基于强化学习的SSD长尾延迟优化方法综述

刘成1)

1)(华中科技大学计算机科学与技术学院 湖北 武汉 430074)

摘 要 固态硬盘（SSD）在各种系统中得到广泛的应用，从实时云服务系统到企业服务器系统。SSD具有写前擦除的特性，因此我们需要地址转换和垃圾回收等闪存管理方法来管理SSD中的数据，而垃圾回收（GC）和I/O合并是闪存系统中产生长尾延迟问题的原因之一。长尾延迟比99%处的平均延迟大100倍以上，无法满足实时系统和质量要求严格系统的服务质量需求，因此，长尾延迟的研究与优化显得尤为重要，是近年来学术界和工业界共同关注的热点。为进一步深入研究强化学习对固态硬盘（SSD）长尾延迟的优化效果，该文通过梳理相关研究工作，总结了两种利用强化学习减轻SSD长尾延迟的技术方法。在请求间隔时间使用垃圾回收和针对不同I/O模式使用合适的I/O合并方式。针对每一种方法，该文深入分析了其思路动机、主要原理和算法细节，最后给出了强化学习针对SSD长尾延迟优化的优势。

**关键词** 垃圾回收；I/O合并；强化学习；长尾延迟；固态硬盘

Survey on Reinforcement Learning to Reduce Long Tail latency in SSD

LIU Cheng1)

1)(School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074)

**Abstract** Solid State Drive (SSD) is widely used in various systems, from real-time cloud systems to enterprise server systems. SSD has the characteristics of erasing before writing, so we need flash management methods such as address translation and garbage collection to manage data in flash memory. Garbage collection (GC) and I/O consolidation are the causes of long-tail latency problems in flash memory systems. one. The long-tail latency is more than 100 times larger than the average latency at 99%, which cannot meet the service quality requirements of real-time systems and systems with strict quality requirements. Therefore, the research and optimization of long-tail latency appear to be particularly important in recent years in academia and industry. Hot spots of common concern. In order to further study the optimization effect of reinforcement learning on the long-tail latency of solid-state drives (SSD), this article summarizes two technical methods to reduce the long-tail latency of SSD by combing related research work. Use garbage collection at the request interval and use appropriate I/O merging methods for different I/O modes. For each method, the article analyzes its motivation, main principles and algorithm details in depth, and finally gives the main research points and development directions of reinforcement learning for SSD long-tail latency optimization

**Key words** Garbage Collection; I/O Merge; Reinforcement Learning; Long-tail Latency; Solid State Drive

# 引言

固态硬盘现在已经广泛的部署在各种计算机系统中，从实时云服务系统到企业服务器系统。因此，针对SSD的长尾延迟优化对于系统保障服务质量具有现实意义。SSD在各个方面都与传统硬盘不同，例如随机访问性能高，功耗低与写前擦除等特性。因此垃圾回收（GC）和I/O合并都会导致长尾延迟问题。

由于写前擦除的特性，SSD使用一个闪存管理层（FTL）来管理数据，当有新数据写入时，FTL会将新数据写入分配的新页中，更新逻辑地址和物理地址之间的地址映射信息。如果空闲块不足，则通过回收已使用块中的未使用空间来获得空闲块。要做到这一点，受害者块的有效页被复制到一个新的块。然后擦除受害者块以获得一个空闲块，这个过程称为GC。因为GC过程中的页面复制和块擦除操作非常耗时，会导致长延迟的问题。GC导致的较长的写延迟不仅会增加写延迟，而且还会增加读延迟，因为GC会阻塞后续读请求的服务。目前已经存在各种各样的研究来减轻SSD中的长尾延迟。Kang等人[1]提出了基于强化学习（RL）的GC，即RLGC方法，利用存储访问中的请求间隔时间来隐藏GC延迟。RLGC调度器使用RL了解系统行为，并利用空闲时间调度GC，确保能够最小化GC导致的延迟增长。GC调度器决定要执行多少部分GC操作来减少GC引起的延迟，如页面复制的数量。RLGC使用Q-table方法来存储对应状态和动作的q值。然而状态信息的数量越大，长尾延迟的降低也就越大。但是，为了保存大量状态信息，大型Q表会消耗大量的内存。因此Kange等人[2]Q表缓存（QTC）方法减低Q表内存开销，同时将在大量的状态候选中最近被访问过的状态存储起来，并通过有效地利用大量的状态进一步减少长尾延迟。然而QTC通过零初始化来初始化Q表中的条目，增加了学习的难度与效果。因此Kang等人[3]提出Q值预测神经网络(QP Net)来初始化Q值。QP Net学习系统运行时的行为，从在插入一个新的条目到QTC时，能够提供良好的初始Q值。这改进了QTC上的Q-learning，最终有助于进一步减少长尾延迟。

I/O起源于传统的机械硬盘，它会合并尽可能多的连续请求，并将它们作为一个请求来服务。通过减少占I/O延迟最大比例的磁盘查找延迟，它不仅可以提高系统吞吐量，还可以提高请求服务时间。相反，I/O合并对SSD的系统吞吐量的主要好处是更好地利用了设备并行性和摊销的I/O完成开销。最近的研究表明，I/O合并会影响单个I/O的响应时间。由于共享完成时间，所有合并的I/O的服务在处理批中的最后一个请求之前无法完成。一方面，合并请求的大小增加延长了请求服务响应时间。另一方面，SSD的尾延迟可能比机械硬盘长得多，I/O合并将延迟扩散到所有合并的请求。在延长的尾延迟和合并请求和系统吞吐量所经历的延迟变化之间存在权衡。由于不同应用程序的I/O工作负载的多样性，基于静态阈值的方法不能完全解决各种应用程序在追求系统吞吐量时带来的挑战，因此，静态解决方案要么会降低系统吞吐量，要么在缓解长尾延迟问题方面效率较低。因此Chao Wu等人[4]提出一种基于强化学习(RL)的I/O合并决策自适应方法，该方法I/O调度程序队列中未决请求的特征动态执行I/O合并该系统设法提供高吞吐量和低I/O性能变化以及减少尾部延迟。

为了进一步深入研究强化学习在SSD长尾延迟问题的优化，接下来本文首先介绍以上四篇文献所使用的强化学习技术，然后将详细描述以上四篇文献的主要原理、关键技术和优化效果，并对强化学习方法在长尾延迟上的应用特点进行总结。

# 背景

强化学习的目标是解决此类问题：环境感知型代理人如何选择最佳行动以通过学习最大化一定的回报。图1描述了智能体（例如，这项工作中的批处理管理器）和环境（例如，存储系统）之间的交互架构。每个时间步骤中的状态转换都由来自代理的操作激活，该操作根据学习的策略进行选择。之后，从环境中感知到下一个状态，以及识别最后一个动作正确性的奖励。RL的目标是学习可以最大化长期预期收益的政策。

根据马尔可夫决策过程（MDP），当前状态的奖励应由当前奖励和未来奖励组成，如公式（1）所示

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

其中为当前状态的奖赏或即时奖赏，表示奖励在状态,也就是未来的奖励对即时奖励。未来奖励的积累旨在预测当前行为对未来状态的影响。状态是时刻步长，是学习的策略，因此表示状态的累积奖励。由于其随机性，预测时间越长，预测结果越不准确。因此，对每一个未来奖励都加一个折扣因子。

给定累积奖励的计算函数，下一个问题是推导长期奖励并据此做出决策。本文利用Q-learning算法来探究长期报酬与相应状态-行动对之间的关系，其基本表达形式为公式（2）

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

这个公式是通过学习策略，选择一个行动的最大奖励,并获得在状态-动作处的长期奖励。贝尔曼（Belman）公式（3）是策略学习的迭代解，提出通过确定的行动的加权累积奖赏更新每个状态-行动的Q值。

|  |  |  |
| --- | --- | --- |
|  |  | (3) |

其中，是学习速率。这些迭代结果保存在Q表中，Q表作为数据库存储每个状态动作对的学习Q值。因此，在智能体在决策时可以参考Q表。

其次，在当前状态下存在未学习的动作时，通过算法来选择动作。这让智能体能够“随机选择一个行动或在所有学习行动中选择一个奖励最大的行动。

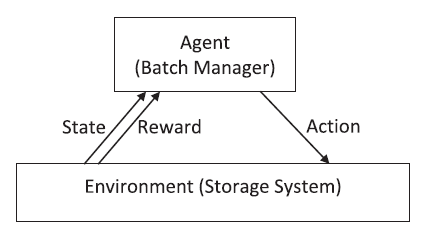


图1 智能体与环境之间的交互

# 研究方法

## 基于强化学习的垃圾回收（RLGC）

（1）思路动机

图2显示了60K请求的请求间隔时间的分布，以及我们实验中使用的实际工作负载。x轴表示请求间隔时间，y轴表示每个区间中请求的频率。如图2所示，存储系统有当量空闲周期。可以利用这样的空闲时间执行GC操作。这个问题的难点是当前空闲期的长度是未知的。为了解决这个问题，有几种技术使用在设计时确定的固定策略。因此，由于不同的程序运行或阶段，它们在适应动态变化的存储访问行为方面受到限制。因此，Kang等人[1]一种强化学习辅助的自适应GC方法，它在线学习存储访问行为并调整GC以减少长尾延迟。

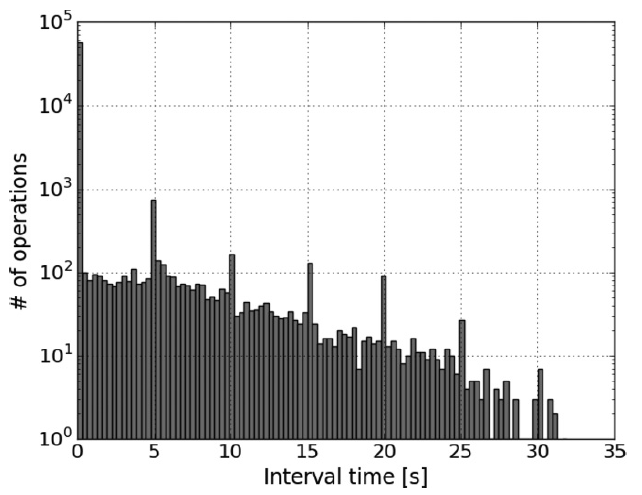


图2 请求间隔分布

（2）算法细节

此算法通过利用空闲时间隐藏GC延迟来减少长尾延迟，以及尽量减少GC诱导的阻塞。在本节中，我们介绍了一个RL辅助的GC调度器来隐藏GC延迟和一个积极的细粒度部分GC方案来减少阻塞时间。

提出的RL辅助GC调度程序是用惰性方式触发的。因此，只有当访问请求到达存储且空闲页面数量低于阈值时，才会触发它。当触发时，它选择一个动作。因为RLGC方法是基于局部的操作是执行一些部分GC操作，例如，从一个受害块复制5页到一个空闲块。因此，GC调度器选择一个动作，即确定在服务当前请求之后将执行多少个部分GC操作。当调度程序选择了一个操作并且块准备被擦除时，执行擦除操作。在这种情况下，块不是执行动作，而是被擦除。注意，第k个请求的响应时间给出了第k-1个请求的奖励。

完成请求后，GC调度器计算响应时间。因为我们的目标是减少长尾延迟，所以我们需要在奖励中反映响应时间。因此，在前面提到的公式（3）中，只有在完成到下一个请求并计算相应的奖励之后，我们更新当前状态s和动作a的Q值

在强化学习中，状态需要表示历史，这有助于最大化奖励。使用以前的请求间隔，现在的请求间隔和以前的动作作为状态。请求间间隔是一个重要的历史信息，因为它反映了存储流量的强度。因此，如果间隔很大，RL辅助的GC调度器倾向于采取更积极的行动，例如，更多的部分GC操作。前面的操作可以总结最近的历史记录，也可以作为GC调度程序的决策。从智能体的角度来看，主机和SSD子系统共同构成了环境，请求间隔表示主机的状态。前面的动作既可以表示SSD子系统的动作，也可以表示主机的动作。这是因为前面的操作不仅起到了总结最近历史的作用，也起到了GC调度程序决策的作用，但也会影响SSD子系统的状态，即页复制繁忙或空闲。例如，如果前面的操作是复制大量页面，那么SSD子系统的当前状态往往是繁忙的，将这三个成分分别划分为多个区间，2个区间表示以前的请求间隔，17个区间表示当前请求间隔，2个区间表示以前的动作，这样总共有68个(=2×17×2)状态。

对于奖励而言，需要以较小的响应时间分配较大的奖励，还需要惩罚响应时间过长的动作。图3显示了奖励函数，奖励范围在- 0.5到1之间。例如，如果响应时间很大(大于阈值t3)，就会分配一个负奖励来惩罚该行为。图3中的奖励函数的阈值需要根据存储访问的特征进行调整。一组固定的阈值不会覆盖存储访问中的不同场景。因此，此算法根据存储访问的特征设置阈值，因此设置了三个阈值t1、t2和t3，分别为70%，90%，99%处的响应时间。因此，即使存储访问行为发生变化，阈值也可以根据响应时间的新分布进行调整。

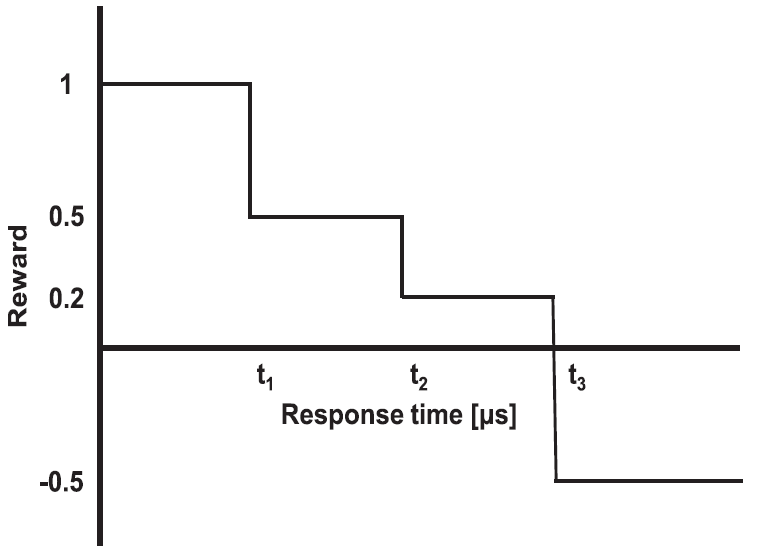


图3 奖励函数

图4显示了提出的RL辅助GC调度程序的伪代码。对于每个存储请求，GC调度器将空闲块的数量与阈值进行比较(在我们的实验中为10个块)。如果>=，调用函数来基于概率判断执行开发或探索。例如，在概率下随机选择一个动作或在概率下按Q表匹配的状态-动作对选择一个动作。在连续请求的情况下，如果请求间隔为零，不会触发GC调度程序。在完成请求并获得当前请求的响应时间之后，执行选定的操作，即部分GC操作后用当前请求的响应时间调用奖励函数。最后，更新前一个请求的Q表条目。正如前面提到的，我们更新了与前一个请求相关联的Q表条目。

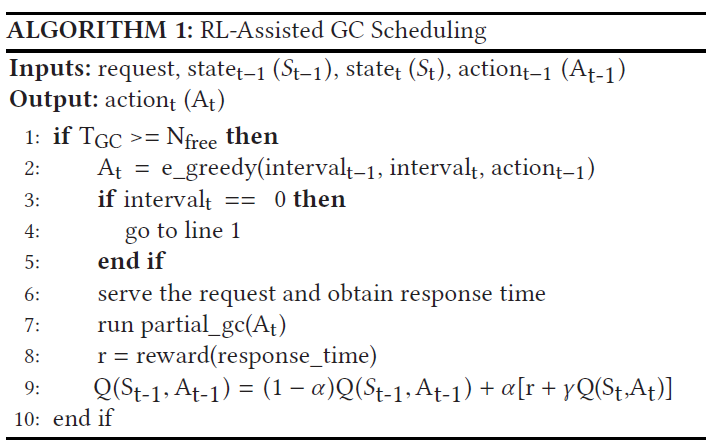


图4 RLGC调度算法

## 使用Q值缓存表动态保留关键状态（QTC）

（1）思路动机

为了减少GC引起的长尾延迟，已经研究了应用RL的技术-RLGC，在RL模型中，使用多个状态来表示环境。例如，工作负载特征和SSD内部信息。使用代表给定环境的适当状态，我们称之为关键状态，对于获得成功的解决方案至关重要。

在RLGC工作的基础上，进行了实验，发现长尾延迟随着状态数量的变化而变化。图5显示了在home1(我们在实验中使用的工作负载之一)中状态数量增加时的99.9999百分位的延迟。结果表明，增加状态数(在本例中通过使用更细的区间)倾向于减少延迟。另一方面，如图5所示，增加状态的数量并不总是能够持续地提高性能，因为大多数RL辅助应用程序的设计空间并不总是单调或线性的。这与作为状态使用的信息以及如何将它们划分为多个区间密切相关

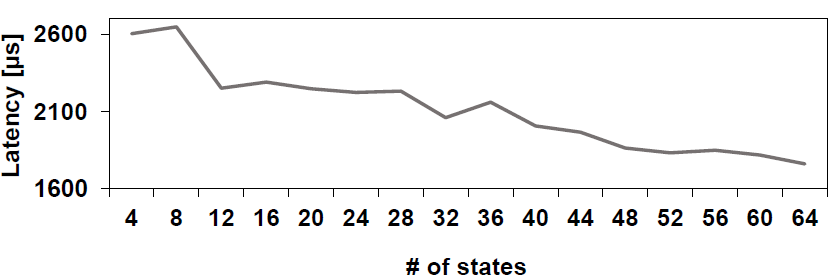


图5 hom1负载中延迟与状态数量的关系

此外，我们还研究了RL解的局部性行为。为此，我们在工作负载中间以相同的周期选择了四个周期，home2(每个周期10000个请求)。我们登记使用了哪些状态，并计算在此期间它们被访问的次数。图6显示了根据访问计数和home2中四个时间段的访问计数排序的前10个状态。图6中显示，每个时期使用的状态是不同的，访问计数也不同。例如，前两个状态占据了第2阶段的大部分访问次数。然而，其他状态的访问数量相对较低，说明排名前两的状态所对应的特征在这一时期占了很大比例。

另一方面，排名较高的状态在第4周期内的访问次数并不高，这表明其特征比其他时期更加多样化。我们继续对各种工作负载进行实验，观察到不同的工作负载具有不同的状态访问模式，但由于页面限制，这里没有显示这些数据。

通过以上两个观察，我们认识到使用更多的状态对于确保更好的性能是必要的，并且创建一个适应动态行为的通用解决方案是必要的。

RLGC使用Q-learning，这是RL方法的一种。在Q-learning中，所有状态对的奖励值Q-value存储在一个称为Q-table的表中。也就是说，随着状态数的增加，Q表的大小也随之增加。在SSD固件中，对代码大小有严格的限制。因此，理想的解决方案需要是动态的，以适应SSD系统不断变化的行为，并允许大量的状态，同时降低Q表的成本，即保持一个小的Q表。

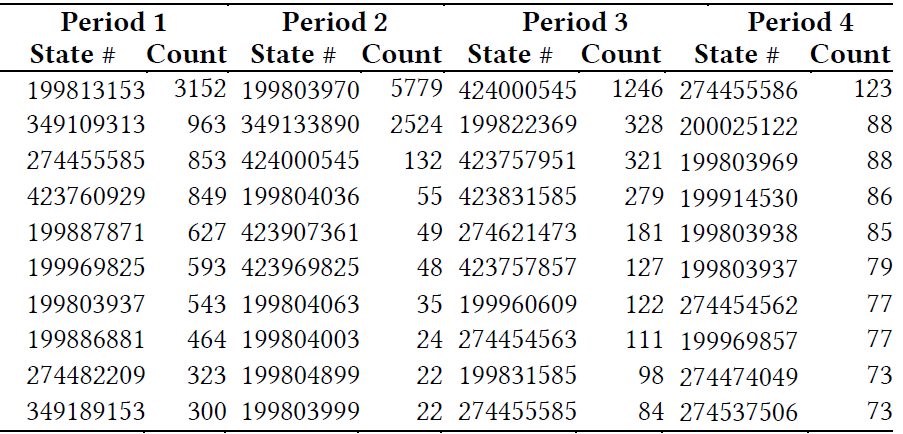


图6 在home2中排名最高的状态和访问次数

（2）算法细节

此算法基于RLGC。RLGC使用部分GC作为GC方法。为了利用请求间隔即空闲时间，RLGC使用强化学习。在RL实现中，重要的数据结构是存储Q值的Q表。Q表条目的数量为状态和动作。在嵌入式系统中采用RL辅助解决方案时，需要使用小的Q表。

智能体学习工作负载的行为，并确定GC触发的时间点，以及作为一个动作要执行多少个部分GC操作。RLGC使用从三种类型的预定义信息中获得的固定状态集，在设计时分成68个状态。由于固定的小状态集，RLGC在进一步减少长尾延迟方面有很大的局限性，这需要大量的状态和适应应用到存储的工作负载的动态行为。

此方法采用动态关键状态管理技术，克服了RLGC的局限性。我们使用大约个状态候选项和17条信息的组合，如图8所示。使用了一个小的Q表缓存（QTC），而不是以前的RLGC的Q表作为存储状态动作对的Q的方法，QTC以最近最少使用（LRU）的方式使用替换表中条目。根据部分GC中页复制的数量存在对应的三个动作，因此三个QTC缓存用于要执行的三个动作。

当存储系统工作时，状态是确定的，奖励是根据所采取的行动来计算的。状态和对应的Q值存放在Q表缓存中，如果确定的状态已经在Q表缓存中，则更新Q值。如果确定的状态在Q表缓存中不存在，那么Q值将作为一个新条目添加。此时，如果表缓存中没有更多的空闲条目空间，那么根据LRU策略将清除其中一个现有条目。通过这个过程，提出的RL辅助GC调度器使用小的Q表缓存，即小的内存资源，进一步减少了长尾延迟。

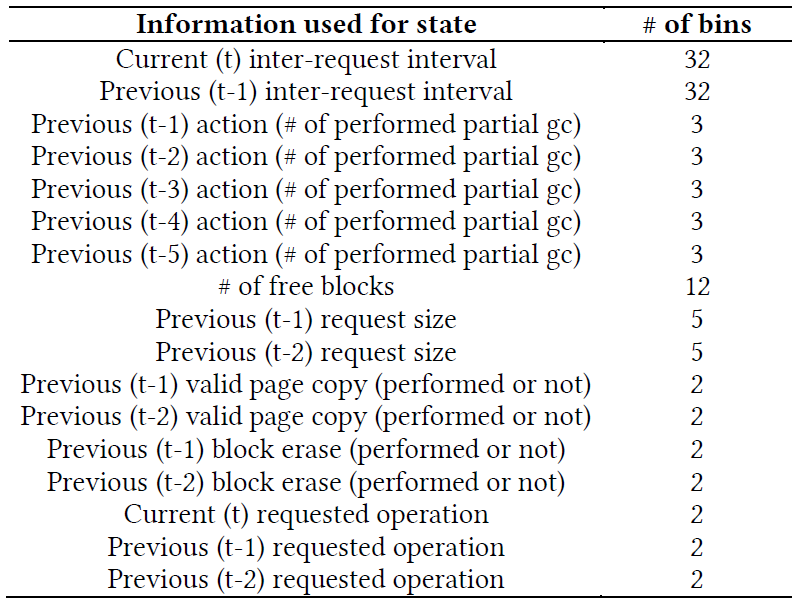


图7 状态信息和区间数

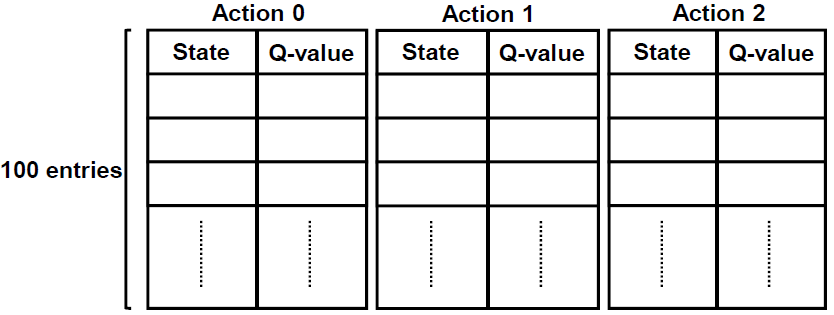


图8 Q表缓存结构

我们的GC调度程序选择具有最大Q值的动作。当确定一个动作的状态Q值对在Q表缓存中找不到时，与使用完整Q表进行Q学习的区别就出现了。当状态Q值对还没有添加到Q表缓存或状态Q值对已经被清除时，就会发生这种情况。如果GC调度程序无法在所有Q表缓存中找到给定状态的Q值，那么它将选择动作0页复制动作。这是一种保守的方法，以降低由于缺乏信息而增加的延迟的可能性。

## 使用Q值预测网络初始化Q表缓存

（1）思路动机

使用更多的状态信息具有详细表达环境的优势，并且可以进一步减少长尾延迟。QTC可以在小内存空间中使用许多候选状态。然而，在QTC中被替换的状态会丢失先前学到的Q值。因此，它依赖于QTC中新插入的状态Q值对的Q值零初始化。零初始化增加了新插入状态的学习时间，最终降低了对这些状态的动作选择的质量。

为了减轻零初始化的问题，可以增大QTC的大小，从而减少被替换状态的学习Q值的损失。然而,大QTC会导致极高的内存成本。例如，为了保持所有的状态（状态），Q表的大小可以达到98 GB，这是令人望而却步的昂贵的嵌入式系统。

特别是，一个大型的QTC往往包含大量的状态，而这些状态是学习不充分的，它们被称为不成熟的状态。这种不成熟的状态会产生不恰当的操作选择，从而阻止我们降低延迟。

（2）算法细节

图9展示了所提出的解决方案的整体架构，该解决方案集成了QTC和一个称为QP网络的小型神经网络。QTC学习短期行为，在给定状态下选择动作。相反，QP网络被训练来学习系统的长期行为。由于QP网具有Q函数的全局视角。当QTC中插入新的条目时，它可以提供一个良好的初始Q值。

QP网络与QTC的这种集成提供了一个基于RL的SSD解决方案的低成本高性能实现。从实验中可以看出，在学习Q函数的关键状态和Q函数的全局视角时，QTC和QP网络在学习的过程中代价都很小。最后，与使用一个小的Q表和仅使用QTC的情况相比，集成的解决方案进一步降低了长尾延迟。

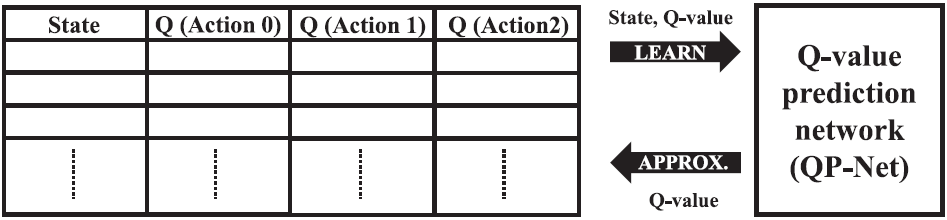


图9 QTC和QP网络

QP网络的结构如图10所示，QP网络是一个多层感知器(MLP)。我们选择两层MLP作为QP网络结构，是基于两层MLP可以逼近任意非线性函数的事实。如前所述，我们的想法是由AC模型中的评论家模型所激发的。我们还认为评论家模型是典型的MLP设计。为了确定两层MLP的具体配置，我们进行了敏感性分析，我们改变了每个隐层上神经元的数量，这将在我们的实验中给出。输入层接收17个输入，这17个输入所描述的候选状态中使用的17条信息的归一化值。输出层有3个输出，输出值为动作0、1和2页复制的预测Q值。

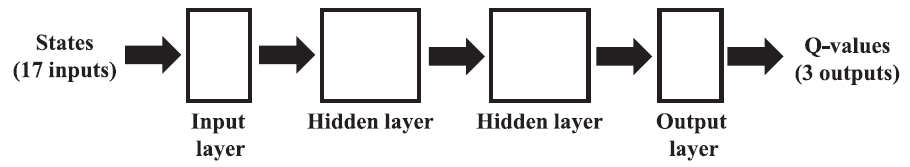


图10 QP 网络结构

每当QTC更新时，我们都会对QP网络进行培训。这是因为QP网络需要学习q函数的整个行为。QP网络类似于AC模型中的批评网络，即当出现新的奖励时，两者都会进行训练。对Q函数的整个行为进行训练的QP网络具有优势，特别是当未知状态出现时，需要预测未知状态的初始Q值，以及当先前排除的状态重新插入QTC时，如果QP网络只针对被驱逐的QTC条目进行训练，那么它就不能对未知状态做出有用的预测。QP网络是用所有这三种方法训练的QTC的Q值，但三个Q值中只有一个被更新。这样的培训使QP网络能够学习行动之间的相对重要性以及状态依赖性Q值。空间插入新项时QTC，我们利用QP Net预测的所有三个Q值来初始化新插入状态的三个Q值。

需要最小化QP网络大小，以减少由于QP运行而带来的额外运行时和内存成本。我们进行了敏感性分析，通过改变两个隐藏层的神经元数量来确定合适的QP网络大小。图11显示了3D 512 Gb和3D 128GB中，闪存不同大小的QP网络的结果(99.9999百分位的总工作负载的平均延迟)。3D 512 Gb闪存具有最低的延迟时，每隐藏层有50个神经元；而3D 128Gb闪存具有最低的延迟时，每隐藏层100个神经元。

容量较小的闪存需要更大的QP网络，可以分析如下。在3D 128Gb闪存中，由于容量较小，与3D 512Gb闪存相比，执行了更多的GC，创建了1.38倍多的状态。因此，为了考虑更多的状态，就需要一个更大的QP网络。

尽管QP网络的大小是在设计时确定的(每个隐层有50/100个神经元，512个Gb / 128 Gb闪存)。当闪存的类型确定时，我们不排除在运行时动态地调整QP净的可能性大小取决于可用的资源，例如，存储容量变化由于老化，和应用程序的行为，例如大量的活动状态导致低QTC命中。在这种情况下，研究进一步减少延迟的可能性将是未来的工作。

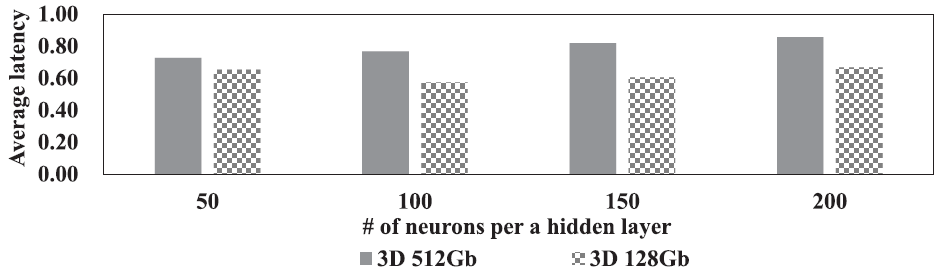


图11 延迟和QP网络大小的关系

QP网络和QTC整体的运行结构如图12所示，首先，SSD接收来自主机的请求并开始提供服务。智能体在服务请求时，如向闪存写入数据时，更新QTC(更新前一个动作对应的条目或插入一个新的state-Q值对)和QP网，同时利用奖励和新状态的信息。然后，智能体根据新状态选择一个动作。在完成当前请求的服务之后，智能体执行所选择的操作，例如，1页复制。然后重复上述步骤。提出的GC解决方案仅在SSD收到写请求时触发。

关于空闲时间有两个方面。首先，我们的解决方案旨在通过在空闲时执行动作来利用空闲时间这是因为空闲时间可能在完成请求之后到来。其次，我们的解决方案并没有充分利用空闲时间，特别是非常长的空闲时间，因为只有当请求到达SSD时才会触发部分GC操作。开发长时间空闲时间将是今后工作中一个很有前途的课题

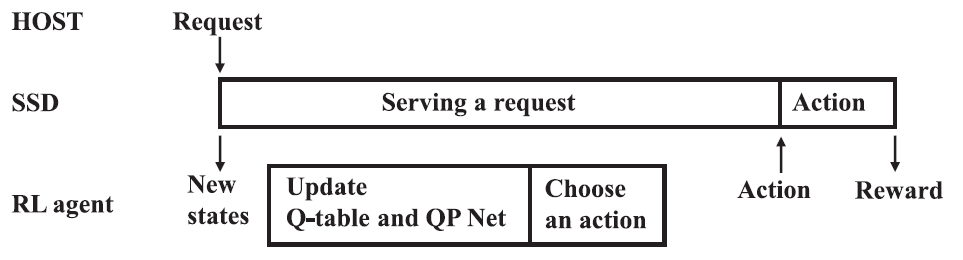
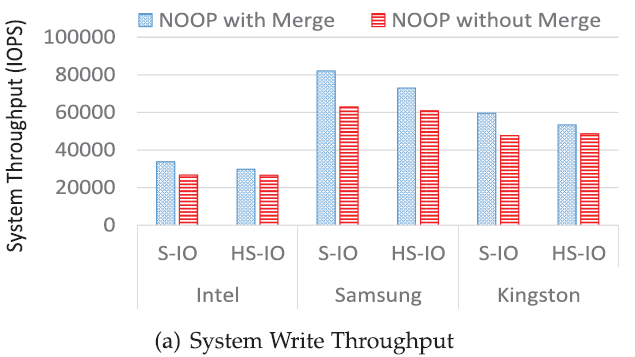


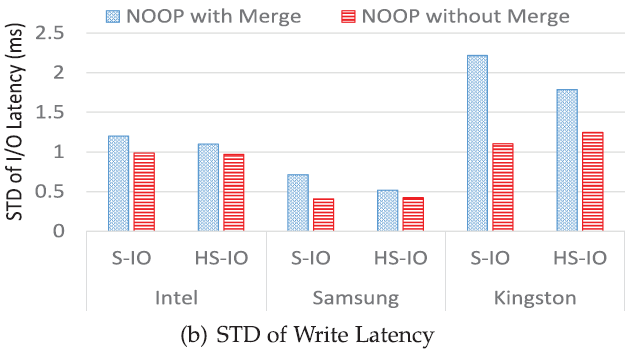
图12 操作概述

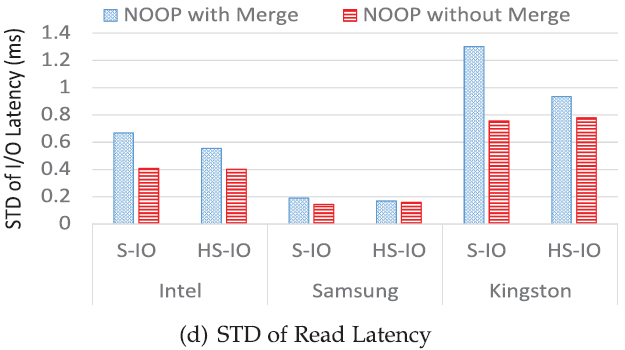
## 基于强化学习的动态I/O合并

（1）思路动机

为了研究I/O合并对系统吞吐量和I/O性能变化的影响，实验在来自不同制造商(Intel，三星和金斯顿)的SSD上进行。我们使用FIO来生成具有不同模式的I/O工作负载(全部是顺序的或50%顺序和50%随机的混合4 KB I/O)和blktrace来捕获I/O系统的性能。为了使用多线程模拟真实的工作负载环境，线程号被设置为64。每个实验都进行了10次，取平均值作为结果，采用NOOP作为默认调度方案。图13显示了有和没有I/O合并的系统之间的吞吐量和I/O性能变化比较。







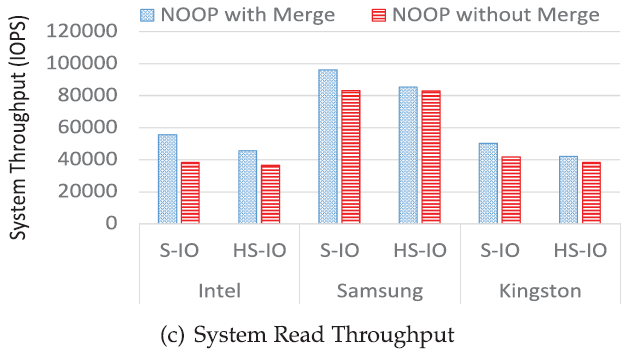


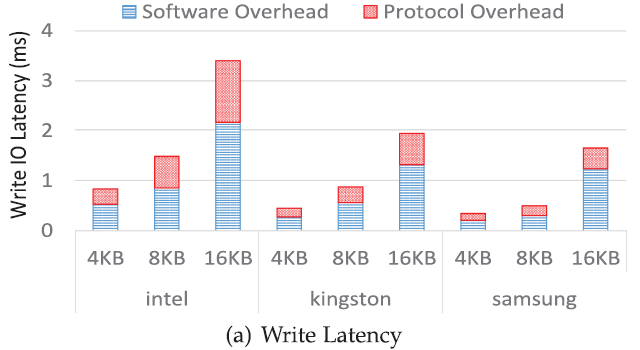
图13 I/O合并前后的系统性能变化

如图13所示，I/O合并大大提高了系统吞吐量。其中，S-IO表示所有I/O请求是顺序的，HS-IO表示一半I/O请求是顺序的，其他的是随机的。对于顺序写请求，NOOP调度器的系统吞吐量I/O合并平均比没有I/O合并的系统获得21.4%的收益。对于顺序读请求，I/O合并的好处平均为20.5%。混合I/O负载的系统吞吐量的提升有限，其中写请求平均提升了12%，读请求平均提升了16%。原因在于由于顺序I / O请求更少而减少了I/O合并操作。

I/O合并对系统吞吐量的好处有两个方面。首先，I/O合并可以提高设备内部并行性的利用率。放大I/O请求后的尺寸稍后可以在设备级别拆分为更多的子请求，并分配给更多的空闲通道和芯片。其次，由于所有合并的请求共享相同的完成时间，I/O完成开销被这些请求平摊。

I/O延迟主要包括软件开销、设备访问开销和I/O完成开销。软件开销表示处理I/O调度时操作系统花费在I/O栈上的时间。I/O合并对系统吞吐量的好处主要来自于增强了设备并行性和摊销I/O完成延迟的利用。为了验证I/O合并带来好处的原因，在一组真实的SSD设备上进行了实验。

实验结果如图14所示，设备访问和I/O完成开销占所有I/O请求的很大一部分。对于读请求，这两个开销占I/O延迟的34.3%，而写请求的值为36.5%。因为I/O合并的优势主要来自设备内部并行性的增强利用和摊销I/O完成开销，这些评估结果揭示了I/O合并影响系统性能的原因。



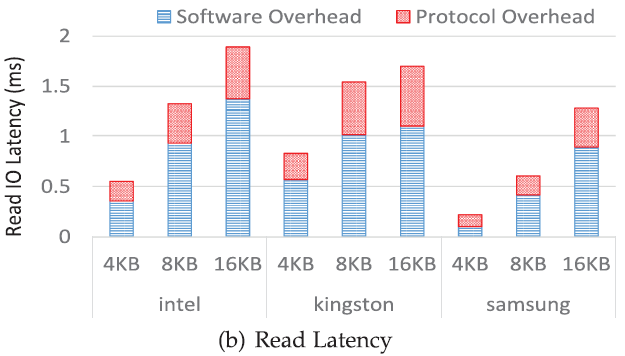


图14 I/O延迟在不同SSD上的不同大小请求的细分

如图13所示，虽然提高了系统吞吐量，I/O合并也扩大了STD(I/O延时的标准偏差) 。但是对于顺序I/O请求，写请求的I/O性能变化平均被放大36.9%，读请求的性能变化平均放大了35.0%。较大的性能变化表示I/O延迟显著偏离平均延迟，这意味着多个请求可能会经历较长的最坏情况延迟。

为了揭示I/O合并对最坏情况延迟的影响，实验详细阐述了从SSD设备捕获的1%尾部请求延迟的累积分布。图15展示了分布情况。对于写请求，带有I/O合并的系统在99.9%上导致了平均2.09毫秒的I/O延迟，没有I/O合并的系统相比延长了19.4%。对于读请求，99.9%的平均延迟为2.25毫秒，延长了22.9%。如前所述，较高的最坏情况延迟会使系统缓慢且影响用户体验，这就需要优化传统的I/O合并技术。

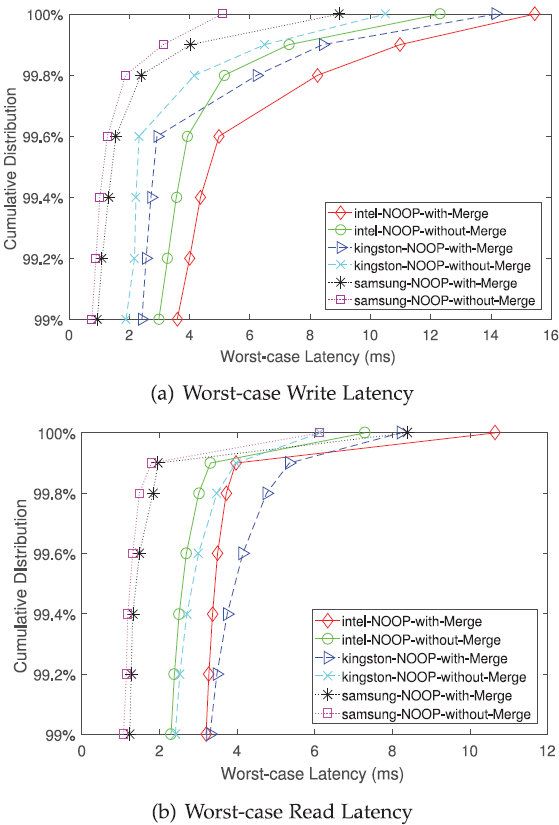


图15 SSD的I/O长尾延迟分布

根据结果，我们可以得出结论，采用I/O合并技术时需要在系统吞吐量和QoS的要求之间做出更好的权衡。这里主要观察的是，现有的I/O合并方法不能同时获得高I/O吞吐量和小性能变化，不能满足SSDs中QoS性能的要求，为了追求理想的QoS性能，我们应该重新审视SSD的现有合并方法。

影响I/O合并效果的主要因素有两个，要合并的I/O请求的大小和数量。为了更好地理解各种合并方法的影响，下面给出了一个简单的示例，队列中有固定的请求数量。

假设队列中有4个读请求，并发给4个闪存芯片，由图14b评价结果可知，假设4KB请求设备访问的开销为80 us，I/O完成开销为5 us。软件开销设置为4 us。此外，由于I/O延迟的标准差的计算需要最近一秒内I/O延迟的平均值，平均延迟设置为250 us。然后通过请求数量计算系统吞吐量(等于4)除以SSD服务所有四个请求所需的时间。I/O延迟的标准差由公式（4）定义

|  |  |  |
| --- | --- | --- |
|  |  | (4) |

其中为标准差，表示请求的数量，这里被设为4，表示第个请求的延迟，表示过去一秒内所有请求的平均延迟。另外，假设每个芯片（chip）有2个模具（die），每个模具有2个平面（plane）。因此，可以在一个芯片中同时处理四个4 KB的子请求。在本例中，大小小于或等于16 KB的请求的访问开销为80 us，而512 KB请求的访问开销为2560 us。

对于等大小模式，当没有合并请求时，I/O调度程序依次向设备驱动层分派I/O请求。由于软件开销，所有芯片在不同的时间被激活以适应不同的请求。在这种情况下，所有芯片活动的时间减少了，这意味着设备并行性不能被充分利用，从而降低了系统性能。此外，由于对所有请求的设备访问都是在很短的时间内完成的，系统必须经历四次I/O完成开销，这进一步降低了系统性能。这种退化可以通过将所有请求合并为一个请求来解决。系统管理发出所有请求相应的芯片同时和共享一个I/O完成开销在这些请求。如图16所示，与没有合并策略相比，所有合并策略将提高系统吞吐量4倍以上。虽然STD增加了3.7 us，对于通常在几百微秒内完成的I/O请求，对系统性能的影响是微不足道的。换句话说，应该合并此模式的所有请求。

对于大小不同的模式，I/O合并会显著增加I/O性能变化。如图16所示，与没有合并策略相比，所有合并策略的STD提高了334.99 us，系统吞吐量提高了2倍以上。将小请求合并为大请求会降低小请求的延迟，从而导致尾延迟过长，违反QoS要求。这个问题可以通过在I/O合并期间使用小请求先服务(SRFS)的部分合并策略来解决。与所有合并策略相比，使用SRFS的部分合并可以减少I/O性能变化高达335.54 us，同时提供较高的系统吞吐量(比所有合并策略低3.34%)。

由于不同模式合并需求的特殊性，需要提出一种能够学习不同模式特征并根据学习结果动态合并请求的模型。这一分析促使我们提出了基于强化学习的合并方法，该方法具有自适应决策能力和计算简单性。

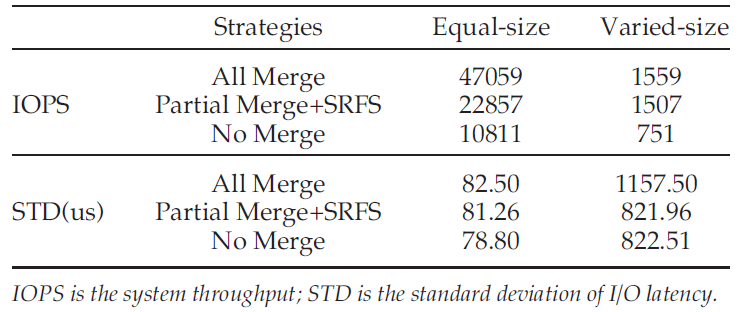


图16 不同的合并策略对不同I/O模式的影响

（2）算法细节

强化学习中状态的定义应该代表历史，并为智能体提供足够的信息来学习和做出相应的决策。本工作采用RL的目的是通过适当的I/O合并决策来抑制性能变化，维持较高的系统吞吐量，以适应不同的工作负载模式。因此，状态的定义应考虑I/O模式状态的特征，包括请求号、大小和读/写操作。

首先，读和写请求不能合并在一起。如果下一个合并请求是写请求，系统检查I/O调度器队列中的所有写请求，以决定合并请求的数量和大小。设分别为读写请求的状态数，因此，所有状态的数量。

其次，状态应该记录每个队列请求的大小。但是，假设最大的请求大小为2 M，那么就有512种可能的I/O大小(2 MB/4 KB = 512)。对于深度为64的队列，读或写请求的状态号是，这使得它无法维护。相反，我们仔细分析实际I/O工作负载的所有请求的大小分布。对于请求中经常出现的大小，我们将每个大小定义为一个大小区域。对于其他很少出现的尺寸，我们将它们合并为一个区域

最后，对于队列中的请求号，我们关心每个区域中的相对请求数量，而不是绝对请求数量。例如，如果队列中有更多的4 KB请求和更少的64 KB请求，我们希望通过找到正确的计数和大小阈值来做出适当的合并决策。

假设大小区域的数量定义为，设0到表示每个大小区域的相对I/O数量。0表示该区域中没有请求，1表示该区域的请求数量最少，2表示该区域中的请求号是所有区域中第二少的，表示这个区域有最大数量的I/O请求，表示所有请求都在这个区域。公式（5）表示状态数与大小区域之间的关系

|  |  |  |
| --- | --- | --- |
|  |  | (5) |

其中，表示状态的数量，表示大小区间的数量，乘数2表示读和写请求的状态总数，表示所有请求的大小都位于一个区域时的情况，表示所有请求的大小都位于区域的情况，表示从中取出个区域的可能性，1表示所有区域都有相同数量的I/O请求，表示在区域内的I/O数量相同，表示在区域内所有请求的相关分布。

动作在RL模型中起着重要的作用，它使状态向目标移动。同时，动作定义需要覆盖状态所表达的信息，这样智能体才能学习不同状态之间所有信息的区别。由于本工作中的状态涉及到请求大小、计数和读/写，操作被定义为分别针对读和写请求的大小阈值和计数阈值的组合。设计了16个计数阈值和8个大小阈值作为动作用于选择，因此对于每个状态有个动作。有了大小和计数阈值的约束，一个悬而未决的问题是应该合并哪个请求。对于SSD，有各种I/O排序算法。SSD的两种常见解决方案是先来先服务(FCFS)和小请求先服务(SRFS)。为了进一步提高系统性能，我们在模型中采用SRFS策略来合并具有更高优先级的小请求。为了避免请求饥饿，根据最后期限I/O调度程序的最后期限配置分配请求等待时间的最后期限。

在强化学习中，智能体通过奖励来指示相应动作的正确性。换句话说，奖励的定义应该代表框架的目标。因此，代理可以识别当前状态与目标之间的偏差。在这项工作中，我们的目标是最小化I/O性能变化，并通过决定合并请求的数量和大小提供最大的系统吞吐量。因此，这项工作的目标可以描述为通过选择和来最大化吞吐量和最小化STD，定义如公式（6）所示

|  |  |  |
| --- | --- | --- |
|  |  | (6) |

其中，表示奖励，第个合并请求的大小，表示合并请求的数量，表示系统的吞吐量，表示性能变化。

为了在开发和探索之间取得平衡，采用策略。在学习的初始阶段，采用一个激进的值(90%)来执行更多的勘探，这意味着随机选择动作的可能性被设置为90%，以便模型能够经历尽可能多的状态动作对。在那之后，被设置为10%，以更好地利用学到的结果，但智能体仍然可以执行探索以避免局部最优。如果开发被选择为任何没有学习过的状态-动作，勘探将被执行代替。

图17描述了提议的I/O合并框架的架构。①在I/O调度程序中，当新请求被插入到队列中时，提议的方案被触发。②队列中所有被智能体感知的信息，包括请求大小、读/写属性和请求数量，以标识当前状态。③④批次管理器（BM）采用策略执行探索和开发，BM在开发过程中，通过查找Q表来选择一个Q值最大的动作，该动作属于当前状态，例如，合并队列中的所有读/写请求。为了探索，BM会随机选择一个动作来学习更多未知动作。⑤I/O调度器根据BM确定的动作合并I/O请求，然后分送下一个排序的请求到块设备层。⑦⑧⑨合并的请求完成服务，智能体获取请求的完成时间并减去插入时间以计算延迟时间。BM维护了一个缓冲区列表，用于存储过去一秒内所有请求的插入时间和延迟时间。⑩在Q表对应的条目中推导和更新上一个状态动作对的奖励。

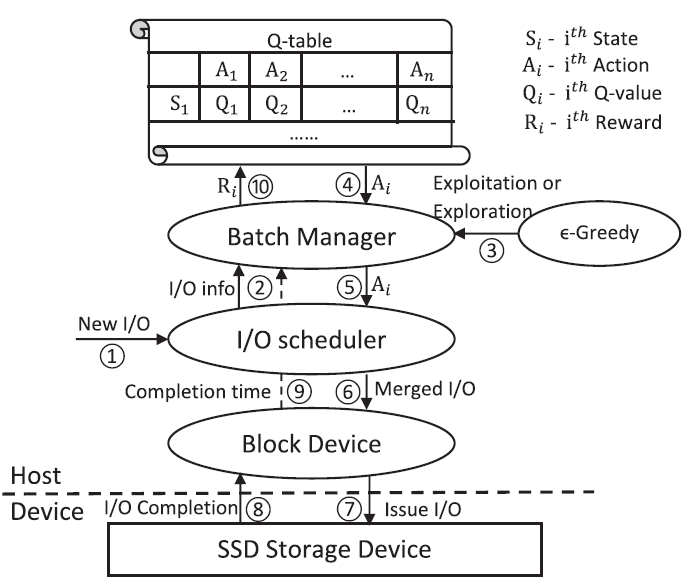


图17 基于强化学习的I/O合并方法

# 论文总结

本文从垃圾回收和I/O合并两个方向研究了基于强化学习的SSD长尾延迟问题的优化方法。垃圾回收的页面复制和块擦除是非常耗时，因为会增加延迟同时会堵塞后续的请求。因此本文调研的方法通过强化学习感知I/O请求行为，确定请求间隔时间来选择合适的垃圾回收动作，从而降低长尾延迟，随后基于强化学习中状态量的数量与尾延迟时间成反比这一发现提出Q表缓存来动态保存尽可能多的状态量。同时提出Q值预测网络来初始化Q表缓存来解决零初始化导致的学习效率低的问题，从而进一步提升长尾延迟优化效果。针对I/O合并具有不同I/O模式的访问请求时产生的长尾延迟问题，调研方法提出一种能够学习不同模式特征并根据学习结果动态合并请求的模型，在保持I/O合并对系统吞吐量的好处同时又缓解了性能变化导致的长尾延迟问题。

通过调研发现，强化学习作为一种机器学习方法，能够解决原有静态处理方法，即在算法设计时已确定调度方式，对具有不同访问特征的请求数据适应性较差的问题，它使SSD能够在线学习请求的访问特征后，能够根据学习结果动态调整自身的测率并选择合适的动作来处理请求，从而提升长尾延迟优化效果。

参 考 文 献

[1] Kang W, Shin D, Yoo S. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD. ACM Transactions on Embedded Computing Systems, 2017, 16(5s): 1-20.

[2] Kang W, Yoo S. Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in SSD. Proceedings of the 55th Annual Design Automation Conference. San Francisco, USA, 2018: 1-6.

[3] Kang W, Yoo S. Q-Value Prediction for Reinforcement Learning Assisted Garbage Collection to Reduce Long Tail Latency in SSD. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019, 39(10): 2240-2253

[4] Wu C, Ji C, Li Q, et al. Maximizing I/O Throughput and Minimizing Performance Variation via Reinforcement Learning Based I/O Merging for SSDs. IEEE Transactions on Computers, 2019, 69(1): 72-86.