Just-in-Time Garbage Collection for High-Performance SSDs with Long Lifetimes

Sangwook Shane Hahn, Hoyoon Jun, Jaeyong Jeong and Jihong Kim
Department of Computer Science and Engineering, Seoul National University, Korea

Garbage collection (GC) is the essential operation for NAND flash-based storage systems. Because of the NAND's erase-before-write constraint, when writing new data to the NAND page, an out-place update is used by writing new data to a new NAND page instead of updating the original page. An out-place update policy generates invalid pages with old data that must be reclaimed by garbage collection.

Since GC requires valid page migrations as well as block erasures, it incurs a significant performance overhead, in particular, if GC is necessary for the current write request (i.e., when a foreground GC (FGC) operation is necessary). In order to hide the performance penalty of a foreground GC operation, background garbage collection (BGC) is commonly used when a storage system is idle so that most foreground GC operations can be avoided. Although background GC operations can be effective in reducing the performance penalty of foreground GC operations, invoking background GC operations too aggressively can significantly degrade the lifetime of NAND-based storage systems by erasing blocks too early. Selecting the right time for invoking a background GC operation is important because it can affect both the performance and lifetime of NAND-based storage systems as shown on page 2 of the poster.

In this poster, we propose a just-in-time (JIT) GC technique, called JIT-GC, that performs background garbage collection operations only when necessary, thus improving both the performance and lifetime of NAND-based storage systems. As shown on page 2 of the poster, JIT-GC aims to overcome the shortcomings of existing BGC policies. An aggressive BGC policy, which reserves a large free space, can avoid most FGC operations and achieves high performance. However, its performance improvement comes with a shortened lifetime. On the other hand, a lazy BGC policy, which maintains a small free space, does not sacrifice the lifetime of a SSD but it can significantly degrade the performance.

In order to perform garbage collection in a just-in-time fashion, we need to know future write demand in advance because the timeliness of GC invocations depends on how much future writes are performed. To estimate the future write demands, we exploit the page cache. Since most file I/O requests are processed through the page cache, our heuristic takes advantages of *how* the page cache handles file I/O requests and *when* it moves data to a SSD. For example, since most writes are managed using a write back policy in the page cache, most new writes will be written to the SSD by a special flusher thread after some delay (e.g., 30 seconds). By scanning the page cache, it is possible to estimate with a high accuracy when the flusher thread will write which data to the SSD by how much. JIT-GC exploits these estimated results in invoking GC operations. An example page cache shown on page 3 of the poster illustrates that for each 5-second interval $I_{[T_c+5\cdot j, T_c+5\cdot(j+1)]}$ (in short I_j) (where $0 \leq j \leq 5$), we estimate the amount of expected writes W_j. That is, W_0 = 25 MB, W_1 = 15 MB, W_2 = W_3 = W_4 = 0 B, and W_5 = 100 MB.

Given future write estimates, W_j's, from the page cache, we decide how much free space should be reclaimed for each I_j. For example, as shown on page 3 of the poster, if W_j's are almost zero, no GC is necessary. On the other hand, if W_5 is very large, we proactively reclaim free space from earlier intervals so that W_5 can be written to a SSD without FGC operations. Furthermore, since our heuristic can tell which data (in the SSD) will be invalidated in a near future by the flusher thread, similarly as proposed in Zombie Chasing [1], we pass the addresses of soon-to-be-invalidate pages to an FTL so that BGC avoids choosing blocks with many soon-to-be-invalidate pages as victim blocks.

The key weakness of our page cache-based heuristic is that our heuristic cannot handle the case when a nontrivial amount of data is written to the SSD directly, bypassing a page cache. For such a case, it can be difficult to estimate the amount of future writes because such writes can happen at any time. In JIT-GC, direct writes, which bypass the page cache, are managed by using a dedicated over-provisioning space. Although a sufficient over-provisioning area can service direct writes without invoking FGC operations, maintaining a large over-provisioning area has the same disadvantage of an aggressive BGC technique. In order to maintain the right amount of the over-provisioning area for direct writes, JIT-GC employs a heuristic predictor for estimating future direct writes. Our heuristic estimates future direct writes using the cumulative data histogram of a given I/O traffic.

In order to evaluate our proposed scheme, we implemented JIT-GC on a modified Samsung SSD (SM843T) [2] connected to a PC host running the Linux kernel. We implemented the future write predictor module in the Linux kernel (version 3.11.1). In order to trigger GC in the modified SSD, we used the SG_IO (SCSI generic I/O) ioctl command which allows the host to send SCSI commands to a storage device. The three benchmarks, Yahoo Cloud Serving Benchmark [3], Filebench (a file-server workload benchmark) [4] and TPC-C (one of online transaction processing benchmarks), were used for our evaluations. Our evaluation results show that JIT-GC improves the overall system performance by increasing IOPS by up to 220% over the lazy BGC policy which invokes BGC when the free space becomes less than 3% of the total capacity of the SSD. The lifetime is also improved by lowering WAF by up to 38% over the aggressive BGC policy which invokes BGC when the free space becomes less than 9% of the total capacity of the SSD.

References

[1] Y. Lee, et al., "Zombie Chasing: Efficient Flash Management Considering Dirty Data in the Buffer Cache," *IEEE Transactions on Computers*, 2013.

[2] SAMSUNG 843T Data Center Series, http://memorysolution.de/mso_upload/out/all/SM843T_Specification_v1.0.pdf

[3] B.F. Cooper, et al., "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143-154, 2010.

[4] P. Sehgal et al., "Evaluating Performance and Energy in File System Server Workloads," in *Proc. USENIX Conf. File and Storage Technologies*, pp. 19-33, 2010.

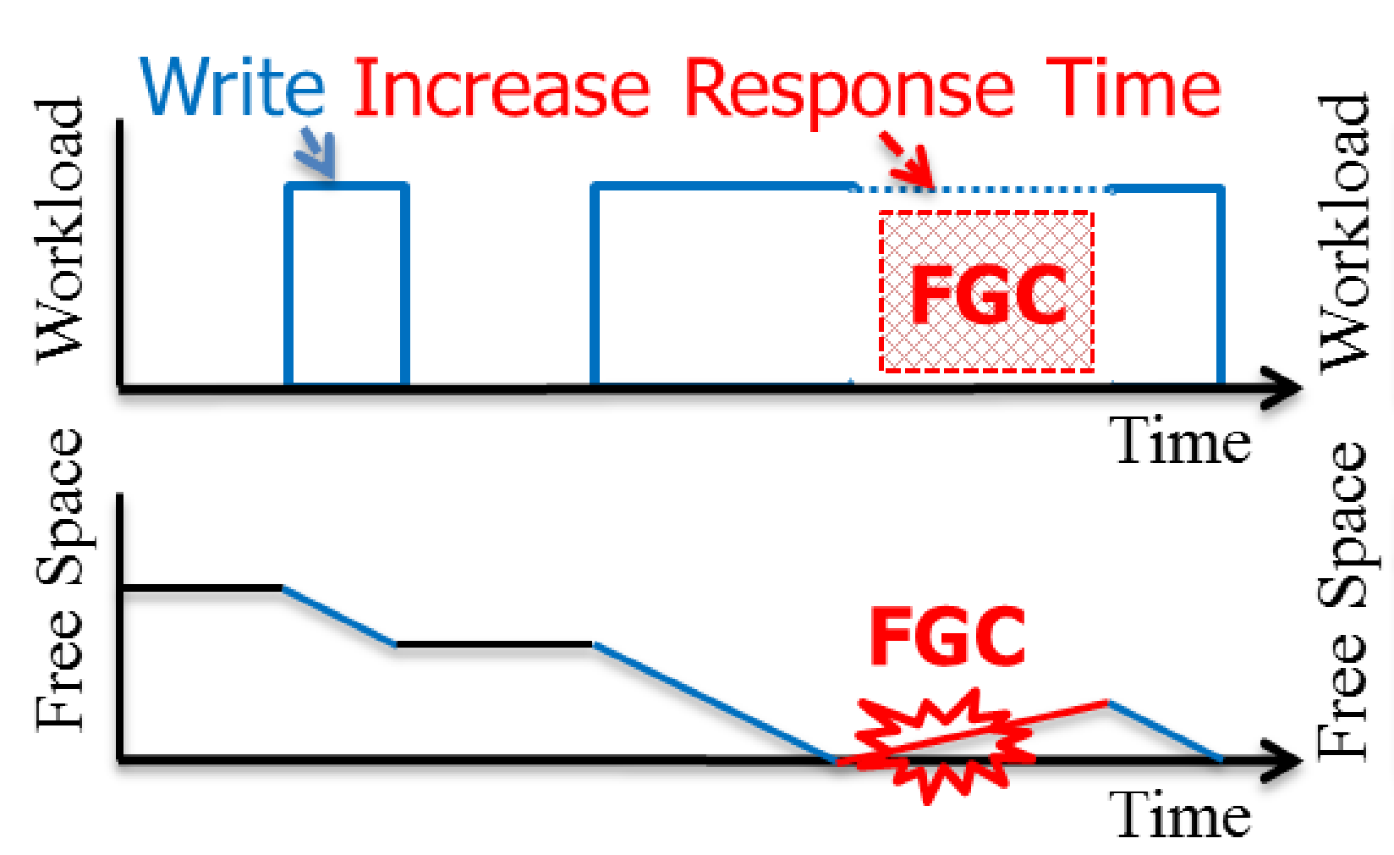
Just-in-Time Garbage Collection for High-Performance SSDs with Long Lifetimes

Sangwook Shane Hahn, Hoyoon Jun, Jaeyong Jeong and Jihong Kim
Department of Computer Science and Engineering, Seoul National University

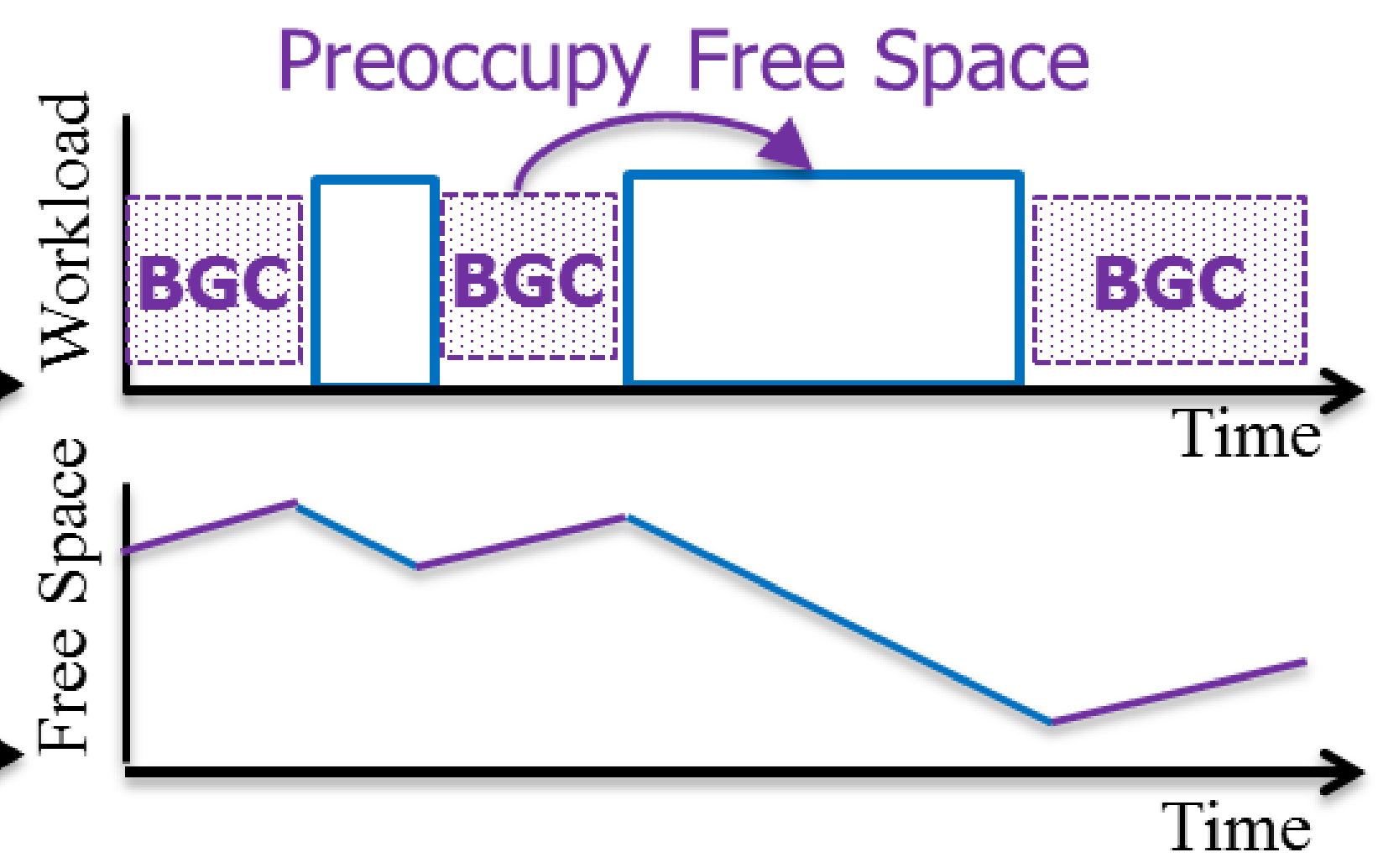
Garbage Collection in SSD

- **Unique NAND characteristics**
  - Out-place update
  - Different operation unit (Read/Write: Page, Erase: Block)

- **Foreground GC (FGC)**
  - Invoked at empty free space

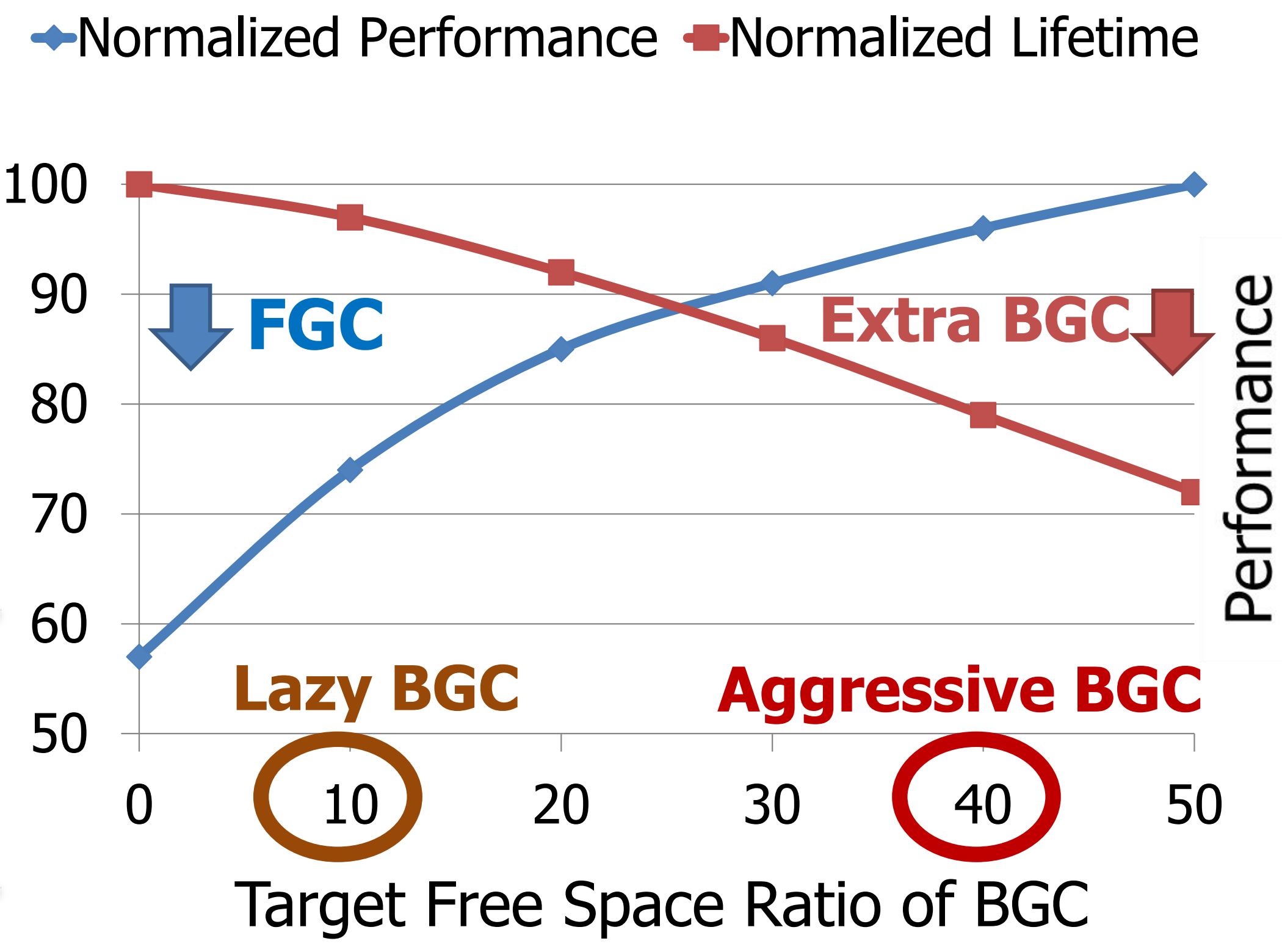


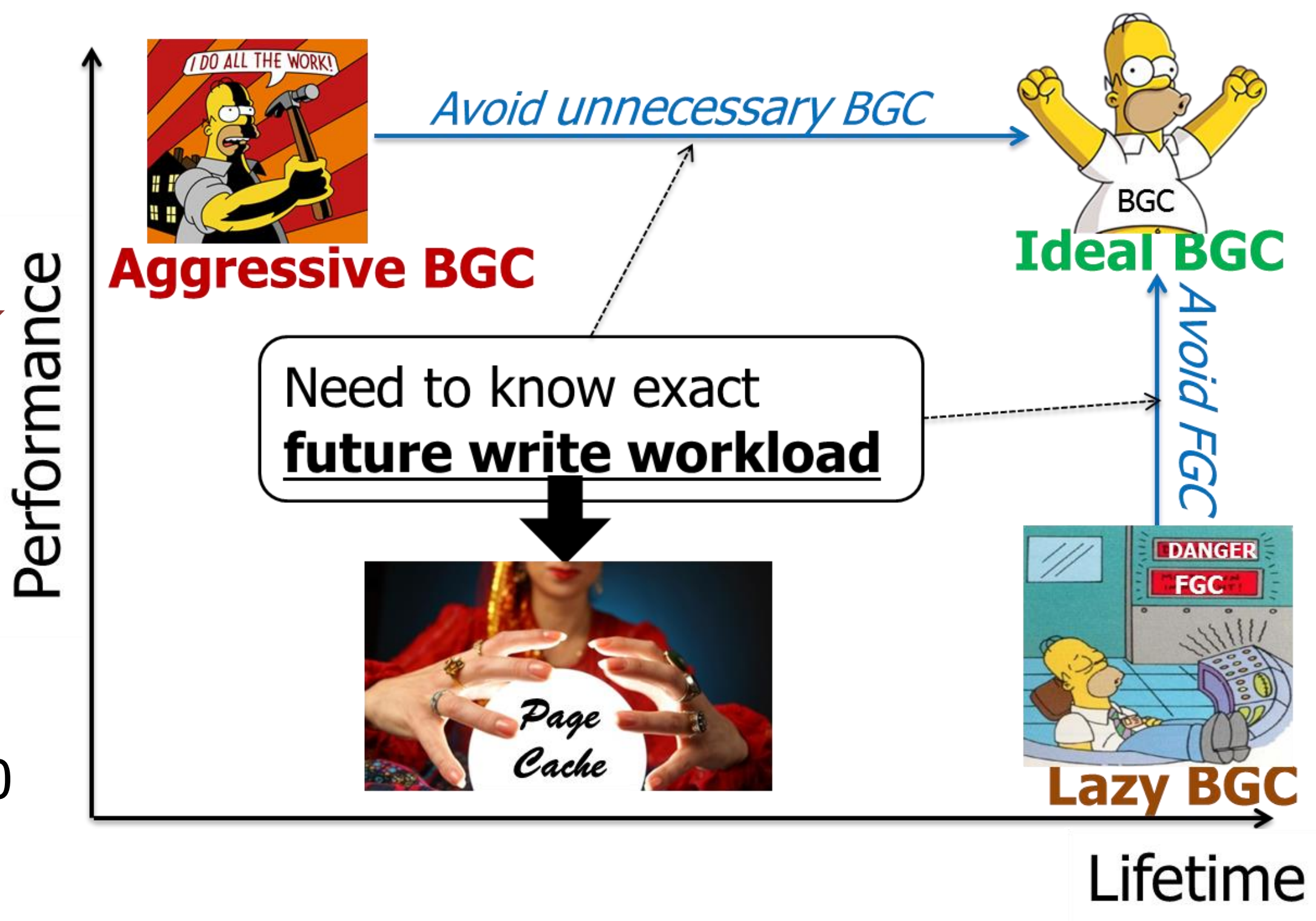
- **Background GC (BGC)**
  - Invoked at idle time



Tradeoff between Performance and Lifetime 2

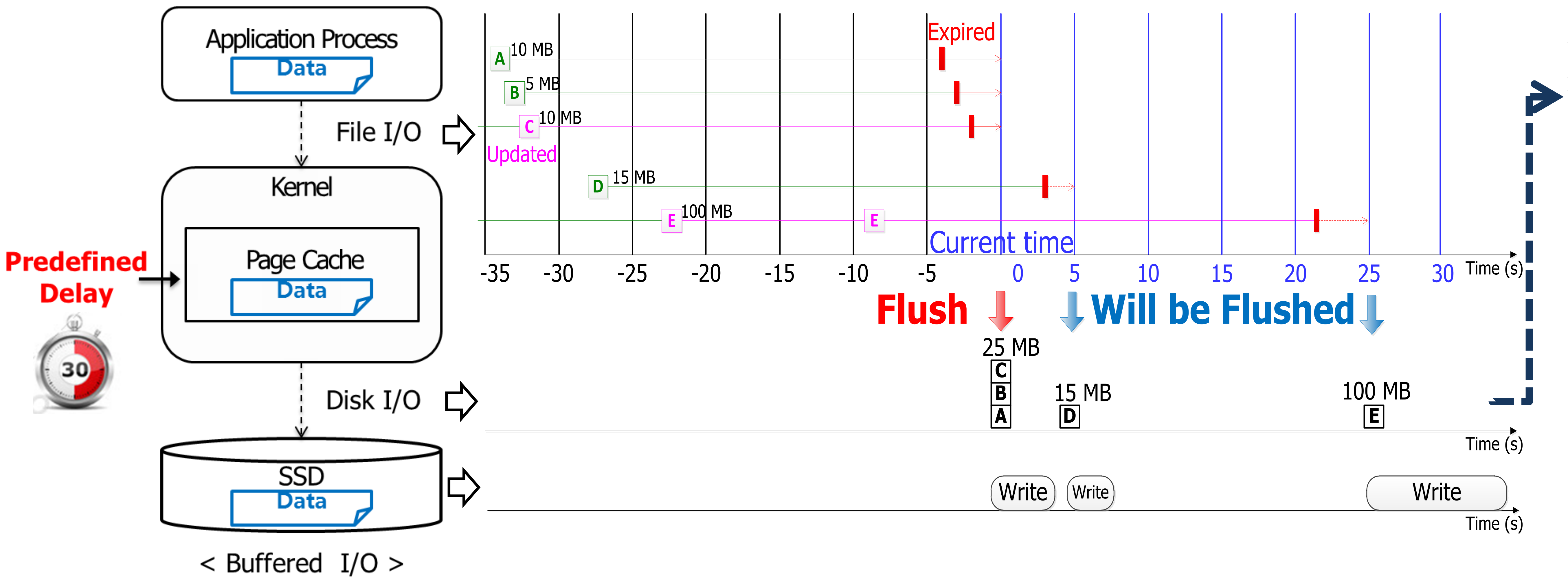
- **Dilemma of BGC**
  - Need to choose one of performance and lifetime





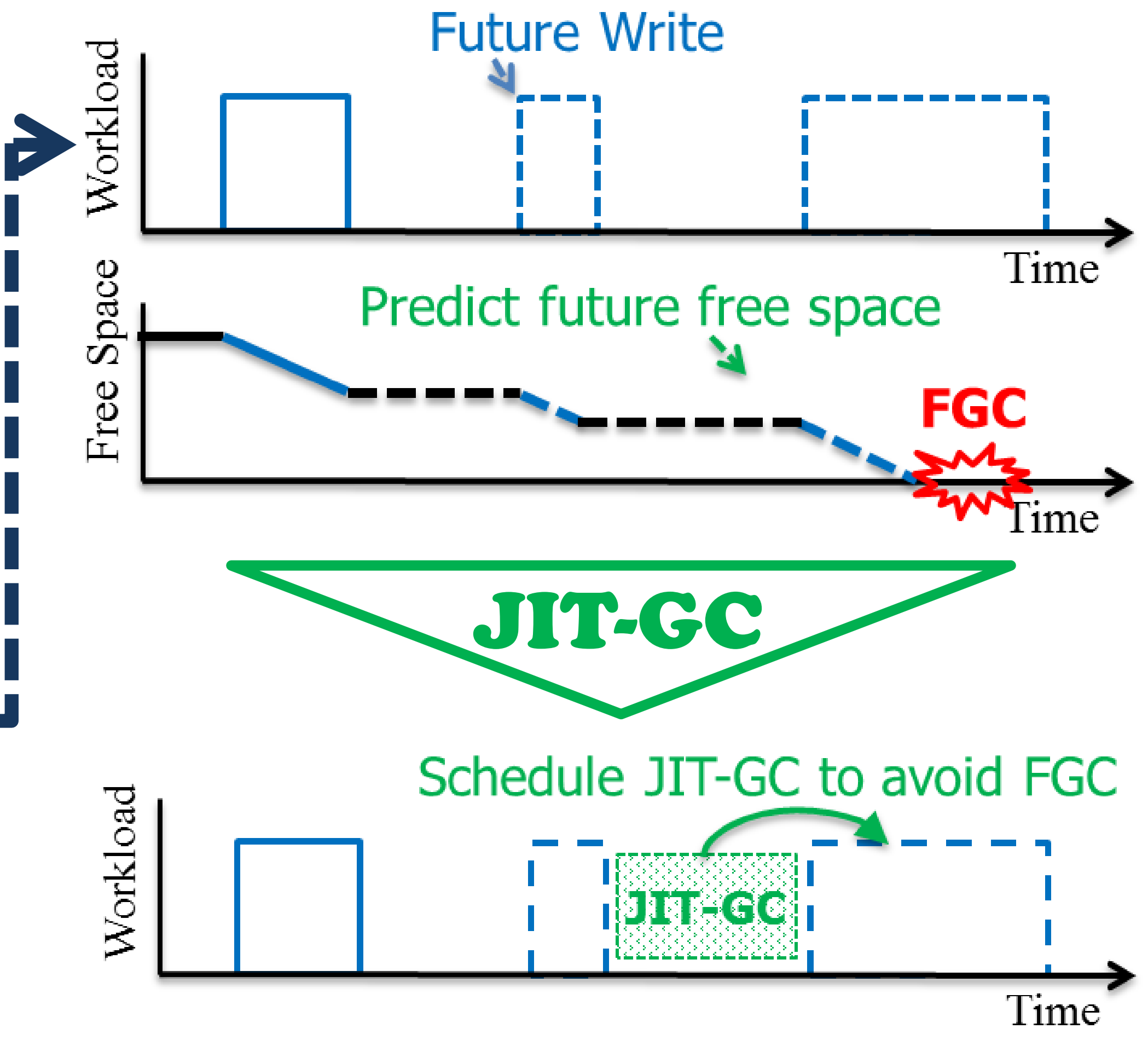
Estimation of Future Write Workload 3

- **Page cache with write-back policy**
  - Buffered I/O: performance improvement from data locality



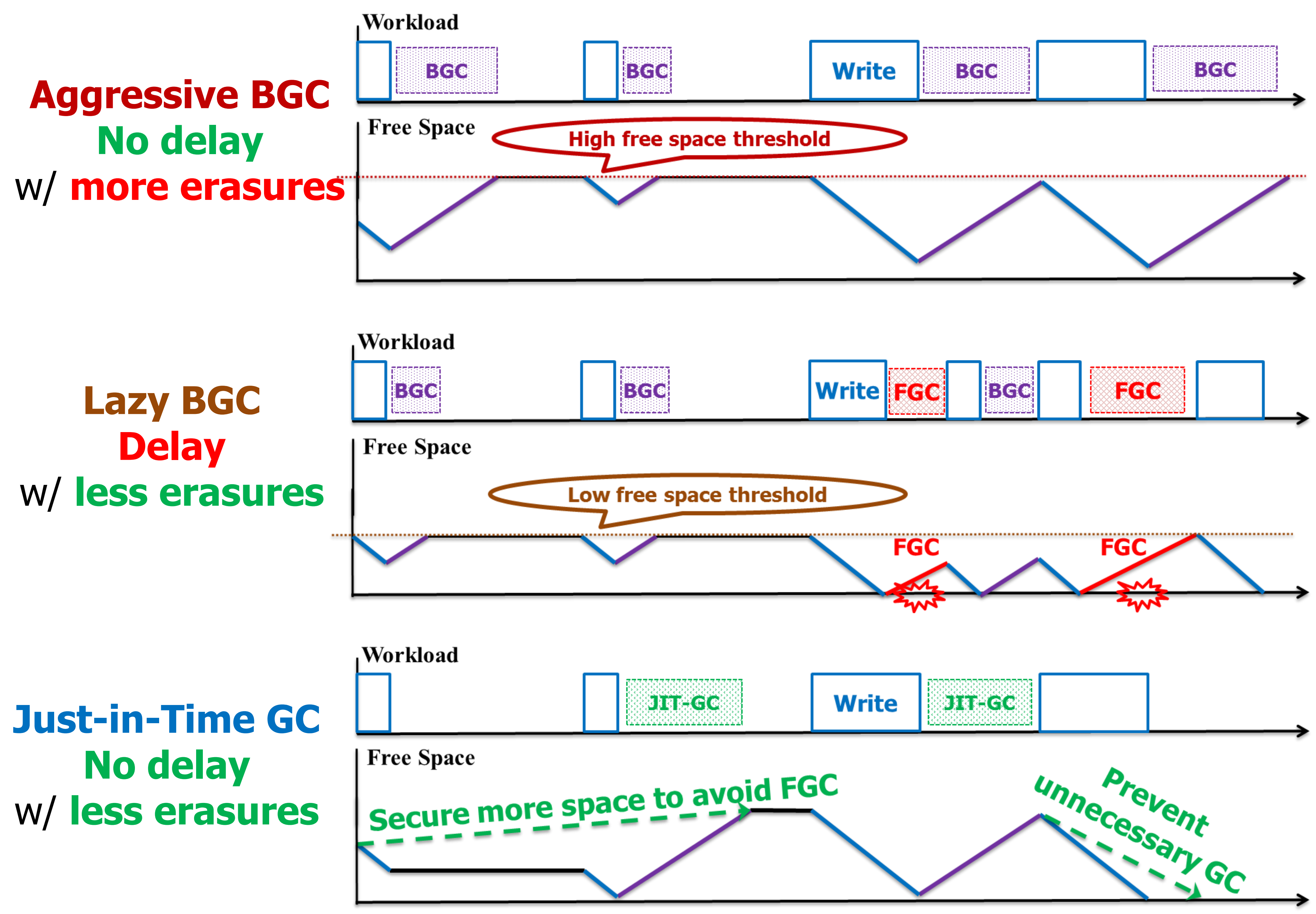
Just-in-Time GC 4

- **Our approach**
  - Perform GC **only when necessary** by monitoring both kernel and SSD



Example Scenario 5

- **Comparisons between JIT-GC and existing BGC schemes**
  - Measure response time delay and number of GC under same workload



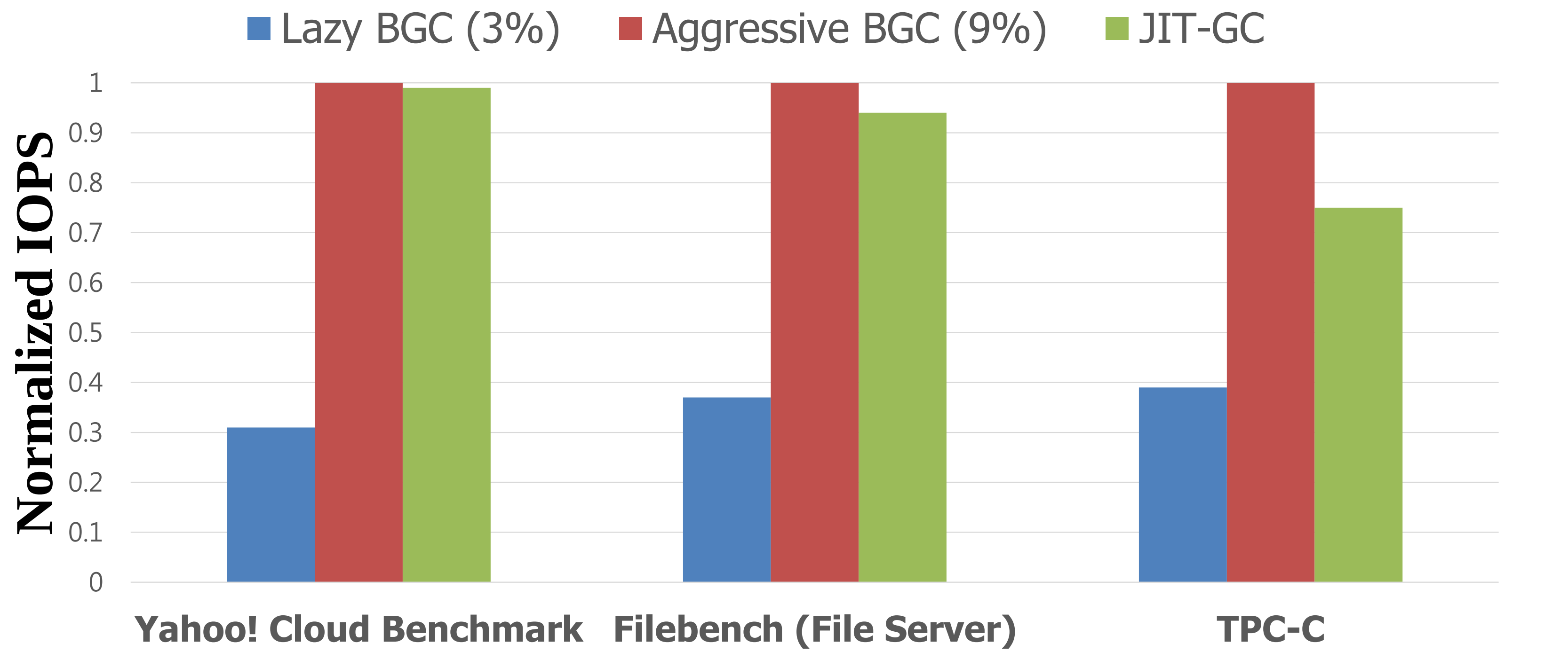
Aggressive BGC
No delay
w/ **more erasures**

Lazy BGC
Delay
w/ **less erasures**

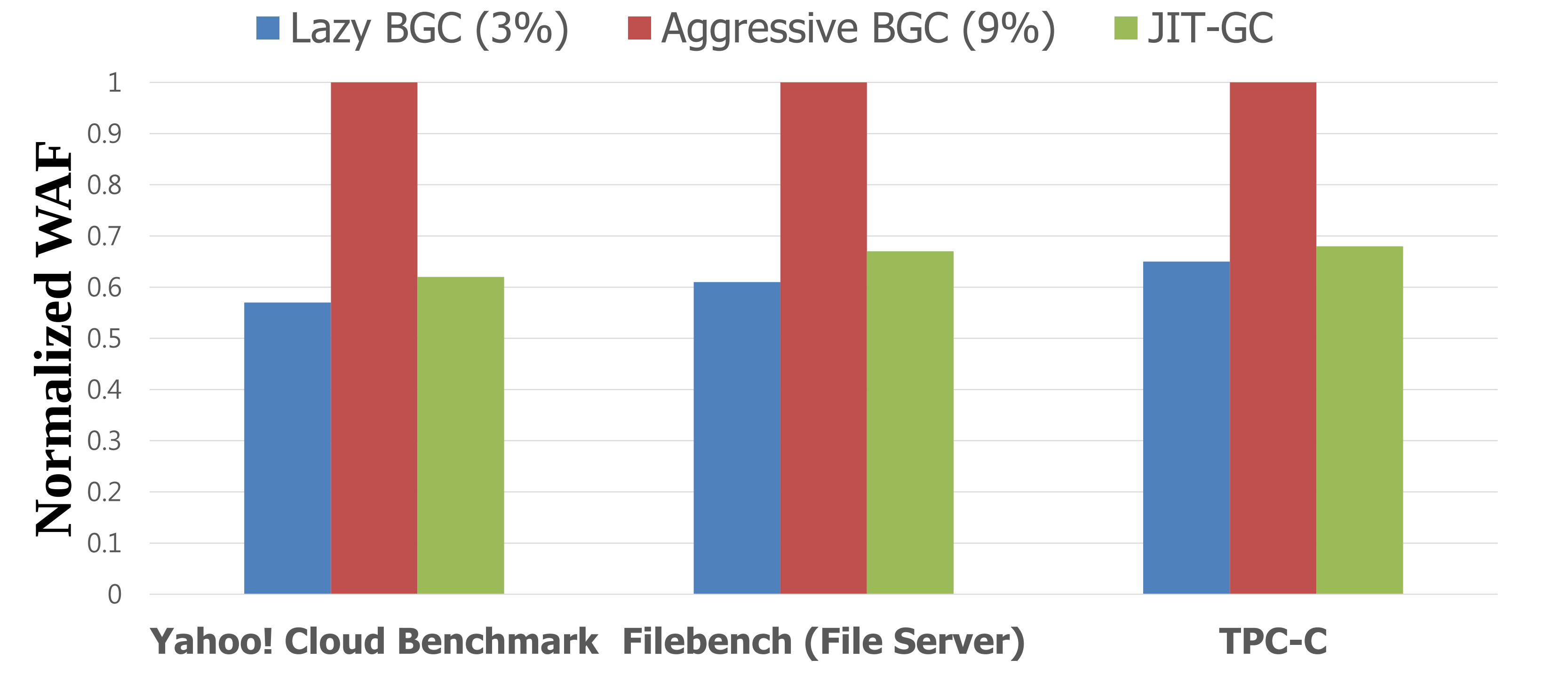
Just-in-Time GC
No delay
w/ **less erasures**

Experimental Results 6

- Normalized IOPS (Input/Output Operations Per Second)



- Normalized WAF (Write Amplification Factor)


Lazy BGC (3%)
Aggressive BGC (9%)
JIT-GC
Normalized WAF
Yahoo! Cloud Benchmark
Filebench (File Server)
TPC-C