# ZNS SSD感知的数据存储管理机制研究

摘 要：ZNS SSD（Zone Namespaces SSD）是最近出现的一种新型SSD。它允许将SSD整个空间划分为多个Zone，并且每个Zone内只允许顺序写入。ZNS SSD可以有效提升SSD的读写吞吐，降低数据存储时的写放大。但是，ZNS SSD的顺序写和Zone划分等限制也对现有的存储分配和垃圾回收策略提出了挑战。其中两个主要的问题是：（1）数据写入时如何结合ZNS的特性，即要求具备ZNS友好性；（2）垃圾回收时如何降低代价以避免影响系统性能的稳定性。针对这两个问题，本文提出了一种ZNS SSD感知的数据存储管理机制。该机制的创新思路主要包括两个方面：（1）针对数据写入时要求具备ZNS友好性的问题，我们将数据从插入到垃圾回收的过程视作一个生命周期，提出了根据数据的访问模式和工作负载特征来计算数据生命周期的方法，进而提出了基于生命周期的数据插入算法；（2）针对Zone垃圾回收问题，由于数据的生命周期在不同的工作负载环境下具有显著差别，我们提出了一种数据生命周期方差感知的Zone回收算法，以降低Zone回收时的数据迁移代价。由于目前商用的ZNS SSD还未上市，我们利用ZNS SSD模拟器libzbd开展了实验验证。结果表明，我们提出的基于生命周期的数据插入算法与两种基线方法（平均插入和轮询插入）相比具有更稳定的插入性能，所提出的生命周期方差感知的垃圾回收算法比常规垃圾回收算法在时间性能上提升了9%，并且性能稳定性更高。

关键词：ZNS SSD；数据存储；数据插入；垃圾回收；数据生命周期

#### Research on ZNS SSD-Aware Data Storage Management Mechanisms

**Abstract**: ZNS SSDs (Zone Namespaced SSDs) are a new type of SSDs. It allows the entire SSD space to be divided into multiple Zones, and only sequential writes are allowed within each Zone. ZNS SSDs effectively improve read/write throughput of SSDS and reduce write magnification. However, the sequential write and Zone partitioning of ZNS SSDs pose challenges to existing storage allocation strategies. How to adapt the existing storage management mechanism to ZNS SSD is the main problem to be solved in this topic. Due to the timeliness of the data, it is invalid at the end of its life cycle, which can change under different workloads, and existing data placement strategies for **conventional** SSDS are mostly segment-based and cannot be applied to the sequential write within Zones feature of ZNS SSDS. To solve this problem, according to the data access mode and workload characteristics, a ZNS SSD aware data placement strategy is designed, and compared with the traditional data placement strategy, to demonstrate the advantages of this method. Specifically, the inserted and modified data is placed according to the life cycle of the data, and the variance of the data life cycle in each Zone is used for management and garbage collection based on the calculation of the conventional garbage collection strategy. Experiments show that the life-cycle based insertion algorithm has a great improvement in stability compared with average insertion and polling insertion algorithm, and the time performance is slightly reduced due to the calculation overhead of life cycle. The life cycle variance aware garbage collection algorithm is 9% better than the conventional garbage collection algorithm in terms of time performance and is more stable.

**Key words**: ZNS SSD; Data placement; Garbage collection; New Storage; Data lifetime

**1 引 言**

近年来，ZNS SSD(Zone Namespaced SSD)作为一种新型的SSD(Solid State Drive)接口，受到了工业界和学术界的普遍关注[1][2]。ZNS SSD是基于用于SMR (Shingled Magnetic Recording)HDD的ZAC/ZBC标准和Open-Channel SSD标准而提出的一种新型SSD。如图1所示，ZNS SSD将磁盘基于空间分布抽象成不同的Zone，Zone内部只能顺序写[3]。主机根据抽象后的接口来管理磁盘I/O。这样划分可以让几乎同时失效的数据存放在一起，减少写放大的次数，提高SSD的性能和寿命。相较于传统SSD在设备上使用FTL (Flash Translation Layer)处理存储单元，需要写前擦除等操作，ZNS SSD减少了DRAM的使用和SSD的超额配置，提高了SSD的利用率[4]。

图示

描述已自动生成

Fig.1 Internal data placement of conventional SSD versus ZNS SSD

图1 传统 SSD 与 ZNS SSD 的内部数据放置[3]

但是，ZNS SSD的顺序写和Zone划分等限制也对现有的存储分配和垃圾回收策略提出了挑战。在存储分配方面，ZNS SSD要求数据插入时必须考虑Zone内部只能顺序写的特性，即要求具备ZNS友好性。由于Zone不能随机写，因此当Zone内的数据发生更新时只能采取异地更新机制，这要求我们定期地执行Zone垃圾回收操作。垃圾回收操作要求重置（Reset）整个Zone，如果Zone内存在着较多的有效数据，则将引入较高的数据迁移代价。因此，如何降低Zone垃圾回收的代价从而避免垃圾回收对系统性能稳定性的影响，是ZNS SSD上的数据管理必须考虑的另一个关键问题。

本论文针对ZNS SSD在数据存储分配和垃圾回收方面存在的问题，提出了一种ZNS SSD感知的数据存储管理机制，并通过实验证明了该机制的有效性。总体而言，本论文的主要工作和贡献包括：

（1）针对ZNS SSD数据写入时要求具备ZNS友好性的问题，我们将数据从插入到垃圾回收的过程视作一个生命周期，提出了根据数据的访问模式和工作负载特征来计算数据生命周期的方法，进而提出了一种基于生命周期的数据插入算法；

（2）为了降低Zone垃圾回收的代价，基于数据的生命周期在不同的工作负载环境下具有显著差别这一观察，本文提出了一种数据生命周期方差感知的Zone回收算法，以降低Zone回收时的数据迁移代价。

（3）我们利用ZNS SSD模拟器libzbd开展了实验验证（目前商用的ZNS SSD还未上市）。结果表明，我们提出的基于生命周期的数据插入算法与两种基线方法（平均插入和轮询插入）相比具有更稳定的插入性能，所提出的生命周期方差感知的垃圾回收算法比常规垃圾回收算法在时间性能上提升了9%，并且性能稳定性更高。

本文后续内容安排如下：第2节讨论了相关工作，第3节介绍了ZNS SSD感知的数据存储管理机制的设计目标以及总体的算法思路，第4节介绍了具体的算法实现，第5节给出了对比实验结果，最后第6节总结了全文并展望了未来的研究方向。

2 相关工作

传统的SSD使用FTL管理，这些信息对于主机来说都是透明的，而ZNS SSD利用主机管理SSD，便可以利用这些信息进行更加智能的管理，比如ZNS SSD可以选择数据放置的位置，可以进行垃圾回收与损耗均衡[4]。图2显示了基于FTL的传统SSD与ZNS SSD在设备层和主机层设计的主要区别。

传统SSD中的数据放置策略[5]，如FTL(Flash Translation Layer)在基于页的存储中使用的DFTL策略，或基于日志的存储中使用的SAST(Set-Associative Sector Translation)策略，都基于段(2MB或4MB)而非Zone(0.5GB或1GB)来进行数据放置与垃圾回收，若将这些策略直接移植至ZNS SSD上则会导致更长的延迟，并且无法体现ZNS SSD分区管理的特性。

数据生命周期指数据被存入磁盘到变为无效的时间。有很多现有算法都将数据的生命周期息运用于SSD的数据管理，以加快数据的增删改查操作。但由于无法准确获得数据的生命周期，所以只能根据对该数据的历史操作来预测数据的生命周期。在针对传统SSD的数据放置算法中，如Wear-Leveling算法，通常使用数据写次数、读取频率和回收频率来定义数据的生命周期[6]。现有的在ZNS SSD上的数据放置策略，大都结合具体的存储结构来界定数据的生命周期，并不能及时体现数据在变化的工作负载下的生命周期[7][8]。

图示

描述已自动生成

Fig.2 Comparing conventional SSDS with ZNS SSDS

图2 传统 SSD 与 ZNS SSD 的对比[4]

由于在Zone内只能顺序写，因此很多基于传统 SSD抽象接口开发的应用和系统不能直接应用在 ZNS SSD上。此外，由于 ZNS SSD移除了设备内的垃圾回收，因此需要用户自行设计垃圾回收机制，带来了额外的复杂性。Bjørling等基于ZNS SSD开发出了 ZenFS文件系统来适配 RocksDB的文件接口，目前已经成功提交到 RocksDB的社区[9]。此外，他们也对 F2FS（Flash-Friendly File System）[10]进行了改造以适配 ZNS SSD。Han等人在 ZNS接口的基础上更进一步，提出了ZNS+接口[11]。ZNS+是一种对 LFS感知的 ZNS接口，将用户定义的垃圾回收产生的有效数据拷贝放到设备内部进行，从而减少 I/O操作，提升文件系统垃圾回收的效率。Choi等对于设计使用ZNS SSD时所需的垃圾回收提出了新的方法[7]。他们建议针对 ZNS SSD进行细粒度的垃圾回收，并根据数据热度将细粒度数据存储在不同的分区内。

**3 ZNS SSD感知的数据存储管理**

本节主要讨论ZNS SSD感知的数据存储管理机制的设计目标以及设计细节。

**3.1 设计目标**

由于ZNS SSD的垃圾回收代价较大，且会影响正常的读写操作，不同生命周期数据的放置会影响垃圾回收的进程。同时随着工作负载的变动，数据的生命周期也会发生改变，需要及时对数据的防止做出调整。因此设计目标如下：

（1）刻画数据的生命周期。根据数据的历史信息，定义数据的生命周期，并根据生命周期存放在不同的Zone中。

（2）降低垃圾回收代价。希望能减少单位时间内垃圾回收次数，降低单次垃圾回收代价。

（3）优化数据存放。根据变化的数据生命周期调整Zone中数据的存放，使得ZNS SSD在整个工作负载下都能够稳定的执行数据操作。

围绕上述目标，ZNS SSD感知的数据存储管理机制重点解决两个关键问题：ZNS SSD感知的数据插入问题以及ZNS SSD感知的垃圾回收问题。本文提出了基于数据生命周期的插入策略和垃圾回收策略。具体而言，当包含一组键值的数据页插入Zone时，根据页中键的历史生命周期以及各个Zone中键的生命周期，算法选择最接近的Zone插入；当工作负载发生变化，Zone中键的生命周期的方差变大时，考虑将该Zone加入垃圾回收队列，回收时将Zone内有效块重新插入最优的Zone中。下面详细讨论相关的算法设计。

**3.2 基于生命周期的数据插入**

ZNS SSD感知的数据插入算法需要利用数据的生命周期这一信息，将需要插入的数据分配到最佳的Zone中，且需要根据变化的工作负载，通过调用垃圾回收来优化数据放置。

当有数据要插入SSD时，算法需要选择合适的Zone将其写入。而依据的信息便是键的生命周期与Zone中已有数据的生命周期。

**定义 3.1 键生命周期**：对于一条键值记录，其对应键的生命周期为距离对该键上一次写的对其他键的更新次数。 ■

因此，当对某一键进行更新或删除时，其他键的生命周期值加1，而该键对应的生命周期值变为0。生命周期值小的键，意味着在此工作负载下，其值的有效期短。数据的生命周期不仅可以有效地表征某一数据的生命周期，也无需更多的内存开销来保存多余的数据历史信息，还可以根据变化的工作负载更新数据生命周期的分类。为了保证记录的生命周期的时效性，我们设置了键的最大生命周期。新插入的键值数据由于先前没有记录，其生命周期也设置为最大生命周期。

定义了数据的生命周期后，我们可以在此基础上继续定义页的生命周期和Zone的生命周期。

**定义 3.2 页生命周期**：一个页的生命周期定义为该页中生命周期最小的键的生命周期。 ■

这是因为当该页中生命周期最短的键发生改变时，需要将该页全部读出，该页所在块也被标记为无效。

**定义 3.3 Zone生命周期**：一个Zone的生命周期定义为该Zone中所有页的生命周期的平均值。 ■

如图3所示，当页插入时，选择生命周期与该页的生命周期相差最小的Zone进行插入，这样能在总体上保证相近生命周期的数据被存放在了同一个Zone中，这些数据会在未来相近的时间变为无效数据，提升单次垃圾回收所回收的无效块的比例，降低单位时间的垃圾回收次数。

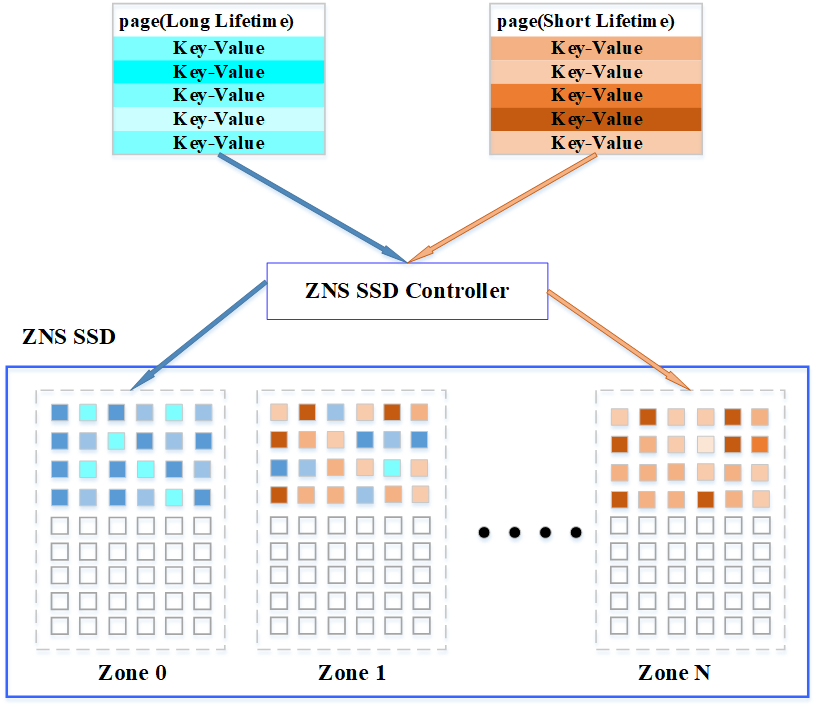


Fig.3 Illustration of the data insertion based on lifetime

图3 基于生命周期的数据插入示例

**3.2.1 初始化与生命周期更新**

初始化过程（见算法1）将所有的Zone都重置并恢复到初始状态，将Zone的生命周期记录和垃圾回收队列清空，各个Zone的空闲率、垃圾率等数据也设为初始状态。

|  |  |
| --- | --- |
| 算法1 初始化 Initialize() | |
| Input: 无  Output: 无 | |
| 1 | reset zones |
| 2 | clear zoneLifetime and gcQueue |
| 3 | zoneEmptyTable ← 1 |
| 4 | zoneGarbageRateTable ← 0 |

在对Zone进行插入或修改后，需要对键和Zone的生命周期更新。算法2给出了生命周期更新的过程。若有键被修改，则将Zone中的其他键生命周期加 1，除非它已经达到最大生命周期。将Zone的生命周期更新为各页的生命周期的平均值，插入页所在的块的生命周期更新为页的生命周期，所在Zone的生命周期为有效块的平均值。

|  |  |
| --- | --- |
| 算法2 生命周期更新  refreshLifetime(page, pageLifetime, blockAddr, zoneId, OP) | |
| Input: 插入数据页page，插入数据页的生命周期 pageLifetime, 插入块的地址blockAddr, 插入zone编号 zoneId, 导致插入操作的指令OP  Output: 无 | |
| 1 | if OP = MODIFY then |
| 2 | for all key do |
| 3 | if keyLifetime[key] != MAX\_LIFETIME then |
| 4  5  6  7  8  9 | keyLifetime[key]++  end  end  end  blockLifetime[blckAddr] ← pageLifetime  zoneLifetime[zoneId] ← avg(blockLifetime in zoneId) |

**3.2.2 数据插入与修改**

在有数据插入时，首先根据插入的页中的键的生命周期中生命周期最短的键来作为插入页的生命周期。如果此时ZNS SSD 中Zone的生命周期记录较少，无有效信息来判断页的存放，则根据Zone的空闲率，平均存放；否则，选取与插入页生命周期最接近的一个Zone进行存放，并返回存放的块地址。存放后将该Zone和存入键的生命周期进行更新，最后调用垃圾回收检测算法。算法3给出了数据插入的过程。

|  |  |
| --- | --- |
| 算法3 数据插入 Insert(page, keySet, OP) | |
| Input: 插入数据页page, 插入的键集合keySet，导致插入操作的指令OP  Output: 返回页插入的block地址 | |
| 1 | pageLifetime ← max(keyLifetime[i in keySet]) |
| 2 | if avg(zoneEmptyTable) < 0.2 then |
| 3 | zoneIn ← maxIndex(zoneEmptyTable[i]) |
| 4  5  6  7  8  9  10  11 | else  zoneIn←minIndex(|pageLifetimezoneLifetime[i]|)  end  blockAddr=writePage(page,zoneIn)  OP=ADD  refreshLifetime(page,pageLifetime,blockAddr,zoneIn, OP)  GCDetect();  return blockAddr |

算法4给出了数据修改和删除的过程。对数据的修改及删除操作时，首先找到该key所在block地址，将其读入内存并将该block标为无效块，将读出的页中相应的key修改为新的value，或value为空将该key删除，该key的生命周期变为 0，再重新组织成新的页，调用插入算法插入。

|  |  |
| --- | --- |
| 算法4 数据修改和删除 ModifyDelete(key, value) | |
| Input: 修改键 key, 修改值 value  Output: 无 | |
| 1 | \*keyRead, \*valueRead ← readPage(key) |
| 2 | keyRead[key] = value |
| 3 | keyLifetime[key] = 0 |
| 4  5  6  7 | pages←kv2pages(keyRead,valueRead)  for page in pages do  blockAddr=Insert(page, keyRead, MODIFY)  end |

**3.3 生命周期方差感知的垃圾回收**

数据存储管理的另一个问题是垃圾回收。由于ZNS SSD的顺序写限制，Zone内部的写指针只能向后移动，当数据变化时，无法进行就地更新，只能将原先存放该数据的块标为无效块，再将新数据写入。这样的设计使得ZNS SSD面对频繁的修改和删除时，Zone的垃圾率会增长得很快。所以当Zone的空闲率降低，垃圾率提高时，需要调用垃圾回收操作来回收无效块，释放空间以存储更多数据。常规的垃圾回收算法是当SSD中空间不足时调用垃圾回收，在加入垃圾回收队列后性能有所提升[8]。

当工作负载发生变化时，根据上一个工作负载而安排的数据布局就会失效，体现在键值的生命周期发生变化，导致Zone内页的生命周期变得相差较大，这一信息可以由Zone内页的生命周期的方差来体现，当Zone内生命周期的方差过大时，由Zone内页生命周期平均值来表征该Zone的生命周期变得不再精确，此时便需要将这种内部页生命周期差距较大的Zone进行垃圾回收，来保证数据插入算法的有效性。所以将Zone的生命周期方差添加到触发垃圾回收的条件中，用当前工作负载下键的生命周期来重新分配数据。

算法5给出了垃圾回收的过程。当触发垃圾回收时，根据对回收Zone中块的状态，将有效块读取并重新组织成新的数据页，调用数据插入算法写回SSD中。当进行了数据的插入或修改时，根据Zone的状态参数来判断各Zone是否需要进行垃圾回收，并将需要垃圾回收的Zone 加入垃圾回收队列gcQueue中。若该Zone的空闲率低于阈值*se*时对其进行检测，若Zone的垃圾率达到*sg*或者当Zone的生命周期方差大于*sl*时，将该Zone加入垃圾回收队列；而当垃圾回收队列中的 Zone大于垃圾回收队列上限*gl*时，触发垃圾回收。这些阈值如表1所示，他们的最佳值将通过实验来确定。

|  |  |
| --- | --- |
| 算法5 垃圾回收 GarbageCollection() | |
| Input: 无  Output: 无 | |
| 1 | for zone in gcQueue do |
| 2 | for block in zone and block is valid do |
| 3 | key, value ← readPage(block) |
| 4  5  6  7  8  9  10  11  12 | if data read is large enough to form a page then  page ← kv2pages(keyRead, valueRead)  Insert(page, keyRead,ADD)  end  blockStatus ← empty  end  clear zoneLifetime[zone]  end  clear gcQueue |

Table1 Parameters of the GCDetect() algorithm

表1 垃圾回收检测算法的参数

|  |  |
| --- | --- |
| 参 数 | 说 明 |
| *sl* | 触发垃圾回收的生命周期方差阈值 |
| *sg* | 触发垃圾回收的Zone中非空闲部分的垃圾率 |
| *se* | 触发垃圾回收的Zone的空闲率 |
| *gl* | 垃圾回收队列长度 |

在数据插入过程结束时，我们需要判断是否需要执行垃圾回收操作。该过程通过算法6所示的GCDetect()算法来实现。

|  |  |
| --- | --- |
| 算法6 垃圾回收检测 GCDetect() | |
| Input: 无  Output: 无 | |
| 1 | getZoneEmptyRate(zoneEmptyTable) |
| 2 | getZoneGarbageRate(zoneGarbageRateTable) |
| 3 | for zone in all zones do |
| 4 | if zoneEmptyTable[zone]<se then |
| 5  6  7  8  9 | if zoneGarbageRateTable[zone]>sg || LifetimeVarience(zone)>sl then  add zone into gcQueue  end  end  end |
| 10  11  12 | if len(gcQueue)>gl then  GarbageCollection()  end |

**4 实验与分析**

**4.1 实验环境**

由于目前还没有商用的ZNS SSD上市，因此本文使用了ZNS SSD模拟器libzbd[12]来模拟ZNS SSD。实验测试了在不同工作负载下，ZNS SSD感知的数据插入和垃圾回收算法相较于常规算法，在运行时间、垃圾回收代价以及稳定性上的优势。实验环境配置如表2所示。服务器操作系统为Ubuntu，内核版本为5.4.0，gcc版本为9.3.0。模拟的ZNS SSD的参数如表3所示。

Table2 Server configuration

表2 服务器环境配置

|  |  |
| --- | --- |
| 服务器组件 | 说明 |
| OS | Ubuntu 20.04.1 LTS |
| CPU | Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10G |

Table3 Parameters of simulative ZNS SSD

表3 模拟ZNS SSD的参数

|  |  |  |
| --- | --- | --- |
| ZNS SSD参数 | | 说 明 |
| Zone模型 | Host-Managed | |
| 容量 | 1.047GB | |
| 扇区大小 | 512Bytes | |
| Zone数量 | 16 | |
| Zone大小 | 64MB | |

实验使用自行搭建的工作负载对ZNS SSD感知的数据放置算法进行测试，键长度为4字节，范围在0到 220~1。工作负载中共有4∗106次写操作，其中更新操作（包括修改和删除）的占比为50%，其中20%是对5%的热点数据做更新，热点数据经过一段时间后会随机变化，共变化10次，以此来模拟工作负载的变化。首先将本实验中所涉及的阈值进行实验，分析他们对本算法在性能与稳定性上的影响。由于ZNS SSD感知的数据放置对数据插入和垃圾回收都做了优化，所以对比实验分为两部分，第一部分将数据插入改为常规的平均插入和轮询插入，第二部分将垃圾回收的触发条件仅设为空闲率与垃圾率，不加入生命周期方差的判定。

**4.2 实验结果分析**

**4.2.1 垃圾回收检测算法中阈值对结果的影响**

首先对垃圾回收检测算法中所涉及的阈值进行了实验测试，如图4(a)和图4(b)，其中触发垃圾回收的Zone中非空闲部分的垃圾率se与Zone的空闲率se均控制在40%时，执行时间与稳定性最高。

对于生命周期方差的阈值sl，如图4(c)所示，当阈值在3000以下时，时间性能较好，但是由于生命周期方差阈值较低，在ZNS SSD的空闲率较低时，一次垃圾回收操作中回收的有效块插入其他Zone会导致该 Zone 的生命周期方差达到阈值，触发多次垃圾回收操作，造成系统性能的波动；当阈值在5000以上时，由于无法识别数据放置不合理的Zone，导致ZNS SSD在空闲率较低时的总垃圾率较高，从而时间性能较差。而当阈值在4000左右时，时间性能较好，且不会导致连续的垃圾回收操作的触发，能及时优化数据放置。

最后考虑垃圾回收队列的长度对时间性能的影响，如图4(d)，可以看出将多个Zone一起回收可以大幅提升时间性能。而当垃圾回收队列变长时，算法及时识别出个别Zone的生命周期方差发生变化，也不会及时进行垃圾回收操作，无法达到优化数据放置的目标，所以当垃圾回收队列设定为2时，系统的性能与稳定性最佳。

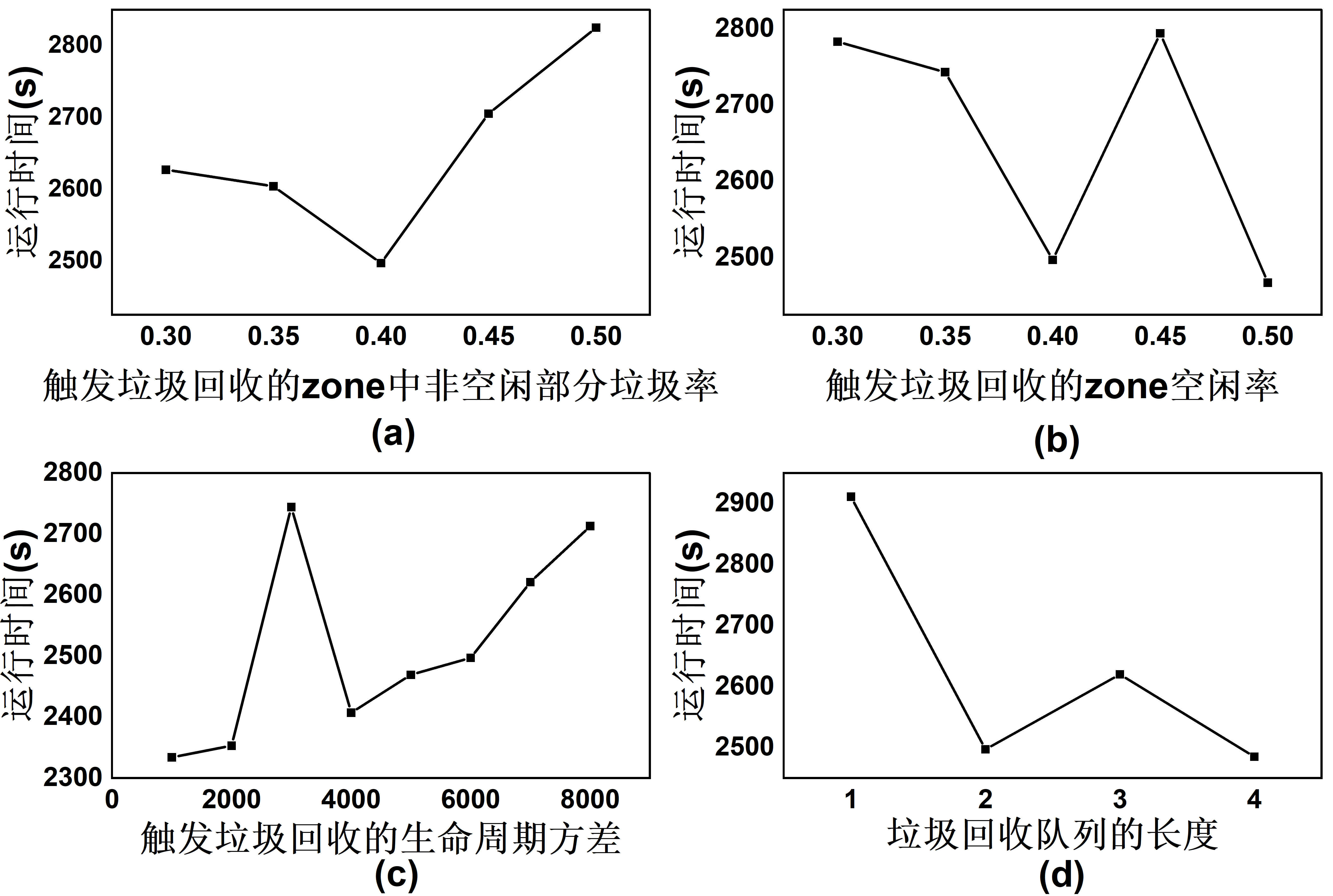


Fig.4 The impact of thresholds on performance in GCDetect() algorithms

图4 垃圾回收检测算法中阈值对性能的影响

**4.2.2 数据插入性能**

这一部分我们对比了基于生命周期的数据插入与平均插入、轮询插入的性能差异。垃圾回收策略均为生命周期方差感知的垃圾回收策略，𝑠𝑔 = 0.4, 𝑠𝑒 =0.4, 𝑠𝑙 = 6000, 𝑔𝑙 = 2。

如图5, 在执行垃圾回收的次数上基于生命周期的数据插入算法相较于平均插入、轮询放插入均减少了 50%；相较于常规垃圾回收减少了。从图中我们也可以看出，两种简单的算法由于并未根据生命周期对数据进行分类，所以触发垃圾回收操作的诱因是较低的空闲率与较高的垃圾率，所以会在某一阶段连续触发垃圾回收操作，来回收多个 Zone；而基于生命周期的数据插入算法在工作负载发生偏移时，基本上都会触发一次垃圾回收操作以调整数据的分布，这样使得ZNS SSD 在面对变化的工作负载时也可以稳定的进行数据的插入与修改。

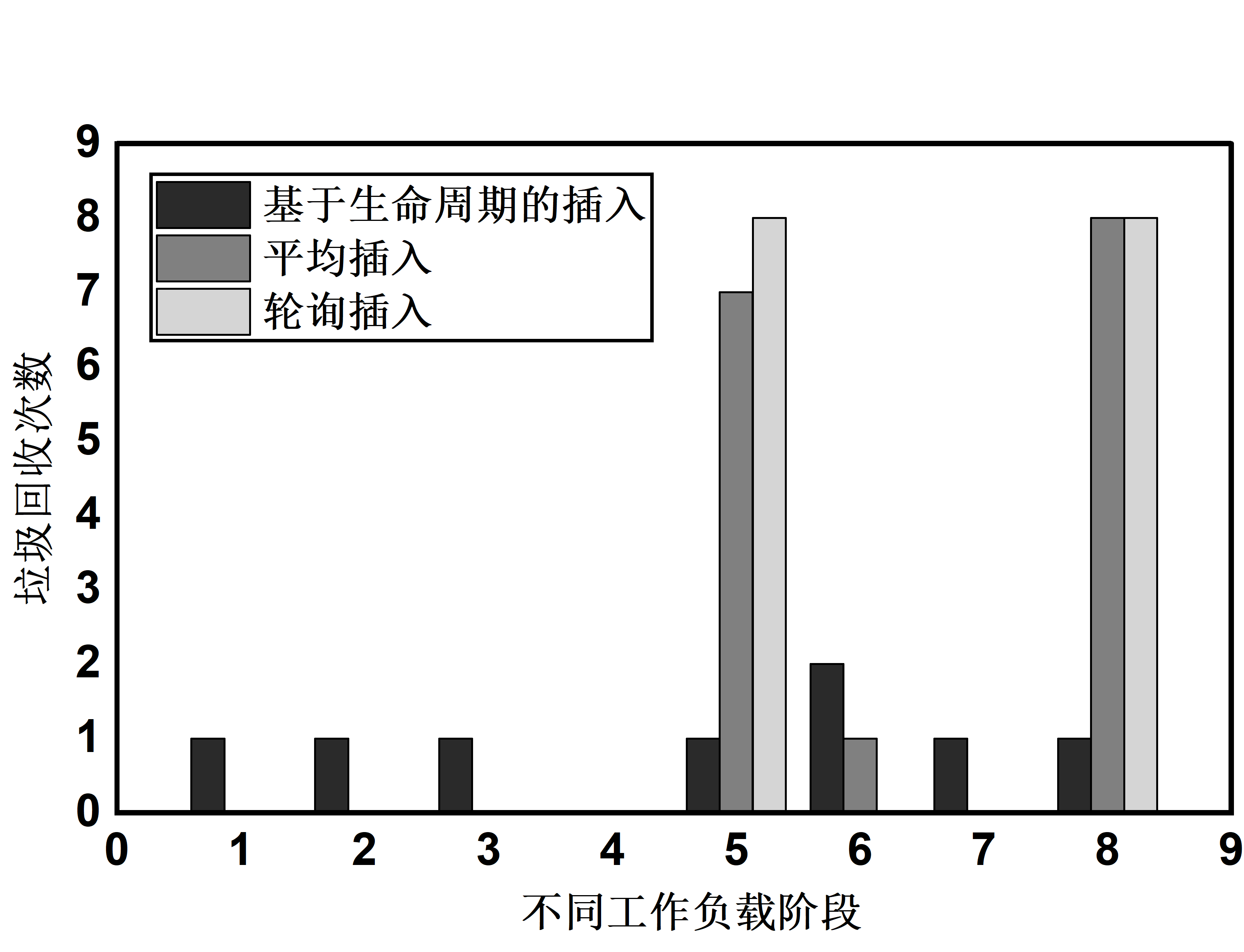


Fig.5 Comparison of garbage collection times of different insertion algorithms under varying workload phases

图5 不同插入算法在变化的工作负载阶段下的垃圾回收次数对比

如图6，在运行总时间上，由于 ZNS SSD 感知的数据放置算法每次插入数据块时，需要找出与其生命周期最接近的块，所以相较于平均放置、轮询放置的运行时间增加了 11.7% 和 8.0%，分别为 2497s，2234s 和 2311s。但是由于这两种算法的垃圾回收较为集中，导致在其进行垃圾回收时出现了不稳定的波动，而基于生命周期的插入算法由于识别了各个工作负载的变动，在不同的工作负载下都进行较少的垃圾回收，使得整个过程中操作执行较为稳定。

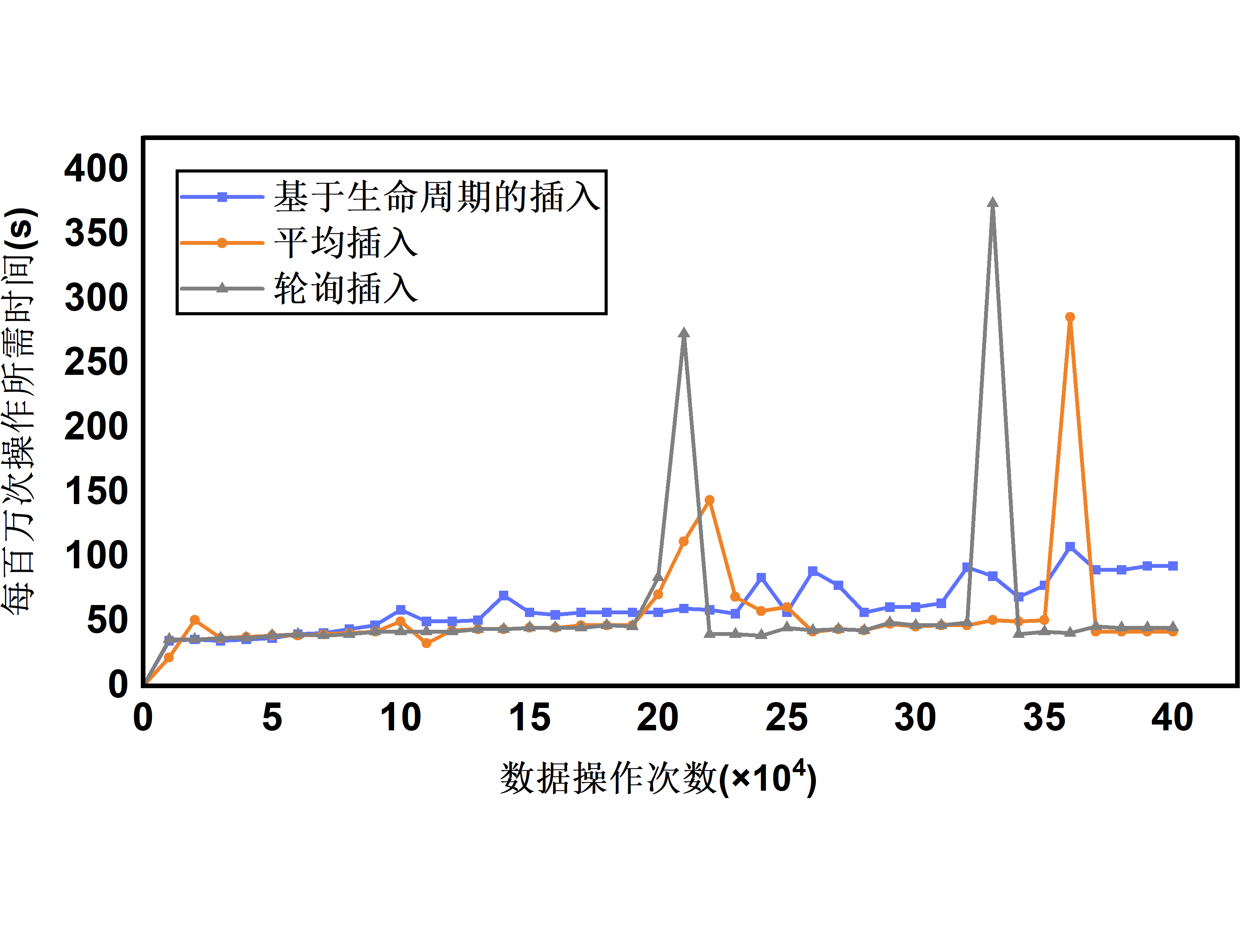


Fig.6 Performance fluctuations of different insertion algorithms under varying workload phases

图6 不同插入算法在变化的工作负载阶段下性能波动

**4.2.3 垃圾回收性能**

在加入生命周期方差作为垃圾回收的触发条件后，运行性能较常规的垃圾回收算法提升了 9%，分别为 2741s 和 2497s。如图7所示，在Zone空闲率降低时，生命周期方差感知的垃圾回收算法越能体现其优势。由图8我们可以看出，生命周期方差感知的垃圾回收比常规垃圾回收更加稳定，根据工作负载变化调整垃圾回收的时机，使得当 ZNS SSD 空闲率降低时，执行数据操作更加稳定。

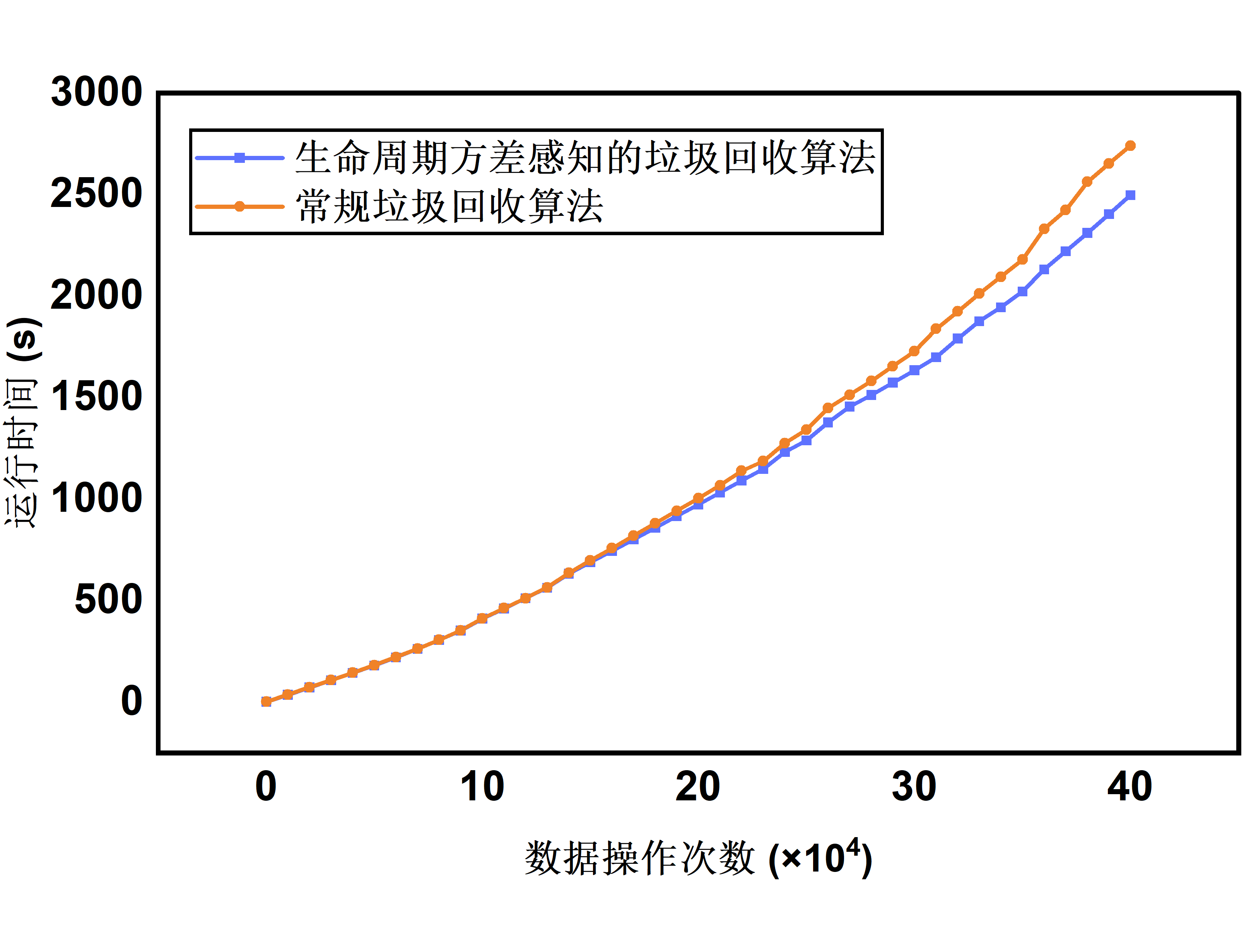


Fig.7 Performance comparison of different garbage collection algorithms over varying workload phases

图7 不同垃圾回收算法在变化的工作负载阶段下的性能对比

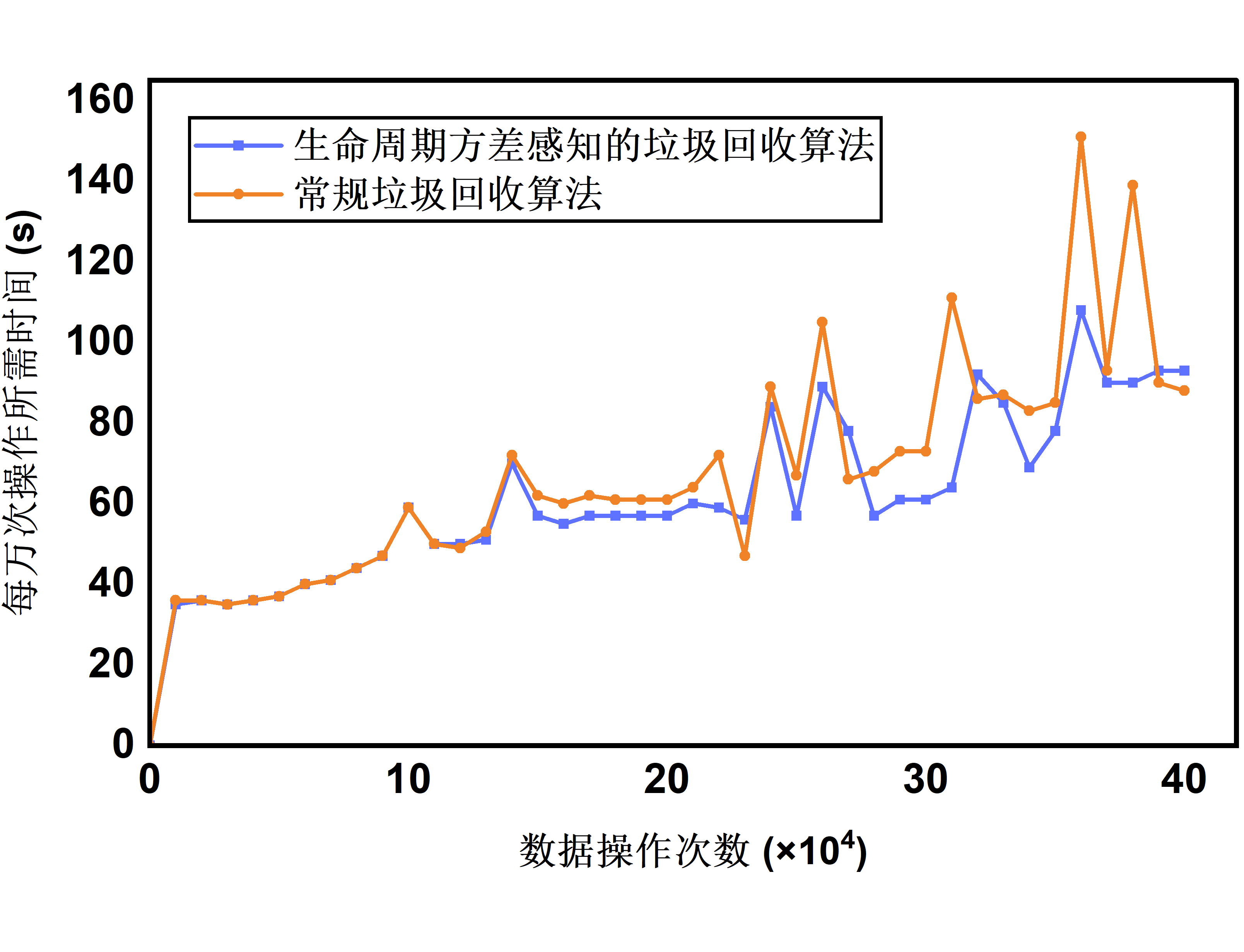


Fig.8 Performance fluctuations of different garbage collection algorithms under varying workload phases

图8 不同垃圾回收算法在变化的工作负载阶段下性能波动

**4.2.4 空间代价**

本实验中，由于需要记录键值、页和Zone的生命周期，在内存中额外需要8.4MB的空间来存储这些信息，这对现今的内存来说影响不大。

**5 结束语**

ZNS SSD是近年来出现的一种新型SSD。本文研究了ZNS SSD感知的数据存储管理问题，提出了基于数据生命周期的ZNS SSD感知的数据插入算法和垃圾回收策略。实验表明，当触发垃圾回收的Zone中非空闲部分的垃圾率与Zone的空闲率为40%，生命周期方差阈值在6000，垃圾回收队列长度为2时，我们提出的算法的性能与稳定性最佳。同时，基于生命周期的插入算法比平均插入、轮询插入算法在稳定性上有很大提升，生命周期方差感知的垃圾回收算法比常规垃圾回收算法在时间性能上提升9%，且更加稳定。

References:

1. Bjørling M. From open-channel SSDs to zoned namespaces[C]. Linux Storage and Filesystems Conference (Vault 19). 2019, 1.
2. Stavrinos T, Berger D S, Katz-Bassett E, et al. Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete[C]. Proceedings of the Workshop on Hot Topics in Operating Systems. 2021: 144-151.
3. Western Digital Corporation. Zoned Stoage [EB/OL]. zonedstorage.io , 2022.
4. Shin H, Oh M, Choi G, et al. Exploring performance characteristics of ZNS SSDs: Observation and implication [C]. 2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA). IEEE, 2020: 1-5.
5. 林子雨.闪存数据库概念与技术 [M]. 人民邮电出版社. 2015
6. Chang L P. On efficient wear leveling for large-scale flash-memory storage systems[C].Proceedings of the 2007 ACM symposium on Applied computing. 2007: 1126-1130.
7. Choi G, Lee K, Oh M, et al. A New LSM-style Garbage Collection Scheme for ZNS SSDs [C].12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20). 2020.
8. Oh G, Yang J, Ahn S. Efficient Key-Value Data Placement for ZNS SSD[J]. Applied Sciences, 2021, 11(24): 11842.
9. M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. Le Moal, G. R. Ganger, and G. Amvrosiadis, ZNS: Avoiding the block interface tax for flash-based SSDs [C], in USENIX ATC, pp. 689–703, 2021.
10. C. Lee, D. Sim, J. Hwang, and S. Cho, F2FS: A new file system for flash storage [C], in FAST, pp. 273–286, 2015.
11. K. Han, H. Gwak, D. Shin, and J. Hwang, ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction [C], in OSDI, pp. 147–162, 2021.
12. westerndigitalcorporation/libzbd [EB/OL]. https://github.com/

westerndigitalcorporation/libzbd, 2022

1. Maheshwari U. From blocks to rocks: A natural extension of zoned namespaces[C]. Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems. 2021: 21-27.
2. Purandare D R, Wilcox P, Litz H, et al. Append is Near: Log-based Data Management on ZNS SSDs[C]. 12th Annual Conference on Innovative Data Systems Research (CIDR’22). 2022.
3. Chan H H W, Liang C J M, Li Y, et al. HashKV: Enabling Efficient Updates in KV Storage via Hashing[C], 2018 USENIX Annual Technical Conference (USENIX ATC 18). 2018: 1007-1019.
4. Dayan N, Bonnet P. Garbage collection techniques for flash-resident page-mapping FTLs[J]. arXiv preprint arXiv: 1504.01666, 2015.
5. Ding J, Minhas U F, Chandramouli B, et al. Instance-optimized data layouts for cloud analytics workloads[C]. Proceedings of the 2021 International Conference on Management of Data. 2021: 418-431.
6. He S, Sun X H, Feng B, et al. A cost-aware region-level data placement scheme for hybrid parallel I/O systems[C]. 2013 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2013: 1-8.
7. Aghayev A. Adopting zoned storage in distributed storage systems[J]. Ph. D. dissertation, 2020.