Flash-Optimized Index Structures for Embedded Systems

Devesh Agrawal, Shashi Singh, Deepak Ganesan, Ramesh Sitaraman, and Yanlei Diao
{*dagrawal,shashi,dganesan,ramesh,yanlei*}*@cs.umass.edu*,
Department of Computer Science,
Univeristy of Massachusetts,
Amherst MA 01003.

Flash memories are in ubiquitous use in many embedded systems, including sensor networks, PDAs, mobile phones, digital cameras and others. Flash-based embedded systems present two key challenges in designing index structures: (a) flash has fundamentally different read/write characteristics from other non-volatile media such as magnetic disks, and (b) the limited memory on embedded platforms requires careful allocation of memory resources. In this paper, we present the Buffered-B+ Tree, a novel index structure that is designed to minimize pages and bytes accessed on flash, thereby minimizing energy cost and response time. Our work is inspired by prior research on buffered data structures designed for sorting and bulk-loading that minimize I/O for updates by fully utilizing available memory to buffer operations. However, unlike previous schemes, the Buffered-B+ tree uses an online adaptive algorithm that enables it to be efficient for immediate lookup, in addition to being adaptive to workload characteristics. Our evaluation shows that the Buffered-B+ tree offers upto 20x performance benefits over alternate flash-based approaches across a range of workloads, datasets, and memory constraints.

1. INTRODUCTION

Flash memories are in ubiquitous use in many embedded systems, including sensor networks, PDAs, mobile phones, embedded wireless routers, digital cameras and other devices. Flash memories offer numerous benefits that make them an ideal fit for battery-powered embedded systems: small size, low cost, low power consumption, high capacity, and greater shock resistance. The extremely low power consumption of flash memory is particularly important in energy-constrained sensor network applications since it enables us to view the sensor network as a distributed data archival and querying system [9]. In such a system, voluminous sensor data (e.g. image/audio/vibration data) can be stored and indexed locally at sensors, and queries can be pushed inside the network to retrieve the most relevant data. Flash-based index structures are also required in other embedded systems. For example, flash file systems require efficient index structures to quickly locate items, and some of them use B-trees for indexing [2]. In addition, mobile phones are being employed in a variety of roles that require indexing on flash, ranging from low-power embedded databases [1], to embedded search engines.

A key challenge in designing index structures for flash is that, as a storage medium, flash has fundamentally different read/write characteristics from other non-volatile media such as magnetic disks. The first difference is that, unlike disk writes, flash writes are immutable and cannot be updated in-place—once written, a data page must be erased before it can be written again. Moreover, the unit of erase often spans multiple pages, further complicating flash management. Second, flash memory has a very different cost model from magnetic disks — reads and writes to flash have very low fixed cost, and reads are cheaper than writes. Besides flash characteristics, embedded systems are often *memory-constrained* with memory sizes ranging from as little as a few kilobytes (for sensor platforms such as the Mote [20]) to a few megabytes (for PDAs). Both flash and memory constraints fundamentally impact the energy consumption and response time of a flash-based database system: on one hand, the idiosyncrasies of flash require fundamentally different techniques to limit read, write and erase operations and to optimize for a different cost model, and on the other hand, the constrained memory sizes require memory optimizations that can maximize energy efficiency across a wide range of sensor devices with diverse memory constraints.

There have been numerous proposals to design flash optimized storage systems [7, 8], database systems [15], and index structures [15, 18, 24, 25]. A majority of these approaches focus on the problem of minimizing the overhead resulting from the lack of in-place updates. When data in a flash page is updated, these schemes avoid re-writing the entire page by storing "deltas" to the page in a separate location on flash. We show that such delta-based approaches are fundamentally unsuitable for designing flash-based index structures. This is because they attempt to optimize for node writes, but end up greatly increasing the cost of node reads, thereby increasing overall cost. Our view is that the main deficiency of existing techniques to design flash-optimized indexes is that they focus solely on storage-layer optimizations and seek to keep the index structure itself unchanged. As a result, the index structure is not designed to either optimize for the constraints and cost function of flash memory, or to fully exploit available memory on embedded platforms.

In this paper, we present the Buffered-B+ Tree, a novel index structure that is designed to minimize both pages as well as bytes accessed on flash, thereby achieving the twin objectives of minimizing energy costs as well as response time. Our work is inspired by prior research on buffered data structures designed for sorting and bulk-loading that minimize I/O for updates by fully utilizing available memory to buffer operations [4, 5]. The principal idea behind these techniques is to have a large buffer associated with each node (or group of nodes) of a B-tree, and to buffer operations on a sub-tree until the buffer fills up. Such an approach amortizes each read and write request to the storage media over a large number of insert or lookup operations, and also makes maximal use of available memory. The key limitation of these techniques is that they were designed for bulk-loading, hence their performance suffers dramatically when lookups need to be answered immediately. This is because such "immediate" lookup requests incur the overhead of performing a linear scan through large flash-resident buffers, which is typically far more expensive than the gains obtained by buffering updates. In fact, we show that while a Buffer Tree is significantly better than a standard B+ tree when lookups can be delayed, it performs considerably worse when even a small fraction of the lookups (1%) need to be processed immediately.

The central idea of this paper is that we can design an index structure that provides the best-of-both-worlds — it can provide the benefits of a Buffer-tree for insertions, while providing the benefits of a B+-tree for immediate lookup. We achieve this objective by starting with a flash-optimized B+ Tree, and attaching buffers to different nodes such that each buffer batches updates to a sub-tree. Then, we adapt the size of each buffer using an online algorithm that estimates the cost and benefit of buffering updates based on the currently observed lookup/insertion workload. This technique allows us to combine the benefits of a write-optimized Buffer tree for a write-intensive workload together with a read-optimized B+-tree for read-intensive workload, and to choose the ideal buffer size for any intermediate workload. We call our index structure a Buffered-B+ tree to signify this adaptive capability.

Translating the benefits of a Buffered-B+ tree into practice is non-trivial since it requires that we carefully deal with flash memory limitations and cost functions, as well as the memory limitations of embedded platforms. An important contribution of our work is optimizing the Buffered-B+ tree for flash by: (a) optimizing the layout of buffers and index nodes on flash, (b) determining the optimal node size across a range of workloads and memory sizes, and (c) identifying how to allocate limited memory (RAM) for caching buffers and nodes.

In summary, the contributions in this paper are four-fold:

- We present an analysis of the Buffer Tree for immediate lookups, and a comparison of the strengths and limitations of the B+ tree and Buffer tree.
- This study leads us to the design of the Buffered-B+ tree, a novel index structure for flash memories that is efficient, adaptive, and versatile, and is fundamentally better suited for flash memories than existing approaches. We present a 2-competitive online algorithm for adapting buffer sizes of a Buffered-B+ tree, as well as a heuristic with lower memory consumption.
- We present a full implementation of the Buffered-B+ Tree on flash that optimizes for flash characteristics and the memory constraints of embedded systems.
- Finally, we present a full evaluation of the Buffered-B+ tree and show that it is upto 20x better than all existing flash-based indexing techniques across a range of workloads, datasets, and memory constraints.

2. DESIGN CONSIDERATIONS

In this section, we discuss flash characteristics, and how, in conjunction with memory constraints, they impact the design of an index structure.

2.1 Flash Memory Characteristics

NAND flash memories present some important differences from magnetic disks in terms of their constraints and cost model that impact the design of an index structure:

Flash memory hardware characteristics: Flash memories have three key hardware constraints:

- **No in-place updates:** Although flash is an energy-efficient non-volatile storage medium, it is fundamentally different from other devices such as disk due to its *no-overwrite* nature—once a data page is written to flash, it can not be updated or rewritten and must be erased prior to being written again. The smallest unit that can be erased on flash, termed an *erase-block*, typically spans few tens of pages, which makes a *read-modify-write-operation* prohibitively expensive since it necessitates copying of all valid pages within the erase block, then erasing the block, and finally copying the valid pages and updated page back to the block. Therefore, it is preferable to write the updated data page to a *different* location, rather than erasing and rewriting the old block.
- **Byte-level access:** An important distinction between a disk and flash is that while the smallest amount of data that can be accessed on a disk is a sector (typically 512 bytes), flash memory can be accessed at the granularity of a byte. Each access involves first clocking in the address of the page and the byte offset, after which a sequence of bytes can be read out or written. As we will see, the ability to access flash at a small granularity is crucial since it enables us to design techniques that optimize the total number of bytes accessed as well as the total number of pages accessed.
- **Limited writes per page:** A third constraint of flash memory is that a page can be written to only a limited number of times (typically between 1-4) in a non-overlapping manner.

Flash memory Read/Write Cost: Table 1 shows the energy and latency costs involved with the read and write operations of two different classes of NAND flash memories available today. There are three notable characteristics of the cost function:

- **Low Fixed Cost:** Flash memory is a purely electronic device and has no moving parts. As a result, it has no rotational and seek delay to access a page, unlike magnetic disks. The fixed access cost for flash is, therefore, much closer to per-byte read and write costs than in the case of disks. The ability to cheaply access any page in flash introduces questions about how to best optimize index structures for flash, which is one of the problems that we address in this work.

			Write	Read
SLC Small Block*, 1Gb, 0.5KB page, 16KB erase block	Energy	Fixed	24.54μJ	4.07μJ
		Per-byte	0.0962μJ	0.105μJ
	Latency	Fixed	274us	69us
		Per-byte	1.577us	1.759us
SLC Large Block, 1Gb, 4KB page, 64KB erase block	Energy	Fixed Cost	2.06 μJ	7.78 μJ
		Per-byte	0.002μJ	0.002μJ
	Latency	Fixed	94.4μs	25 μs
		Per-byte	0.042 μs	0.042 μs

Table 1: Flash read/write energy and latency numbers for the Toshiba TC58DVG02A1FT00 small block flash, and the Samsung K9F1208X0C large block flash. For the Toshiba flash, the numbers are measured using an oscilloscope and an add-on board for the Mica2 Mote fabricated by us. For the Samsung flash, the numbers are obtained from the datasheet.

- **Energy vs Latency Cost:** Table 1 also shows that the energy and latency cost functions are very similar to each other in terms of the ratio of fixed to per-byte costs. This is expected since the longer an operation takes, the more energy it consumes. The similarity between the cost functions suggest that optimizing an index structure for energy will optimize latency, and vice-versa.

2.2 Relation to Prior Work

A key problem in designing flash-based index structures is how to handle updates to a node that has already been written to flash. A trivial solution to the problem is to re-write an entire node upon every update — for example, every update to a B+ tree node can result in an out-of-place rewrite of the entire flash page corresponding to the node to a different location in flash. A flash translation layer (FTL) hides this low-level movement of the physical page by exposing only logical page numbers to the index structure, hence pointers are not impacted by the page re-write. Clearly, such an approach is inefficient and results in numerous expensive node re-writes.

Existing techniques for reducing the overhead of out-of-place updates treat updates to a flash page as a list of deltas to the page. Instead of performing an expensive re-write of the original flash page, the deltas to the page are stored separately, thereby reducing the flash write overhead. When a page needs to be read back to memory, both the base page, as well as the pages where the deltas are stored are retrieved, and the flash page is re-constructed. Such a delta-based write-optimization is at the core of many recent techniques [15, 18, 25], all of which use this basic mechanism but differ in how they pack deltas into a flash page, and how they layout delta pages in erase blocks on flash.

Is such a delta-based approach suitable for tree-based index structures on flash? Delta-based optimizations increase the cost of reads since additional delta pages need to be read at each level of the tree, but decrease the cost of writes since inserted keys are packed into a delta page before being written to flash. A delta-based approach to constructing tree-based index structures such as a B+ tree relies on the fact that the savings in terms of fewer page writes outweighs the cost of reading deltas for every tree access. However, Table 1 shows that the cost of writing a page (fixed cost + a page of bytes) is not significantly higher than the cost of reading a page — the ratio is 1.5x for the large block flash, and 5.5x for the small block flash. This suggests that even reading one or two extra delta pages at each level of the tree makes a delta-based scheme more expensive than even a standard B+ tree with a node re-write upon every update.

The memory limitations of embedded devices exacerbates the drawbacks of a delta-based approach. The limited caching opportunity in small memory platforms leads to greater number of evictions of partially filled delta pages to flash. Since a flash page can be only written a limited number of times, this results in more delta pages being written to flash, each of which is only partially filled. This in turn increases the cost of insertion and lookups since more pages need to be accessed. In Section 6, we provide a more detailed study that considers numerous possible optimizations to a delta-based scheme and show that delta-based B+ trees are not effective to reduce the overhead of out-of-place rewrites.

Our Approach: Our view is that the main deficiency of these techniques is that they focus solely on storage-layer optimizations and seek to keep the index structure itself unchanged. Instead of this approach, we ask how an index structure can be re-designed to be aware of the flash memory cost function and limitations, and to fully exploit available memory to optimize performance. This question leads us to the Buffer tree [4], an index structure for bulk-loading that is fundamentally designed to minimize I/O for updates by fully utilizing available memory to buffer operations. In the following sections, we describe the Buffer tree in greater detail, and how we can take advantage of some of its benefits to design a flash-optimized index structure.

3. BUFFER TREES FOR INDEXING

Buffer trees [4] were originally designed for minimizing the I/Os (i.e., operations to read or write a page (block) of elements of size B) required for sorting and bulk loading, hence optimized for insertions.[1] In our work, we re-examine the buffer tree algorithm in the context of online index operations, including its performance for both insertions and lookups (Section 3.1). We further perform a comparative analysis to gain insights into the strengths and limitations of buffer trees and B+ trees. Results of this analysis lead to the design of a more advanced tree structure that offers truly efficient support for both insertions and lookups over flash, which we describe in the next section.

3.1 Buffer Trees

The key idea underlying buffer trees is that, instead of performing insertions or lookups one at a time, we can perform multiple insert and lookup operations all at once. This way, node accesses from the root to the leaves are shared among these operations, yielding a better amortized cost for each operation. To allow multiple operations to be performed all at once, a buffer tree operates in a **lazy batched** manner: it attaches a buffer to each node, containing operations to be performed on the subtree rooted at that node, but without actually performing the operation. When the buffer finally fills up, all of its contained elements are pushed in a batch to the buffers of the child nodes. This process proceeds until elements are pushed all the way to the leaf buffers and eventually to the leaf nodes themselves. Buffer trees also

[1] Buffer trees also support deletions with the same cost as insertions. For ease of composition, we omit deletions in the discussion of this paper and refer the reader to [4] for details.

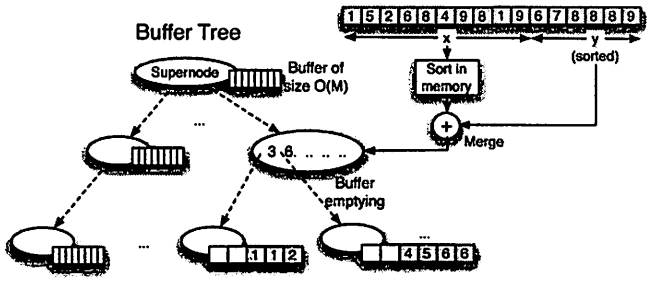


Figure 1: A Buffer tree of height three is shown together with the buffer emptying process for one of the nodes.

Parameter	Symbol	Value(s)
Max. of elements in main memory	M	variable
Max. of elements in a page (block)	B	variable
Node size (in num. of elements)	D	$\frac{M-1}{2}$
Node occupancy	(a,b)	$b = D$, $a = \frac{D}{4}$
Buffer size (in num. of elements)	U	$\frac{M-1}{2}$
Num. of elements in the tree	N	variable
Height of the tree	H	$\leq \log_a \frac{N}{a} + 1$

Table 2: Parameters of the Buffer Tree Algorithm

fully utilize memory to reduce I/Os by employing large nodes as well as large buffers that amortize the node access cost over many elements.

Data Structure. Like a B+ tree, a buffer tree is a *balanced* tree where all leaf nodes reside at the same level from the root of the tree. With respect to *node occupancy*, a buffer tree is modeled as an (a,b)-tree, where each non-leaf node has between a and b child pointers, with $a \leq (b-1)/2$, and the root has between 2 and b child pointers (B+ tree is a special (a,b)-tree with $a = (b-1)/2$). Buffer trees differ from B+ trees in two main aspects: Each node in a buffer tree is a *super-node* whose size D in number of elements is on the order of the memory size M (D is also known as *degree* of the tree). size U is also $\Theta(M)$. Table 2 summaries the parameters of a buffer tree. For succinctness of our discussion, we assume certain values of these parameters as shown in the table. In particular, we set a super-node and its buffer to be roughly half the size of the memory so that they fit in memory together.[2]

Insertions. We first present the buffer tree insertion algorithm. When receiving a request to insert an entry into the buffer tree, we construct a new element consisting of the entry to be inserted and a time stamp. When we have collected B such elements in main memory, we sort them by value and insert them as a block into the buffer of the root stored on the external memory. If the buffer is not full, we stop here. Otherwise, we incur the buffer-emptying process.

Buffer emptying at a non-leaf node. When buffer emptying occurs at a non-leaf node, the basic operations are to load the buffer and the super-node into main memory, remove elements from the buffer, and distribute them to the child buffers through lookups in the node.

More specifically, when emptying the buffer at a non-leaf node v, we view the buffer as two components: the batch of most recent insertions y that caused the buffer overflow, and the elements x in the buffer before these insertions, with $|x| < U$ and $|x| + |y| \geq U$. We also know that y are sorted (more on this below). The emptying process takes three steps: (1). Read elements in x into main memory and sort them, which takes $\frac{|x|}{B}$ I/Os. (2). Read y (already sorted) to merge with x. As we merge the elements, we push them down the tree via lookups in v, and place them in the appropriate child buffers. Given our memory size, it is easy to see that we only need to read y and v once in this step, leading to $\frac{|y|+D}{B}$ I/Os for this step. Furthermore, since we distribute the elements in sorted order to the child buffers, the write cost is simply $\frac{|x|+|y|}{B}$ I/Os. (3). If a child buffer becomes full after receiving y_i (sorted) elements from this process, we empty it recursively. A buffer emptying operation is illustrated in Figure 1.

If we first ignore the recursive calls, the sum of the first two steps is $\frac{2|x|+2|y|+D}{B}$. So, the amortized cost over $|x|+|y|$ elements is $\frac{1}{B}(2+\frac{D}{|x|+|y|})$. Since $|x|+|y| \geq U$, the amortized cost for pushing an element to a child buffer is:

$$O\left(\frac{1}{B}\left(2+\frac{D}{U}\right)\right) \quad (1)$$

By plugging in the values of D and U from Table 2, we have $O(\frac{1}{B})$ I/O to push an element to a child buffer. Since the emptying process can propagate from the root to the leaves, the cost of pushing an element all the way down to a leaf buffer is $O(\frac{H}{B})$.

Buffer emptying at a leaf node. The buffer emptying process at a leaf node is similar to above, except that elements are now removed from the buffer and inserted to the leaf node, which may cause node splits. The cost of this process hinges on the number of splits that can occur when inserting $|x|+|y|$ elements into the leaf node. Earlier work offered upper bounds on the number of splits for an (a,b)-tree [12]. Its theorem indicates that the maximum number of splits in our case is $\frac{4(|x|+|y|)}{b-2a} + H$. So, the number of splits caused by an element is $\frac{4}{b-2a} + \frac{H}{|x|+|y|}$, or $O(\frac{1}{b-2a} + \frac{H}{U})$.

Also, each node split takes $O(\frac{D}{B})$ I/Os. Finally, by plugging in values of a, b, D, and U from Table 2, we again have $O(\frac{H}{B})$ for the amortized cost of inserting an element into a leaf node and splitting nodes bottom-up.

Lookups. As mentioned above, buffer trees were designed for batched insertions and lookups. In the *delayed lookups* approach, a lookup request causes an element to be created and buffered at the root, just like an insertion request. The lookup is not performed until its element is pushed to a leaf node in the buffer emptying process. Thus, a lookup has the same amortized cost, $O(\frac{H}{B})$, as an insertion. However, it experiences a delay up to the time taken to fill all the buffers on the root-to-leaf path.

In the *immediate lookup* approach, whenever we receive a lookup request, we search the buffer tree immediately by scanning the nodes on a root-to-leaf path as well as their buffers. Thus, we address lookups immediately, but have to pay $O(\frac{MH}{B})$ I/Os for each lookup, where the factor M refers to the size of super-nodes and buffers. Note that we cannot achieve a better lookup bound by just delaying the lookup response a little bit. If the delay is so small that the size of the batch is not on the order of M, the slightly delayed lookup cost is still $O(\frac{HM}{B})$, the same as immediate lookup.

Buffer Trees vs B+ Trees: Table 3 shows that while buffer trees are significantly better than the B+ tree for

[2]Other values that satisfy the buffer tree definition also obey the results presented in this paper.

	Insertion	Immediate Lookup
B+ tree	$O(\log_B N)$	$O(\log_B N)$
Buffer tree	$O(\frac{1}{B}\log_M N)$	$O(\frac{M}{B}\log_M N)$

Table 3: I/O costs of Buffer Trees and B+ Trees.

insertions, it is significantly worse for immediate lookups. The lower insertion cost of buffer trees is because of the log factor with the base M (in contrast to the base B for B+ trees). The high immediate lookup cost for the buffer tree arises due to the large factor M due to the need to scan the large nodes and large buffers on a root-to-leaf path to perform immediate lookups. In other words, the very choices of large nodes and buffers that improved insertion performance penalize lookup performance.

Results of our comparative analysis lead to two questions that we address in the rest of the paper: (1). Can we adjust the super nodes in buffer trees so that they require a reduced node access cost for lookups while still contributing to the low insertion cost? (2). Can we also adjust the buffers so that we can achieve a balance between the lookup cost and the insertion cost?

4. Buffered-B+ TREE

In this section, we propose a new index structure, called **Buffered-B+ tree**, that supports efficient immediate lookup and yet has a low insertion cost. It achieves this by making two key modifications to the original buffer tree structure. First, it replaces each super-node with a subtree. Second, it adapts the buffer size of each subtree based on the workload seen by that subtree. The following two subsections explain these techniques in detail.

4.1 Replacing a Super Node with a Subtree

To address the problem of high node access cost, the Buffered-B+ tree replaces a super-node of size $O(\frac{M}{B})$ blocks with a subtree of comparable size. Each node of the subtree has a fanout $F = O(B)$, and the height h of the subtree is chosen to be $O(\log_F \frac{M}{B})$.[3] Since M is typically quite large, it can be assumed that $h \leq \frac{M}{B}$. In addition, each subtree also has a buffer of size $O(\frac{U}{B})$ blocks associated with it. This way, the buffer tree can be seen as a regular B+ tree of height $H = O(\log_F N)$, augmented by buffers at every h^{th} level node. A Buffered-B+ tree is depicted in Figure 5.

The key insight is that this change retains the benefits of the Buffer tree in terms of batched insertions, but reduces the immediate lookup cost. This can be seen as follows:

Insertions: Applying the same analysis as done in Section 3.1, we can show that the amortized cost of emptying an element from a buffer to the next level buffer (placed h levels down the tree) is at most $O(\frac{1}{B} \cdot (2 + \frac{M}{U})) = O(\frac{M}{U \cdot B})$. To compute the cost of pushing an element all the way down to the lowest level buffer, we multiply the above cost with the number of buffers on the path, $O(\frac{H}{h})$, where H is the total height of the tree and h is the height of a subtree for one buffer. Therefore, the amortized top-down cost of insertion (ignoring splits) can be shown to be:

$$O\left(\frac{M}{U \cdot B} \cdot \frac{H}{h}\right) \quad (2)$$

An analysis similar to the one done in section 3.1, shows that the amortized cost of splits caused by emptying the buffer of leaf subtree is at max $O(\frac{\frac{M}{B}+H-h}{U})$. Since $h \leq \frac{M}{B}$, it can be shown that the splitting cost is not more than the top down cost of insertion. Hence, we claim that when the buffer size U is comparable to the size M of the subtree, the total insertion cost is $O(\frac{1}{B} \cdot \frac{H}{h})$. Given $\frac{H}{h} = O(\log_F \frac{N}{B} / \log_F \frac{M}{B})$ $= O(\log_{\frac{M}{B}} N)$, the insertion cost becomes:

$$O\left(\frac{1}{B}\log_{\frac{M}{B}} N\right) \quad (3)$$

As can be seen, this cost has the same order as the buffer tree insertion cost shown in Table 3.

Immediate lookups: While the insertion cost remains unchanged, changing the node size is able to lower the immediate lookup cost. The node access cost for a lookup is now reduced to reading only $O(\log_F N)$ nodes, each of size $O(B)$. We can see that this is a huge improvement over the node access cost of a lookup over a buffer tree, which is linear in M. However, the buffer access cost of a lookup remains unchanged as we still need to scan $\frac{H}{h}$ buffers of size $O(\frac{U}{B})$ blocks each. Hence, the buffer scan cost can be given by $O(\frac{U}{B} \cdot \frac{H}{h}) = O(\frac{U}{B}\log_{\frac{M}{B}} N)$.

Therefore, the total cost of an immediate lookup is:

$$O(\log_F N) + O\left(\frac{U}{B}\log_{\frac{M}{B}} N\right) \quad (4)$$

It can be seen that the lookup cost here, assuming $U = \Theta(M)$, still has the same order as the corresponding cost of the buffer tree $O(\frac{M}{B} log_M N)$ as shown in Table 3.

A similar approach of placing buffers at every h^{th} level of the tree was employed in [5]. One distinction is that, that approach was used to efficiently bulk load a tree, whereas ours is to reduce the node access cost of immediately lookups. Second, such an approach of attaching buffers to subtrees does not completely solve the problem of high lookup costs, as showed above. This motivates us to consider using an adaptive buffer size to also reduce the buffer access cost in lookups.

4.2 Adaptive Buffering

The Buffered-B+ tree adaptively controls the buffer size of each subtree to improve the lookup cost while maintaining a low insertion cost. The buffer size is controlled by adaptively deciding whether or not to prematurely empty the buffer in response to a lookup or an insert operation. The key rationale behind doing so is that emptying a buffer reduces the buffer size to zero, hence eliminating the need for buffer access for future lookups.

The right point at which to empty the buffer is workload dependant. From Equation 2, we can see that the insertion cost is inversely related to U (the *effective* buffer size in this case, since we prematurely empty buffers), whereas Equation 4 shows that the lookup cost is linear in U. This trade-off indicates that the optimum value of U depends on the workload: we would want U to be small for a read intensive workload, and high for a write intensive workload. The decision of when to empty the buffer is taken at a per subtree level, since different parts of the tree could be seeing a different workload depending on the key distribution.

In the following subsections, we first present an optimal online algorithm *ADAPT* that adaptively decides when to empty the buffer. Then, we present a simplification of this

[3]To avoid confusion, we use F to denote the node size in a Buffered-B+ tree, as opposed to D for a buffer tree.

algorithm that requires a much smaller amount of state but achieves good performance in practice.

4.3 An optimal deterministic online algorithm for Adaptive Buffering

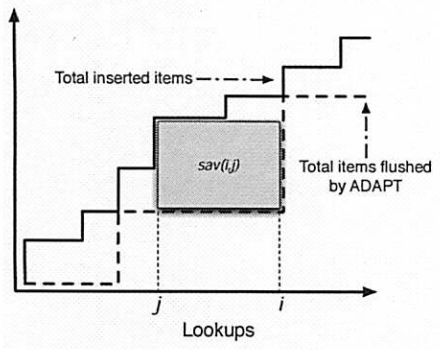


Figure 2: Algorithm ADAPT.

We describe algorithm $ADAPT$ which processes a sequence of insert and lookup operations in an online fashion and decides at each stage whether or not to empty the buffer. A typical sequence may be

$$\{INS_1, INS_2, \cdots L_i, L_{i+1}, \cdots INS_j, \cdots\}$$

where INS_i and L_i are the i^{th} insert and i^{th} lookup respectively. The sequence of operations processed by ADAPT can be graphically represented by plotting the i^{th} lookup on the x-axis versus the total inserted items preceding the i^{th} lookup on the y-axis (see solid staircase-like line in Figure 2). The vertical segments of this plot represents items inserted into the buffer, and the horizontal segments represents lookups made subsequent to the insertions. Likewise, the actions taken by ADAPT can be represented by plotting the i^{th} lookup on the x-axis versus the total items emptied by ADAPT prior to the i^{th} lookup in the y-axis (see dotted stair-case-like line in Figure 2). The vertical segments of this plot represents the algorithm emptying the buffer, while the horizontal segments represents lookups processed without emptying.

The goal of ADAPT is to minimize the total cost of processing the sequence of operations. ADAPT operates in an online fashion making decisions at the current time without knowing the future. The competitive ratio of ADAPT is the ratio of the cost of ADAPT to the cost of OPT (an *offline* optimal algorithm that knows the *entire* sequence in advance) for a worst-case sequence of operations.

Let s_j denote the number of buffer entries in the buffer at the j^{th} lookup, δ denote the cost of reading the subtree and γ denote the cost of reading (and writing) a buffer-entry during a buffer emptying operation. We make the following simplifying assumptions in this analysis. We assume that the cost of scanning the buffer is a linear function of the number of buffer entries, *i.e.* $\beta \cdot s_j$ for the j^{th} lookup. We also assume that the buffer emptying cost can be written as $\gamma \cdot s_j + \delta$, since it is composed of the cost of reading the subtree δ and the cost of reading (and writing) the s_j buffer entries given by $\gamma \cdot s_j$.

4.3.1 Algorithm description

When the i^{th} lookup is received, ADAPT considers every suitable $j < i$ since the last flush and computes $sav(i,j)$ which is the savings in the cost of lookup processing if it had (in hindsight) emptied the buffer at the j^{th} lookup. If ADAPT had emptied the buffer at the j^{th} lookup, it would have saved scanning s_j entries for the next $(i-j+1)$ lookups. Therefore, the savings $sav(i,j)$ can be written as $(i-j+1)\cdot\beta\cdot s_j$. Pictorially, $sav(i,j)$ is simply β times the area of the shaded rectangle in Figure 2. Let the cost of emptying the buffer at the j^{th} lookup be $empty(j)$ which equals $\gamma \cdot s_j + \delta$. If there exists a j such that the savings $sav(i,j) \geq empty(j)$, ADAPT empties the buffer prior to processing the i^{th} lookup. Intuitively, ADAPT looks to see if an empty operation at some previous j^{th} lookup would have decreased its overall cost *in hindsight*, and if such a lookup exists it empties the buffer now. In particular, if OPT knowing the future *could have* emptied the buffer at some j^{th} lookup to decrease overall cost, ADAPT follows suit, albeit with some delay resulting in some extra lookup cost. Note that up to U counters may be needed to track the relevant values of s_j and $sav(i,j)$, where U is the maximum size of the buffer. In addition, ADAPT will also empty a buffer if the size limit of U is exceeded, i.e. if there have been $U+1$ inserts after the last empty operation.

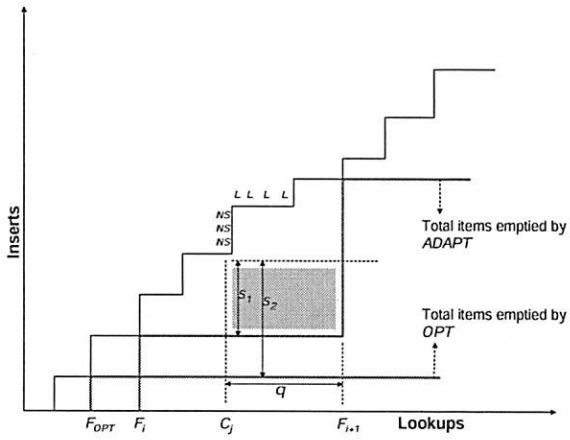


Figure 3: Proof of Lemma 1

4.3.2 Cost analysis

Let $cost(ADAPT)$ (resp., $cost(OPT)$) be the total cost incurred by ADAPT (resp., OPT). This total cost can be broken down into 3 components: (1) $bef(ADAPT)$ is total cost incurred by ADAPT corresponding to the *fixed* component δ of buffer-emptying cost. (2) $bev(ADAPT)$ is the total cost corresponding to the variable component $\gamma \cdot s_i$ of buffer-emptying cost. (3) $lp(ADAPT)$ is the total cost corresponding to the variable component $\beta \cdot s_i$ of lookup-processing cost. The total cost incurred by ADAPT is $cost(ADAPT) = bef(ADAPT) + bev(ADAPT) + lp(ADAPT)$. Similarly, $cost(OPT) = bef(OPT) + bev(OPT) + lp(OPT)$. The total cost of buffer-emptying is $be(ADAPT) = bef(ADAPT) + bev(ADAPT)$ (similarly for $be(OPT)$). In the following lemmas, we bound each of these 3 cost components for

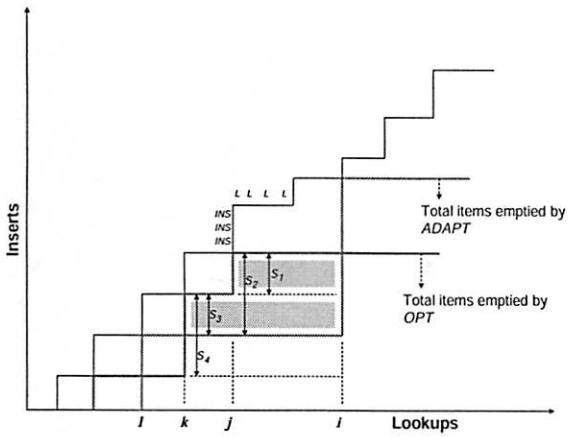


Figure 4: Proof of Lemma 3

ADAPT.

Lemma 1: There exists an optimal OPT such that between any two consecutive empty operations of ADAPT there must be at least one empty operation of OPT.
Proof: Let F_i and F_{i+1} be two consecutive empty operations of ADAPT (see Figure 3). F_{OPT} denotes the empty operation of OPT. We have the following two cases:
(1) Empty F_{i+1} was caused by the buffer getting full, i.e. when U elements were already in the buffer and the $(U+1)^{th}$ element arrived. In this case we know that in the interval between events F_i and F_{i+1}, there would have been $U+1$ inserts. Thus, OPT must also empty at least once in this interval.
(2) Empty F_{i+1} was not caused because the buffer got full. Thus, ADAPT emptied the buffer to optimize cost. For contradiction, assume that OPT did not empty in the interval between F_i and F_{i+1}. Let F_{OPT} be the OPT empty immediately preceding F_i. In case OPT did not empty before empty F_i of ADAPT, F_{OPT} is the *first* empty of OPT that occurs when the algorithm starts. The algorithm starts with the buffer being empty, and we consider it as the *first* empty operation. In Figure 3 red-line shows the behavior of OPT and green-line shows that of ADAPT. In the interval (F_i, F_{i+1}), red-line must be below green-line because in this interval, buffer size of OPT must be no less than that of ADAPT. ADAPT would have taken the decision to carry out empty F_{i+1} because one of the counters, in the interval (F_i, F_{i+1}), would have hit zero. Let C_j be that counter, corresponding to the lookup L_j (shown in Figure 3). s_1 is the buffer size of ADAPT at lookup L_j. s_2 is the buffer size of OPT at L_j. q is the number of lookups that arrived in the event-interval $[L_j, F_{i+1}]$ (including the lookup which resulted in the event F_{i+1}). Empty F_{i+1} was carried out because C_j hit zero, i.e. $\beta \cdot q \cdot s_1 \geq \gamma \cdot s_1 + \delta$. Here, $\beta \cdot q \cdot s_1$ is the savings in the cost of lookup processing if ADAPT had (in hindsight) emptied the buffer at the j_{th} lookup. $\gamma \cdot s_1 + \delta$ is the cost of emptying at j_{th} lookup. This implies $\beta \cdot q \geq \gamma + \delta/s_1 > \gamma + \delta/s_2$, i.e. $\beta \cdot q \cdot s_2 > \gamma \cdot s_2 + \delta$. This suggests that, had OPT emptied at L_j, it would have further decreased its cost. But this is not possible because OPT is optimal. Hence, we arrive at a contradiction. □

Lemma 2: $bef(ADAPT) \leq bef(OPT)$
Proof: The *first* empty of ADAPT occurs when the algorithm starts and the buffer is already empty. By arguments similar to the one given in Lemma 1, there must be at least one OPT empty between this *first* empty of ADAPT, and its next empty operation. Lemma 1 states that between any two real empties of ADAPT later on, there must be at least one OPT empty. We associate each ADAPT empty with the immediately preceding OPT empty. We conclude that the total number of ADAPT empties is no more than the total number of OPT empties. Each empty operation contributes a constant, δ, to $bef(ADAPT)$ and $bef(OPT)$. The Lemma immediately follows. □

Lemma 3: $bev(ADAPT) \leq bev(OPT) + \gamma \cdot U$
Proof: When the buffer of fixed constant size U gets filled, there must be an *empty* on the subsequent insert operation. We consider the total number of elements (each element emptied from the buffer is counted separately) that could be emptied by ADAPT and OPT during the entire course. The difference in this number between any two algorithms is at most U, corresponding to at most U elements still remaining in the buffer at the end of the course. The *bev* part is proportional to the total number of elements emptied during the entire course. Hence, in the worst case, $bev(ADAPT)$ may be off by a constant $\gamma \cdot U$ from $bev(OPT)$. The Lemma immediately follows. □

Lemma 4: $lp(ADAPT) \leq lp(OPT) + be(OPT)$
Proof: Firstly, we have $lp(ADAPT) \leq lp(OPT) + lp(ADAPT - OPT)$, where $lp(ADAPT - OPT)$ is the extra lp cost that ADAPT incurs over the lp cost incurred by OPT. We then show that $lp(ADAPT - OPT) \leq be(OPT)$, which proves the lemma.

Consider Figure 4. Red staircase line shows ADAPT and green staircase line shows OPT. ADAPT empties consecutively at l^{th} and i^{th} lookups. By Lemma 1, there is at least one OPT empty in between. Figure 4 shows two of them: at k^{th} and j^{th} lookups. The shaded area shown in the figure is a typical portion of $lp(ADAPT - OPT)$. We show that this is at most the cost OPT incurred to empty the buffers at j and k. Thus each such portion of $lp(ADAPT - OPT)$ before an ADAPT-empty can be attributed to empty costs incurred by OPT before this ADAPT-empty and after the immediately preceding ADAPT-empty. Using Lemma 1, we can then conclude that $lp(ADAPT - OPT) \leq be(OPT)$.

The shaded portion in Figure 4 is divided into two rectangles. For the top rectangle, ADAPT's savings if it had emptied at j, is $\beta \cdot (i-j) \cdot s_2$ (excluding i^{th} lookup which caused an *empty* before its execution). Now ADAPT emptied at i because one of the counters went below zero (including the cost of i^{th} lookup). Thus, cost of accumulated lookups excluding i^{th} lookup, is at most the *initial* value of any of the previous counters (Note: initial value of a counter is $\gamma \cdot s + \delta$, s being the buffer size at the counter position). In particular, $\beta \cdot (i-j) \cdot s_2 \leq \gamma \cdot s_2 + \delta$, i.e. $\beta \cdot (i-j) \leq \gamma + \delta/s_2 < \gamma + \delta/s_1$. Or, $\beta \cdot (i-j) \cdot s_1 < \gamma \cdot s_1 + \delta$, i.e. portion of $lp(ADAPT - OPT)$ denoted by the top rectangle is at most the cost of emptying by OPT at j. Similarly for the bottom shaded rectangle, $\beta \cdot (i-k) \cdot s_3 \leq \gamma \cdot s_3 + \delta < \gamma \cdot s_4 + \delta$, i.e. portion of $lp(ADAPT - OPT)$ denoted by the bottom rectangle is at

most the cost of emptying by OPT at k. □

Theorem 1: ADAPT is 2-competitive, *i.e.*, the cost incurred by ADAPT is within a factor of 2 of the optimal offline algorithm OPT.
Proof: Using Lemma 2, Lemma 3 and Lemma 4, we have: $bef(ADAPT)+bev(ADAPT)+lp(ADAPT) \leq bef(OPT)+bev(OPT)+lp(OPT)+be(OPT)+\gamma \cdot U$
Or, $cost(ADAPT) \leq cost(OPT)+be(OPT)+\gamma \cdot U$
Or, $cost(ADAPT) \leq 2 \cdot cost(OPT)+\gamma \cdot U$ (since $be(OPT) \leq cost(OPT)$)
Here $\gamma \cdot U$ is a constant. Hence, it follows that ADAPT is 2-competitive. □

Theorem 2: ADAPT has the best possible competitive ratio for a deterministic algorithm, i.e. no deterministic algorithm can be better than 2-competitive.
Proof: *Ski-rental* [27] is a special case of our problem by considering the following mapping. Suppose we get just *one* insert and sequence of lookups after that. Each lookup costs β, which corresponds to the cost of *renting* a ski. An empty operation costs $\gamma+\delta$, which corresponds to the cost of *buying* a ski. Since we know that even the simple case of ski-rental cannot have a deterministic online algorithm with competitive ratio less that 2, the same lower bound holds for our more general problem. □

4.4 A simplified algorithm

Although we showed *ADAPT* to be 2-competitive, it suffers from two problems that complicate a direct implementation. First, it requires U counters, one for reach possible state of the buffer. Since this state has to be maintained for each buffer in the Buffered-B+ tree, doing so is infeasible for the low memory regimes that we target. Second, it has a higher computational overhead as it needs to update every counter on processing a lookup. Therefore, in our implementation, we use a simplified heuristic that is motivated by algorithm *ADAPT*, called *Adapt-Simple*.

Algorithm 1 *Adapt-Simple:* Adaptive buffer emptying algorithm

```
if requestType is an insert then
  if bufferSize < U then
    Do not empty the buffer
  else
    Empty the buffer
  end if
end if
{The following only concerns lookup requests}
subtree ← Subtree receiving the lookup
α ← Current accumulated cost of buffer scanning
bufferReadCost ← Est. cost of reading this buffer
subtreeReadCost ← Est. cost of reading the entire subtree
bufferEmptyCost ← subtreeReadCost + bufferReadCost
if subtree is a leaf then
  bufferEmptyCost += Est. cost of writing entire subtree
else
  bufferEmptyCost += Est. cost of writing entire buffer
end if
if α + bufferReadCost < bufferEmptyCost then
  Scan the buffer to answer the lookup request
  α += Cost of scanning the buffer
else
  Empty the buffer
  α ← 0 {Clear the counter}
end if
```

Adapt-Simple assumes that the future workload is well described by the workload seen in the past and heuristically estimates whether to empty the buffer. At a high level, *Adapt-Simple* keeps track of the accrued cost, of scanning the buffer, it has paid since the last buffer empty operation. It decides to prematurely empty when this accrued cost is more than the estimated cost of buffer emptying. Similar to *ADAPT*, Adapt-Simple waits till the lookup processing cost accrues to a certain threshold that warrants incurring the cost of emptying the buffer. Algorithm 1 provides the pseudocode for the algorithm.

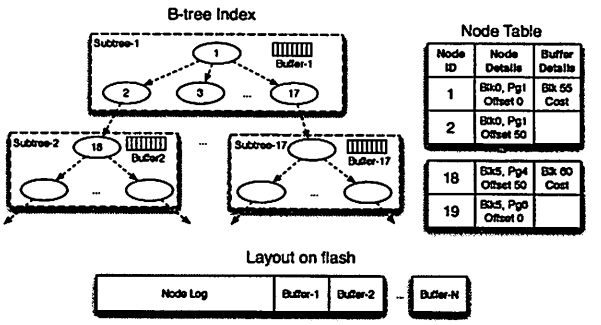


Figure 5: A Buffered-B+ tree having $h = 2$ and a fanout of 16 is shown. Nodes 1 and 18 are roots of their subtrees respectively and have a buffer associated with them.

5. IMPLEMENTATION

Having discussed the design of the Buffered-B+ tree, we now turn to the implementation of the index structure on flash. There are three key aspects of our implementation: (a) the indexing layer including how insertions, buffer empties and immediate lookups are carried out, (b) the flash storage layer including layout of the nodes and buffers, and (c) memory allocation between the node and buffer caches.

5.1 Index Layer

The implementation of the Buffered-B+ tree augments a standard B+ tree with buffers attached to every node having a height that is a multiple of the subtree height h (Figure 5 shows an example with $h = 2$). An in-memory *Node Table* maps each node to its current location on flash. Subtree root nodes that are associated with a buffer also have additional buffer-specific metadata associated with them. Our implementation requires about four bytes of node-table metadata for each node, and about 8 bytes of metadata for each subtree. Since the number of subtrees is an order of magnitude lower than the number of nodes, we believe this additional metadata overhead justifies the savings obtained by Buffered-B+ trees.

Insertions: Insertions to the Buffered-B+ tree are carried out in a lazy batched manner. An insertion involves creating a new "buffer-entry", that is inserted into the root subtree's buffer. Since a buffer needs to be sorted in memory, the maximum buffer size is constrained by the total memory M. In our implementation we set this to $3M/4$.

Adapt-Simple: The *Adapt-Simple* algorithm can trigger a buffer empty operation in two cases (a) prematurely in response to a lookup (b) when the buffer size exceeds the maximum buffer size U, which can result in cascading empties down the tree. It has three components that are es-

timated online: (a) subtree read/write costs, (b) buffer read cost, and (c) buffer empty cost for non-leaf subtrees. We assume that the entire sub-tree is read from flash for each emptying operation. This is typically the case since a batch of updates are applied at the same time, hence it is likely that the updates need all the nodes to be read. The cost of reading the buffer is known since we can keep track of the size of each buffer and the number of entries in it. To estimate the buffer empty cost, we keep track of the average packing ratio of the buffer pages (*i.e.* number of buffer entries per buffer page on flash), and use this to compute the number of flash pages written, and hence the cost for emptying the buffer.

Buffer Emptying: To empty a buffer, we first load and sort the head $|x|$ of the buffer in memory. Next it is merged on the fly with the rest of the buffer $|y|$, which is still on flash, and passed through the subtree. If it is a non leaf buffer, then we route the buffer entries through the emptying subtree into the buffer of the destination subtree. For example, as shown in Figure 5, buffer entries in the buffer of subtree 1 may be deposited in the buffers of subtree 2 through 17. In case it is a leaf buffer, then the sorted buffer entries are inserted into the leaf subtree. We allow the subtree to split multiple times as the entire leaf buffer is emptied. The newly created leaf subtrees are then inserted into the parent subtree, which in turn might split. In case the subtree has a non-empty buffer, its buffer is also split. The whole splitting process may cascade up to the root, in which case a new root subtree might need to be created.

Immediate lookup: The Buffered-B+ tree does immediate lookup much like the B+ tree. However, on encountering a subtree root node on the way to the leaf, it also scans the buffer associated with that subtree. For example, in Figure 5, if a lookup request reaches subtree 2, we will also need to scan buffer 2. If the key to be looked up is not found in the buffer we send it further down the tree.

5.2 Flash Layer

Node size: The cost model and tradeoffs of flash memory require careful engineering of node sizes to optimize them for this media. Previous studies have shown that the huge gap between seek latency and the transfer rate for magnetic disks favors larger B+ tree node sizes (16-32KB [11]) because they amortize the cost of going to the disk and produce shallower trees. In contrast to magnetic disks, the fixed cost of page reads and writes for flash is much closer to the per-byte cost (as shown in Table 1). This makes it important to consider the total number of bytes accessed on flash since it constitutes a significant fraction of the overall cost. The flash memory cost model favors small B+ tree nodes and deeper trees for two reasons: (a) smaller node sizes reduce the total read cost since fewer bytes need to be read for each lookup or insert operation, and (b) smaller node sizes reduce the cost of an out-of-place rewrite since fewer bytes needs to be re-written. Somewhat surprisingly, existing approaches to design B-trees or its variants for flash memories do not consider this problem, and set the node size to be equal to a physical page, which is typically between 0.5KB - 4KB. As we will show later in Section 6, a lower fanout (node size) leads to tremendous cost savings for all flash-based indexing schemes.

Node layout on flash: Since each Buffered-B+ tree node is much smaller than a page in size, it is important to pack multiple nodes into a flash page to amortize the cost of flash writes over multiple nodes. Our implementation packs multiple nodes into a single page, and stores these pages on flash in a log-like manner. Every write to a node results in an out-of-place re-write to the head of the log. An important point here is that fewer such out-of-place node re-writes are generated by the Buffered-B+ tree in comparison with a B+ tree due to the effect of batching. Thus, each out-of-place rewrite of a node is typically amortized over multiple keys being inserted to the node. Whenever a node is re-written to a new location, the in-memory Node Table is updated to reflect the new location. In order to facilitate garbage collection to free up space, the node log is split into two halves. When one of the halves fills up, the valid nodes of the Buffered-B+ tree are recursively copied and written to the second half, and the first half is erased. Since the node log is fairly large (many hundreds of erase blocks), erases are an infrequent operation. For typical NAND flash memories, the erase cost is much lower than the page write cost. For example, the block erase cost (block = 32 pages) for the Toshiba flash in Table 1 is approximately 2.5% of the block write cost. Hence the cost of erasing the node log is a relatively small fraction of the overall cost.

Buffer layout on flash: The buffer sizes in the case of the Buffered-B+ tree are typically considerably larger than the node sizes. Therefore, we do not need to pack different buffers together into pages. Instead, each buffer is laid out as a small flash partition consisting of a couple of erase blocks. Pages within a buffer partition are written in a log like manner. This approach also simplifies garbage collection since the partition can be erased completely after it is emptied. Moreover, the buffer can also be emptied and erased when the partition becomes full. Since the number of bytes read is important to minimize for flash, we store metadata in each buffer page that tells how many buffer entries are in the page. Therefore, each page read involves first reading the metadata, and then the corresponding number of bytes.

5.3 Memory Allocation

Memory allocation presents an important implementation challenge in our system. We now describe how memory is used by different system components, and then discuss how it is partitioned across them. The Buffered-B+ tree partitions the memory given to it into two parts: (a) the Node Cache and (b) the Buffer Pool. The Node Cache is responsible for caching the new and recently accessed nodes of the Buffered-B+ tree in memory. An LRU eviction policy is used to evict old nodes[4]. In addition it also ensures that the nodes of the subtree, whose buffer is being emptied, are read at most once from flash. In order to do so, it pins the nodes belonging to the subtree being emptied.

The Buffer Pool is responsible for storing the tail pages of each buffer. This enables the buffer pool to improve write-coalescing since it can potentially allow more buffer-entries to be packed into a buffer page before the page is flushed to flash. This improves both the cost of writing the buffers as well as the cost of reading the buffers for immediate lookup since fewer fixed page read costs are incurred. Evictions from the buffer pool can occur in two cases: (a) a buffer needs to be loaded into memory for emptying, and (b) a new tail page is added to a buffer. The eviction policy

[4]We experimented with other eviction policies such as LIFO but did not find much difference between the two

for the buffer pool is "highest-packed buffer page" *i.e.* the buffer page having the largest number of buffer entries in it is flushed. This increases the packing ratio of the buffer pages, leading to less expensive reads for reasons discussed earlier.

How should we allocate memory between the Node Cache and Buffer Pool? We allocate most of the system memory to the buffer pool. The key reason is that the buffer read and write costs contribute to a significant share of the overall immediate lookup cost. Hence, packing of the buffer entries is critical to reduce this cost. In contrast, giving less memory to the node cache has a low impact on the overall performance because nodes of a subtree are accessed together resulting in a smaller working set.

6. EVALUATION

In this section, we evaluate the performance of the Buffered-B+ Tree over a NAND flash simulator. Unless otherwise mentioned, we used the cost function of the Toshiba 1Gb NAND flash shown in Table 1. This flash does not allow any page rewrites and has a page size of 512 bytes. We find that the results in terms of energy and latency were very close to each other, hence we present results only in terms of energy as the performance metric. The simulator was designed such that the memory given to it could be varied. Moreover, each index structure was responsible for managing its own memory.

Each experiment was performed by applying a workload over a pre-constructed tree containing 50K keys. The total costs obtained were normalized by the workload size to report the cost per operation. Most of the experiments use an index workload comprising a random mix of lookups and inserts with a given lookup-to-insert-ratio q, where q denotes the likelihood of getting a lookup over an insertion. The workload size was adjusted for each q such such that a total of 200 thousand insertions took place. Unless otherwise stated the keys being inserted and queried were *i.i.d* from a uniform distribution ranging from 1 to 10000.

The Buffered-B+ tree was configured such that it used 25% of its memory for its node cache, while the rest of the memory was used by the buffer pool. The fanout of the Buffered-B+ tree was fixed to 16, and the subtree height h was set to two. These parameters work well across a wide range of workloads and memory regimes (see Section 6.3.6). We exclude the memory usage of the Node Table (approximately 40 KB) in the results that we present since the sizes of the Node Tables for different schemes were almost identical. We note that, for memory limited platforms, the Node Table can be maintained on flash as a multilevel tree as shown in [3].

6.1 Evaluation of existing flash-based indexes

In this section, we validate our claim that delta-based approaches for designing tree-based index structures are inefficient even in comparison to a well-tuned standard B+-tree for flash. We implement the delta-based approaches at a layer below the B+ tree. Each "delta" in our implementation describes the high level operation performed on the B+ tree node, for example, insertion of a new key-pointer pair or updation of a child pointer. Each delta based scheme uses an in-memory Node Table that maps a B+ tree node to a "base node" and a list of delta pages. In order to read a node, we first read the base node into memory and then apply the deltas mentioned in the delta pages in order. To prevent the Node Table size from growing unbounded, we limit the number of delta pages associated with each node to four [24, 25]. Similar to the Buffered-B+ tree implementation, nodes are written out as a log and an LRU based Node cache is employed to cache new and recently accessed nodes. We implement four different delta based schemes:

Basic Delta: This scheme is based on [24], and packs deltas from different nodes into a single delta page that is written out to the flash as a log. The basic drawback of this approach is that deltas belonging a node are scattered over the flash and hence many delta pages may need to be read to reconstruct a single node, thereby significantly increasing the cost of node reads. The remaining three techniques present solutions for this problem.

BFTL: BFTL [25] improves upon the basic delta approach by coalescing deltas belonging to a single node and writing them into fewer flash pages. This reduces the number of flash pages that must be read to reconstruct a node. Updates in BFTL proceeds in three phases: 1. a maximum of N updates to the B+-tree are kept in a 'reserve buffer', 2. when the reserve buffer is full, the deltas generated by the insertion of N keys into the tree are kept in an in memory delta pool, and 3. deltas contained in the delta pool are flushed to the flash such that the deltas concerning the same node are written to the same (or consecutive) pages. We ensure that whenever memory is not being used by the reserve buffer or delta pool, it is used by the node cache. We set $N = 60$ based on the value recommended in [25].

IPL: In IPL [15], each nodes gets its own dedicated delta page which only contains deltas belonging to that particular node. Each dirty cached node has an associated log buffer in memory which stores the deltas corresponding to that node. Whenever a dirty cached node is evicted, we write its corresponding log buffer to a flash page. In the IPL scheme, the pages in an erase block are statically partitioned into node pages and log pages. In our implementation, each erase block is partitioned into 16 node pages and 16 log pages since these settings worked best. Whenever a new node page residing in erase block needs to be dirtied, we allocate a new flash log sector on the erase block. The erase block is "merged" if it is full by reading, merging, and rewriting all node pages in it to a new erase block.

Consolidation: The Consolidation scheme addresses the problem of long chain of deltas in the Basic Delta scheme by using an online algorithm to adaptively decide whether a node should be maintained as a list of deltas or as a single page on flash. This scheme is inspired by the FlashDB algorithm [18], which proposes an adaptive scheme to decide whether a node should be maintained on flash as a list of deltas or as a single page on flash. In this approach, each node N maintains a counter C that tracks two costs: (a) the cost of adding a new delta by incrementing C by $NodeWriteCost$, and (b) the cost of reading the delta chain by decrementing C by $PageReadCost * i$, where i denotes the length of the delta chain. The node N is consolidated when $C < 0$, *i.e.* when the cost of storing deltas outweighs the benefits.

To have a fair comparison between the schemes, we first determine the optimal fanout for each scheme and pick the value that gives the best results for the scheme (see Section 6.3.6 for a discussion of the impact of fanout). We find that a fanout of 8 gives the best results for the B+ tree, a

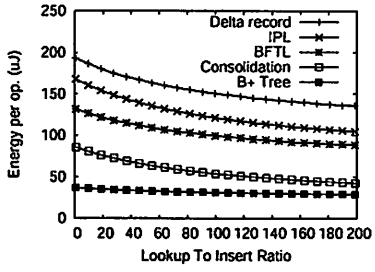


Figure 6: Comparison of delta based schemes with a tuned B+ tree

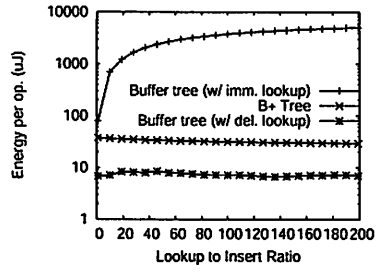


Figure 7: Immediate lookup on a regular buffer tree

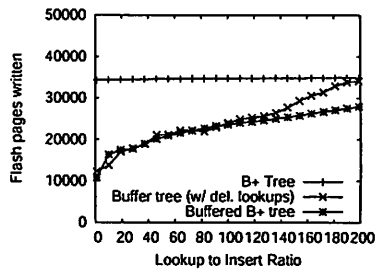


Figure 8: Impact on Garbage Collection

fanout of 16 for the Basic Delta, BFTL, and Consolidation schemes, and a fanout of 84 for the IPL scheme.

Figure 6 shows how the total cost per operation varies as the lookup-to-insert-ratio q is varied from 1% to 200%. This experiment was performed with 64KB of memory. The results show that a B+ tree with optimal fanout does better than all the delta-based techniques. This is because a low fanout greatly reduces the node write (and read) costs for a B+ tree. Delta based techniques avoid the cost of rewriting a node, but incur a much higher node read overhead. Since node writes are quite cheap in a low fanout B+ tree, the delta based techniques do more harm than good.

The differences between the four delta-based schemes is also worth noting. The Basic Delta scheme is the worst since it has to read a long list of delta pages for each node. IPL improves upon the Basic Delta scheme as it packs deltas into dedicated delta pages. However, it needs to perform a large number of merge operations which increases its cost. The BFTL approach does better than both IPL and Basic Delta as it achieves a good amount of delta packing and yet doesn't forgo the benefits of storing the deltas in a log. The Consolidation scheme performs the best among these schemes as it has a low read overhead for frequently read nodes like top level nodes and reduces the number of node rewrites required for infrequently read nodes like leaf nodes.

6.2 B+ Tree vs Buffer Tree

We now turn to a comparison of the B+-tree with optimal fanout against the Buffer tree. We consider two cases of the Buffer tree in this study: (a) Buffer tree with delayed lookups *i.e.* lookups are buffered with the insertions, (b) Buffer tree with immediate lookup *i.e.* the lookups scan the nodes and buffers and are executed immediately. Figure. 7 shows that the Buffer tree with batched lookups and insertions is about 5x better than the B+ tree with optimal fanout. This shows that buffering updates has considerable benefits over a scheme that does not use buffering at all. However, the graph also shows that merely doing immediate lookups on an unmodified Buffer tree is not a good idea as the large linear scan costs of lookups dominates over the benefits obtained by batching the insertions. In fact, the Buffer tree with immediate lookup performs considerably worse than a B+ tree even for a very small lookup-to-insert-ratio ratio of just 1%. It can also be seen that the cost of immediate lookup on a buffer tree rises very rapidly as the workload becomes more read-intensive. This motivates the need for an adaptive structure that can obtain the benefits of write batching in a write-intensive workload, yet perform well for a read-intensive workload.

6.3 Performance of the Buffered-B+ Tree

In this section, we show that the Buffered-B+ tree scales gracefully over a spectrum of workloads and memory constraints, and provides considerable performance benefits.

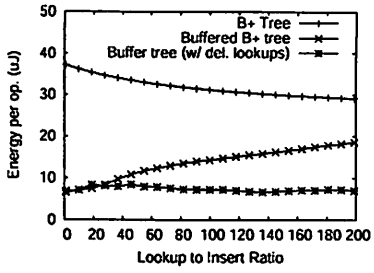
6.3.1 Impact of Workload

We use two traces to evaluate the effectiveness of Buffered-B+ tree. The first is a "Uniform" workload where keys drawn from a uniform distribution. The second is a "Image Sensor" workload comprising visual terms in an image search engine. This workload is a sequence of features contained in 200 images that were sampled from a camera sensor network for indoor object detection. Each image is first processed using a feature extraction algorithm (SIFT) to extract unique features from the image, which are then mapped to visual words that are indexed [19]. A total of 64KB of memory was given to the index in both cases. The keys to be looked up were drawn randomly from the given trace.

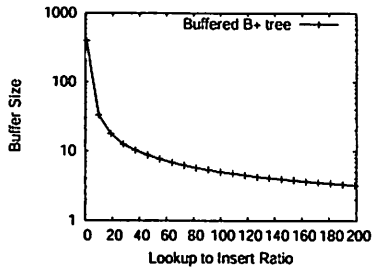
Uniform Workload: Figure 9(a) shows how the performance of Buffered-B+ tree varies gracefully as the workload changes from a write-intensive to read-intensive. The Buffered-B+ tree works as well as the Buffer tree for write-intensive workloads because it amortizes the cost of each insertion heavily by batching. As the fraction of reads increases, the Buffered-B+ tree responds by adapting the buffer size appropriately as shown in Figure 9(b). We see that Buffered-B+ tree quickly realizes that buffers are doing more harm than good and reduces their size rapidly as the number of lookups is ramped up. However, the Buffered-B+ tree doesn't completely turn off the buffers and is always able to amortize insertion costs which helps it to perform better than a B+ tree even in a read-intensive regime.

Sensor Workload: We now evaluate how the Buffered-B+ tree compares to the Buffer tree and the B+ tree for the Image Sensor trace (see Figure 9(c)). We find that the behavior is very similar to that shown in Figure 9(a). The Buffered-B+ tree performs about 4x better than the B+ tree for write intensive workloads and about 20% better for read intensive workloads.

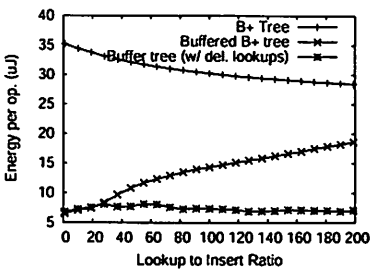
We also ran the Buffered-B+ tree against other sensor traces and find that it gives excellent performance in general. The only cases were the Buffered-B+ tree performs worse than the B+ tree is when the workload is highly temporally correlated. In those cases, caching becomes very effective, and the Buffered-B+ tree's memory allocation policy (only a quarter of the memory to the cache) impacts its performance by about 30%. We believe we can address this problem by dynamically adjusting the split of memory across the buffer



(a) Cost



(b) Adaption of buffer size



(c) Cost with Image sensor trace

Figure 9: Effect of workload

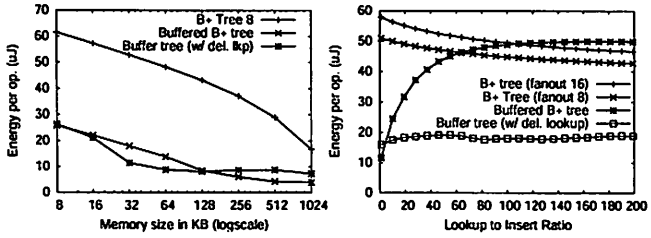


Figure 10: (a) Adaption to memory under write intensive workload (b) Effect of workload under low memory

pool and the node cache, which is a topic of future work.

6.3.2 *Impact of Memory*

In this set of experiments, we vary the memory available to the index structures from 8KB to 1MB (excluding memory given to the Node Table), the former representing Mote-class embedded platforms, and the latter representing PDA-class embedded platforms. Figure 10(a) shows how the performance varies with memory by considering a write-intensive workload with lookup-to-insert-ratio of 5%. To ensure that the index structures don't completely fit within the high amount of memory given to them, we increased the workload size to 1 million keys and started with an pre-constructed tree having 200 thousand keys in it. The results demonstrate that in a write-dominated workload, the Buffered-B+ tree out-performs the B+ tree with optimal fanout by 2.5 - 4x across the spectrum of memory constraints.

We now consider an extremely low memory scenario. Figure 10(b) shows the performance of Buffered-B+ tree for a varying workload in low memory (8KB). The graph shows that the Buffered-B+ tree is better than the optimized B+-tree when the lookup-to-insert-ratio is less than 60%, whereas the cost of Buffered-B+ tree becomes about 5-10% worse than the B+ tree when the fraction of lookups exceeds this point. The primary reason for this behavior is that the Buffered-B+ tree uses a larger fanout of 16 compared to the optimized B+ tree which has a fanout of 8 (for reasons described in Section 6.3.6). At read intensive/low memory settings, the Buffered-B+ tree is very similar to a B+ tree with fanout of 16 since it has very low buffer size. As shown in the figure, this difference in fanout accounts for much of the difference between the two techniques. A second, less significant, reason is because of the difference in the amount of memory given to the node cache. Since the node cache is smaller for the Buffered-B+ tree than the B+ tree, more evictions occur for the former, leading to increased cost. We believe that some of these problems of the Buffered-B+ tree can be rectified if there is prior knowledge of the fraction of lookups, as well as by dynamically adjusting the memory given to the node cache.

These results show that the Buffered-B+ tree is significantly better than the B+ tree across a wide range of workloads and memory constraints.

6.3.3 *Impact on Garbage Collection*

We now look at the impact of different techniques on garbage collection at the storage layer to free space. The impact on garbage collection depends on the number of pages written by a scheme to flash — the more the number of pages written, the greater the number of times that garbage collection is invoked. Figure 8 shows the number of flash pages written for each of the three index structures. The results show that the Buffered-B+ tree uses 3.5x fewer pages on flash than the B+ tree for write-intensive workloads, and is consistently better than the B+ tree with optimal fanout. This shows that the Buffered-B+ tree triggers considerably fewer garbage collection and erase operations.

6.3.4 *Effect of Flash Type*

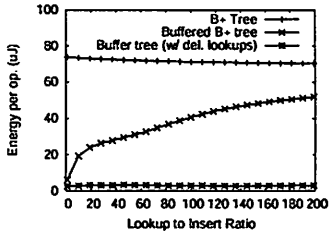


Figure 11: Performance with large block flash

The results that we have shown so far are based on the Toshiba small block flash. We now consider the Samsung large block flash detailed in Table 1. We use response time as the metric since the datasheet does not provide exact energy numbers. This experiment was done with 64KB of main memory. The results are shown in Figure 6.3.4. We find that that the choice of a particular type of flash hardly affects the relative merits of the Buffered-B+ tree, and the result is very similar to Figure 9(a). In fact, the Buffered-B+ tree performs even better for the large block flash, with gains of upto 15x for a write-intensive workload. This is be-

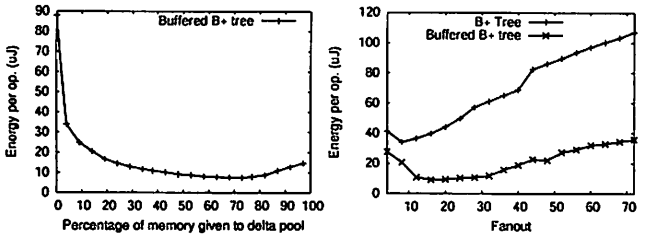


Figure 12: (a) Impact of memory allocation across buffer pool and node cache (b) Effect of fanout

cause the Samsung large block flash has even greater asymmetry between read and write fixed costs, which enables the Buffered-B+ tree to obtain more write-amortization.

6.3.5 Effect of Memory Allocation

We now evaluate the impact of memory allocation between the node cache and the buffer pool for the Buffered-B+ tree. Our experiments until now have used a simple allocation where one 25% the total memory is allocated to the node cache, and the rest of the memory is allocated to the buffer pool. Figure 12(a) shows the sensitivity of the performance of the Buffered-B+ tree to the split across the node cache and the buffer pool, when the lookup-to-insert-ratio ratio was maintained at 25 % and a total of 64 KB was given to the index. As discussed earlier in Section 5, the buffer pool serves to increase the packing of buffer pages on the flash. This packing is quite important, as the more packed a buffer is, the lesser is the cost of reading it from flash. On the other hand, larger the node cache, lower will be the cost of accessing the nodes. Overall, we find that giving more memory to the buffer pool over the node cache is preferable (the minimum in the graph occurs around 75%), hence our decision to allocate 75% of the memory to the buffer pool.

6.3.6 Effect of fanout

Figure 12(b) validates the optimal fanout choices that we made for the Buffered-B+ tree and B+ tree. The two trees were given 64KB of memory and a uniform random workload having 25% lookup-to-insert-ratio. It can be seen that the B+ tree cost is lowest when the fanout is 8, which corresponds to roughly one-tenth of a page. This result can also be shown analytically: in our implementation each key-pointer pair occupies 6 bytes and each node has a header of 4 bytes, thus the size of a node with fanout f is $S(f) = 6 \cdot f + 4$. Hence, the cost of reading the node, as given by Table 1, is $C(f) = 4.07 + 0.105 * S(f)$. Since the tree height is inversely proportional to $\log f$, we wish to minimize the cost of accessing a leaf node which is given as $\frac{C(f)}{\log f}$. It can be shown that this function attains a minima when $f = 8$.

The Buffered-B+ tree has a higher optimum fanout of 16, as compared to the B+ tree. This is because a lower fanout increases the number of subtrees, which translates to an increased number of buffer emptying operations, thereby increasing the cost of having a low fanout. Therefore our implementation of Buffered-B+ tree uses a fanout of 16. Since a subtree should fit within memory, the maximum subtree height h is constrained to be just two levels to fit within 8KB of RAM.

7. RELATED WORK

In this section, we survey related work that we have not discussed previously in this paper (delta-based flash-based data management techniques including BFTL [25], IPL [15], and FlashDB [18] are discussed in Section 6.1).

Flash-Based Indexes and DBMS: The μ-tree [13] tries to minimize the impact of cascading updates on flash in the absence of an FTL. It does so by packing the entire path from the root to the leaf node into a single page on flash. This is orthogonal to the problem that we address of minimizing the accesses to flash. Microhash [26] is a specialized hash table index for flash resident streams. However, its technique of reverse chaining pages belonging to the same hash bucket does not generalize easily to tree based indices. [6] introduces auxiliary data structures for reducing updates to file organizations on portable devices, however they do not support indexes. LGeDBMS [14] demonstrates the DBMS support for mobile phone applications over simulated flash memory, but lacks technical details.

Other Embedded DBMS: PicoDBMS [21] introduces a smart card (EEPROM) based database platform. EEPROM differs from flash in that it supports in-place updates but has very limited storage and extremely high write costs. As a result of which the proposed techniques do not extend easily to flash based index design. Another EEPROM based solution, DELite [22] proposes to incorporate indexes in the data itself to save storage and to allocate memory among query operators for reduced cost.

Flash File Systems: There has been significant work on flash file systems but most of these (e.g., YAFFS [7], JFFS2 [23]) consume too much memory and do not have index-specific optimizations. Similarly, file systems for mote-class sensors such as Matchbox [10] and Capsule [16] do not support complex tree-based indexes. ELF[8], a delta based file system is inefficient for index construction because of reasons described in Section 6.1.

Special-Purpose Tree Indexes. A log structured design for optimizing writes in a disk based B+ tree was proposed in [11]. Log structured write optimizations are ill suited for flash, because of the absence of seek time. LHAM [17] is a special purpose tree like index for write intensive workloads. It works by partitioning the index across multiple trees, the first few of which are kept entirely in memory. In contrast, our work is to build a single B+ tree over flash in an efficient manner.

8. CONCLUSION

In this paper, we presented the design, analysis, and implementation of the Buffered-B+ tree, a flash and memory-optimized index structure designed for embedded systems. The key novelty in the Buffered-B+ tree is its ability to achieve the benefits of buffered updates while still being effective for immediate lookups across a range of workloads. Our work has a number of novel contributions including, (a) a 2-competitive online buffer adaptation algorithm for the Buffered-B+ tree and a simplified heuristic, (b) a full implementation of the Buffered-B+ tree that addresses flash limitations and cost functions, as well as memory limitations on embedded platforms, and (c) a comprehensive evaluation of the Buffered-B+ tree that demonstrates substantial performance benefits across a range of workloads, datasets and memory constraints.

Finally, we are exploring a number of avenues for future work including: (a) integration of the Buffered-B+ tree with a database system, (b) use of the index structure in a camera sensor network search engine, and (c) a randomized version of the online algorithm that we presented in this paper.

9. REFERENCES

[1] Enea polyhedra flashlite. http://www.enea.com.

[2] Reiserfs v4. www.namesys.com/v4/v4.html.

[3] D. Agrawal, G. Mathur, G. Niv, D. Ganesan, Y. Diao, and P. Shenoy. A memory-adaptive flash storage substrate for sensor data management. Technical Report 07-53, UMass Amherst, 2007.

[4] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In *WADS '95*.

[5] L. Arge, K. Hinrichs, J. V., and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *ALENEX 99*.

[6] C. Bolchini, F. A. Schreiber, and L. Tanca. A context-aware methodology for very small data base design. *SIGMOD Records*, 33(1), 2004.

[7] A. O. Company. YAFFS: Yet another flash filing system. http://www.aleph1.co.uk/yaffs/.

[8] H. D., M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys 2004*.

[9] Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy. Rethinking data management for storage-centric sensor networks. In *CIDR 2007*.

[10] D. Gay. Matchbox: A simple filing system for motes. http://www.tinyos.net/tinyos-1.x/doc/matchbox.pdf.

[11] G. Graefe. Write-optimized B-trees. In *VLDB 2004*.

[12] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17, 1982.

[13] D. Kang, D. Jung, J. Kang, and J. Kim. μ-tree: an ordered index structure for NAND flash memory. In *EMSOFT 2007*.

[14] G. Kim, S. Baek, H. Lee, H. Lee, and M. Joe. LGeDBMS: a small DBMS for embedded system with flash memory. In *VLDB 2006*.

[15] S. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD 2007*.

[16] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys 2006*.

[17] P. Muth, P. ONeil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In *VLDB 1998*.

[18] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN 2007*.

[19] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR 2007*.

[20] J. Polastre, R. Szewczyk, and D. E. Culler. Telos: enabling ultra-low power wireless research. In *IPSN 2005*.

[21] P. Pucheral, L. Bouganim, P. Valduriez, and C. Bobineau. PicoDBMS: Scaling down database techniques for the smartcard. *The VLDB Journal*, 10(2-3), 2001.

[22] R. Sen and K. Ramamritham. Efficient data management on lightweight computing device. In *ICDE 2005*.

[23] D. Woodhouse. JFFS: The journalling flash file system. http://sourceware.org/jffs2/jffs2.pdf.

[24] C. Wu, L.-P. Chang, and T.-W. Kuo. An efficient b-tree layer for flash-memory storage systems. In *RTCSA 2003*.

[25] C. Wu, T. Wei, and L. P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Systems*, 6(3), 2007.

[26] D. Zeinalipour-Yazti, S. L., V. K., D. Gunopulos, and W. Najjar. MicroHash: An efficient index structure for flash-based sensor devices. USENIX 2005.

[27] R.M. Karp. On-line algorithms versus off-line algorithms: how much is it worth to know the future?. In Proc. IFIP World Computer Congress, 1992, pp. 416-429.