Elastic Scheduling for Microservice Applications in Clouds

Sheng Wang, Zhijun Ding, Senior Member, IEEE, and Changjun Jiang

摘要

微服务被广泛用于灵活的软件开发。目前而言，容器已经成为微服务的首选部署技术，因为它启动速度快，开销低。然而，容器层使任务调度和云中的自动伸缩变得复杂。现有算法不适应由虚拟机和容器组成的两层结构，经常忽略流工作负载。为此，论文提出了一种针对微服务的弹性调度(ESMS)，它将任务调度与自动伸缩集成在一起。ESMS 的目标是在满足期限限制的情况下最小化虚拟机的成本。论文将微服务的任务调度问题定义为一个具有期限约束的成本优化问题，并提出一种基于统计的策略来确定流工作负载下的容器配置，随后提出一种基于紧急度的工作流调度算法，该算法可以分配任务并确定可扩展实例的类型和数量。最后，论文将新容器到虚拟机的映射建模为一个变大小的装箱问题，并对其进行求解，以实现虚拟机和容器的集成扩展。通过与现有算法的比较，验证了 ESMS 在提高交期成功率和降低成本方面的能力。

关键词：自动伸缩；云计算；容器；微服务；任务调度

1 引言

随着云计算、虚拟化与容器等技术的发展，万物上云已成为可能。对于云服务使用者，尤其是Iass/Pass 服务使用者而言，在使用云资源时需要考虑如下两个主要问题：

（1）面对不确定的任务工作流，如何确定一种调度策略，以实现更高效的服务性能；

（2）面对租赁的云计算资源，如何确定一种伸缩策略，在低成本的预算下，确保任务的顺利完成。

往往这两种问题是具备相关性的，即不同的调度策略，可能会导致系统对资源的需求不同，从而需要不同的伸缩策略。例如，在不考虑调度的伸缩策略中，可能会出现由于调度所需资源不足而导致服务性能下降等情况，反之亦然。

在此前提出的云环境下的工作流调度算法，一些算法只关注了单个工作流的调度，而非多工作流，涉及到的伸缩也仅是不考虑容器层面的虚拟机简单扩展；另一些算法则不考虑后续工作流，而是基于已知所有工作流的特征。同时，一些基于 MDP、回归模型等的自动伸缩算法也不涉及到具体的调度策略，在一定程度上影响了系统性能。即便一些调度算法与基于规则的伸缩算法结合，其评估结果也显示出，延迟时间及平均成本与预期结果存在着明显差异。

本次选择复现的论文提出了一种基于容器-虚拟机双层结构，并集成自动伸缩的微服务弹性调度算法（ESMS）。该论文提出了基于紧急度的工作流调度算法（UWS），以及能够确定容器配置的统计策略（IFFD）。ESMS 算法能够分配调度任务并确定可扩展实例的类型和数量，并在满足任务的约束期限下最小化虚拟机成本。该算法适用于同时考虑基于任务期限的调度与低成本伸缩的微服务云环境。

论文的主要贡献如下：

（1）提出了一种融合任务调度和自动伸缩的微服务弹性调度算法。利用基于紧急度的调度算法来获得调度方案，并准确地确定要伸缩的新实例。然后通过两层缩放算法得到缩放方案；

（2）为提高微服务流工作负载下的利用率，设计了一种基于统计信息的配置求解算法。生成的配置用于创建新的容器和分配截止日期，以提高截止日期分布的准确性；

（3）为了解决虚拟机和容器的两层伸缩问题，在任务调度中考虑了容器镜像对容器初始化的影响。将自动伸缩问题构造为 VSBPP，以获得最优的虚拟机伸缩方案和容器部署方案，使得虚拟机成本最小化.

2 相关工作

本篇论文的研究工作有两方面：任务调度和伸缩策略。目前云环境下的工作流调度算法主要分为：期限约束的工作流调度（性能需求表示为工作流的期限，目标是在满足期限约束的情况下，使得工作流的执行成本最小化）和预算约束的工作流调度（在用户定义的预算范围内最小化工作流的完成时间）。而云环境下的伸缩算法可分为三类：基于规则的伸缩方法，基于时间序列的分析方法，以及基于模型的分析方法。

许多任务调度算法采用简单的策略向外扩展实例，例如创建最便宜的服务实例或采用基于规则的扩展策略。一个新的实例将直接在 VM 上创建。但是，在由容器和 VM 组成的两层结构中，这些策略不能解决如何确定容器的配置、应该创建哪种类型的 VM 以及应该将 VM 的新容器放置在哪种位置等问题。虽然存在部分文献都描述了微服务的两层资源结构，但没有讨论相关的伸缩问题，只考虑了容器的伸缩，忽略了 VM 的伸缩。而一些伸缩算法不涉及具体的调度方法，会影响系统性能。这些模型的结果可能与实际调度结果不一致，不能准确反映实际对资源的需求。

2.1 云环境下的任务调度

（1）期限约束的工作流调度：

Abrishami 等人定义了局部关键路径（PCPs），并提出了 IC-PCP 算法以最小化总成本[1]。Q.Wu 等人在 HEFT 中扩展了向上秩（upward rank），并定义了概率向上秩，提出了两种工作流调度算法—ProLiS 和 LACO[2]。H.Wu 等人提出了一种四步算法 MSMD，通过减少虚拟机数量和实例时间来最小化成本[3]。Thiago 等人提出了 R-粒度方法，以加快有截止时间约束的工作流调度[4]。

（2）预算约束的工作流调度：

Zheng 等人对经典的 HEFT 进行了扩展，提出了 BHEFT，即通过评估备用预算来选择可能的最佳服务[5]。Bao 等人建模了微服务的性能，提出了 GRCP 算法，使用贪心策略将任务映射到现有或新的实例[6]。Qin 等人提出了剩余最便宜预算，以满足预算约束，保证解决方案的可行性[7]。

2.2 自动伸缩云

（1）基于规则的方法：

该方法被广泛应用于行业，如:Amazon EC2。这些方法需要一些阈值或基于先验知识的扩展规则来触发响应动态工作负载的扩展操作。Zheng 等人设计了两个基于规则的自动标度器，并将其应用到基于 SLA 感知微服务的框架 SmartVM[8]。

（2）基于时间序列的分析方法：

该方法使用一组数据点来发现重复模式或预测未来的值，例如未来的请求量或周期趋势，通常是通过机器学习技术。Kan 等人使用二阶自回归移动平均方法预测短时间后的每秒请求量[9]。Fernandez

等人利用五种不同的统计模型来适应四种类型的工作负载[10]。

（3）基于模型的分析方法：

该方法通过建立数学或概率模型来预测性能和作出决策，最典型的例子是排队论。排队理论被用来对微服务的响应时间和供应资源进行建模。此外，基于马尔可夫过程的强化学习、回归模型和基于 Petri 网的分析模型也是应用广泛的技术。

3 本文方法

3.1 本文方法概述

本篇论文将系统模型分为四层：装载科学工作流的工作负载 Workload，工作流层（Workflow Layer），微服务实例层/容器层（Microservice Layer/Container Layer），虚拟机层（Virtual Machine Layer），如图 1所示。

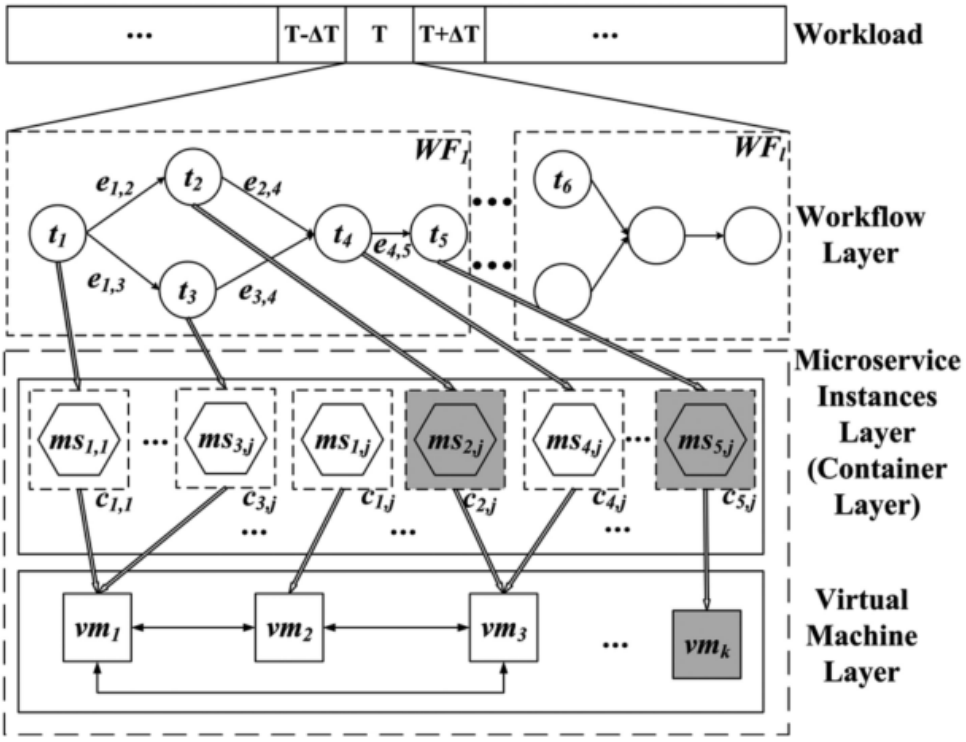

图 1: 系统模型

本篇论文提出 ESMS 算法，即 UWS+IFFD。算法基于多个科学工作流组成的请求流 Request Queue，并加以不同的工作负载（Stable，Increase，Decrease）作为任务输入。在不同的时间区段内工作流的数量不尽相同，可存在多个工作流。

算法处理 T 时间段内的多工作流，将其合并为单个工作流，计算工作流，及其包含的子任务的相关信息（如：WF 的最后期限 DDL，子任务的最早开始时间 EST，子任务的紧急度 urgency 等）。随后交给 UWS 进行微服务与容器映射，其中包含 CE 迭代配置，基于子任务紧急度的容器分配，初始容器-VM 映射。最后由 IFFD 进行适配，其中将会对未被适配的实例进行最佳适配（Best-Fit），未被 BF 适配的实例将会继续进行 FFD（First Fit Descending），最后再根据 VM 所剩资源进行 VM 调整。系统架构如图 2所示。

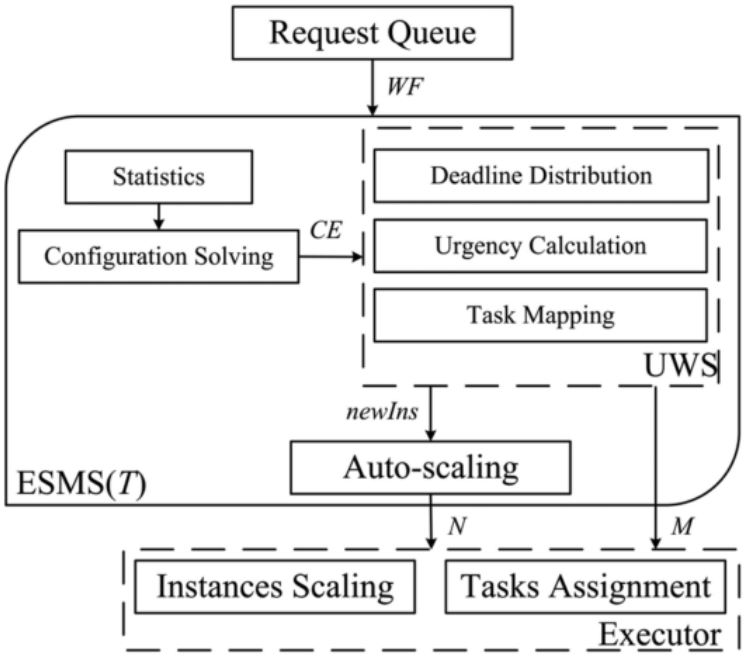

图 2: 系统架构

3.2 模型相关设置

本文将涉及到的虚拟机资源（包括但不限于 CPU，内存，显存等）离散化，如离散化后的内存资源最小单位为 500MB，这些离散化资源所代表的计算能力被拟合为 ECU，最小单位为 0.5，并根据文献设置 VM 和容器的启动时间[11]。

本文使用四种科学工作流：Montage（I/O 密集型），LIGO（CPU 密集型），SIPHT（CPU 密集型），GENOME（数据密集型），并具备有详细的工作信息：名称，计算工作量，数据传输量和任务之间的依赖关系，并将任务的计算工作量定义为其在标准计算服务上的执行时间。科学工作流的相关模型见 3。工作流应用程序将用作基于微服务的应用程序，每种类型的任务对应一种微服务，每一种微服务只能处理一种类型的任务，文献中的论文提供了每个任务的执行成本和其它详细信息[12][7]。

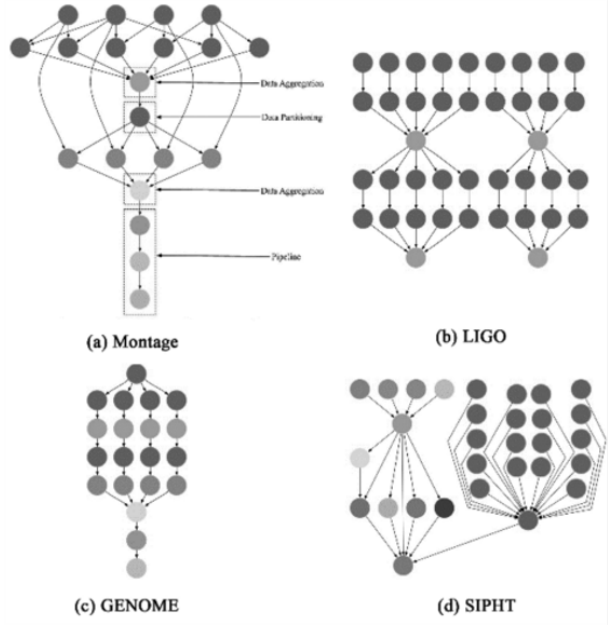

图 3: 科学工作流模型

根据 Wikipedia access traces 设置三种工作负载，设定该工作负载的执行时间。在执行时间内，请求速率呈稳定的为 stable，请求速率不断增加的为 Increase，请求速率不断减小的为 Decrease。ESMS

处理每一段 T 时间存在的工作流，进行任务调度与资源伸缩。T 时间段的工作负载存在多个工作流将会被合并成单个工作流。计算该单个工作流和每个子任务的 DDL，其中工作流的 DDL 由关键路径的长度与 factor（1.0-2.0）共同决定，而子任务 DDL 由最早完成时间决定（EFT，由最早开始时间 EST 和执行时间 ET 决定）。

3.3 基于紧急度的工作流调度

UWS 首先执行容器配置。定义 CE 为每一个装载有微服务的容器具体配置，容器配置将分为以下四步进行：

（1）用最少的资源将 CE 中的所有元素初始化，即用最低的配置作为每一个容器的基本配置；

（2）假设第 j 个类型的微服务的每一个实例都运行在一个配置为 ce-j 的容器上，使用配置方案 CE 计算当前的 makespan 和成本。其中 makspan 为当前配置工作流的预计完成时间，makespan 将由所有子任务中最大的 EFT 决定；而成本计算按容器资源与 VM 资源比例最高计算（如：VM 配置为 2vCPU 和 8GB 内存，每 0.1 美金一个小时，而当前容器配置为 1vCPU 和 2GB 内存，那么将按 CPU 比例进行价格计算，即容器单价为 0.1*(1/2)=0.05 美金）；

（3）如果预期的 makespan 大于工作流的 DDL，那么将当前第 j 个类型的微服务对应的容器配置替换为包含更多资源的配置，通常增加的资源配置为一个离散化资源单位。当具有 h 种类型的微服务时，逐一迭代替换增加每一种类型微服务的配置，获取对应的增益，并选取收益最大的 CE 返回步骤（2）；

（4）当前迭代的 makespan 小于工作流的 DDL 时，迭代便停止，当前的 CE 便是最佳配置。

计算每一子任务的秩以及子任务 sub-DDL。其中秩 rank 通过执行时间 ET + max 数据传输时间 TT + 关联的后续子任务的秩计算得出，即当前子任务到退出任务的预期执行时间长度。子任务 DDL 则通过工作流 DDL 乘以秩的比例计算得到。再计算子任务的紧急度 urgency，通过 (sub-ddl - 实际结束时间 XFT)/hop，其中 hop 为当前子任务剩余的关键路径上的任务数，这也反应了当 hop 越大，紧急度越小，当前子任务的紧急程度越高，优先级越高，应被立即处理，对应的容器应被首先分配。

UWS 算法主体循环处理未被分配的任务。首先优先处理未被分配的任务，计算每个任务的松弛度 Laxity（sub-ddl - 最早结束时间 EFT），即当前子任务还剩多少时间就能被调度。

1. 当 Laxity≥0 时，当前子任务能够在截止时间内完成，那么将当前子任务分配给具有最小 incrCost（即当前容器的配置增量花费）的容器；

2. 当 Laxity<0 时，当前子任务不可能在截止时间内完成，那么根据 minSpeed（当前的执行时间）分配容器：

1）minSpeed < 0 或 minSpeed < VM 最大速度，选择 EFT 最小的方案：a) 现有 EFT 最小的容器；b) 最大 VM 速度的容器;

2）minSpeed < 当前容器速度，将任务分配给当前容器；

3）当前容器速度 < minSpeed < VM 最大速度，创建并分配给刚好大于 minSpeed 的容器。

最后对当前分配了容器的微服务，与装载由该镜像的 VM（已有）进行最佳适配，剩下未被适配的容器加入 newIns 集合，该集合中的容器将会在 IFFD 中进行适配。

3.4 容器与虚拟机的自动伸缩

在 UWS 中未被分配的容器 newIns 将继续采用最佳适配，映射到已有的虚拟机中。若仍有容器未被适配，即 newIns 不为空，那么进行 FFD 适配，流程如下：

（1）容器按 ECU 从高到小排序，虚拟机按类型排序（按计算能力由强到弱排序）；

（2）固定当前虚拟机类型 i，遍历 newIns，ECU 高的容器优先分配给 ECU 高的 VM，当前 VM 没有足够的资源分配时，新建同类型 i 的 VM 进行分配，直至 newIns 集合为空，记录当前成本 cost；

（3）取分配到最后一次新建的虚拟机中的容器，重新加入 newIns，撤销该 VM，取新的 VM 类型（i+1），继续重复步骤（2），直至 VM 类型被遍历完为止；

取 cost 最小的配置，并重新整理对应的 VM 集合，即存在一些 VM 还剩下一些资源未被使用，但可以缩小为更低类型的 VM，以减小成本。此时的任务调度，以及伸缩策略是满足约束期限的，且是最低成本的。

4 复现细节

本篇复现论文没有源代码，所有相关代码均为本人独自实现，所有代码可参考 GitHub 上提交的代码。原篇论文仅给出 UWS 以及 IFFD 的大致伪代码，以下将给出更具体的实现。

4.1 源代码实现

（1）实体类以及初始化实现：

实体类应该包括：VM（虚拟机），Container（容器），Task（任务），Workflow（工作流）等。其中 VM 的属性应该有：MAX-SPEED（所有虚拟机中的最大速度），NETWORK-SPEED（网络速度），TYPE-NO（类型），ECU（计算能力），UNIT-COST（单位），LEASE-TIME（租用时间），IMAGES（装载的镜像）等。Container 的属性应该有：ECU（计算能力），VM-TYPE（VM 类型），TASK-ID（任务 ID）等。Task 的属性应该有：ID（任务 ID），NAME（任务名），TASK-SIZE（任务大小），RANK（秩），URGENCY（紧急度），OUT-EDGES（出边），IN-EDGES（入边）等。Workflow 的属性应该有 deadline（工作流的最后期限），factor（DDL 的影响因子）等。

合并工作流需要先从 DAX（由科学工作流文件提供）中读取任务，并根据文件为每个任务设置出入边（即前继任务和后继任务），最后为合并后的单个工作流设置入口 entry 和出口 exit，相关伪码实现如下：

Procedure 1 Read DAX file and merge the Workflow

Input: DAX-FILE *file*
Output: MultiWorkflow *singleWF*
$taskSet$ = readDAXFile(*file*)
$entry$ = newTask("*entry*")
$exit$ = newTask("*exit*")
for *task* ***in*** *taskSet* **do**
  if *task.InEdges() not empty* **then**
    $edge$ = newEdge(*entry*, *task*)
    $task.InEdges().add(edge)$
  end
  if *task.OutEdges() not empty* **then**
    $edge$ = newEdge(*task*, *exit*)
    $task.OutEdges().add(edge)$
  end
end
$addTask2Workflow(entry)$
$addTask2Workflow(exit)$

（2）计算相关信息的实现：

工作流的 DDL，通过公式 DDL = factor * cpLength 计算得到，其中 cpLength 为关键路径长度，因此需要先对整个工作流进行拓扑排序，随后计算关键路径长度，得到。若需要计算某一工作节点的 hop，那么还需要计算从该工作节点开始到 exit 结束节点的 AOE 网的拓扑序列和反拓扑序列，从而求得关键路径上的节点数，即 hop。图 4为计算工作流的实际代码实现，图 4为计算任务节点 hop 的实际代码实现。

```
private void calDDL() {
    //calculate from the basis configuration
    double speed = VM.ECU[VM.SLOWEST];

    for(int j=this.size()-1; j>=0; j--) { //find the biggest value(that can decide whether the task have finished)
        // review the handwriting paper
        double bLevel = 0;
        double sLevel = 0;
        Task task = this.get(j);
        for(Edge outEdge : task.getOutEdges()){
            Task child = outEdge.getDestination();
            bLevel = Math.max(bLevel, child.getBLevel() + (double) (outEdge.getDataSize() / VM.NETWORK_SPEED));
            sLevel = Math.max(sLevel, child.getSLevel());
        }
        task.setBLevel(bLevel + (task.getTaskSize() / speed)); //rank(ti)
        task.setSLevel(sLevel + (task.getTaskSize() / speed));
    }

    this.deadline = this.get(0).getBLevel() * factor;
}
```

图 4: 计算工作流的实际代码

```
public int calHop(Task entry, HashMap<Task, Allocation> revMapping) {
    //calculate the ve
    List<Task> topoList = topoSort(entry);
    for (Task task : topoList) {
        double ve = 0.0;
        List<Edge> inEdges = task.getInEdges();
        for (Edge edge : inEdges) {
            double speed = revMapping.get(task).getContainer().getECU();
            ve = Math.max(ve, edge.getSource().getVe() + (double) (edge.getDataSize() / VM.NETWORK_SPEED) + (task.getTaskSize() / speed));
        }
        task.setVe(ve);
    }

    //calculate the vl
    List<Task> reTopoList = reTopoSort(entry);
    for (Task task : reTopoList) {
        double vl = task.getVe();
        List<Edge> outEdges = task.getOutEdges();
        for (Edge edge : outEdges) {
            double speed = revMapping.get(task).getContainer().getECU();
            vl = Math.min(vl, edge.getDestination().getVl() - ((double) (edge.getDataSize() / VM.NETWORK_SPEED) + (task.getTaskSize() / speed)));
        }
        task.setVl(vl);
    }

    int count = 0;
    //find the task which ve = vl
    for (Task task : topoList)
        if (Math.abs(task.getVe() - task.getVl()) < 1e-6) count++;

    return count == 0 ? 1 : count;
}
```

图 5: 计算任务节点 hop 的实际代码

（3）UWS 的实现：

代码首先执行容器迭代配置 CE，基于 CE 计算出子任务的秩及其 DDL，随后计算出每一个子任务的紧急度，按紧急度排序每一个未被分配的任务，创建 newIns 集合。对于每一个未被分配的任务，尝试计算每一个可以装载该任务的容器对应的松弛度，如果松弛度不为负数，则记录它的成本，最终选择最低成本的容器进行分配。如果不存在松弛度为非负的，那么计算 minSpeed，根据上一大节所述的策略进行分配容器，最后进行最佳适配。图 6以及 7为 UWS 实际代码实现。

```
public List<Container> execute() {
    CalHandler.setCondition(workflow, solution, vms);
    CalHandler.calCE();

    System.out.println("------------------------------------------------------------");
    System.out.println("After CE, it's makespan is: " + solution.calMakespan());
    System.out.println("After CE, it's cost(by single fit) is: " + solution.calCost( flag: false));
    System.out.println("After CE, it's cost(by ratio, do not consider VM's basic resource) is: " + solution.calCost( flag: true));
    System.out.println("------------------------------------------------------------");

    //cal rank and subDDL
    workflow.calTaskRankBasedCE(revMapping);
    workflow.calSubDDL();
    CalHandler.calUrgency();

    List<Task> unassigned = CalHandler.getSortedTaskList();// ordered by urgency
    List<Container> newIns = new ArrayList<>();
    for (Task task : unassigned) {
        Container chosen = null; //the chosen one to the task

        Container origin = revMapping.get(task).getContainer();
        double incrCost = Double.MAX_VALUE;
        boolean laxityNeg = false;

        for (Container container : holdMapping.get(task)) {
            double laxity = CalHandler.calLaxity(task, container);
            if (laxity < 0) laxityNeg = true;
            else {
                double currentCost = solution.calCost( flag: true);
                if (incrCost > currentCost) { //minimum incrCost
                    incrCost = currentCost;
                    chosen = container;
                }
            }
        }
```

图 6: UWS 实际代码

```
    if (!laxityNeg) {
        CalHandler.changeTaskContainer(task, chosen);
    } else {
        double minSpeed = CalHandler.calMinSpeed(task);
        if (minSpeed < 0 || minSpeed > VM.MAX_SPEED) {
            Container c1 = CalHandler.getMinEFTInstance(task);
            Container c2 = new Container(VM.MAX_SPEED);
            holdMapping.get(task).add(c2);
            chosen = CalHandler.setMinEFTContainer2Task(task, c1, c2);
        } else if (minSpeed < origin.getECU()) {
            chosen = origin;
            CalHandler.changeTaskContainer(task, origin);
        } else {
            Container c1 = new Container(minSpeed);
            chosen = c1;
            CalHandler.changeTaskContainer(task, c1);
        }
    }
    assert chosen != null;
    VM vm = Fitter.bestFit(vms, chosen,  flag: false);
    if (vm == null) newIns.add(chosen);
    else conf.addVM(vm);
}
return newIns;
```

图 7: UWS 实际代码

（4）IFFD 的实现：

IFFD 由两个主要的适配方法组成—BF 和 FFD。BF 是根据容器的需求，分配最合适当前容器资源的 VM，直至容器被分配完成，本次的最佳适配采用折半查找，查找符合对应容器资源的 VM，分配完容器后，存在 VM 仍有空余资源。BF 的实现代码如图 8所示，其中 flag 标志将用于判断是否需要考虑当前 VM 是否存在该容器的镜像。FFD 适配则是根据容器配置和 VM 配置进行的适配，具体流程

可参考上一节，FFD 能够把所有未被分配的实例进行 VM 分配，其实现代码见图 9。最后根据 VM 的资源进行类型替换，以最小化成本。

```java
public static VM bestFit(List<VM> vms, Container container, boolean flag) {
    double needECU = container.getECU();
    List<VM> vmList = vms.stream().filter(vm -> flag || vm.getImages().contains(container))
            .sorted((o1, o2) -> (int) (o1.getResidueECU() - o2.getResidueECU())).collect(Collectors.toList());
    int low = 0;
    int high = vmList.size()-1;
    while (low < high) {
        int mid = (high+low)/2;
        double residueECU = vmList.get(mid).getResidueECU();
        if (Math.abs(needECU - residueECU) < 0.0001) {
            break;
        }
        else if (needECU < residueECU) high = mid - 1;
        else if (needECU > residueECU) low = mid + 1;
    }
    if (low >= vmList.size()) return null;

    VM vm = vmList.get(low);
    if (vm.getResidueECU() < container.getECU()) return null; //have not residue ECU
    else {
        container.setVMType(vm.getType());
        vm.setResidueECU(vm.getResidueECU() - container.getECU());
        vm.getLoadedContainer().add(container);
        return vm;
    }
}
```

图 8: BF 实际代码

```java
public static HashMap<VM, List<Container>> FFD(List<VM> vms, int VMType, List<Container> newIns) {
    for (Container container : newIns) {
        boolean fitted = false;
        VM fittedVM = null;
        for (VM vm : vms) {
            if (container.getECU() < vm.getResidueECU()) {
                fittedVM = vm;
                container.setVMType(vm.getType());
                vm.setResidueECU(vm.getResidueECU() - container.getECU());
                fitted = true;
                break;
            }
        }
        //need to create a new VM to fit this
        if (!fitted) {
            VM newCreateVM = new VM(VMType);
            vms.add(newCreateVM);
            container.setVMType(VMType);
            newCreateVM.setResidueECU(newCreateVM.getResidueECU() - container.getECU());
            fittedVM = newCreateVM;
        }

        List<Container> containerList = fitMapping.computeIfAbsent(fittedVM, k -> new ArrayList<>());
        containerList.add(container);
    }

    //deep clone, because the map will not change, but the VM/Container object's value will change
    return SerializationUtils.clone(fitMapping);
}
```

图 9: FFD 实际代码

4.2 实验环境搭建

本次实验使用的是 Java 实现的仿真平台，该平台可以读取 DAX 格式文件中的工作流，并生成工作流请求序列。本项目创建为 Maven 项目，基于 JDK17，最低版本支持 JDK8。同时，本次实现也加入了 Lombok，以简化相关类的方法（Getter，Setter 等），因此需要配置 Lombok 相关插件配置，如果不配置 Lombok 插件，只需要将类相关的方法补充完整，并去除 Lombok 注解即可。由于 Java 的可移植性，该程序可以在任何支持 Java 的操作系统上运行。

4.3 界面分析与使用说明

本次使用的是命令行的方式驱动程序的执行。一开始运行程序需要输入读取的 DAX 文件的路径，以及 factor 因子，见图 10。输入完成后回车，程序就可以自动执行。

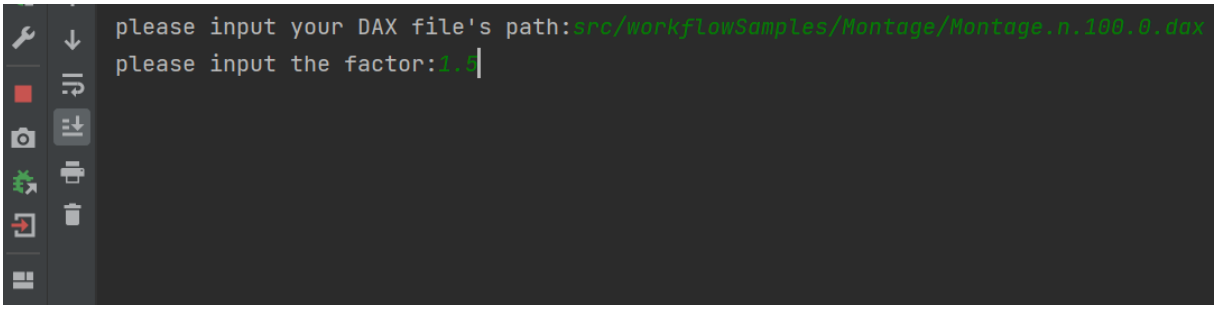

图 10: 程序输入

执行完成后，命令行将会打印一系列程序执行信息，见图 11。其中包括输入的 factor，是否成功读取了 DAX 文件，工作流的 DDL，未执行算法前最基础的 makespan，执行 CE 配置后算法的 makespan，cost（打印出的第一个成本是视当前容器为一个 VM 时的成本，第二个成本是忽略当前 VM 的具体配置的成本），经过 UWS 算法后还未分配的容器数，以及最后的算法执行完后的具体虚拟机数，最终成本，每一种类型虚拟机的具体数量，算法成功率，每一台 VM 具体信息及其所分配的容器。

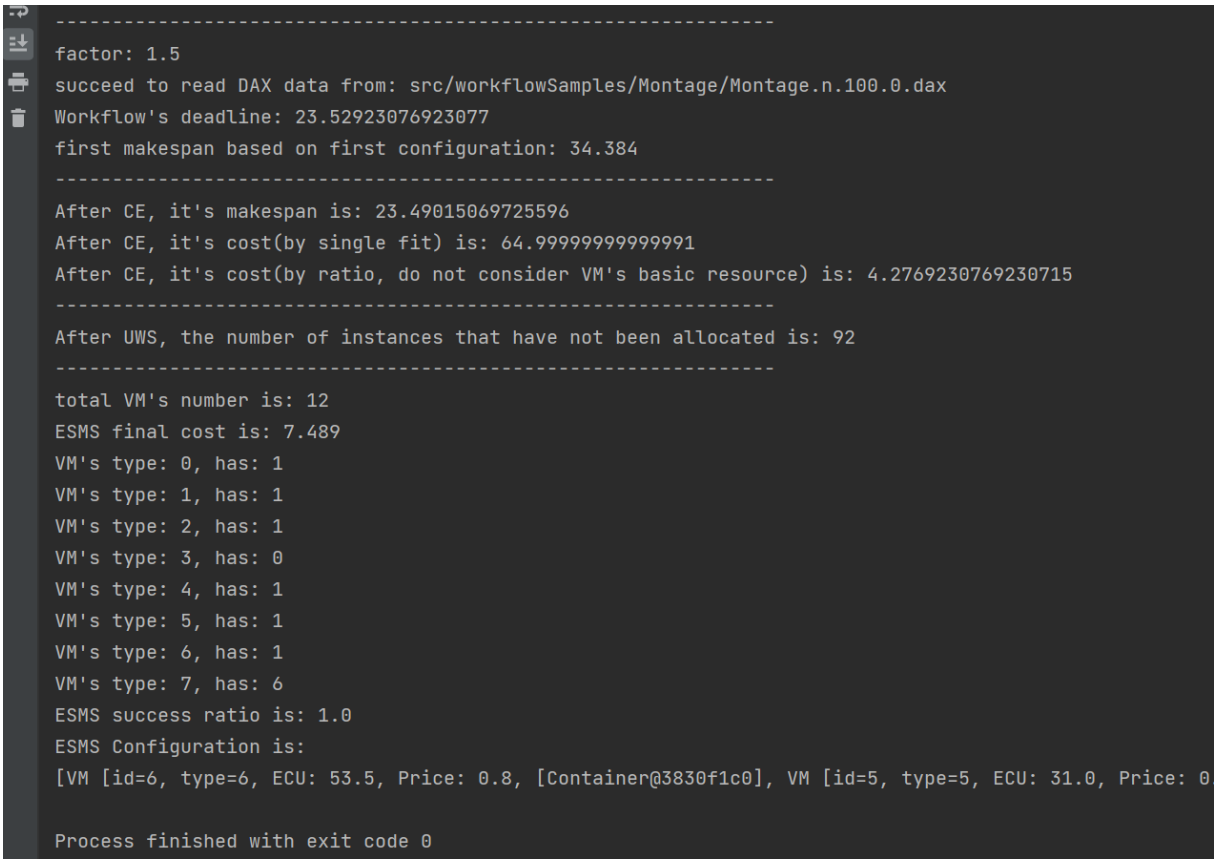

图 11: 程序输出

当我们将负载因子由小到大进行改变，即工作流的 DDL 变大了，约束变宽松了，会发现最后仿真结果得到的 VM 数量，和实际成本均有所减少，这也从另一方面说明了复现的算法具有可行性，具体实验结果可见第五节。

5 创新点

论文存在以下的值得改进的地方：

（1）没有提及初始的微服务实例的类型和数量，容器数量和配置，虚拟机数量，以及任务与微服务实例的对应关系；

（2）请求流和工作流是提前确定的，这样才能计算出每一个子任务的具体信息，算法才能进行下

去，但现实中每个微服务是无法确定它的具体某一个请求的工作量和具体执行时间的；

（3）UWS 时间复杂度高，可达到 n 的 5 次方；

（4）只考虑了新增 VM 的水平伸缩。

因此本次复现工作的创新是提出了一种云中成本最小化的通用资源伸缩算法—CSLV（Clustered resource Scaling and resource optimization of Lotka-Volterra model），算法关注容器的资源需求，从聚类以及优化过程等抽象层面出发解决通用性资源伸缩下的成本最小化问题。整个模型从容器资源出发，引入了容器层的聚类结构，实现基于虚拟机资源指标，请求速率的方差以及缓冲队列的伸缩判断机制，并基于 Q-Learning 实现 MDP 拟合伸缩过程，提出临近虚拟机资源搜索，离散化资源垂直伸缩，水平伸缩需求交付下一层算法解决的模型策略。下一层算法基于生态学模型拟合水平伸缩需求与虚拟机水平伸缩数，提出 k 曲率步长的资源相位点优化过程，最终得到最优的虚拟机水平伸缩数，该算法将通过 CloudSim 仿真实验对比三种伸缩算法—本次复现的 IFFD 算法，Best-Fit 算法，FFD 算法，其对应的实验结果将在第 6.2 节进行展示。

5.1 基于聚类与 Q-Learing 的通用容器伸缩算法

5.1.1 资源聚类划分与容器伸缩模型

目前大部分的微服务是单一的功能点，所使用的资源相仿，如提供数据存储的微服务更多依赖的是存储空间，网络带宽等资源，而一些后端逻辑业务处理微服务则需要更多的 CPU 型计算资源[13]。一般而言，一个容器对应于一个微服务，而多个容器对应于一台虚拟机资源，也就是说装载需要资源类型相同的微服务的容器之间所需硬件资源相仿。因此有必要进行对容器的聚类划分。

式 (1) 表示了容器的所需资源，可以根据式 (1) 提供的维度对所需不同资源的容器进行了聚类划分，若 dimension 维度过高，可进行 PCA 降维。不失一般性，本文假定将已有的容器划分为 K 类。

$$dimension = [CPU, Memory, disk, ..., etc] \tag{1}$$

本文参考了 Kubernetes 中的 HPA 以及 VPA 弹性伸缩模型，即一种 Pod 类型可以伸缩到不同虚拟机中。注意：Pod 本质上是容器[14]。本文将伸缩的粒度放大至容器层面，并根据当前请求速率进行容器的伸缩，当请求速率持续上升，或者猝发时，容器有必要进行拓展资源，当拓展资源时，又可能会涉及到可能会造成虚拟机的伸缩。在本节中，重点关注容器层造成的水平/垂直伸缩，确定容器的伸缩后，硬件层的 VM 水平伸缩将交给第 5.2 节的 Lotka-Volterra 模型处理。本节的模型见图 12。

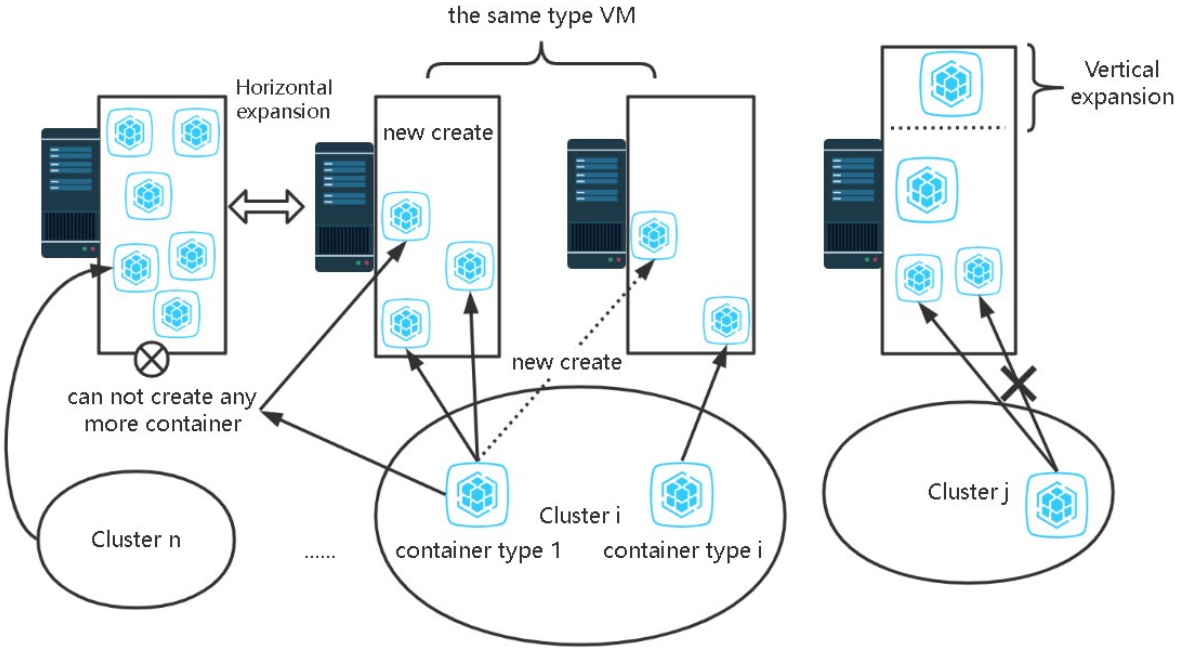

图 12: 容器伸缩模型

本文的容器伸缩时机受两种指标影响：VM 使用资源空间，当前微服务请求速率。前一种指标代表的意思是：如果目前 VM 资源不足以装载更多的新容器，需要进行资源拓展；如果当前 VM 资源过剩，当前容器不需要太多副本，需要进行资源缩减。后一种指标代表的意思是：当前请求容器中的微服务功能的数量增多，而当前不存在更多的容器资源处理请求时，需要进行容器拓展，反之亦然。可以把两种指标都指向同一个问题：如何在只关注请求数而不关心具体任务执行的情况下，在容器数量改变的时候，进行成本最小化的容器伸缩。

5.1.2 通用伸缩算法

定义两种类型的请求: 猝发异常型请求, 非猝发异常型请求. 假定请求为 Request, 将时间段 Ti 内的请求数量进行方差处理, 并与时间段 Ti-1 的方差进行比较, 若经过几轮比较后方差由大变得越来越小, 说明当前时间序列内的请求为非猝发异常型请求, 即当前请求可能为持续高并发的请求, 需要考虑容器的伸缩. 如果方差保持波动大的情况, 说明当前时间序列的请求为猝发异常型请求, 即当前请求仅存在一段波峰, 或者遭到恶意请求, 需要对这段方差波动大的请求进行缓和. 建立每个容器的可变长队列 Q, 队列长度见式 (2), 即队列长度为平均单位时间内处理的请求数.

$$|Q| = \frac{|Request - time|}{Time} \tag{2}$$

由于请求对于容器而言是透明的, 队列 Q 采用 FIFO 处理到达的请求. 队列 Q 存在的意义是: 对请求进行缓和. 收集容器所处的虚拟机的资源占用指标 metric, 当容器所处的 VM 对应的主要资源的 metric 上升, 当前请求数方差由大逐渐变得越来越小, 并且队列满, 此时选择请求压力最大的容器类型, 进行下述的基于 Q-Learning 的容器扩容处理. 即此时的请求已是平时处理的两倍, 请求持续保持高并发请求, 并且当前 VM 资源占用率持续上升, 必须进行扩容处理. 缩容与上述内容思路相反. 示意图如图 13所示.

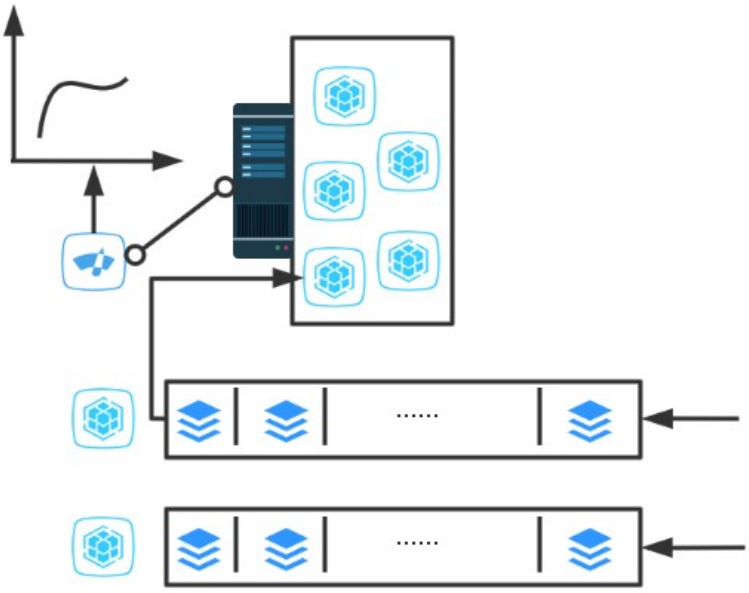

图 13: 需要扩容情况

我们参考[15]对资源进行离散化, 即每次伸缩的资源具有固定的单位数量. 利用 MDP 进行建模, 对 MDP 的几个要素进行定义: 智能体: 云; 策略: 选择配置以满足当前容器的伸缩需求; 状态: 当前容器; 回馈奖励: 成本/平均每个容器处理的请求数, 即若要使得奖励最大, 那么每一个请求的成本应该是最低的. 定义执行动作: 找到当前容器类型所处的聚类, 优先搜索距离当前容器类型最近的另一容器, 搜索方向根据容器拓展或缩减而定, 即如果是容器需要进行拓展, 按图 14分析, 应该在右半部分开展搜索工作, 对搜索到的容器, 查询对应虚拟机是否可以继续装载当前容器, 否则继续寻找下一个容器类型, 直至右半部分的 VM 都搜寻完毕, 若无法进行分配那么尝试以最小离散资源单位为增量单位进行资源垂直伸缩, 否则进行水平伸缩, 图 14为示例图.

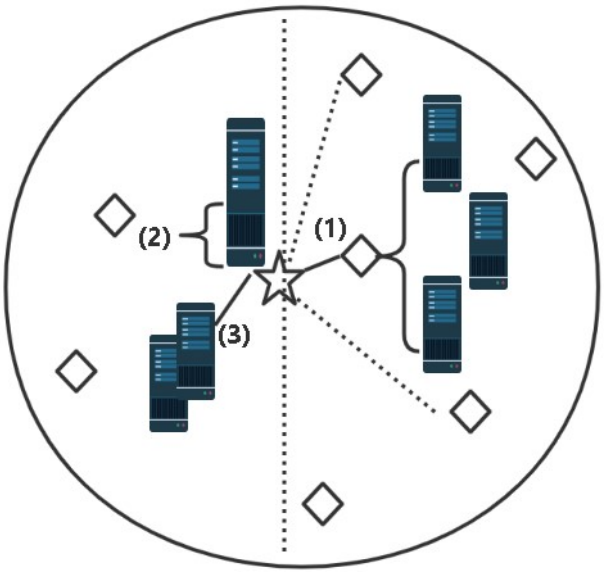

图 14: 基于聚类的动作选择

使用 Q-Learning 对上述 MDP 进行实现, 公式见式子 (3). 由于 Q-Learning 选择动作时遵循地策略和更新动作值函数时遵循的策略不一致, 即此处的 Q-Learning 使用的是 Off-Policy.

$$Q(St, at) = Q(St, at) + \alpha[Rt + max - \alpha Q(St + 1, a) - Q(St, at)] \tag{3}$$

其中:

(1) α 为学习率, 学习率越大, 保留此前训练的效果就越少;

(2) λ 为折扣因子, 折扣因子越大,max-αQ(St+1, a) 所起的作用就越大;

(3) St 表示在 t 时刻系统的状态, at 表示在 t 时刻选择的动作;

(4) Rt 表示在时刻下, 系统状态为 St 时, 选择 at 所获得的即时奖励;

(5) max-αQ(St+1, a) 表示下一状态中, 选取使得价值函数最大的动作时产生的价值, 此处使用的是贪心算法, 若使用 Sara 算法, 那么此处应被替换为 Q(St+1, at+1), 即 Sara 算法属于 On-Policy, 即依旧使用更新动作值函数时所使用的选取策略.

在本次的 Q-Learning 算法中使用了 ε-greedy 策略, 也就是在进行 max-αQ(St+1, a) 计算时, 此前的算法会直接选择最大值 max-αQ(St+1, a), 而其它值却得不到选择. 在算法中引入一个常量 ε, 当概率大于 ε 时, 算法将按照贪心策略选择使动作值函数 Q(St+1, a) 得到最大值的动作 max-α, 否则算法将随机选取下一动作, 这样就避免了算法陷入局部最优的问题.

经过学习后, 若选择的最佳动作是当前容器需要进行水平伸缩, 那么将该需求进行记录, 交给下一节的 Lotka-Volterrra 模型进行进一步地优化.

5.2 基于 Lotka-Volterra 模型的 VM 资源优化

5.2.1 Lotka-Volterra 模型

经典的 Lotka-Volterra 是两个相互依赖的种群生长的生物学模型. 该模型处理任意两种动物, 并检验生存的相互依赖性. 这种相互依赖可能是因为一个物种作为另一个物种的食物. 以兔子 (猎物) 和狐狸 (捕食者) 为例. 这是两个相互依赖的物种, 兔子是狐狸的食物来源. 根据 Lotka-Volterra 模型定义, 本文构建了两个式子描述种群间的数量变化关系. 式 (4) 和式 (5) 反应了物种数量随时间的变化数量.

$$\frac{dP}{dt} = \alpha P - \beta PQ \tag{4}$$

$$\frac{dQ}{dt} = \delta PQ - \gamma Q \tag{5}$$

其中,P 为猎物的数量,Q 为捕食者的数量,α 是猎物在无捕食情况下的自然繁殖率,β 是猎物每次遭遇捕食的死亡率,γ 是捕食者在无食物情况下的自然死亡率,δ 是捕食者的繁殖率. 式子 (4) 和式子 (5) 代表两个种群随时间的增长率. 当种群数量稳定时, 式 (4) 和式 (5) 均等于 0, 即调整参数后, 存在一个稳定点, 此时种群数量都围绕着一个静止点而变化, 图 15所示为相位图, 其中横纵坐标分别反应了两个种群的种群数量, 图中横坐标表示为捕食者种群, 纵坐标为猎物种群.

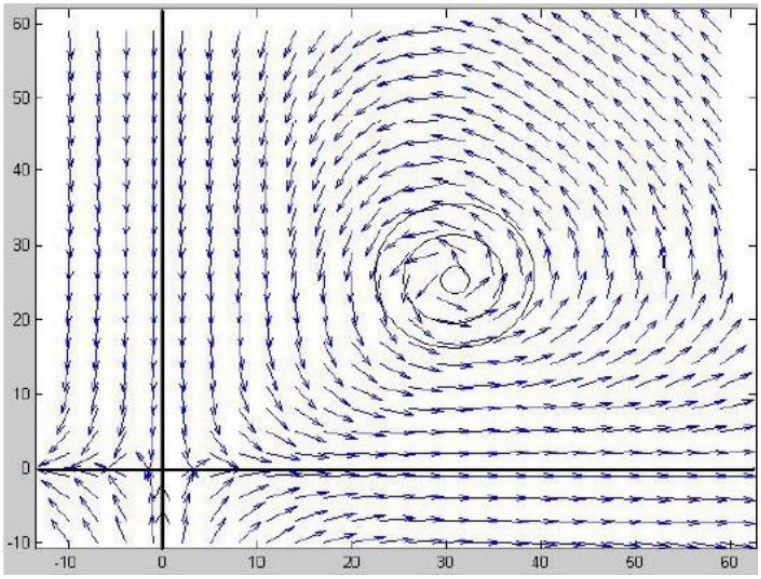

图 15: 静止点相位图

5.2.2 响应容器伸缩需求的分配

在初始时, 静态地根据容器伸缩需求分配虚拟机资源时, 会导致资源过剩, 或者资源供应不足, 如存在新租用的虚拟机空闲时间占了绝大部分的租用时间, 因为其仅响应了一次伸缩需求, 没有装载再多的容器, 计算资源被闲置. 因此, 可以应用 Lotka-Volterra 模型进行动态按需添加资源. 将虚拟机比作猎物, 容器伸缩需求 (Require) 比作捕食者. α 为虚拟机在没有 Require 的情况下的新建率, β 为系统中虚拟机由于租期到期而导致的释放率,γ 为容器伸缩需求得不到虚拟机处理的失败率, δ 为请求速率. 这些参数将被用于控制云系统. 根据容器伸缩需求数与虚拟机的比例, 将系统分为过载, 平衡负载, 以及欠载三个阶段. 系统过载是单个虚拟机负责的需求数在单位时间内过高; 系统平衡负载是单个虚拟机处理的需求数恰好, 并且存在资源余量; 系统欠载则是虚拟机资源过剩, 判定某一虚拟机在租期结束后需要释放资源, 以减少成本. 令 (Xs,Ys) 为平衡点 stable, 以 stable 为原点向外扩张的同心圆, 以及圆上的不同位置反应了系统的不同阶段, 目标是不断根据变化的请求数, 修正 stable 位置, 从而确定 VM 资源数. 建立初始的 Require-VM 相位图, 设定时间段 t 大于单位时间, 取 t 时间段内单位时间内最大需求数 Rmax, 和最小需求数 Rmin, 设这两点在相位图中分别为 uncertain-max,uncertain-min. 此时这两点在相位图中处于 VM=Xs 的直线上, 即处于 stable 点的垂直端, 并分别位于不同的同心圆上. 一个例子如图 16所示.

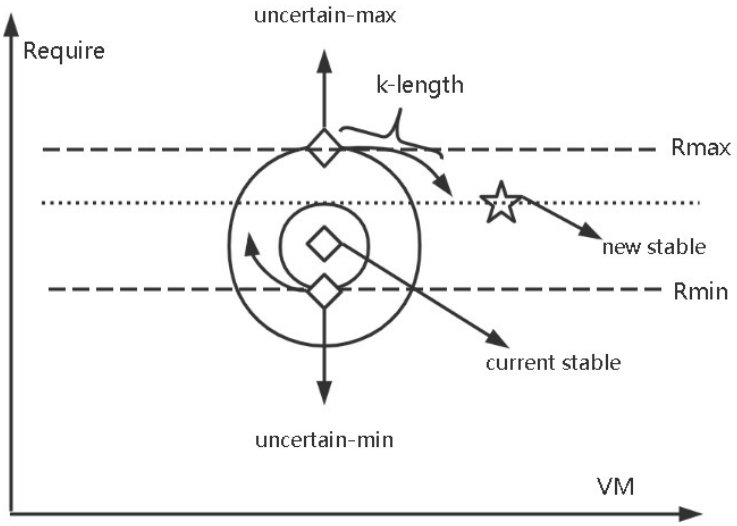

图 16: 资源优化过程

对两点进行优化, 此时 uncertain-max 应按顺时针沿该点在圆上小于该同心圆的曲率的 k 长度进行移动. uncertain-min 应按顺时针沿该点在圆上小于该同心圆的曲率的 k 长度进行移动. 注意: 此处的 k 类似于机器学习中的学习率, 范围在 [0,πR/4], 其中 R 要比当前的同心圆半径要略大, 其意义是步数不能大于这个曲率的圆的 1/8 周长. 顺逆时针方向的确定应由是否够满足合理的资源分配这一条件来确定. 如 uncertain-max 在 stable 上方, 说明当前已处于系统过载的阶段, 沿顺时针方向能够增加虚拟机数量从而确保能够满足比当前更大的伸缩需求, 如果 uncertain-max 在 stable 下方, 即当前系统处于系统欠载阶段, 那么此时应该进行逆时针优化, 促进虚拟机缩小操作. 在优化过程中, 两点不能越过 Require=Rmax 以及 Require=Rmin 这两条直线, 即优化过程始终满足最高需求以及最低需求. 当 uncertain-max 与 uncertain-min 处于同一 require 水平时，优化停止，并按照式 (4) 与式 (5) 相等这一条件找到新的 stable 点, 此时 stable 点能够满足当前系统的平均伸缩需求, 维持系统处于平衡负载阶段, 对应的虚拟机数为最终需要伸缩的成本最小化的最优数量.

6 实验结果

6.1 复现实验结果分析

实验采用了三种算法进行对比，这三种算法分别是：ProLiS，SCS，IC-PCPD2。实验的指标为任务成功率和实际成本，实验条件为不同的工作流和不同的工作负载，以及不同的 factor 因子。详细的实验结果分析如下：

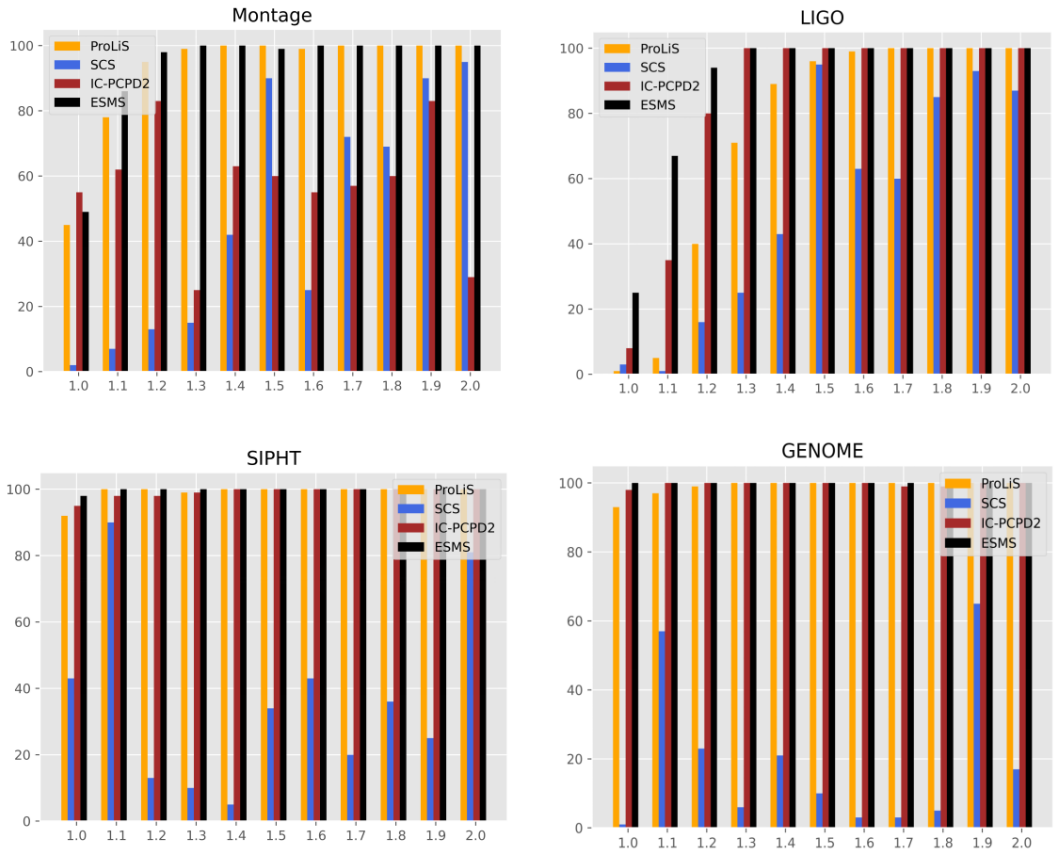

图 17: 不同工作流以及稳定工作负载下的成功率

从图 17可以看出，随着 factor 因子的增加，成功率逐渐增加，但 SCS 算法除外。结果表明，ESMS 系统的性能优于其他系统。从图 18中的数据点数量可以看出，ESMS 可以获得比其他方法更可行的解决方案。特别是在 LIGO 实验中，ESMS 的成功率分别比 ProLiS、SCS 和 IC-PCPD2 高 18.31%、36.39% 和 6.02%。SCS 的成功率波动很大，没有可行的解决方案。SCS 算法不同于其他三种算法。SCS 首先在 EDF 调度算法前通过负载向量估计所需的实例数。然而，负载向量的预测结果与 EDF 调度算法的需求之间存在微小差异，这导致一些任务被迫排队。由此产生的延迟在工作流执行期间逐渐累积，最终导致工作流超时。此外，随着因子的增加，VM 的成本效益类型发生变化，其配置也逐渐减少。因此，即使最后期限因素放宽，成功率也不会增加。另外三种算法都是基于任务调度的，可以通过一种调度方案来确定资源需求，因此它们的成功率较高。结果还证明了任务调度与自动伸缩相结合的有效性。

IC-PCPD2 的性能也不稳定：它可以获得高成功率、低成本和接近 ESMS 的 GENOME 可行解数量，但在 Montage 中无法获得任何可行解。对于 SIPHT，图 19中显示 IC-PCPD2 的平均成功率接近 ESMS，但 IC-PCPD2 只有三种可行的解决方案，如 18所示。这是因为 IC-PCPD2 的大多数成功率高于 99%，但低于 100%。尽管其他三种工作流的成功率差异很小，但 ESMS 的成本最低。

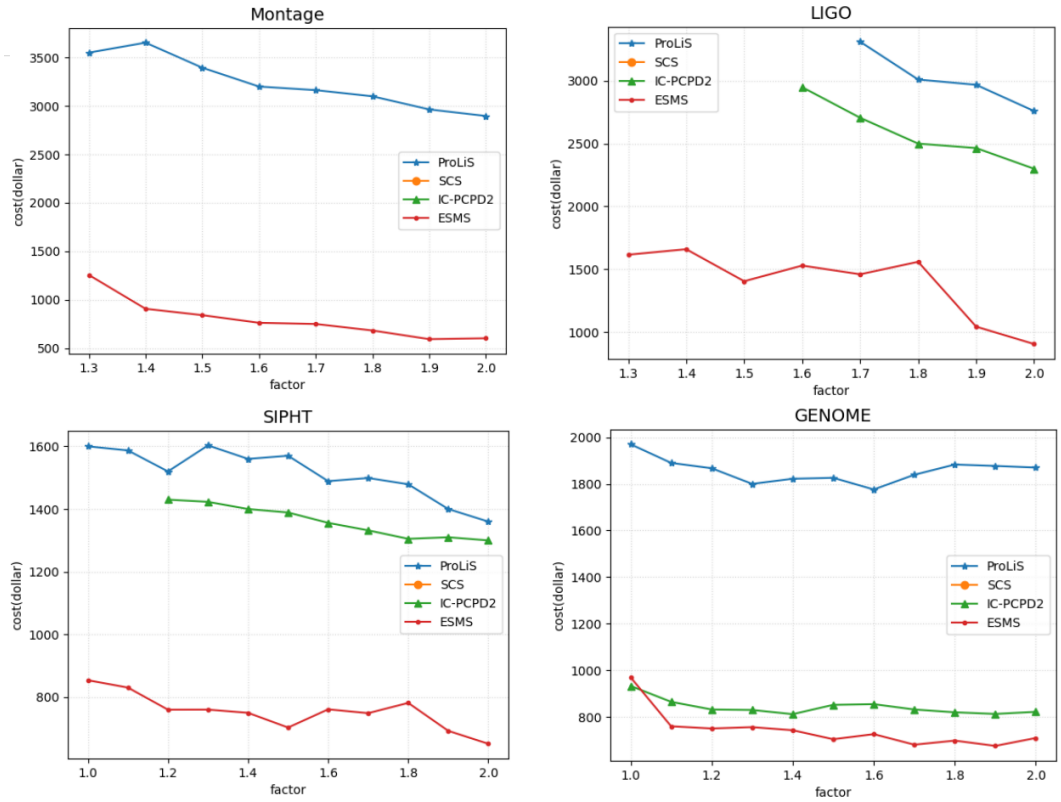

图 18: 不同工作流以及稳定工作负载下的成本

在成本方面，从具有不同因子值的实验中收集所有可行的解决方案，标准化它们的成本并计算平均成本，如图 14所示。ESMS 的成本显著低于 ProLiS，分别降低了 79.08%、6.80%、15.37% 和 18.29%。除了 IC-PCPD2 在 Montage 中没有可行的解决方案外，ESMS 在其他三种工作流中的成本分别比 IC-PCPD2 低 4.57%、0.58% 和 11.39%。

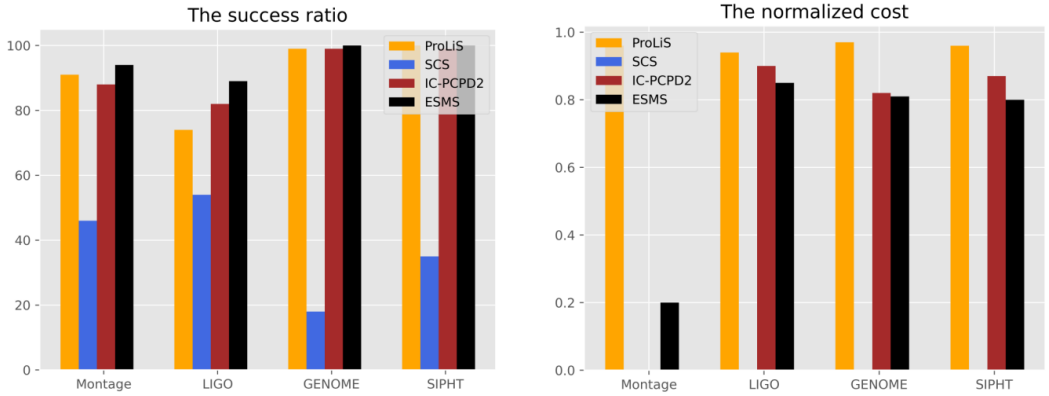

图 19: 不同工作流下的平均成功率和平均标准化成本

在图 18中，ProLiS 的成本始终很高，并且不会随着因子的增加而降低。这是由于期限分配不合理造成的。在 Montage 中，mAdd 和 mShrink 之间的数据传输更多，ProLiS 使用最快的速度分发截止日期，导致 mAdd 的子截止日期太小。因此，在任务 mAdd 之前执行的所有任务都必须在短时间内完成，这需要更多的实例和更高的速度，从而导致更高的成本。例如，在某个工作流实例中，mAdd 的子截止日期为 10.5909，工作流的生成时间为 53.7460，远小于截止日期 71.9009。ESMS 使用 CE 对应的速度分配截止日期，使 mAdd 的 33.8911 子截止日期和最大完成时间 69.6119 更接近截止日期，减少了不必要的成本。在其他三个工作流中，ESMS 的成本最低，因为每种类型的实例都添加了合理的冗余，从而减少了资源浪费。

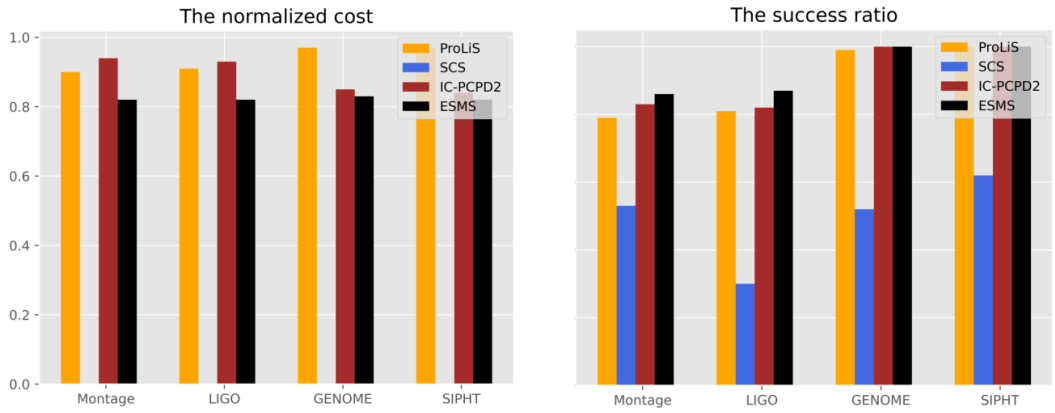

图 20: 不同工作负载模式下的平均成功率和平均标准化成本

在不同工作负载模式的实验也显示出与上述类似的结果：1）SCS 的成功率仍然是随机的，可行解的数目为 0。因此，在图 20中没有成本条。2）在 LIGO 的三个实验中，ESMS 保持了最低的成本和最高的成功率。3）在 GENOME 的三个实验中，ESMS 优于 ProLiS 和 SCS，ICPCPD2 的性能接近 ESMS。

6.2 创新实验结果分析

本次创新实验关心的并不是任务调度的成功率/是否满足最后期限的约束, 而是本次资源伸缩算法带来的经济效益, 即最终花费成本是本次实验指标之一. 由于容器的伸缩并不一定会带来虚拟机的水平/垂直伸缩, 因此本次实验还关心虚拟机数量的变化率 (拓展/缩减虚拟机均会改变虚拟机的数量, 其意义是在当前时刻虚拟机数量相比初始时数量的比率). 同时对于 Lotka-Volterra 模型算法中的 k-length 进行统计分析, 得出本次实验最佳的 k-length, 并用于接下来的对比实验.

通过改变不同容器水平伸缩需求量范围, 实验以平均需求量以及新增虚拟机成本作为 k-length 的统计分析依据. 在正式实验中, 我们固定任务调度算法为 UWS, 并将 CSLV 算法同期对比其它三种资源伸缩算法, 分别为 BFB 算法,IFFD 算法,FFD 算法, 最终统计成本, 以及虚拟机数量的变化率. 我们设置了三种请求速率类型的实验, 每种实验均运行四种科学工作流.

将需求量的范围设定在 [0-100], 即平均需求量的范围也为 [0-100].k 的范围在 [0,πR/4], 因此将 πR/4 作为基本单位, 测试了 (0.0,1.0] 范围内的 10 个数, 即从 0.1 每次递增 0.1 的 10 个参数 λ. 这些参数分别对应 k 在范围内的不同值. 对这些参数分别进行 10 组实验, 每组实验均持续固定时间, 重复实验 100 次, 取平均值, 实验结果见图 21.

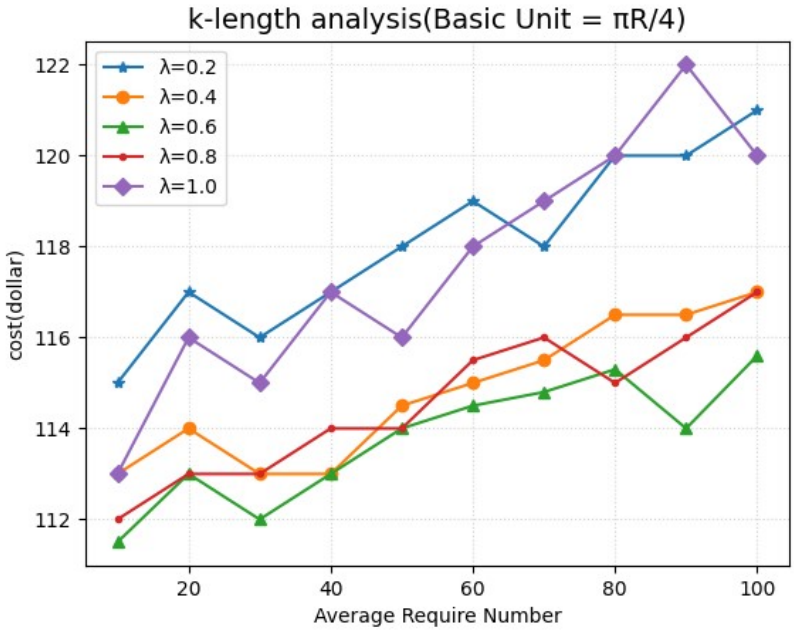

图 21: k-length 参数分析

从图 21可知, 当参数过小或过大时, 每次同心圆的点走的步数过小或过大, 得到的最终 stable 点精确度不够高, 造成成本增加, 并且波动较大, 也就是虚拟机分配增加. 因为步数过小或过大, 并不能适应两个 uncertain 点的变化率, 即两个点所处的同心圆曲率偏差太大, 且步数导致优化过程太长/太短, 造成优化不精确. 在参数靠近中心位置时, 随着需求数增加, 成本变化维持在一定范围内. 在 λ 处于 0.4-0.6 附近时, 成本接近所有参数的最小范围, 并且波动较小. 因此接下来我们使用 [0.4-0.6] 的中间值 0.5 进行接下来的对比实验.

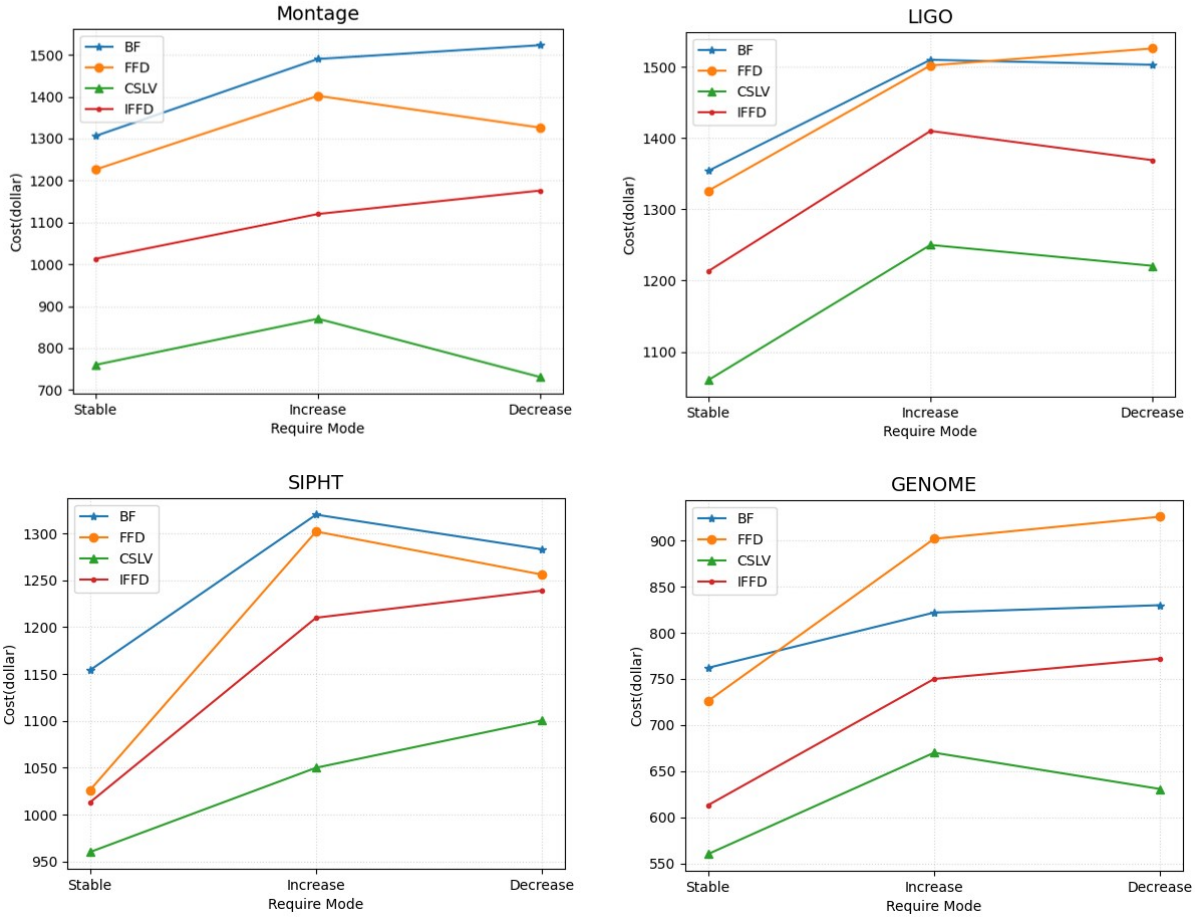

图 22: 成本对比

由于 CSLV 将使用不同资源的容器进行划分, 并且考虑了水平与垂直伸缩, 仅关注请求速率, 而不关注具体执行的任务调度, 同时加入了队列以应对猝发异常型的请求. 因此, 在图 22中, 使用 CSLV 算法后得到的成本要比其它三种对比算法低, 并且比与 UWS 搭配的 IFFD 算法成本要低 20% 左右. 由于 CSLV 算法能够进行 VM 资源的释放, 面对 Decrease 类型的请求时, 能够使用队列缓和猝发的请求, 并且随着请求速率的降低, CSLV 算法能够及时地释放一些资源, 以达到减少成本. 对于 BestFit 和 FFD(First Fit Descend), 这两种算法虽然适用于资源分配, 但不针对具体调度, 不关心成本问题, 只关心资源是否能够分配完, 因此这两种算法的成本较高. 而 IFFD 算法则是 BF 与 FFD 的结合, 并针对本次实验的科学工作流做出了对应的调整, 因此整体成本会比 BF 和 FFD 要小, 但由于没有关注所有 VM 的资源利用, 也没有关注垂直伸缩, 仅关心 UWS 得到的容器是否得到分配, 即其本质与 BF 与 FFD 一致, 因此整体成本要比本论文提出的 CSLV 算法要高.

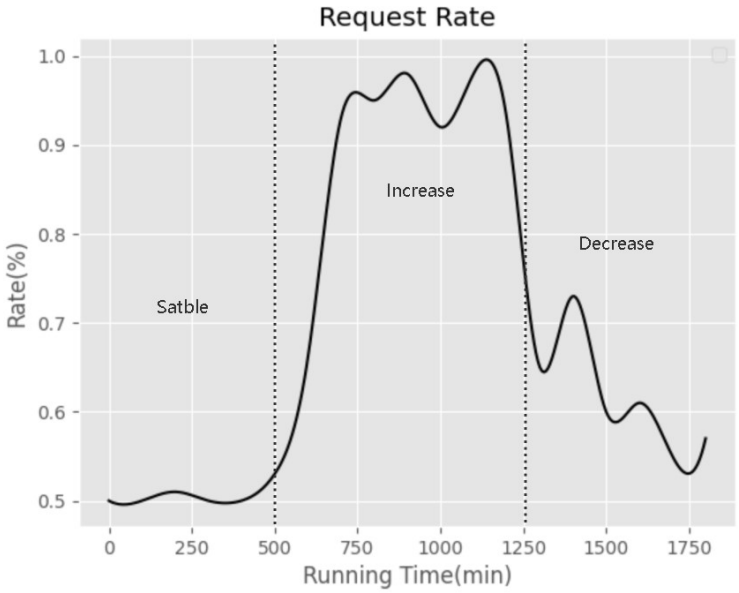

图 23: 三阶段请求速率

本次实验生成了在 Stable-Increase-Decrease 三种模式交替变化下的请求速率, 三阶段请求速率曲线见图 23. 并在该请求速率下运行四种工作流应用程序, 得到四种算法的虚拟机数量的变化速率, 见图 24.

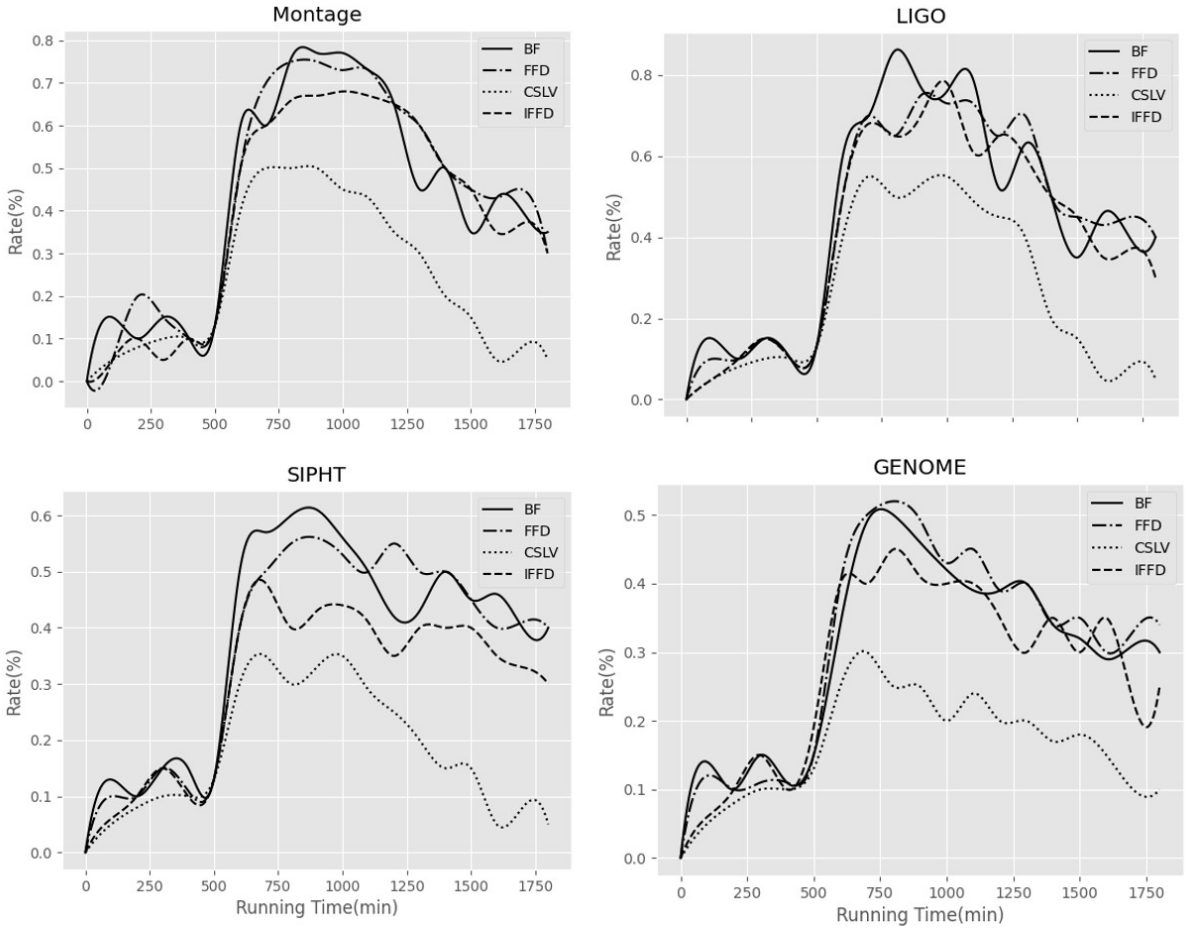

图 24: 虚拟机数量变化率

从图 24中可以看出:CSLV 算法的虚拟机变化率始终比其它算法要低, 这也反应了 CSLV 算法的成本要比其它算法要低. 从图中可以看出 BF,FFD,IFFD 很难适应请求率大幅度变化的场景, 因为对于这些算法而言, 请求增加就是需要新建虚拟机处理. 在 Increase 模式下,CSLV 算法与其它三种算法的 VM 变化率相比要小, 并且波动幅度也较小, 这是因为算法通过容器的聚类伸缩算法减少了不必要的 VM 水平与垂直伸缩, 同时本文提出 Lotka-Volterra 模型算法也使得伸缩需求与 VM 数量始终保持在平衡点范围. 在 Decrease 模式下, 由于请求速率下降, 对应请求数量下降, 因此不需要更多的虚拟机来进行处理, 因此所有算法的 VM 变化率均下降,CSLV 算法波动较小, 对比的三种算法波动较大, 这是因为在 Decrease 模式下, 请求数量的方差波动大, 三种算法不能适应这种波动, 而 CSLV 具有队列以及指标的判断机制, 从而缓和了这些波动。

7 总结与展望

本文介绍了论文的相关背景与目的，其它与本文相关的工作，以及论文中提出的弹性调度算法 ESMS。该算法用于云中微服务，并以集成的方式处理任务调度和自动伸缩。算法基于工作负载的统计信息确定容器配置，并适应流式工作负载。不仅如此，算法在考虑容器图像的影响下，实现了基于 VSBPP 的缩放算法，解决了由于引入容器层而引起的两层缩放问题。在个人创新部分，我也引入了通用的资源水平与垂直伸缩算法。最终，通过使用工作流应用程序的仿真实验，验证了复现算法在提高云中微服务成功率和成本的能力。

不仅如此，本文也基于复现工作提出了一种创新性的通用资源伸缩算法—CSLV，用于云中容器-虚拟机双层架构的资源成本最小化场景。该算法首先根据容器资源划分聚类，并引入了缓冲队列和资

源指标作为监控容器伸缩的机制，在资源伸缩时，采用 MDP 建模，将资源伸缩奖励化，提出的伸缩策略可分为临近资源伸缩，水平/垂直伸缩。对于需要进行 VM 水平伸缩的需求，将其与 VM 之间的数量关系转化为生态学模型，并提出了优化过程。最后使用 CloudSim 进行仿真实验，实验结果验证了本论文提出的算法在最小化成本，提升在资源伸缩场景下的鲁棒性以及通用性上具有不错的效果。

在未来，我们可以计划研究如何为基于微服务的系统构建一个通用的性能模型，以准确预测不同配置下每个微服务的响应时间。其次，考虑到云资源的多样性，Spot 等其他计费模型也将会是另一个主要的研究方向。

参考文献

[1] ABRISHAMI S, NAGHIBZADEH M, EPEMA D H. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds[J]. Future generation computer systems, 2013, 29(1): 158-169.

[2] WU Q, ISHIKAWA F, ZHU Q, et al. Deadline-constrained cost optimization approaches for workflow scheduling in clouds[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(12): 3401-3412.

[3] WU H, HUA X, LI Z, et al. Resource and instance hour minimization for deadline constrained DAG applications using computer clouds[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 27(3): 885-899.

[4] GENEZ T A, BITTENCOURT L F, MADEIRA E R. Time-discretization for speeding-up scheduling of deadline-constrained workflows in clouds[J]. Future Generation Computer Systems, 2020, 107: 1116-1129.

[5] ZHENG W, SAKELLARIOU R. Budget-deadline constrained workflow planning for admission control [J]. Journal of grid computing, 2013, 11(4): 633-651.

[6] BAO L, WU C, BU X, et al. Performance modeling and workflow scheduling of microservice-based applications in clouds[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(9): 2114-2129.

[7] QIN Y, WANG H, YI S, et al. An energy-aware scheduling algorithm for budget-constrained scientific workflows based on multi-objective reinforcement learning[J]. The Journal of Supercomputing, 2020, 76(1): 455-480.

[8] ZHENG T, ZHENG X, ZHANG Y, et al. SmartVM: a SLA-aware microservice deployment framework [J]. World Wide Web, 2019, 22(1): 275-293.

[9] KAN C. DoCloud: An elastic cloud platform for Web applications based on Docker[C]//2016 18th international conference on advanced communication technology (ICACT). 2016: 478-483.

[10] FERNANDEZ H, PIERRE G, KIELMANN T. Autoscaling web applications in heterogeneous cloud infrastructures[C]//2014 IEEE international conference on cloud engineering. 2014: 195-204.

[11] MAO M, HUMPHREY M. A performance study on the vm startup time in the cloud[C]//2012 IEEE Fifth International Conference on Cloud Computing. 2012: 423-430.

[12] JUVE G, CHERVENAK A, DEELMAN E, et al. Characterizing and profiling scientific workflows[J]. Future generation computer systems, 2013, 29(3): 682-692.

[13] HOENISCH P, WEBER I, SCHULTE S, et al. Four-fold auto-scaling on a contemporary deployment platform using docker containers[C]//International Conference on Service-Oriented Computing. 2015: 316-323.

[14] GUERRERO C, LERA I, JUIZ C. Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications[J]. The Journal of Supercomputing, 2018, 74(7): 2956-2983.

[15] WANG S, DING Z, JIANG C. Elastic scheduling for microservice applications in clouds[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 32(1): 98-115.