Research Repository UCD

Title	On Computational Models for Flash Memory Devices
Authors(s)	Ajwani, Deepak, Beckmann, Andreas, Jacob, Riko, et al.
Publication date	2009-08-20
Publication information	Ajwani, Deepak, Andreas Beckmann, Riko Jacob, and et al. "On Computational Models for Flash Memory Devices." Springer, August 20, 2009. https://doi.org/10.1007/978-3-642-02011-7_4.
Series	Lecture Notes in Computer Science (LCNS, volume 5526)
Publisher	Springer
Item record/more information	http://hdl.handle.net/10197/9903
Publisher's statement	The final publication is available at www.springerlink.com.
Publisher's version (DOI)	10.1007/978-3-642-02011-7_4

Downloaded 2026-05-01 23:34:00

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)

© Some rights reserved. For more information

On Computational Models for Flash Memory Devices *

Deepak Ajwani[1], Andreas Beckmann[2], Riko Jacob[3], Ulrich Meyer[2], and Gabriel Moruz[2]

[1] Department of Computer Science, Aarhus University, Denmark.
[2] Institut für Informatik, Goethe-Universität Frankfurt am Main, Germany.
[3] Computer Science Department, TU München, Germany.

Abstract. Flash memory-based solid-state disks are fast becoming the dominant form of end-user storage devices, partly even replacing the traditional hard-disks. Existing two-level memory hierarchy models fail to realize the full potential of flash-based storage devices. We propose two new computation models, the general flash model and the unit-cost model, for memory hierarchies involving these devices. Our models are simple enough for meaningful algorithm design and analysis. In particular, we show that a broad range of existing external-memory algorithms and data structures based on the merging paradigm can be adapted efficiently into the unit-cost model. Our experiments show that the theoretical analysis of algorithms on our models corresponds to the empirical behavior of algorithms when using solid-state disks as external memory.

1 Introduction

In many practical applications, one needs to compute on data that exceeds the capacity of the main memory of the available computing-device. This happens in a variety of settings, ranging from small devices, such as PDAs, to high-performance servers and large clusters. In such cases, the cost of data transfers between disk and the main memory often proves to be a critical bottleneck in practice, since a single disk transfer may be as time-costly as millions of CPU operations. To capture the effect that memory transfers have on the running time of algorithms, several computational models have been proposed over the past decades. One of the most successful of these models is the *I/O-model*.

I/O-model. The I/O-model, as defined in [1], is a two-level memory model. It consists of a CPU, a fast internal memory of size M and a slow external-memory of infinite size. The CPU can access only data stored in the internal memory, and data transfers between the two memories are performed in chunks of B consecutive data items. The I/O-complexity of an algorithm is given by the number of memory transfers, or I/Os, performed. Many problems have been studied in this model and efficient algorithms have been proposed. For comprehensive overviews we refer the interested reader to [2, 3].

* Partially supported by the DFG grant ME 3250/1-1, and by MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

Flash memories. In the recent years, a new trend has emerged in the storage device technology – that of solid-state disks based on flash memory. Flash memories are non-volatile, reprogrammable memories. Flash memory devices are lighter, more shock resistant and consume less power. Moreover, since random accesses are faster on solid-state disks compared to traditional mechanical hard-disks, flash memory is fast becoming the dominant form of end-user storage in mobile computing. Many recent notebook and netbook models have already replaced traditional mechanical hard-disks by flash memory disks. Market research company In-Stat predicted in July 2006 that 50% of all mobile computers would use flash (instead of hard-disks) by 2013.

Flash memory devices typically consist of an array of memory cells that are grouped into *pages* of consecutive cells, where a fixed amount of consecutive pages form a *block*. Reading a bit is performed by reading the whole page containing the given bit. When writing, we distinguish between changing bits from 1 to 0 and from 0 to 1. To change a bit from 0 to 1, the device first "erases" the entire block containing the given bit, i. e. all the bits in the block are set to 1. However, changing a bit from 1 to 0 is done by writing only the page containing it, and each page can be programmed only a small number of times before it must be erased again. Reading and writing pages is relatively fast, whereas erasing a block is significantly slower. Each block can sustain only a limited number of erasures. To prevent blocks from wearing prematurely, flash devices usually have a built-in micro-controller that dynamically maps the logical block addresses to physical addresses to even out the erase operations sustained by the blocks.

Related work. Recently, there has been an increased interest in using flash memories to improve the performance of computer systems. This includes the experimental use of flash memories in database systems [4–6], using flash memories as caches in hard-disks (e. g. Seagate's Momentus 5400 PSD hybrid drives), Windows Vista's ReadyBoost, i. e. using USB flash memories as a cache, or integrating flash memories into motherboards or I/O-buses, e. g. Intel's Turbo Memory technology [7].

Most previous algorithmic work on flash memories deals with wear leveling, i. e. block-mapping and flash-targeted file systems (see [8] for a comprehensive survey). There exists very little work on algorithms designed to exploit the characteristics of flash memories. Wu et al. [9, 10] proposed flash-aware implementations of B-trees and R-trees without file system support by explicitly handling block-mapping. More recently, efficient dictionaries on flash disks have been engineered [11]. Other works include the use of flash memories for model checking [12] or route planning on mobile devices [13, 14].

Our contributions. Owing to the lack of good computation models to help exploiting the particular characteristics of flash devices, there is no firm theoretical foundation for comparing algorithms. In this paper, we propose two computational models for flash devices that exploit their constructive characteristics – the general flash model and the unit-cost flash model. These models can be used as a basis for a theoretical comparison between different algorithms on flash

memory devices. While the general flash model is very generic and is especially suitable for studying lower bounds, the unit-cost flash model is appealing for the design and analysis of algorithms. In particular, we show that a large number of external-memory algorithms can be easily adapted to give efficient algorithms in the unit-cost flash model. Interestingly, we observe that external-memory algorithms based on the merging paradigm are easy to adapt in the unit-cost flash model, while this is not true for algorithms based on the distribution paradigm. We conduct experiments on several algorithms exhibiting various I/O-access patterns, i.e. random and sequential reads, as well as random and sequential writes. Our experiments confirm that the analysis of algorithms on our models (particularly, the unit-cost flash model) predicts the observed running-times much better than the I/O model. Our experiments also show that the adaptations of these algorithms improve their running-times on solid-state disks.

2 Models for flash memory

In this section we propose and discuss models for flash memories. We first discuss the practical behavior of flash memories. We then propose two models of computation, a *general flash model* and a *unit-cost flash model.* They are both based on the I/O-model, but use a different block size for reading than for writing.

Flash memory behavior. Due to constructive characteristics, in practice flash memories have a significantly different behavior compared to hard disks [15–17]. In Figure 1 we give empirical results showing the dependence of throughput on the block size when performing random reads and writes, as well as sequential reads and writes. We used two different disks: a 64 GB Hama SSD drive and a Seagate Barracuda 7200 rpm 500 GB hard-drive. The main difference concerns the relative performance of random reads and random writes. For hard-disks random reads and random writes provide similar throughput, whereas for the SSD drive random reads provide significantly more throughput than random writes, especially for small block sizes. Furthermore, the throughput of random accesses converges to the throughput of the corresponding sequential accesses at different block sizes, implying different block sizes for reading and writing. Also, the throughput provided by sequential reads is nearly the same as the throughput provided by sequential writes for most flash devices [15].

The key characteristic of the flash devices that we model is the different block sizes for reading and writing. For the general flash model we also consider different throughput for reading and writing. To keep our computation models simple enough for algorithm design, we abstract away the other flash-memory characteristics, such as effects of misalignment, limited endurance etc.

General flash model. The general model for flash memory devices is similar to the I/O model, with the exception that read and write block sizes are different and that they incur different costs. The general flash model assumes a two-level memory hierarchy, with a fast internal memory of size M and a slow external

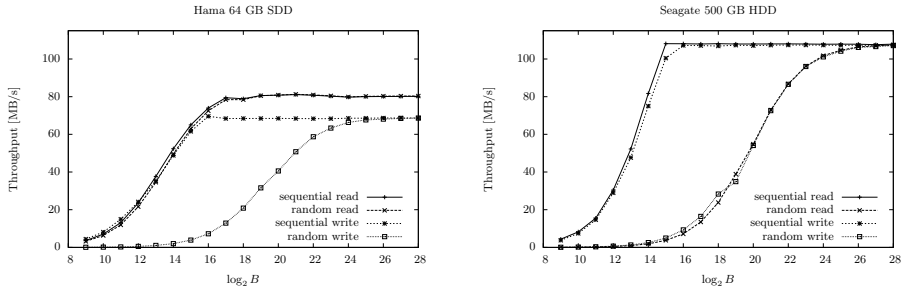

Fig. 1. Performance summary of solid-state disks (left) vs. hard disks (right). The x-axis shows the block size (in bytes), in logarithmic scale.

flash memory of infinite size. The input and output data reside on the external flash memory, and computation can only be done on data residing in the internal memory. Read and write I/Os from and to the flash memory occur in blocks of consecutive data of sizes B_r and B_w respectively. The complexity of algorithms is $x + c \cdot y$, where x and y are the number of read and write I/Os respectively, and c is a penalty factor for writing. Similarly to the I/O-model, the parameters M, B_r, B_w, and c are known to the algorithms. Typically, we assume $B_r \leq B_w < M \ll N$ and $c \geq 1$. We note that the I/O-model is a particular case of this general model, when $B_r = B_w = B$ and $c = 1$.

Unit-cost flash model. The fact that in the general flash model c may take arbitrary values implies arbitrary relative costs between read and write I/Os. This complicates the reuse of existing external-memory algorithms and algorithmic techniques. In [15] it was shown that for most flash devices the throughput provided by reads and writes is nearly the same, assuming proper block sizes, i.e. B_r and B_w are set so that the maximum throughput is achieved on random I/Os. This means that, in spite of different read and write block sizes, the access time per element is nearly the same. The unit-cost flash model is the general flash model augmented with the assumption of an equal access time per element for reading and writing. This simplifies the model considerably, since it becomes significantly easier to adapt external-memory results. For the sake of clarity, the cost of an algorithm performing x read I/Os and y write I/Os is given by $xB_r + yB_w$, where B_r and B_w denote the read and write block sizes respectively. Essentially, the cost of an algorithm in this model is given by the total amount of items transferred between the flash-disk and the internal memory.

For both models, we note that "items transfered" refers to all the B_r (B_w) elements moved during a read (write) I/O and not just the useful elements transfered. Also, our models can be adapted to obtain hardware-oblivious models.

Relating unit-cost models to external-memory models. We turn to exploring the relation between the unit-cost models and the external-memory models.

Lemma 1. *Any algorithm designed in the unit-cost flash model which transfers $f(N, M, B_r, B_w)$ items can be simulated by an external-memory algorithm with $B = B_r$ which performs $f(N, M, B_r, B_w)/B_r$ I/Os.*

Consider some algorithm A in the unit-cost flash model, which transfers $f(N, M, B_r, B_w)$ items. Denote by $f_r(N, M, B_r, B_w)$ the total cost for read I/Os and let $f_w(N, M, B_r, B_w)$ be the total cost for write I/Os. The algorithm is executed as an external-memory algorithm with a block size $B = B_r$ as follows. Read operations are done in blocks of size B_r and therefore the reads incur $f_r(N, M, B_r, B_w)/B_r$ I/Os, whereas writes are done in blocks of size B_w which implies that each write incurs B_w/B_r I/Os. We obtain that all the writes take $(f_w(N, M, B_r, B_w)/B_w) \cdot (B_w/B_r) = f_w(N, M, B_r, B_w)/B_r$ I/Os.

The simulation in Lemma 1 provides an efficient mechanism for obtaining lower bounds in the unit-cost flash model, as stated in Lemma 2.

Lemma 2. *A problem that requires $\Omega(L(N, M, B))$ I/Os in the I/O-model requires $\Omega(B_r \cdot L(N, M, B_r))$ items transferred in the unit-cost flash model.*

3 Algorithms for the unit-cost flash model

Typical external-memory algorithms manipulate buffers using various operations, such as merging and distributing. Given that in the unit-cost flash model the block sizes for reads and writes are different, algorithms can merge $\mathcal{O}(M/B_r)$-ways and distribute $\mathcal{O}(M/B_w)$-ways. Since $M/B_r > M/B_w$, merging is preferred to distributing because more buffers can be manipulated simultaneously. A surprisingly large body of merging-based external-memory algorithms (and data structures) can be easily adapted to get efficient and sometimes even optimal algorithms (and data structures) in the unit-cost flash model, sometimes by simply setting the block size B to B_r. In this section we show a few typical examples of how simple changes lead to efficient algorithms in the unit-cost flash model.

3.1 Sorting

Sorting N records in the I/O-model requires $\Omega(N/B \log_{M/B} N/B)$ I/Os [1]. Using Lemma 2, we obtain that sorting N elements needs $\Omega(N \log_{M/B_r} N/B_r)$ items to be transfered in the unit-cost flash model.

To sort in the unit-cost flash model, we use multi-way mergesort, which is optimal in the I/O-model, and we show that it achieves optimality also in the unit-cost flash model. The algorithm splits the input into $\Theta(M/B)$ subsequences, recursively sorts them, and in the end merges the (sorted) subsequences. The I/O-complexity is $\Theta(N/B \log_{M/B} N/B)$ I/Os. For the unit-cost flash model, different costs are achieved depending on the number of subsequences the input is split into. Splitting the input in $\Theta(M/B_w)$ subsequences yields an algorithm that transfers $\mathcal{O}(N \log_{M/B_w} N/B_w)$ items, whereas splitting $\Theta(M/B_r)$-ways yields the optimal $\Theta(N \log_{M/B_r} N/B_r)$ cost.

Lemma 3. *Sorting N elements can be done by transferring $\Theta(N \log_{M/B_r} N/B_r)$ items in the unit-cost flash model.*

3.2 Data structures

In this section we give brief descriptions of efficient implementations for search trees and priority queues in the unit-cost flash model.

Search trees. For searching, we show how to adapt the B-trees used in the I/O-model to obtain an efficient implementation in the unit-cost flash model. We employ a two-level structure. The primary data structure is a B-tree with a fan-out of $\Theta(B_w)$; each node of the primary structure is stored also as a B-tree, but with nodes having a fan-out of $\Theta(B_r)$. Searches and updates transfer $\mathcal{O}(B_r \log_{B_r} N)$ items.

Priority queues. Several optimal external-memory priority queues have been proposed [18–21]. Each of them takes amortized $\mathcal{O}(1/B \log_{M/B} N/B)$ I/Os per operation. However, only the cache-oblivious priority queue in [20] translates directly into an optimal priority queue in unit-cost flash model, taking amortized $\mathcal{O}(\log_{M/B_r} N/B_r)$ items transfered per operation. This is because it only merges buffers, whereas the other priority queues also employ distribution and achieve only amortized $\mathcal{O}(\log_{M/B_w} N/B_w)$ transfered items. We note that priority queues are the core of time forward processing, a technique widely employed to achieve efficient external memory graph algorithms.

3.3 BFS

For BFS on undirected graphs $G(V, E)$ in the unit-cost flash model, we focus on the randomized external-memory algorithm by Mehlhorn and Meyer [22]. For ease of exposition, we restrict ourselves to sparse graphs, i.e. $|E| = \mathcal{O}(|V|)$. The algorithm starts with a preprocessing phase, in which the input graph is rearranged on disk. This is done by building $|V|/\mu$ disjoint clusters of small diameter ($\mathcal{O}(\mu \cdot \log |V|)$ with high probability (whp.)) that are laid contiguously on disk. In the BFS phase, the algorithm exploits the fact that in an undirected graph, the edges from a node in BFS level t lead to nodes in BFS levels $t-1$, t or $t+1$ only. Thus, in order to compute the nodes in BFS level $t+1$, the algorithm collects all neighbors of nodes in level t, removes duplicates and removes the nodes visited in levels $t-1$ and t. For collecting the neighbors of nodes efficiently, the algorithm spends one random read I/O (and possibly, some further sequential read accesses depending on the cluster size) for loading a whole cluster as soon as a first node of it is visited and then keeps the cluster data in some efficiently accessible data structure (hot pool) until all nodes in the cluster are visited. The preprocessing and BFS phases together require $\mathcal{O}(\text{scan}(|V|) \cdot \mu \cdot \log |V| + \text{sort}(|V|))$ I/Os (reading and writing) whp. plus another $\mathcal{O}(|V|/\mu)$ read-I/Os. In the I/O-model, choosing $\mu = \Theta\left(\sqrt{B/\log |V|}\right)$ implies a total cost of $\mathcal{O}(|V| \cdot \sqrt{\log |V|/B} + \text{sort}(|V|))$ I/Os whp. In the unit-cost flash model this means a total cost of $\mathcal{O}(\ |V| \cdot \mu \cdot \log |V| \ + \ |V| \cdot \log_{M/B_r} \frac{|V|}{B_r} + |V| \cdot B_r/\mu\ )$, which is minimized by choosing $\mu = \Theta\left(\sqrt{\frac{B_r}{\log |V|}}\right)$.

Lemma 4. *Computing undirected BFS on sparse graphs* ($|E| = \mathcal{O}(|V|)$) *in the unit-cost flash model requires* $\mathcal{O}(|V| \cdot \sqrt{B_r \cdot \log |V|} + |V| \cdot \log_{M/B_r}(|V|/B_r))$ *item transfers.*

4 Experimental results

The main goal of our experimental study is to verify the suitability of the proposed unit-cost flash model for predicting the running-time of algorithms using SSD as an external-memory. We want to check how well the behavior of the algorithms on SSDs correspond to their theoretical analysis on the unit-cost flash model. In particular, we look at the improvements from the adaptation process as predicted theoretically on the unit-cost flash model and ascertain if these gains are actually observed in practice. We consider three algorithms which present various I/O-patterns and have very different complexities in the I/O model. First, we consider sorting, which takes $\text{sort}(N) = \mathcal{O}(N/B \log_{M/B} N/B)$ I/Os and performs mainly sequential I/Os. We then move to BFS, which requires $\mathcal{O}(|V| \cdot \sqrt{\log |V|/B} + \text{sort}(|V|))$ I/Os whp. for sparse graphs and causes both sequential and random reads, but no random writes. Finally, the classical DFS implementation performs $\mathcal{O}(|V|)$ I/Os on sparse graphs and does a large number of random reads and writes. We observe the performance of these algorithms when using a SSD as external-memory.

Experimental setup. For algorithms and data structures designed in the I/O-model we use implementations already existent in the STXXL library [23] wherever possible. We show results where the size of blocks in which data is transferred between the internal memory and the flash device is set to both the read and write block sizes of the device. According to our flash models, algorithms read blocks of size B_r and write blocks of size B_w. To comply with this requirement, we implement a translation layer similar to Easy Computing Company's MFT (Managed Flash Technology) [24]. The translation layer prevents random writes of blocks of size B_r by buffering B_r-sized blocks into blocks of size B_w that provide optimal throughput when written to the disk. When using the translation layer, an algorithm reads and writes pages of size B_r. Oblivious to the algorithm, the translation layer logically groups B_w/B_r pages into a block of size B_w, which is written to the flash disk. To do so, $O(1)$ B_w-sized buffers are reserved in the memory, so that when one such buffer gets full it is immediately written to the flash disk. To keep track of the data used, this layer maintains a mapping of the logical addresses of the pages viewed by the algorithm to their actual address on the flash disk. Since this mapping occupies little space and is used only to manage temporary data, the translation layer is stored in main memory throughout the execution of the algorithm. Additionally, the translation layer is responsible for keeping track of the free pages and blocks.

Due to its simplicity and generality, we view the translation layer as a generic easy-to-implement adaptation of I/O algorithms to algorithms in the unit-cost flash model. However, we note that there exist cases where the translation layer

can not be employed, e.g. extremely large inputs when the translation layer may no longer fit into the main memory.

Our experiments were conducted on a standard Linux machine, with an Intel Core 2 Quad 2.4 GHz CPU, 8 GB RAM out of which algorithms are restricted to use only 512 MB, and a 64 GB HAMA flash disk. The smallest block sizes where the disk reaches optimal performance for random reads and random writes are 128 KB and 16 MB respectively, see e. g. Figure 1, and consequently we set B_r and B_w to these values. The code was compiled using GCC version 4.3.

Sorting. For sorting we consider the STXXL implementation, which is based on (cache-aware) multi-way mergesort. The results in Table 1 show that when the block size is set to B_w, the running time is larger than when the block size equals B_r, and the volume of data read and written by the algorithm is larger as well. This behavior is easily explained theoretically by the larger number of recursion levels in the former case, noticeable by the relative ratio between the read/write volumes and the input volume. Also, when using the translation layer we obtain very similar results to when setting the block size to B_r. This behavior is also in line with the theoretical analysis in unit-cost flash model, since the algorithm essentially writes data sequentially, and in this case writing blocks of size B_r yields the same throughput as when writing blocks of size B_w (when using the translation layer). Such a behavior would be inexplicable in the I/O-model, which assumes reads and writes in equally sized blocks for reading and writing. We note that, due to the limited size of the flash disk, we could not sort larger sequences.

input		TL			B_r			B_w		
$\log_2 N$	volume [GB]	RDV [GB]	WRV [GB]	RT [s]	RDV [GB]	WRV [GB]	RT [s]	RDV [GB]	WRV [GB]	RT [s]
25	0.12	0.20	0.25	9.10	0.25	0.25	9.35	0.25	0.25	9.13
26	0.25	0.49	0.50	16.73	0.50	0.50	16.72	0.50	0.50	17.10
27	0.50	0.99	1.00	32.25	1.00	1.00	31.29	1.00	1.00	33.58
28	1.00	1.99	2.00	62.35	2.00	2.00	60.96	3.00	3.00	93.46
29	2.00	3.99	4.00	120.82	4.00	4.00	118.84	6.00	6.00	192.98
30	4.00	8.00	8.00	240.24	8.00	8.00	238.74	12.00	12.00	387.16
31	8.00	16.00	16.00	478.46	16.00	16.00	475.11	32.00	32.00	1002.95
32	16.00	32.00	32.00	946.88	32.00	32.00	950.04	64.00	64.00	2029.41

Table 1. The read volume (RDV), write volume (WRV), and the running time (RT) for sorting N random integers (taking the specified volume) when using the translation layer (TL), setting the block size to B_r and to B_w respectively. RDV and WRV are measured in GB, and RT is measured in seconds.

BFS. We perform experiments on square grid graphs as they have proven to be a difficult graph class [25] for the external-memory BFS algorithm. As shown in Table 2, using the translation layer yields only a small benefit compared to the

$\log_2 \lvert V \rvert$	TL			B_r			B_w		
	pp	bfs	Σ	pp	bfs	Σ	pp	bfs	Σ
READ VOLUME [GB]									
20	0.194	0.000	0.194	0.670	1.924	2.594	0.406	0.094	0.500
22	2.423	5.968	8.391	2.709	8.080	10.789	1.500	0.188	1.688
24	26.943	60.406	87.350	27.187	61.660	88.848	91.922	457.750	549.672
26	108.953	316.341	425.294	109.726	320.881	430.607	364.578	2621.047	2985.625
WRITE VOLUME [GB]									
20	0.594	0.000	0.594	0.560	0.009	0.569	0.250	0.172	0.422
22	2.281	0.094	2.375	2.271	0.104	2.375	1.016	0.234	1.250
24	9.344	1.656	11.000	9.251	1.654	10.905	22.734	0.812	23.547
26	36.750	5.531	42.281	36.783	5.531	42.313	89.938	1.203	91.141
RUNNING TIME [s]									
20	21.5	744.5	766.0	31.5	768.4	799.9	40.5	381.4	421.9
22	95.0	1668.4	1763.4	100.0	1697.0	1797.0	76.2	1126.0	1202.2
24	609.8	4581.2	5191.0	632.9	4570.4	5203.3	1738.2	9184.6	10922.8
26	2426.8	15755.4	18182.2	2524.2	15778.9	18303.1	6824.8	43329.1	50153.9

Table 2. Read/write volumes (in GB) and running times (in seconds) for external-memory BFS with randomized preprocessing on square grid graphs, separated into preprocessing phase (pp) and BFS phase, using block sizes B_r, B_w and the translation layer (TL).

read block size. This is explained by the fact that the algorithm performs no random writes, while random and sequential reads are not affected by the layer.

For preprocessing, using a smaller block size, and consequently a smaller μ, results in smaller running time, since the computed clusters tend to contain fewer nodes and have a smaller diameter. Comparing the preprocessing times for B_r and B_w on the square grid graph in Table 2 confirms this, as preprocessing using B_w takes up to three times as long as when B_r is used.

For the BFS phase, choosing a larger block size reduces the number of random I/Os needed to load clusters, but at the same time potentially increases the size of the hot pool because clusters with bigger diameter tend to stay longer in the pool. This affects the performance adversely if the hot pool no longer fits in internal memory as can be seen in Table 2 for $|V| \geq 2^{24}$. At that point the algorithm using B_w is outperformed by the one using B_r.

DFS. For DFS, we use a straightforward non-recursive implementation of the text-book RAM algorithm. The algorithm explores the graph by visiting for each node the first not yet visited neighbor, and to do so we use two data structures: a vector to mark the nodes visited and a stack to store the nodes for which not all the neighbors have been visited. The key particularity of this algorithm is that it performs extensive random reads to access many adjacency lists, as well as extensive random writes to mark the nodes. For a graph $G = (V, E)$ the unit-cost of the algorithm is given by $\mathcal{O}(|E| \cdot B_r + |V| \cdot B_w)$, since there are $|E|$ read accesses to the adjacency lists and $|V|$ write accesses to mark the vertices visited. The costs for accessing the stack are much smaller since both reads and writes can be buffered. We note that when transferring data in chunks of size

B_r the cost of the algorithm remains $\mathcal{O}(|E| \cdot B_r + |V| \cdot B_w)$, but when the block size is set to B_w the cost increases to $\mathcal{O}(|E| \cdot B_w + |V| \cdot B_w)$.

We conduct experiments which show the running time of DFS when transferring chunks of B_r and B_w consecutive data between the memory and the flash disk, as well as on using the translation layer. Due to extensive running times, we restrict to square grid graphs. We noted that for all input sizes using the translation layer yields better running times than when doing I/Os in blocks of size B_r, which is due to writing many blocks of size B_r at random locations. When the graph fits into the main memory the algorithm is extremely fast. For $|V| \leq 2^{20}$, the running times were below two seconds. However, when the graph no longer fits into the main memory, the running times and the I/O-traffic increase significantly.

For $|V| = 2^{22}$, the running times were of 4 180, 4 318, and 610 000 seconds for the translation layer, B_r, and B_w block sizes respectively. The huge running time for the B_w block size is explained by the huge volume of read data, of about 46 TB, compared to 336 GB read when using B_r-sized blocks and 311 GB when using the translation layer. The volume ratio between B_w and B_r approximately matches $\frac{B_w}{B_r} = 128$. However, the volume of data written was significantly low (less that 300 MB in each experiment). This is due to vector marking the visited nodes completely residing in memory.

Therefore we used another approach and stored the visited information with each node, effectively scattering the bits over a larger range of external memory. Internal memory was further restricted to cache at most half of an external memory data structure. Comparable experiments with block size B_w are not possible in these settings because the internal memory cannot store a required minimal amount of blocks. For $|V| = 2^{21}$ the DFS using the translation layer took 6 064 seconds reading 250 GB and writing 146 GB of data. Using block size B_r instead, the running time increased to 11 352 seconds and read volume of 421 GB, while write volume was 145 GB. The translation layer could serve a fraction of the read requests directly from its write buffers explaining the increase in read volume. While the written volume and write throughput rate were nearly unchanged (145 GB, 77-80 MB/s), the read throughput dropped from 69 MB/s to 46 MB/s. The subobptimal block size used for writing obviously triggers reorganization operations in the flash device that block subsequent operations (reads in our case). This accounts for the major part of the additional running time showing a clear benefit for the translation layer bundling these small random write requests.

5 Conclusions and future research

We proposed two models that capture the particularities of the flash memory storage devices, the general flash model and the unit-cost flash model. We show that existing external-memory algorithms and data structures, based on the merging paradigm, can be easily translated into efficient algorithms in the unit-cost flash model. Relevant examples include sorting, search trees, priority queues,

and undirected BFS. We conduct experiments that the unit-cost flash model predicts correctly the running times of several algorithms that present various I/O-patterns.

For the general flash model, an interesting future direction concerns obtaining lower bounds for fundamental problems, such as sorting or graph traversals, even for extreme cases when we set the penalty factor c to a very large value that allows the algorithm to write only the output. Future investigations in this model include engineering fast algorithms for basic problems, such as sorting.

For the unit-cost flash model, possible topics for future research include identifying problems for which the best external memory upper bounds cannot be matched in the unit-cost flash model.

Promising directions also include introducing relevant computational models that capture other characteristics of the flash devices and yet allow meaningful algorithm design.

References

1. Aggarwal, A., Vitter, J.S.: The Input/Output complexity of sorting and related problems. Communications of the ACM **31**(9) (1988) 1116–1127
2. Meyer, U., Sanders, P., Sibeyn, J.F., eds.: Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]. In Meyer, U., Sanders, P., Sibeyn, J.F., eds.: Algorithms for Memory Hierarchies, Springer (2003)
3. Vitter, J.S.: Algorithms and Data Structures for External Memory. now Publishers (2008)
4. Lee, S.W., Moon, B.: Design of flash-based DBMS: an in-page logging approach. In: SIGMOD Conference. (2007) 55–66
5. Lee, S.W., Moon, B., Park, C., Kim, J.M., Kim, S.W.: A case for flash memory ssd in enterprise database applications. In: Proc. ACM SIGMOD international conference on Management of data. (2008) 1075–1086
6. Myers, D.: On the use of NAND flash memory in high-performance relational databases. Master's thesis, Massachussets Institute of Technology (2008)
7. Matthews, J., Trika, S., Hensgen, D., Coulson, R., Grimsrud, K.: Intel® turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. ACM Transactions on Storage **4**(2) (2008) 1–24
8. Gal, E., Toledo, S.: Algorithms and data structures for flash memories. ACM Computing Surveys **37**(2) (2005) 138–163
9. Wu, C.H., Chang, L.P., Kuo, T.W.: An efficient R-tree implementation over flash-memory storage systems. In: Proc. 11th ACM International Symposium on Advances in Geographic Information Systems. (2003) 17–24
10. Wu, C.H., Kuo, T.W., Chang, L.P.: An efficient B-tree layer implementation for flash-memory storage systems. ACM Transactions on Embedded Computing Systems **6**(3) (2007)
11. Li, Y., He, B., Luo, Q., Yi, K.: Tree indexing on flash disks. In: Proc. 25th International Conference on Data Engineering. (2009) To appear.
12. Barnat, J., Brim, L., Edelkamp, S., Sulewski, D., Šimeček, P.: Can flash memory help in model checking? In: Proc. 13th International Workshop on Formal Methods for Industrial Critical Systems. (2008) 159–174

13. Goldberg, A.V., Werneck, R.: Computing point-to-point shortest paths from external memory. In: Proc. 7th Workshop on Algorithm Engineering and Experiments. (2005) 26–40
14. Sanders, P., Schultes, D., Vetter, C.: Mobile route planning. In: Proc. 16th Annual European Symposium on Algorithms. (2008) 732–743
15. Ajwani, D., Malinger, I., Meyer, U., Toledo, S.: Characterizing the performance of flash memory storage devices and its impact on algorithm design. In: Proc. 7th International Workshop on Experimental Algorithms. (2008) 208–219
16. Birrell, A., Isard, M., Thacker, C., Wobber, T.: A design for high-performance flash disks. ACM SIGOPS Operating Systems Review **41**(2) (2007) 88–93
17. Bouganim, L., Jónsson, B.P., Bonnet, P.: uFLIP: Understanding Flash IO Patterns. In: Proc. 4th biennial conference on innovative data systems (CIDR). (2009)
18. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica **37**(1) (2003) 1–24
19. Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: An optimal cache-oblivious priority queue and its application to graph algorithms. SIAM J. Comput. **36**(6) (2007) 1672–1695
20. Brodal, G.S., Fagerberg, R.: Funnel heap - a cache oblivious priority queue. In: Proc. 13th Annual International Symposium on Algorithms and Computation. (2002) 219–228
21. Brodal, G.S., Katajainen, J.: Worst-case efficient external-memory priority queues. In: Proc. 6th Scandinavian Workshop on Algorithm Theory. (1998) 107–118
22. Mehlhorn, K., Meyer, U.: External-memory breadth-first search with sublinear I/O. In: Proc. 10th Annual European Symposium on Algorithms. (2002) 723–735
23. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Software: Practice and Experience **38**(6) (2008) 589–637
24. Easy Computing Company: Managed flash technology. (`http://www.easyco.com/mft/`)
25. Ajwani, D., Meyer, U., Osipov, V.: Improved external memory BFS implementation. In: Proc. 9th Workshop on Algorithm Engineering and Experiments. (2007) 3–12