APPENDIX

A. *Scalability with Larger Flash Page Sizes*

While the paper currently assumes a 4 KB flash page size, Fraggle's design maintains its robustness and effectiveness as flash memory technology advances to adopt larger page sizes (e.g., 8 KB or 16 KB). For preallocation efficacy, larger page sizes enable larger segment sizes, directly enhancing preallocation effectiveness by securing more contiguous LBA-space file layouts. Meanwhile, logical space waste from unused preallocated regions remains negligible given the 8 ZB scale. In terms of HashFTL performance, a larger segment size enabled by the larger page size is also beneficial for enhancing HashFTL's cache efficiency during sequential file accesses, as more mapping entries can be loaded in one translate page access. Meanwhile, since FTL mapping is maintained at the 4 KB unit, small (e.g., 4 KB) write requests incur no additional write overhead, even with larger flash pages. Therefore, Fraggle's adaptability to larger page sizes (e.g., 8 KB or 16 KB) ensures the core benefits — improved preallocation effectiveness and optimized HashFTL performance — are not only preserved but can even be further enhanced with evolving flash techniques. This confirms Fraggle's long-term viability and relevance amid ongoing advancements in SSD technology.

B. *Device-level GC Process of Fraggle*

We illustrate the device-level GC process of Fraggle. Since NAND flash enforces erase at the unit of blocks and writes at flash pages, GC needs to migrate valid pages in the victim block before erasing it. To identify whether a page is valid, FTL typically compares the physical address of the page and the one recorded in the GTD. If they are the same, the page is valid and needs to be migrated. Otherwise, the page can be directly reclaimed. Besides, GC relies on a reverse mapping table (PPA-to-LPA), typically stored in the out-of-band (OOB) area of the flash page. The GC process includes two types of flash pages, namely data page GC and translation page GC, detailed as follows:

Data Page GC. The OOB of each data page records its corresponding logical page address (LPA). During data page GC, Fraggle first selects the victim flash superblock (i.e., consisting of all flash blocks with the same offset across all parallel units) within the SSD based on the greedy algorithm (i.e., selecting the superblock with the most invalid pages). Then, Fraggle finds a new location (destination) to accommodate valid pages residing in the victim superblock (source). For each valid page, Fraggle copies it from the source to the destination. Then, Fraggle updates the data page's new location by modifying the relevant HashFTL mapping entry. When modifying HashFTL, the only difference from page-level FTL mapping lies in its locating the mapping entry: Fraggle hashes the LPA from the data page's OOB to identify the corresponding Global Translation Directory (GTD) entry, which stores the physical address of the translation page (TPPA). The TP content is then read into SRAM, and a binary search locates the matched mapping entry within the segment. After the data is written to a new physical page address (PPA), this entry is updated. Finally, after the modified TP is persisted to a new flash page, Fraggle updates the corresponding TPPA in the GTD.

Translation Page GC. The OOB of each translation page (TP) stores the index of the corresponding GTD entry. Note that the translation pages and data pages are stored in separate superblocks. When the victim superblock stores TPs, similar to data page GC, fraggle will first find the destination for valid TPs. After TPs are migrated to a new physical address (TPPA), Fraggle updates the relevant GTD entry to reflect the new location.

GC Overhead of HashFTL. The overhead of HashFTL during the GC process mainly refers to the number of translation page accesses, which is highly relevant to the data write pattern. Specifically, when the migrated valid pages are mapped to dispersed logical block addresses (LBAs), excessive translation page (TP) updates occur due to poor mapping cache efficiency. This situation is analogous to the remapping process during file defragmentation. Such a case typically arises due to file fragmentation in the LBA space or random data writes, where both HashFTL and TPFTL incur the same overhead. Conversely, when the LBAs of migrated data pages are consecutive, the number of modified TPs is significantly reduced. This scenario typically occurs during sequential file writes by the host file system. Notably, in such sequential cases, HashFTL modifies four times more TPs than traditional page-level FTL because a segment in HashFTL covers up to 1 MB of consecutive LBA range, whereas a TP of page-level FTL covers 4 MB. Fortunately, HashFTL still introduces minimal overhead in this situation. This is because extensive sequential writes in mobile devices, often associated with bandwidth-intensive operations like application updates, multimedia file downloads, or large data transfers, make data I/O the primary bottleneck. By batching TP modifications and writing them to a superpage — a technique widely adopted in commercial SSDs — the bandwidth consumed by TP updates in HashFTL becomes negligible, amounting to only 0.39% (writing 256 superpages (64 MB) of data requires just one superpage of TP writes). Moreover, since the write workloads of mobile devices are predominantly characterized by 4KB random and synchronous writes [7], [32], the LBAs of migrated pages are likely to be dispersed due to file fragmentation (in F2FS) or random updates. To summarize, HashFTL incurs a garbage collection overhead comparable to that of traditional page-level FTL.