Near-Optimal Offline Cleaning for Flash-Based SSDs

Mansour Shafaei
Northeastern University
Boston, MA 02115
Email: shafaei@ece.neu.edu

Peter Desnoyers
Northeastern University
Boston, MA 02115
Email: pjd@ccs.neu.edu

***Abstract*—We present the first description of offline optimal solutions for FTL and LFS cleaning, focusing on the single-write-frontier demand-clean case. We describe an approximate solution to this problem, based on tree pruning and Monte Carlo Tree Search. Results are presented supporting the accuracy of this approximation, based on both toy problems—validated against exhaustive search—as well as longer runs. Finally we show results for real-world traces, comparing them to both Greedy cleaning for the same single-write-frontier case, as well as to an online dual-write-frontier solution based on hot/cold data identification.**

Optimal cleaning is seen to offer modest improvements over Greedy, by an amount which is in fact negligible in comparison to the improvements possible with a fairly simple dual-write-frontier solution. From these results we draw the following conclusion:

> **Efficient cleaning for real workloads is not a matter of deciding which blocks to *select* for cleaning, but rather of deciding where to *place* incoming data, and then collecting the free space generated by these decisions.**

I. Introduction

NAND Flash, as used in SSDs, may be read or written in units of pages (typically 4 KB or more) but pages may not be re-written until they are erased as part of a larger unit (typically 128 or more pages). Efficiently providing a re-writable block interface on top of this technology requires a log-structured or "out-of-place write" translation layer, where newly-erased storage is used for incoming writes, and a *cleaning* (or *garbage collection*) process is used to reclaim space used by outdated information.

Algorithms for cleaning have been discussed in the literature, and their performance examined both experimentally and in some cases analytically [1], [2], [3], [4], [5], [6]. However, although we are able to measure and predict how well a given algorithm will perform, in all but the simplest synthetic case we do not know the limits of translation layer performance—we can measure whether one algorithm performs better than another, but do not know how much room for improvement there is beyond the better of the two.

We can contrast this with the case of cache replacement, where a simple algorithm (Belady's MIN [7]) provides an upper bound for the performance of any algorithm on a specific workload. This provides a baseline to which real algorithms may be compared, in a sense factoring out the difficulty of a particular workload.

In this paper we present the first attempt (known to the authors) to establish offline optimal bounds to cleaning performance for arbitrary workloads, addressing the specific case of a page-mapped translation layer with a *single write frontier* for both new writes and cleaned pages, and *demand cleaning*, where a block is selected for cleaning at the precise point when we run out of free pages. Since an exact solution to the optimal cleaning problem appears to be NP-Hard we describe an approximation to this bound using tree pruning and Monte Carlo Tree Search.

Performance results from this approximate algorithm on real traces allow us to make the following conclusions:

- near-optimal cleaning, an approximation of optimal offline cleaning based on branch pruning in combination with Monte Carlo Tree Search is computationally feasible for realistic real-world traces;
- accuracy of this algorithm is assessed via exhaustive search on small traces and Monte Carlo simulations of increasing size for large traces;
- near-optimal cleaning is shown to offer modest improvements over online Greedy cleaning in the constrained single-write-frontier demand-cleaned case, resulting in performance which is lower than that obtained by multi-write-frontier online cleaning algorithms. (e.g. hot/cold separation)

The final point has the broadest implications. Cleaning is typically conceived of as a process of *selecting* the optimal block to clean, whether based solely on occupancy (Greedy) or based on predictions of future behavior (Rosenblum's Cost/Benefit [2] heuristic). We believe that these results show the importance of page *placement* on cleaning performance: without multiple write frontiers, allowing pages of similar lifespan to be placed in the same block, even the best offline cleaning algorithm is unable to achieve significant improvements over online Greedy cleaning.

II. Problem Definition and Considerations

We consider a device with T blocks of N_p pages each, giving a physical address space $P = \{b, p\} \mid b \in \{1 \ldots T\}, p \in \{1 \ldots N_p\}$, a logical address space $L = \{1 \ldots N\}$ where $N < T \cdot N_p$, a mapping $M \colon L \to P$ and an address trace of length m, $\{a_i \mid i = 1, \ldots m\} \in L$. The *utilization* ρ of the device is the ratio of valid to total physical pages, which will reach a limit $\frac{N}{T \cdot N_p}$ at the point when all pages in L have occurred at least once in the address trace. At any given time the *contents* of particular physical block b is the set of LBAs $\{l \mid M(l) = (b, p)\}$ for $p \in \{1 \ldots N_p\}$. The *write frontier* is a physical address $w = (b_w, p_w)$ where the next write will be performed; i.e. if $w = (b_w, p_w)$ before write access a_i, then afterwards $M(a_i) = (b_w, p_w)$. We assume an initial value $W = (1, 1)$.

After a write access, if $p_w < N_p$ then the write frontier will be incremented by 1, to $(b_w, p_w + 1)$. Otherwise a new block b'_w will be selected for the write frontier and *cleaned*: if on selection there are v valid pages $\{l_1 \ldots l_v\}$ in b'_w they will be moved to pages $1 \ldots v$ (i.e. M updated so $M(l_i) = (b'_w, i)$) and the write frontier set to $(b'_w, v + 1)$ before write a_{i+1} is performed[1].

Finally, we define a *cleaning schedule* as a sequence of block numbers $\{b_i\}$ which correspond to valid cleaning selections for the translation layer described above; i.e. for $i = 1 \ldots \mid \{b_i\} \mid$, when b_i is selected with $v_i < N_p$ valid pages, after $N_p - v_i$ steps, b_{i+1} will have less than N_p valid pages. The performance goal is to find the shortest such valid schedule; this may also be expressed as the write amplification factor (WAF), equal to $\frac{m + \sum_{n=1}^{|b_i|} v_i}{m}$. In the best case each block selected for cleaning would be empty, giving one cleaning for every N_p write operations and a WAF of 1.0.

Cleaning algorithms which have been proposed for page-mapped translation layers (or equivalent log-structured file systems) include Least-Recently Written (LRW [1]), Greedy, d-Choices [8], and Cost/Benefit [2]. In LRW the oldest block is selected, resulting in round-robin use of blocks. Greedy selects a block with minimal valid pages, and d-Choices approximates Greedy by sampling d pages and choosing the one with minimal valid pages. Finally, Cost/Benefit considers data *age* when selecting a block, under the assumption that new data is more likely to be invalidated in the near future. Greedy has been proven to be optimal for uniform random workloads [6], while Cost/Benefit and d-Choices [8] are observed to outperform Greedy for certain non-uniform workloads; no cases are known where LRW is superior.

To better understand the problem, we examine cleaning decisions more closely, and in particular compare them to decisions made by the Greedy algorithm. Figure 1 shows that all optimal solutions either end in a Greedy step (if the trace exactly fills the last block) or there is an equivalent optimal solution ending in a Greedy step. However as seen in Figure 2, in certain situations a non-greedy choice may result in better performance.

To describe this in more detail, we introduce several concepts. *Page lifetime* is the time until a page is invalidated by another write to the same LBA; the time of this write can be equivalently termed the *death* of the data in that page. Page lifetime is solely a property of the sequence of write

[1]This is equivalent to holding pages in memory while erasing the block undergoing cleaning; in practice those pages would be copied to a reserved free block, then the newly freed block would be erased and reserved for the next cleaning operation.

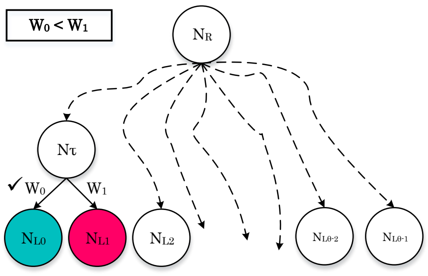

Fig. 1: An example cleaning tree showing no optimal cleaning is achieved unless by picking Greedy in at least one of the cleaning steps (the one at the end where $valid(N_{L0}) \le valid(N_{L1})$). If the next-to-last step of the optimal solution is N_τ, then the path ending in N_{L0} must be valid and optimal.

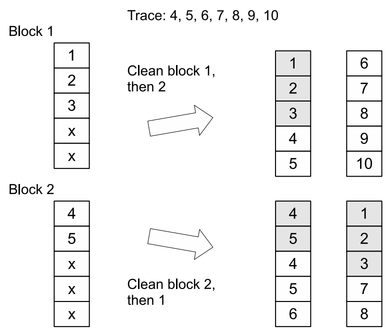

Fig. 2: An example with a short trace and two blocks where a non-greedy choice (block 1) results in a lower total write count than the greedy choice (block 2). pages marked with x are unused; gray pages are invalidated.

operations, as in the example in Figure 3 where we see data written to LBA 1 surviving through the duration of the entire trace, while data in LBA 2 is rapidly over-written. The same trace is shown in Figure 4 being written into two physical blocks.

When selecting a block for cleaning we can examine its *instantaneous write amplification*, the number of pages to be copied if selected for cleaning at that point in time, as well as the *ultimate future write amplification*, the lower limit on

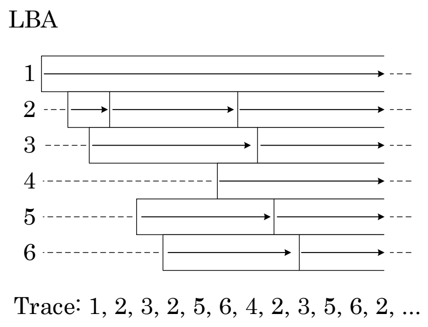

Fig. 3: Logical page lifetime. Each write to an LBA represents a separate page of data, which "lives" until it is invalidated by another write to the same LBA.

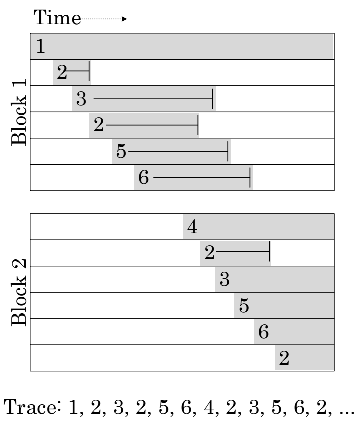

Fig. 4: Data lifetime for the trace in Figure 3 written into two physical blocks.

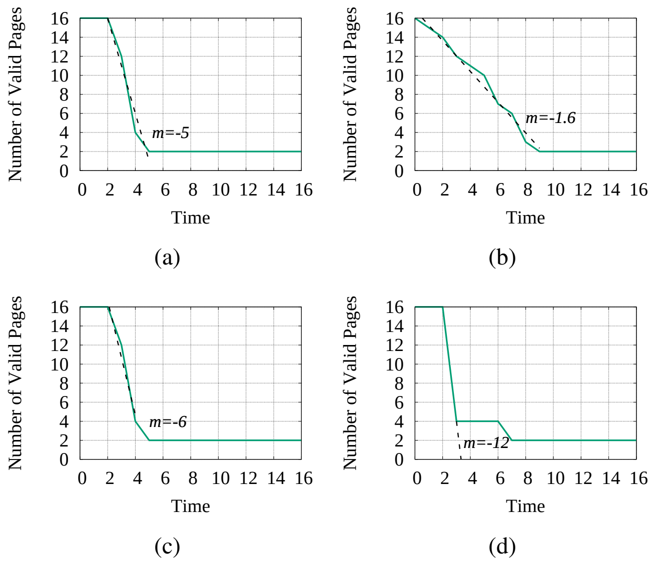

Fig. 5: Examples of death rate metric: block (a) is better than block (b) for all non-static pages as it drops from 16 to 2 valid pages in much shorter time (5-3=2 compared to 9-0=9). Similarly, block (d) is better than (c) for the first 12 non-static pages requiring only 3-2=1 time units compared to 4-2=2.

the number of pages to be copied if it were cleaned at some point in the future. (for many real workloads this is non-zero, as some writes represent *static* data which is never overwritten or erased, such as LBA 1 in Figures 3 and 4.)

III. Optimal Cleaning and its Approximation

Optimal cleaning may be formulated as a decision problem—i.e. whether a valid cleaning schedule of length l exists. For the case of a single write frontier this is clearly in NP, as we can use a cleaning schedule $(b_1, \ldots b_l)$ as a witness and verify it by running the translation layer over (a_i) using a cleaning schedule (b_i) to determine the validity of the schedule. It appears to be NP-Hard, due to the choice of > 1 block for cleaning at each of $O(m)$ different points during the trace; however no proof of this complexity is known to the authors.

We analyze this as a search problem. The search tree has a single root node representing the first cleaning decision, with one edge for each block which could be chosen; each of these paths reaches a node at the next cleaning decision *for that path*, and then splits again according to which block is selected. In other words, each node in the tree to be searched corresponds to the sequence of cleaning decisions leading up to the current choice. Leaf nodes in the graph are ones where the trace $\{a_i\}$ "runs out" or is finished, and an optimal cleaning schedule corresponds to a shortest path from the root to a leaf node. (note that at each decision point there is always at least one block with $v < \rho N_p$, defering the next cleaning event $\lceil 1-\rho \rceil N_p$ accesses in the future, so the length of this path is bounded by $\frac{m}{\lceil 1-\rho \rceil N_p}$.)

As with many search problems of this form, the tree to be searched is huge. For example, assume a 1 GB device comprising 2048 blocks of 128 pages each. For a trace with writes uniformly distributed over LBAs, at every cleaning step almost all the 2048 blocks have $v < N_p$ valid pages and are thus legal candidates for cleaning. Since the minimal cleaning schedule is at least $\frac{m}{N_p}$ long, a naive breadth-first search would explore $O\left(2048^{\frac{m}{N_p}}\right)$ paths. To reduce the complexity of this search we apply two heuristics for approximating the optimal solution: graph pruning and randomized (Monte Carlo) graph traversal, as described below.

A. Graph Pruning (Node Selection)

We prune the search tree using the following heuristics, with the goal of weeding out candidate paths at each step which have little or no possibility of leading to an optimal solution.

- Instantaneous write amplification, or the number of valid pages to be copied from the selected block. Although this is not the only criteria (see Figure 2) it is important, since the mean instantaneous write amplification is minimized by an optimal solution. This metric is calculated for a candidate block *before* cleaning.
- Ultimate future write amplification: the number of static pages in a block, and thus a lower bound on the write amplification of that block when selected for cleaning in the future. Note that this metric is calculated for a candidate block *after* cleaning—i.e. for a candidate with v valid pages, we calculate the number of static pages in the union of those pages and the next $N_p - v$ accesses in the trace.
- Death rate: A block with high rate in page "death" results in few wasted pages before its pages begin to die; after many of them die (over a short period) the block may be cleaned with very low write amplification. Figures 5a and 5b illustrate this metric, showing valid pages vs. time for two blocks which each start with 16 valid pages, and in the long run retain 2 valid static pages. Pages in the first block die within a shorter interval (3 time units) than pages in the second block (9 time units). Rather than measuring death rate directly, we use the slope of this line, i.e. pages

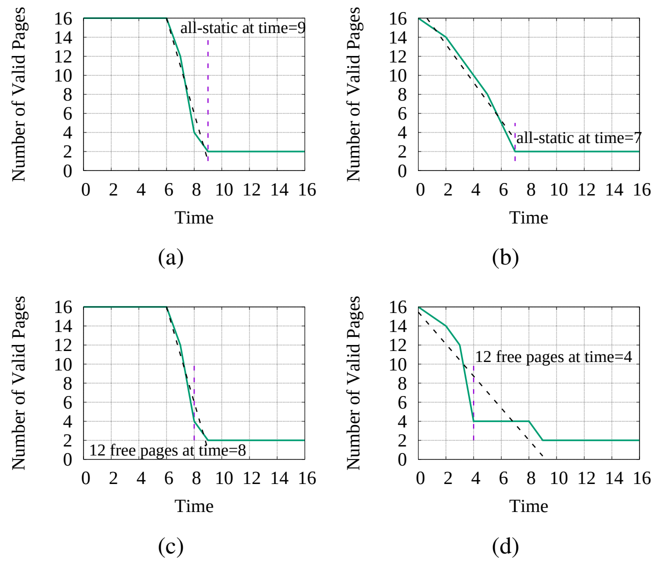

Fig. 6: Absolute death time metric – block (b) is better than (a) for all non-static pages and block (d) better than (c) for the first 12 non-static pages.

invalidated per unit time, as shown by the dashed lines in the figures.

For realistic values of device utilization, most blocks are unlikely to be left uncleaned long enough for all non-static pages to be invalidated. We therefore also consider the rate at which the first j pages in a block are invalidated; this comparison is shown in Figures 5c and 5d for $j = 12$. This metric is computed for the candidate block *after* cleaning.

- Absolute death time, or when space will be available in a block for future cleaning. For example, consider a block immediately after it has been filled, i.e. just before the next block is selected for cleaning, and assume that its pages will all die (nearly) simultaneously at a point far in the future. Before the block's pages begin to die it will be a poor candidate for cleaning, becoming a (very) good candidate afterwards. In Figures 6a and 6b we see blocks where all non-static pages are invalidated by time 9 and time 7, respectively.

  Again we also consider absolute death time for the first k pages, as seen in Figures 6c and 6d, where in the first case one must wait until time 8 for 12 free pages, while the same number of free pages may be collected from the latter block at time 4. This metric is again calculated for the candidate block *after* cleaning.

B. Pruned Search Algorithm

Algorithm 1 uses these metrics to prioritize cleaning selections based on their potential for being part of a high-quality cleaning schedule. At each cleaning step the set of candidate blocks is limited to:

- all candidate blocks with minimal instantaneous write amplification a

Algorithm 1 Finding potential candidate blocks

```
$invalid\_blocks = \varnothing$
$candidates = \varnothing$
$all\_static\_block = False$
for $b$ in $Blocks$ do
    if $n\_valid[b]! = Np$ then
        $invalid\_blocks.append(b)$
    end if
end for
for $(i = min(iwas); i <= max(iwas); i++)$ do // instantaneous write amplifications
    $blocks\_with\_the\_same\_iwa = \varnothing$
    if $i$ in $iwas$ then
        for $j$ in $invalid\_blocks$ do
            if $iwa[j] == i$ then
                if $j$ not in $candidates$ then
                    if $min(death\_time[j]) == \infty)$ then // all-static
                        if $all\_static\_block == False$ then
                            $candidates.add(j)$
                            $all\_static\_block = True$
                        end if
                    else
                        $blocks\_with\_the\_same\_iwa.append(j)$
                        $ufwa\_of\_blocks\_with\_the\_same\_iwa$
                        $.append(ufwa(j))$
                    end if
                end if
            end if
        end for
        for $j$ in $min(ufwa\_of\_blocks\_with\_the\_same\_was)$ : $max(fwa\_of\_blocks\_with\_the\_same\_was)$ do
            if $j$ in $ufwa\_of\_blocks\_with\_the\_same\_was$ then
                for $k$ in $blocks\_with\_the\_same\_was$ do
                    if $ufwa[k] == j$ then
                        $blocks\_with\_the\_same\_ufwa[j].append(k)$
                    end if
                end for
            end if
        end for
        for $ind$ in $blocks\_with\_the\_same\_ufwa$ do
            if $blocks\_with\_the\_same\_ufwa[ind]! = \varnothing$ then
                $add\_blocks\_with\_max\_death\_rate\_for\_the$
                $\_first\_k\_pages(ind)$
                $add\_block\_with\_earliest\_death\_times\_for\_k^{th}$
                $\_page(ind)$
            end if
        end for
    end if
end for
```

- any candidate block with instantaneous write amplification $a + i$ if it has lower ultimate write amplification, higher death rate, or earlier absolute death time *than all blocks with instantaneous write amplification* $< a + i$.

Instantaneous and ultimate future write amplification have straightforward scalar definitions; however death rate and absolute death time depend on k, the pages on which these metrics are compared. For each of the two measurements one could in fact calculate for each of the N_p possible values of k, resulting in a large vector comparison of dubious utility. Instead we calculate these timings for a small number of values of k, with that number determined by the sampling parameter ψ ranging from 1 to Np in Algorithms 2 and 3.

	Complete Graph (Optimal)		Pruned Graph (Near-optimal)		Greedy	
Trace	Traverses	Internal Writes	Traverses	Internal Writes	Traverses	Internal Writes
Uniform	250,740,915	6	170	7	1	10
Normal	9,624,761	2	45	2	1	3
Exponential	9,548,995	6	80	7	1	10
Gamma	10,851,636	15	429	15	1	23

TABLE I: Effects of pruning on search reduction and accuracy. "Internal writes" counts cleaning-related writes, i.e. write amplification.

Algorithm 2 Adding blocks with minimum absolute death times

```
1: for ($k = \psi; k <= Np; k = k + \psi$) do
2:     if $candidates$ != ∅ then
3:         $min\_for\_k^{th}\_page\_cand$=$min(death\_time[x][k]$ for $x$ in $candidates)$
4:     else
5:         $min\_for\_k^{th}\_page\_cand = \infty$
6:     end if
7:     $min\_for\_k^{th}\_page\_blk = min(death\_time[x][k]$ for $x$ in $blocks\_with\_the\_same\_ufwa[ind])$
8:     if $min\_for\_k^{th}\_page\_cand > min\_for\_k^{th}\_page\_blk$ then
9:         for $j$ in $blocks\_with\_the\_same\_ufwa[ind]$ do
10:            if $blocks\_with\_the\_same\_ufwa[ind][j]$ not in $candidates$ then
11:                if $min\_for\_k^{th}\_page\_blk == death\_time[j]$ then
12:                    $candidates.append(j)$
13:                    break
14:                end if
15:            end if
16:        end for
17:    end if
18: end for
```

Algorithm 3 Adding blocks with maximum death rate

```
1: for ($k = \psi; k <= Np; k = k + \psi$) do
2:     if $candidates$ != ∅ then
3:         $max\_death\_rate\_for\_first\_k\_pages\_cand = min(death\_time[x][k] - death\_time[x][1]$ for $x$ in $candidates)$
4:     else
5:         $max\_death\_rate\_for\_first\_k\_pages\_cand = \infty$
6:     end if
7:     $max\_death\_rate\_for\_first\_k\_pages\_blk = min(death\_time[x][k] - death\_time[x][1]$ for $x$ in $blocks\_with\_the\_same\_ufwa[ind])$
8:     if $max\_death\_rate\_for\_first\_k\_pages\_cand > max\_death\_rate\_for\_first\_k\_pages\_blk$ then
9:         for $j$ in $blocks\_with\_the\_same\_ufwa[ind]$ do
10:            if $blocks\_with\_the\_same\_ufwa[j]$ not in $candidates$ then
11:                if $max\_death\_rate\_for\_first\_k\_pages\_blk == death\_time[j]][k] - death\_time[j][1]$ then
12:                    $candidates.append(j)$
13:                    $break$
14:                end if
15:            end if
16:        end for
17:    end if
18: end for
```

C. Graph Traversal

Even though graph pruning greatly reduces the branching factor of the tree to be searched, for realistic I/O traces and device sizes the complexity of the resulting problem is still too great for traditional search algorithms such as depth-first-search (DFS), (with complexity $O(|V| + |E|)$). We instead employ Monte Carlo Tree Search (MCTS) [9] , a stochastic search algorithm. MCTS relies on random node sampling and expansion of the most promising nodes. It consists of four steps which are repeated over and over until a certain condition e.g. the allowed run time is met. The followings are the steps taken in each iteration of the algorithm:

- Selection: Starting from root node N_R, at every node an edge (a child node) is selected to maximize the value of a policy function (π) set to traverse the tree towards the most promising nodes. Upper Confidence Bound (UCB1) is a typical π policy which is defined as follows:

$$\overline{X}_j + \sqrt{\frac{2\ln n}{n_j}} \qquad (1)$$

  where $\overline{X}_j$ is the average number of wins (rewards) achieved by taking node j, n_j is the number of times node j has been visited and n is the total number of visits for all the nodes at that level.
- Expansion: Unless the selection phase traverses the graph from N_R all the way to one of its leaves e.g., N_{Li}, the process is continued by expanding the last selected node's children and choosing node N_E as one of its children to expand the tree.
- Simulation: The process continues with a random (or heuristics-based) playout from N_E to a leaf node N_{Li}.
- Back-propagation: The results of the simulation phase is back-propagated to update the value of nodes from N_E up to N_R.
  Figure 7 depicts these phases with an example.
  In the Simulation phase above we use Greedy cleaning to complete each traverse, which we find accelerating the convergence of the algorithm compared to other approaches (e.g. random).

IV. Evaluation

We implemented our framework in python using roughly 1000 lines of code supporting both optimal and near-optimal cleanings with DFS and MCTS as graph traversal options of choice. The implemented MCTS algorithm supports both greedy and random block selections for its simulation step and

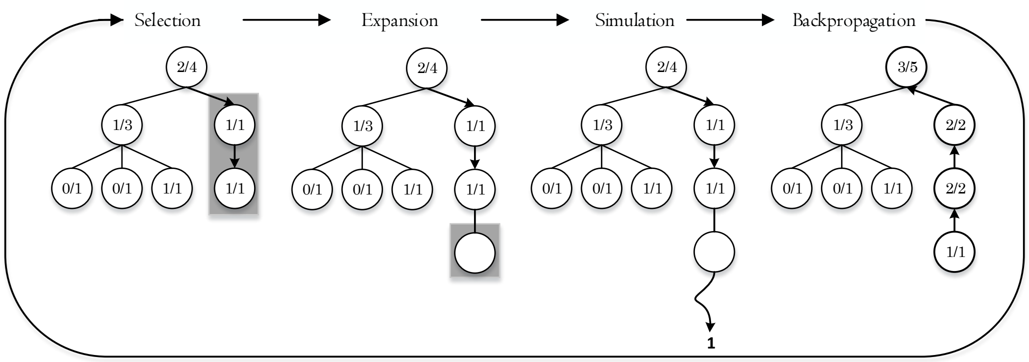

Fig. 7: Steps of Monte Carlo Tree Search (MCTS) where x/y in each node is the reward rate defined as the number of wins (x) over the number of traverses (y). The right-handed branch (highlighted) is selected, expanded, and simulated. Then the weights based on simulation results are back-propagated.

runs for the given number of traverses set by user rather than time.

We use 4 synthetic as well as 10 real-world block traces (from the Microsoft Research Cambridge set [10]) to 1) evaluate the impact of graph pruning and MCTS techniques for this application and 2) to examine the performance of approximate optimal cleaning on real world traces, as compared to a Greedy cleaning baseline.

A. Graph Pruning Effects

Although graph pruning at each cleaning step reduces the complexity of graph search, it raises the question of whether the optimal solution in fact traversed one of the pruned subgraphs, and if so, how much accuracy was lost due to the eventual selection of a sub-optimal path. To answer these questions we search for optimal cleaning schedules for several traces using depth-first search (a) with heuristic pruning and (b) without pruning, in effect performing an exhaustive search of all possible cleaning schedules.

Due to the complexity of exhaustive search we generate 4 synthetic traces of 46 writes each, over a device with 8 blocks and 32 pages total ($N_p = 4$). The footprint (i.e. number of distinct LBAs) of the traces ranged from 15 to 29 pages in each trace and writes in each trace follow a different data distribution i.e. Uniform, Normal, Exponential and Gamma. Results of these tests are shown in Table I, showing both the number of paths explored and the number of internal writes due to cleaning. Although only limited confidence may be drawn from these short traces, the differences between exhaustive search and pruned search were tiny at most (e.g. a write amplification of $\frac{46+7}{46}$ vs $\frac{46+6}{46}$) while Greedy required nearly twice as many internal writes as optimal in each case.

B. Monte Carlo Tree Search

The runtime of MCTS is not the question, as it performs a user-specified number of traversals and reports the best result; instead the question is how many traversals are needed for it to converge to an accurate answer, and how quickly that answer degrades with fewer runs. Using UCB1 and Greedy playout as described above, we measure how many MCTS runs are needed to replicate the results obtained by pruned DFS; results are shown in Table II. With no decrease in accuracy, the number of traverses is reduced by 25% and 79% for Uniform and Normal, and by a factor of 20 or more for Gamma and Exponential.

Trace	Traverses (%)	Accuracy (%)
Uniform	21	100
Normal	75.5	100
Exponential	2.5	100
Gamma	3	50
	5.6	100

TABLE II: MCTS results for short traces vs. DFS with graph pruning.

C. MCTS and Real Traces

We next evaluate MCTS-based approximate optimal cleaning for real-world traces on devices of realistic size. We simulate a device with 64K blocks of 128 4 KB pages and a spare factor of 2%; each simulation was initialized by prepending sequential writes to the entire LBA space to the trace being simulated. In Table III we see results for ten selected MSR traces. In each case we were able to find multiple paths with better performance than Greedy, with a maximum improvement of 4.68% (hm_0); the largest number of better-than-Greedy paths found was 17 (src2_0).

In Figure 8 we see MCTS performance vs. Greedy for 1 to 50,000 MCTS runs. In most cases the improvements due to increasing run count begin to plateau by 16,000 runs; in all cases the improvement for 16,000 runs is very close to that for 50,000 runs in Table III.

D. Near-optimal vs Greedy

Table IV shows the total number of cleanings needed by near-optimal and greedy paths for the MSR traces tested. The two algorithms choose very similar paths, with small

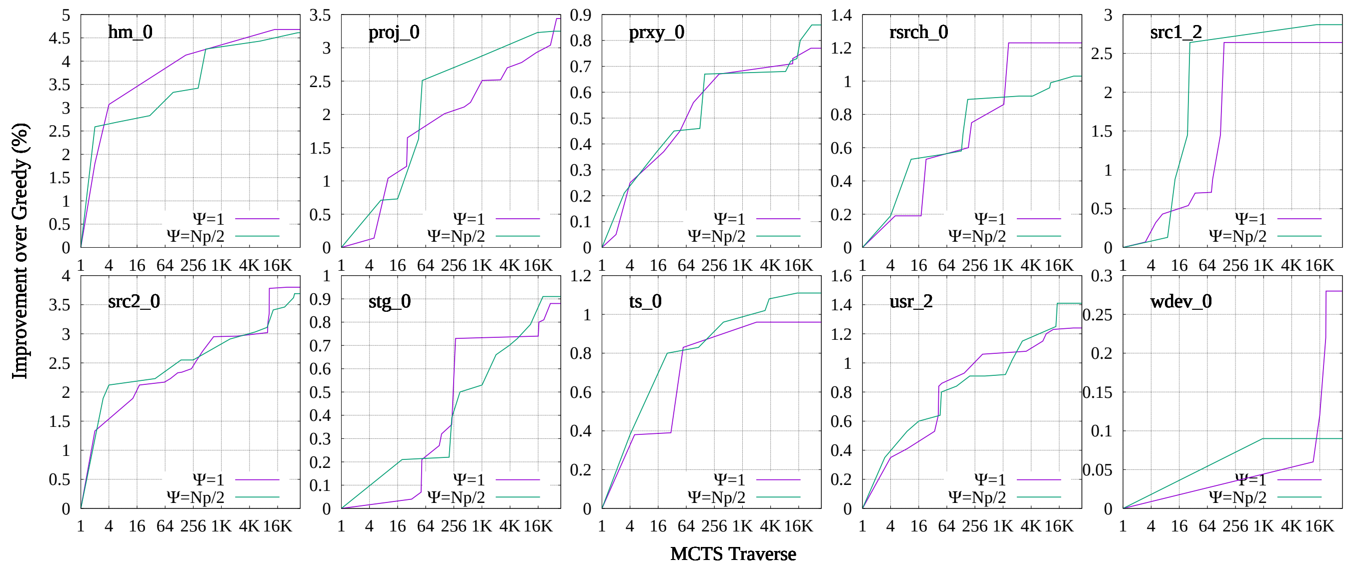

Fig. 8: Near-optimal improvement vs Greedy, 1 to 16,384 MCTS runs.

Trace	Number	Maximum Improvement (%)
hm_0	4	4.68
proj_0	15	3.44
prxy_0	9	0.77
rsrch_0	7	1.23
src1_2	9	2.64
src2_0	17	3.8
stg_0	11	0.88
ts_0	4	0.96
usr_0	13	1.24
wdev_0	4	0.28

TABLE III: The number of paths found better than Greedy for MSR traces along with the their maximum reduction in term of internal writes.

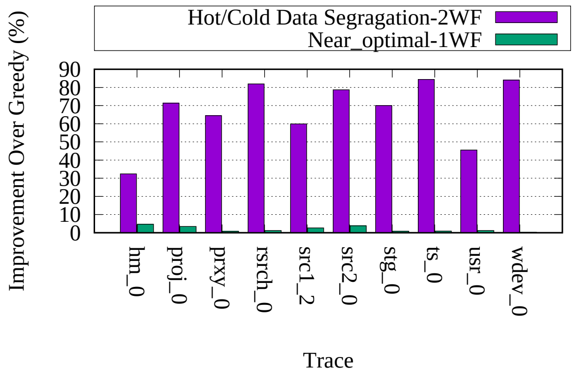

Fig. 9: Near-optimal single-write-frontier performance vs. extent-based hot/cold data segregation.

differences in cleaning performance. To compare the decisions in more detail we express each path as a sequence $n_1, \ldots n_i, \ldots n_m$ where n_i is the number of valid pages seen on the ith cleaning, truncate the longer trace (Greedy), and compute the correlation between these two series, as shown in Figure 10. We do this for each of the better-than-greedy paths discovered, showing the range of correlations across the traces in purple and correlation vs. the best cleaning sequence in green.

E. Device/Trace Size vs Quality of Results

To examine the effect of device and trace size on the quality of results, we prepare a series of traces spanning different LBA ranges, by filtering the MSR traces for a specific LBA range, and replay those traces for simulated device sizes of 128 to 32,768 blocks of 128 pages each, using a 10,000-run cutoff. The results are shown in Figure 11, showing number of blocks on the X axis vs cleaning improvement on the Y axis. We see that for small devices there are cleaning schedules which offer significant improvements over Greedy, while for larger devices the improvements are far more modest.

F. Dual Write Frontier Cleaning

For the demand-cleaned, single-write-frontier case examined above, the improvements over Greedy cleaning are quite small. We compare these with an implementation of dual-write-frontier cleaning with hot and cold data segregation [11], with identical device size, trace, and over-provisioning. Results may be seen in Figure 9; while MCTS-based near-optimal offline cleaning struggles to achieve a 4 or 5% improvement over Greedy, dual-write-frontier cleaning offers performance improvements of 70% or more for half of the traces, with the smallest gain being 30% for trace hm_0.

This appears to be the most significant lesson from this work. Given the restrictions of a single write frontier and demand cleaning, even an offline algorithm can do little better than online Greedy. Given two write frontiers to allow pages to be placed according to their predicted lifespan, a fairly simple online algorithm is able to improve performance by

Trace	hm_0	proj_0	prxy_0	rsrch_0	src1_2	src2_0	stg_0	ts_0	usr_0	wdev_0
Greedy	65,756	342,457	224,556	21,781	90,075	20,115	33,034	24,255	30,851	13,803
Near_optimal	64,807	340,810	224,102	21,774	90,057	20,070	33,016	24,242	30,790	13,803

TABLE IV: Total number of cleanings needed in Greedy and near-optimal paths with 50,000 MCTS runs.

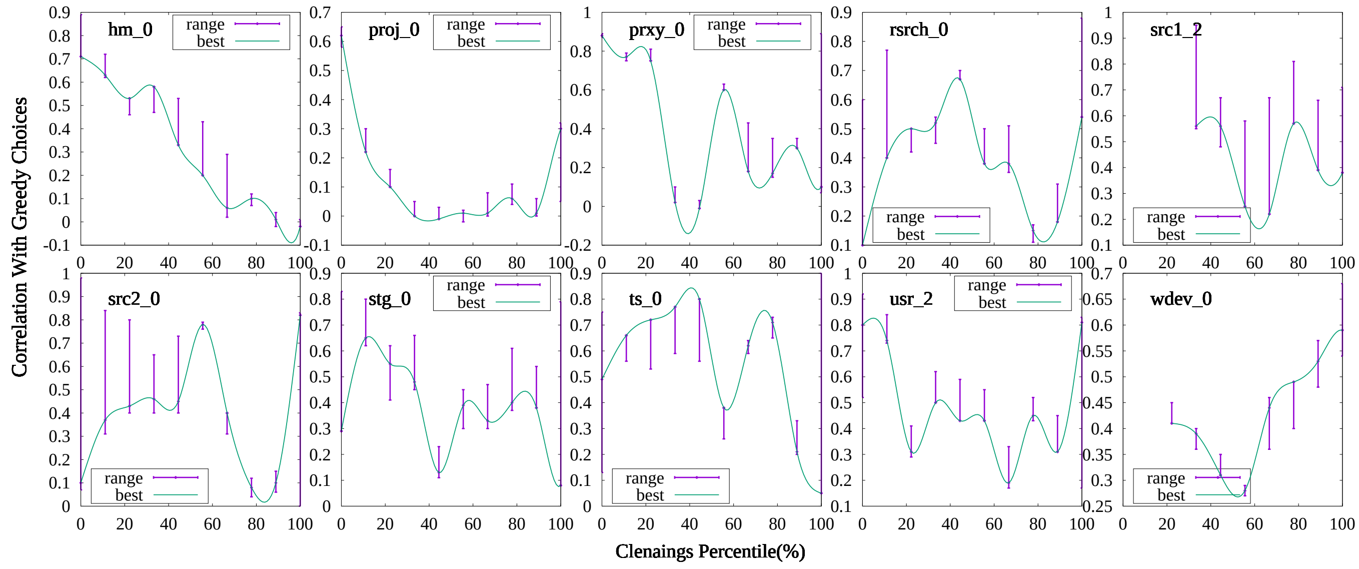

Fig. 10: Pearson correlation between the near-optimal and greedy cleaning paths. (missing sections for src1_2 and wdev_0 are undefined).

nearly a factor of two in many cases, in contrast to negligible improvements ($< 5\%$) for offline near-optimal cleaning.

V. Related Work

A number of works have addressed performance of online flash and LFS cleaning algorithms [4], [12], [5], [3], [8], [1], [6]. To date few works have examined the offline problem: Ben-Aroya and Toledo [13] give performance bounds for the offline wear leveling problem with one free block, and Cheng et al. [14] describe an approximation to offline optimal performance of a cache with flash-like write/erase limitations. In the area of caching Belady's result [7] is well-known, and various offline caching problems have been proven NP-hard: Albers et al. [15] and Brehob et al. [16] prove the NP-hardness of offline optimal replacement for non-standard caches, while Chrobak et al. [17] prove the strong NP-completeness of offline caching for variable-sized items. To date the authors are not aware of any work addressing offline optimal cleaning or its complexity.

VI. Conclusions

We present the first description of offline optimal solutions for FTL and LFS cleaning, focusing on the single-write-frontier demand-clean case. We describe an approximate solution to this problem, based on tree pruning and Monte Carlo Tree Search. Results are presented supporting the accuracy of this approximation, based on both toy problems—validated against exhaustive search—as well as longer runs. Finally we show results for real-world traces, comparing them to both Greedy cleaning for the same single-write-frontier case, as well as to an online dual write frontier solution based on hot/cold data identification.

Optimal cleaning is seen to offer modest improvements over Greedy, by an amount which is in fact negligible in comparison to the improvements possible with a fairly simple dual-write-frontier solution. From these results we draw the following conclusion:

> Efficient cleaning for real workloads is not a matter of deciding which blocks to *select* for cleaning, but rather of deciding where to *place* incoming data, and then collecting the free space generated by these decisions.

References

[1] P. Desnoyers, "Analytic models of ssd write performance," *Transactions on Storage*, vol. 10, no. 2, pp. 8:1–8:25, Mar. 2014.

[2] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *J. Syst. Softw.*, vol. 48, no. 3, pp. 213–231, Nov. 1999.

[3] X.-Y. Hu and R. Haas, "The Fundamental Limit of Flash Random Write Performance: Understanding, Analysis and Performance Modelling," IBM Research - Zurich, IBM Research Report RZ 3771, Mar. 2010.

[4] J. T. Robinson, "Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 4, pp. 29–32, Oct. 1996, aCM ID: 240803.

[5] X. Luojie and B. Kurkoski, "An improved analytic expression for write amplification in NAND flash," in *2012 International Conference on Computing, Networking and Communications (ICNC)*, 2012, pp. 497–501.

[6] Y. Yang, V. Misra, and D. Rubenstein, "On the optimality of greedy garbage collection for ssds," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 2, pp. 63–65, Sep. 2015.

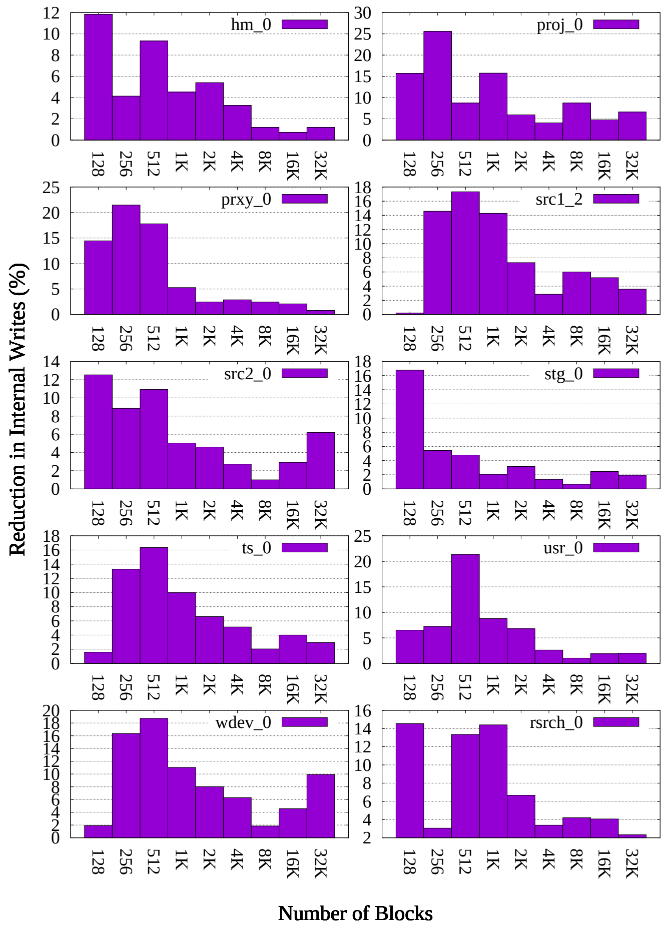

Fig. 11: The effect of device/trace size on the quality of paths found better than Greedy.

[7] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[8] B. Van Houdt, "A mean field model for a class of garbage collection algorithms in flash-based solid state drives," *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 191–202, Jun. 2013.

[9] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game AI," in *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, October 22-24, 2008, Stanford, California, USA*, 2008.

[10] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *Trans. Storage*, vol. 4, no. 3, pp. 10:1–10:23, Nov. 2008.

[11] M. Shafaei, P. Desnoyers, and J. Fitzpatrick, "Write Amplification Reduction in Flash-Based SSDs Through Extent-Based Temperature Identification," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. Denver, CO: USENIX Association, 2016.

[12] J. Menon and L. Stockmeyer, "An Age-Threshold Algorithm for Garbage Collection in Log-Structured Arrays and File Systems," *High Performance Computing Systems and Applications*, vol. 478, pp. 119–132, 1998.

[13] A. Ben-Aroya and S. Toledo, "Competitive Analysis of Flash-Memory Algorithms," in *Algorithms ESA 2006*, 2006, pp. 100–111.

[14] Y. Cheng, F. Douglis, P. Shilane, M. Trachtman, G. Wallace, P. Desnoyers, and K. Li, "Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 379–392.

[15] S. Albers, S. Arora, and S. Khanna, "Page Replacement for General Caching Problems," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '99. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999, pp. 31–40.

[16] M. Brehob, S. Wagner, E. Torng, and R. Enbody, "Optimal Replacement Is NP-Hardfor Nonstandard Caches," *IEEE Trans. Comput.*, vol. 53, no. 1, pp. 73–76, Jan. 2004.

[17] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu, "Caching Is HardEven in the Fault Model," *Algorithmica*, vol. 63, no. 4, pp. 781–794, Aug. 2012.