngine	C_2	10	进程异常	是
B	FontEngine	C_2	10	进程异常	否

任务 A、B 除容错以外的所有测试因素一致。本节编写脚本对并行模糊测试中的常见进程异常进行模拟,并记录了进程出错时刻、主控端监测到该错误的时刻,以及最终的容错对比效果。

容错对比实验如图 8 所示。从图 8 中可看出,测试任务 B 在遇到异常之后,用例的测试执行速度减慢;当所有节点进程都异常退出后,测试用例数最终趋于不变。针对任务 A,基于实验统计从节点出错到任务 A 捕获到错误的平均延迟时间为 167.9 s,错误处理时间则远小于错误捕获延迟,基于此,本框架对一次出错过程的容错处理时间约为 3 min,这在需要长时间运行的大规模并行模糊测试中是十分高效的。从图 8 中也可以看出,任务 A 能迅速监测到测试节点出现异常,重新调度远程节点,基于已有的测试进度继续执行,整个测试过程基本不受影响。

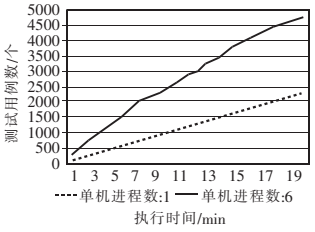

图 7 多层次并行实验

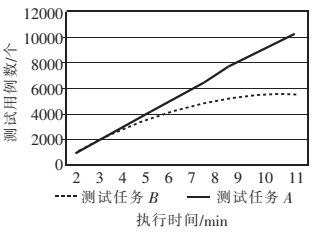

图 8 容错处理实验

基于上述结果,本实验实现了对测试过程中的节点出错监测的全覆盖和对执行异常的实时监测,证明了框架的容错处理过程的有效性。

4.3 大规模并行模糊测试实验

本节主要通过开展对比实验来验证框架在大规模并行处理上的高效性。首先对本框架的流水线处理过程和传统的顺序处理过程进行分析。顺序执行方式的时间开销由三部分组成,即申请 n 个测试节点所需时间($t\_\mathrm{vm}_1, t\_\mathrm{vm}_2, \cdots, t\_\mathrm{vm}_n$)、$n$ 个测试节点准备时间($t\_\mathrm{pre}_1, t\_\mathrm{pre}_2, \cdots, t\_\mathrm{pre}_n$) 以及最长并行执行时间 t_k,对应由式(1)表示。流水线处理过程的时间开销由式(2)表示,即申请到第一个测试节点需要的时间 $t_$vm1、第一个测试节点准备时间 $t_$pre1 以及最长并行测试时间 t_p。根据这两种不同执行方式的时间开销,可利用式(3)计算流水线并行处理方式的加速比。

$$T_k = \sum_{i=1}^{n} t\_\mathrm{vm}_i + \sum_{i=1}^{n} t\_\mathrm{pre}_i + t_k \tag{1}$$

$$T_p = t\_\mathrm{vm}_1 + t\_\mathrm{pre}_1 + t_p \tag{2}$$

$$S = T_k / T_p \tag{3}$$

基于上述理论,本实验在 10、30、50 三个并行度下,针对不同规模的测试任务,开展了并行模糊测试实验;并根据式(3)计算了在上述不同情况下,流水线并行处理相比于顺序处理的加速比 S,其变化趋势如图 9 所示。

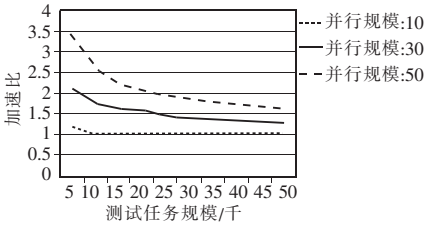

图 9 加速比

根据图 9 可以得出如下结论:

a)控制测试任务规模不变,并行规模越大,加速比越高,说明框架在大规模并行处理方面具有较好的性能。

b)同一并行规模下,随测试任务规模的增大,加速比逐渐减小。对该结果进行分析可知,在并行度不变的情况下,测试节点申请时间和执行准备时间均为定值,并行执行时间 t_k 和 t_p 随着测试任务规模的增大而增大。当任务的执行时间远大于节点申请时间和执行准备时间时,t_k 与 t_p 近似相等,加速比最终趋于 1。

5 结束语

本文基于云平台设计实现了动态资源感知的并行化模糊测试框架,对框架的特色和整体结构进行了介绍,并通过实验进一步论证了框架在提高测试效率、资源利用率以及系统容错

方面的有效性。并行模糊测试是通过生成大量测试用例,提高测试集的完备性来提高单位时间内的异常触发率。而在框架的实际运行中,大部分的测试用例都是无效用例,所以在现有框架基础上引入智能模糊测试技术,是笔者下一步的研究重点。

参考文献:

[1] Sutton M, Greene A. Fuzzing: brute force vulnerability discovery[M]. [S. l.]: Addison-Wesley Educational Publishers, 2007.

[2] 吴志勇,王红川,孙乐昌,等. Fuzzing 技术综述[J]. 计算机应用研究,2010,27(3):829-832.

[3] 吴世忠,郭涛,张普含,等. 基于路径簇规约的并行符号执行方法:中国,201210542210[P]. 2013-04-17.

[4] Wu Bo, Li Mengjun, Zhang Bin, *et al*. Distributed symbolic execution for binary software testing[C]//Proc of IEEE Workshop on Electronics, Computer and Applications. [S. l.]: IEEE Press, 2014: 618-621.

[5] 余啸. 基于动态符号执行的并行化测试数据自动生成[D]. 上海:华东师范大学,2011.

[6] 曹琰. 面向软件脆弱性分析的并行符号执行技术研究[D]. 郑州:解放军信息工程大学,2013.

[7] Boyapati C, Khurshid S, Marinov D. Korat: automated testing based on Java predicates[C]//Proo of ACM International Symposium on Software Testing and Analysis. 2002: 123-133.

[8] Misailovic S, Milicevic A, Petrovic N, *et al*. Parallel test generation and execution with Korat[C]//Proc of the 6th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. New York: ACM Press, 2007: 135-144.

[9] Siddiqui J H, Khurshid S. PKorat. parallel generation of structurally complex test inputs[C]//Proc of International Conference on Software Testing Verification and Validation. [S. l.]: IEEE Press, 2009: 250-259.

[10] 梁洪亮,阳晓宇,董钰,等. 并行化智能模糊测试[J]. 清华大学学报:自然科学版,2014,21(3):14-19.

[11] 王连赢. 文件触发类二进制程序漏洞挖掘技术研究[D]. 北京:北京邮电大学,2015.

[12] Godefroid P, Levin M, Molnar D. SAGE: whitebox fuzzing for security testing [J]. Communications of the ACM, 2012, 55(3): 40-44.

[13] Bounimova E, Godefoid P, Molnar D. Billions and billions of constraints: whitebox fuzz testing in production[C]//Proc of International Conference on Software Engineering. [S. l.]: IEEE Press, 2013: 122-131.

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

[15] Ciortea L, Zamfir C, Bucur S, *et al*. Cloud9: a software testing service [C]//Proc of ACM SIGOPS Operating Systems Review. New York: ACM Press, 2010: 5-10.

[16] Sulley: pure python fully automated and unattended fuzzing framework [EB/OL]. (2014-03-16) [2015-11-28]. http://code. google. com/p/sulley/.

[17] Peach[EB/OL]. (2014-02-23) [2015-11-28]. http://community. peachfuzzer. com/.

[18] Fuzzing for Security[EB/OL]. (2012-04-26) [2015-11-28]. http://blog. chromium. org/2012/04/fuzzing-for-security. html.

[19] Defensics 3. 0 [EB/OL]. (2015-05-20) [2015-11-28]. http://www. codenomicon. com/products/defensics/.

[20] Peach fuzzer for enterprise[EB/OL]. (2014-05-23) [2015-11-28]. http://www. dejavusecurity. com. s3-website-us-east-1. amazonaws. com/files/Peach%20Enterpise%20Fuzzer%20-%20Deja%20Vu Security. pdf.

[21] Xie Yan. Using grid computing for large scale fuzzing[D]. Lisbon: Universidade Nova de Lisboa, 2010.

[22] King J C. A new approach to program testing[C]//Proc of International Conference on Reliable Software. New York: ACM Press, 1975: 228-233.

[23] Chipounov V, Kuznetsov V, Candea G. The S2E platform: design, implementation, and applications [J]. ACM Trans on Computer Systems, 2012, 30(1): 1-49.

[24] KVM[EB/OL]. (2014-01-22) [2015-11-28]. http://www. linux-kvm. org/page/Main_Page.

[25] Qcow2[EB/OL]. (2013-10-09) [2015-11-28]. http://www. linux-kvm. org/page/Qcow2.

[26] FIFO and LIFO accounting[EB/OL] . (2011-09-25) [2015-11-28]. http://en. wikipedia. org/wiki/FIFO_and_LIFO_accounting.

[27] Hash function[EB/OL]. (2015-02-12) [2015-11-28]. https://en. wikipedia. org/wiki/Hash_function.

(上接第 51 页)

[4] Burges C, Shaked T, Renshaw E, *et al*. Learning to rank using gradient descent[C]//Proc of the 22th International Conference on Machine Learning. New York: ACM Press, 2005: 89-96.

[5] Zheng Z, Zha H, Zhang T, *et al*. A general boosting method and its application to learning ranking functions for Web search[C]//Neural Information Processing Systems. 2007: 1697-1704.

[6] Jin J, Lin X. Efficient parallel algorithms for linear ranksvm on gpu[C]// Proc of Network and parallel computing. Berlin: Springer, 2014: 181-194.

[7] Herbrich R, Graepel T, Obermayer K. Large margin rank boundaries for ordinal regression[C]//Proc of Neural Information Processing Systems. 2000: 115-132.

[8] Cortes C, Vapnik V. Support-vector networks[J]. Machine Learning, 1995, 20(3): 273-297.

[9] Chapelle S, Keerthi S. Efficient algorithms for ranking with SVMs[J]. Information Retrieval, 2010, 13(3): 201-215.

[10] Airola T, Pahikkala T S. Training linear ranking SVMs in linearithmic time using red-black trees[J]. Pattern Recognition Letters, 2011, 32(9): 1328-1336.

[11] Kuo T M, Lee C P, Lin C J. Large-scale kernel RankSVM[J]. MIT Press Journals, 2014, 26(14): 781-817.

[12] Dembo R S, Steihaug T. Truncated-Newton algorithms for large-scale unconstrained optimization [J]. Mathematical Programming, 1983, 26(2): 190-212.

[13] Lin C J, Mor′e J J. Newton's method for large bound-constrained optimization problems [J]. SIAM Journal on Optimization, 1999, 9(4): 1100-1127.

[14] Joachims T. Training linear svms in linear time[C]//Proc of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York: ACM Press, 2006: 217-226.

[15] Lee C P, Lin C J. Large-scale linear rankSVM[J]. Neural Computation, 2014, 26(4): 781-817.

[16] Cossock D, Zhang T. Statistical analysis of bayes optimal subset ranking[J]. IEEE Trans on Information Theory, 2008, 54(11): 5140-5154.

[17] Yu Hwanjo, Kim Jinha, Kim Youngdae, *et al*. An efficient method for learning nonlinear ranking SVM functions [J]. Information Sciences, 2012, 209(22): 37-48.