Multi-Buffer Manager: Energy Efficient Buffer Manager for Databases on Flash Memory

Ulpian Cesana and Zhen He
Department of Computer Science and Computer Engineering
La Trobe University
VIC 3086
Australia
ucesana@gmail.com, z.he@latrobe.edu.au

Embedded devices such as personal digital assistants (PDAs), pocket PCs, palmtops, and handheld PCs are increasingly using flash memory for the permanent storage of databases. Databases achieve their fast data access speeds by using a memory manager which manages data pages in a memory buffer. The buffer manager uses a page replacement policy to evict pages when the memory buffer is full. An eviction of a dirty page will result in a write to flash memory. Unfortunately, writing to flash memory consumes a lot more energy than reading. Much of the previous work in page replacement policies has focused on reducing the number of page reads rather than writes. One of the few existing works to consider the effects of flash memory's hardware constraints for database design is Lee et. al.'s [Lee and Moon 2007] In-page Logging (IPL) approach. They demonstrated IPL significantly outperforms traditional disk-based databases when running on flash memory. However, they do not consider the energy efficiency of their approach in terms of the behaviour of the page replacement policy. This paper addresses this issue by presenting the Multi-Buffer manager which is customized for flash databases that uses a logging-based approach for managing updates such as IPL. Extensive experiments show the page replacement policy used plays a pivotal role in the performance of the flash database system. In particular, our Multi-Buffer manager can reduce energy consumption by up to 40% compared to the state-of-the-art clean first flash-based buffer replacement policy (CFLRU).

Categories and Subject Descriptors: H.2.4 [**Information Systems**]: Database Management—*Systems*; D.4.7 [**Software**]: Operating Systems—*Organization and Design*

General Terms: Performance, Algorithms

Additional Key Words and Phrases: flash memory, flash drive, embedded device, database, page replacement, buffer manager, cache manager, energy efficiency

1. INTRODUCTION

Embedded devices such as personal digital assistants (PDAs), pocket PCs, palmtops, and handheld PCs are becoming ubiquitous in the consumer market. It is likely that there will be an increase in demand for applications such as spreadsheet editors, personal organisers, scientific programs, and financial accounting software that require a database installed directly on embedded devices. Flash memory is ideal for storing data for embedded devices because it is small, light-weight, noiseless, energy efficient, and has excellent kinetic shock resistance.

The typical strategy to achieve fast database access is to have a *buffer manager* to manage a subset of the database pages within a fast *RAM buffer*. Whenever a requested page is not found in the buffer, a *page fault* occurs, which involves loading the requested page from flash memory into the RAM buffer. Whenever a page is updated, the buffer manager updates it within the RAM buffer and marks it as 'dirty'. When the RAM buffer is full and

a page needs to be loaded from flash, a *page replacement policy* is used to select a page for eviction. When a dirty page is evicted, it is written (or flushed) into the flash memory.

Using flash memory for the buffer manager poses a problem for the page replacement policy because each eviction of a dirty data page will force a write to flash memory. Flash memory consumes significantly more energy to perform a write operation compared to a read operation. In addition, writes can only occur in blocks that have been previously erased. A block typically contains 64 pages. Therefore, erase costs must also be factored into the write code when flushing a dirty page. Energy in the embedded device's battery is a limited resource. So, to extend the operation time of the embedded device, the embedded device should conserve as much energy as possible. Since writes are much more expensive compared to reads, the database page replacement policy should bias the eviction of clean pages over dirty pages.

A recent state-of-the-art flash database uses the In-page Logging approach (IPL) presented in Lee et. al.'s paper[Lee and Moon 2007]. It takes advantage of the asymmetric read/write speeds of flash memory to improve the performance of a buffer manager on flash memory. When a data page is updated, the changes are written in the form of log pages instead of overwriting the data page itself. So when the data page is loaded from flash memory, the log pages are applied to the original version of the data page. Loading a data page and its associated log pages consumes energy proportional to the number of pages that are read from flash memory because loading a data page requires loading the data page's associated log pages. *This means different data pages may have potentially different read costs.* Therefore, a buffer replacement algorithm that is customized for a logging-based database such as IPL needs to consider the different potential read cost of data pages as well as the high cost of flushing log pages. Despite this fact, the buffer replacement algorithm used in IPL is just the traditional, least recently used buffer replacement algorithm that does not consider these differences in costs.

Most previous work on buffer replacement has mainly focused on virtual memory systems or database buffering systems. These algorithms do not factor in the asymmetric costs of read and write or the potential for different read costs. However, there has been some recent work that has proposed techniques to develop a flash-aware page replacement policy [Park et al. 2004; Park et al. 2006; Tseng et al. 2006]. The general approach is to prefer evicting clean pages versus dirty pages. These techniques succeed in reducing the energy consumption for virtual memory system running on flash memory. However, they do not consider the fact that data pages can have different read costs in logging-based database systems for flash memory.

This paper presents the Multi-Buffer Manager which solves a class of problem inspired by the IPL approach. The Multi-Buffer Manager breaks up the global memory buffer into a set of local buffers of various sizes and a page replacement policy that discriminates between data pages with a different number of associated log pages and dirty pages. The key idea behind the Multi-Buffer Manager is the *resize formula*. The resize formula computes the optimal local buffer sizes in order to minimize the overall energy consumption for a given workload. The Multi-Buffer manager will dynamically recompute the optimal buffer sizes in order to adjust to changes in the workload.

We have conducted extensive experiments comparing our Multi-Buffer manager to the clean first least recently used algorithm[Park et al. 2006] (CFLRU), a simple variant of CFLRU which we created called lowest log count clean first LRU (LLCCFLRU) and the

traditional least recently used (LRU) buffer replacement algorithm running on a single global buffer. The results show Multi-buffer outperforms CFLRU, LLCCFLRU and LRU in all cases tested. Multi-buffer can reduce energy consumption by up to 63%, 40% and 40% compared to LRU, LLCCFLRU, CFLRU respectively. Although Multi-buffer imposed significantly more buffer replacement overhead than its counterparts, we argue that the large energy savings from flash memory access far outweigh the additional buffer replacement overheads.

2. FLASH MEMORY

Flash memory is small, light-weight, shock-resistant, energy efficient[Chiang et al. 1997], noiseless, and has fast data access speeds. These properties of flash memory make it a suitable high-density storage device for embedded devices[Wu and Zwaenepoel 1994a]. There are two types of flash memory: NAND, and NOR. NAND flash memory is cheaper, has higher storage capacity, and overall, consumes less energy and is faster than NOR flash memory. We have chosen to use NAND flash memory in our experiments.

	Read	Write	Erase
Energy Consumption (μJ/4kb)	9.4	59.6	16.5
Access Time (μs/4kb)	284.2	1833.0	499.2

Table I. Typical energy consumption and access time of NAND flash memory[Park et al. 2006]

The characteristics of NAND flash memory are summarised in Table I. Please note that the erase operation is actually done at the block grain and all the values in the table were scaled to 4KB operations for fair comparison. The energy cost and access time of a write is about six times that of a read and about three times that of an erase.

Flash memory is divided into a sequence of large blocks. Each block is typically divided into 64 pages. The typical size of a page is 2KB. Flash memory starts with all of its blocks erased. The flash memory erase operation erases all the pages within one block. Each block in flash memory has a limited number of erase cycles, between 10,000 and 1,000,000. Once a block has been erased a number of times beyond the erase cycle threshold, it can no longer be used[Wu and Zwaenepoel 1994a]. The flash memory write operation is performed a page at a time[Chiang and Chang 1999]. Flash memory does not support *in-place updates* because before a page can be overwritten, the entire block in which it resides must be erased. The typical strategy to update a page is to write out the updated page to an empty page, and mark the original page as invalid. Eventually, a set of invalid pages will accumulate within blocks, so it becomes necessary for a *garbage collector* to erase these pages to reclaim memory.

The following summarizes the various constraints of flash memory:

Write/Read Ratio Cost. Write operations consume more energy than read operations. This can be observed from Table I.

No In-place Update. It is not possible to overwrite an existing page until that page is erased. So, an in-place update has to be performed as an out-of-place update. However, an out-of-place update may consume a lot of energy. Not only is there the energy cost of writing a page, there could be the additional energy costs of the garbage collector which needs to write out valid pages before erasing a block.

Block Wear. Blocks have a limited number of erase cycles (typically 1,000,000 erases). So any block that fails because of block wear will render it unwritable. This has the consequence that the number of blocks that flash memory can write to will diminish throughout the flash memory's life-time. To avoid blocks becoming unwritable before others, the writes have to be spread across its blocks as evenly as possible. This is called *wear-leveling*[Chiang et al. 1997].

The above summary underlines the importance of reducing the amount of writes to flash memory. In this paper, our buffer replacement algorithm biases retaining dirty pages over clean pages in order to reduce energy consumption.

3. RELATED WORK

We discuss five areas of related work: virtual memory systems, database buffering managers, flash specific buffering managers, the flash based database system IPL and log-structured flash file systems.

Frequency-Based Replacement Algorithm (FBR)[Robinson and Devarakonda 1990], and Least Recently/Frequently Used (LRFU)[Lee et al. 1999; 2001] are examples of page replacement policies for virtual memory that improve LRU. Most improve the hit rate by taking into account reference frequency. SEQuence Replacement Algorithm (SEQ)[Glass and Cao 1997], Early Eviction Least Recently Used (EELRU)[Smaragdakis et al. 1999], and Low Inter-references Recency Set (LIRS)[Jiang and Zhang 2002] page replacement policies attempted to resolve the cyclic pattern problem. Unfortunately, all these policies contain user-defined parameters. The Adaptive Replacement Cache (ARC)[Megiddo and Modha 2003] is an adaptable page replacement policy that solved the sequential scan problem.

DEtection based Adaptive Replacement (DEAR) [Choi et al. 1998] is a database buffer manager that resolves the cyclic pattern and sequential scan problems of LRU. Hot Set Model[Sacco and Schkolnick 1982; 1986], and DBMIN[Cornell and Yu 1989] are examples of database buffer managers that use prior knowledge of the workload to guide the page replacement decisions. The reference pattern behaviour is either derived from an access plan generated by a database query optimiser, or by hints that are supplied by the user. The problem for the first approach is that the overhead of executing a query optimiser on an embedded devices may be prohibitively high. The problem for the second approach is that it may not be possible for a user to know the behaviour of the reference pattern in advance in order to derive the hints.

O'Neil et al. [O'Neil et al. 1993] observed that LRU, which was originally intended for reference patterns in code execution, does not work well for database reference patterns. So they developed LRU-K, a database page replacement algorithm that records the last K references of each page in order to better distinguish between frequently and infrequently accessed pages, and it is able to adapt to a changing workload. Johnson et. al. [Theodore Johnson 1994] developed 2Q which reduced the time complexity of LRU-K to constant time.

All buffer replacement algorithms mentioned above are not customized for the asymmetric read versus write cost of flash memory and also do not consider the fact that in a logging based flash database such as IPL, different data pages can have different read costs.

There has been some recent work on the development of flash-aware buffer managers. Park et al.'s Clean-First Least Recently Used (CFLRU)[Park et al. 2004; Park et al. 2006],

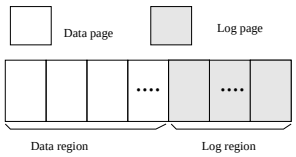


Fig. 1. Flash block layout for IPL.

and Hung-Wei et al.'s HotCache[Tseng et al. 2006] incorporate the asymmetric cost of read and writes of flash memory in their buffer replacement policy. However, these buffer managers do not factor in the fact that different data pages can have different read costs in a logging based flash database such as IPL.

Lee et. al.[Lee and Moon 2007] designed a flash based DBMS that caters for the erase before write characteristic of flash memory by partitioning a block (grain at which data is erased) into a data region and a log region. The data region stores data pages and the log region stores update to the data pages in the form of log pages. They call this approach In-Page Logging (IPL). Figure 1 shows the flash block layout used by IPL. When a data page is updated, a log record is written into a RAM resident log page that is associated with the data. When the log page is evicted from RAM, it is written into the log region of the block its data page resides in. This way, the data pages themselves never need to be written back into the flash memory. Therefore, overcoming the high cost of in-place updates in flash memory. However, when there are no more free pages in the log region of a block, the log region is freed up by erasing the block and writing back the data pages with the log records applied. This is called merging the data region with the log region.

Since data pages are not updated in-place, they can be out-of-date, so whenever a data page is loaded into RAM, all its associated log pages are also loaded in order to reconstruct the updated the data page. This means different data pages can have different associated read costs since different data pages can have different numbers of associated log pages. However, Lee et. al.[Lee and Moon 2007] do not propose any buffer replacement algorithm that is aware of these differences or the fact that writes are much more expensive than reads. In contrast, the approach proposed in this paper address both of these deficiencies in IPL.

IPL can be configured to support recovery. IPL supports recovery by flushing all log pages that are dirtied by a transaction when it commits. Before commit, no log pages are flushed. Upon transaction abort the RAM resident log pages are discarded and the updates to the corresponding data pages are automatically de-applied. Since log pages are not flushed before commit, there is no need to perform an undo during a transaction abort. When merging the data and log pages to free up new space for future log records, the aborted transactions can cause problems since a log record of a transaction that has been aborted must not be merged with the data pages. The problem is overcome by marking log records of aborted transactions so that during merging, those records can be discarded without merging with the data page. During a merge, the log records of the current uncommitted transactions are marked as uncommitted and copied to the newly freed-up log region because we do not know if the current transaction will be aborted or committed. The requirement by IPL's recovery support to flush all log pages at every transaction commit means often many log pages with few update records will be flushed. This results in a high number of page flushes, which is very expensive for flash memory. Therefore, recovery support in IPL will come at a very high energy cost. In this paper, we present

our solution to buffer management in a logging-based flash database such as IPL assuming no recovery support. This allows us to have control over when log pages are flushed and thereby, produce a much more efficient solution. However, our approach can be used with recovery support by controlling the replacement of data pages. This would not allow as much energy to be saved but will still lead to considerable savings in terms of minimising energy used for reading data pages.

There has been a number of log-structured files systems proposed to deal with the operational constraints of flash memory. These file systems include the Journaling Flash File System (JFFS)[Woodhouse 2001] and the Yet Another Flash File System (YAFFS)[Manning 2002]. These file systems organise the entire flash as a log in which flushed dirty pages are appended to the end of the page. The original page that has been dirtied is invalided. A RAM-based buffer is used to keep track of the mapping from logical page id to the current location of the page in the log. This contrasts from log-structured database IPL by considering updates at the coarse page grain as opposed to the finer tuple grain.

Figure 2 shows an example that contrasts the difference between the log-structure database IPL and a log-structured file system. As shown in Figure 2 (a), in IPL, a block is divided into a data region and a log region. When a tuple is updated, a log record is written into an in RAM log page for the block. The log page fills up gradually as other tuples in the same block are updated. This allows three tuple updates from different pages to be grouped together into one log page flush. In contrast, Figure 2 (b) shows the same three tuple updates generating three page flushes for the log-structured file system. The log-structured file system operates solely at the page grain and therefore does not consider separate individual tuples.

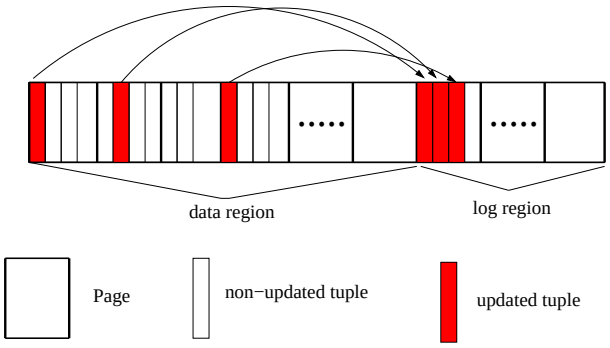
A log-structured buffer manager that addresses the write-erase costs of flash memory was proposed by Jo et. al. called the Flash-Aware Buffer Manager (FAB)[Heeseung Jo and Lee]. It too operates on the page grain, but this works well for the buffer manager because it is intended for use in portable media players that have large blocks of contiguous data. However, it suffers the same problem as IPL in that it does not account for the different read costs of log-page linked data pages.

4. PROBLEM DEFINITION

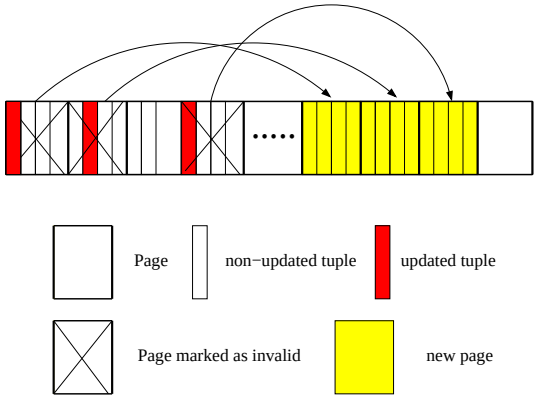
In this paper, we define a class of problem which is inspired by the IPL approach. Suppose we have a logging based database buffer manager that stores data at the page grain. When a data page is updated, the changes made to its contents are recorded by writing log pages to flash memory. The original data page itself is left intact. Whenever an updated data page is loaded from flash memory, its associated log pages are loaded along with it, and the contents of the updated data page are reconstructed by applying the log pages to the original data page.

The problem with this type of buffer manager is that the cost of loading a data page is proportional to the number of log pages that are associated with that data page. The cost of loading an evicted data page that has many log pages is greater than the cost of loading an evicted data page that has few log pages. Therefore, an optimal buffer replacement policy must be aware of the different loading costs for different data pages when deciding which data page to evict. In addition, the optimal buffer replacement policy must also be aware of the cost of evicting a dirty log page.

In this paper, we take a multiple buffer approach to solve this class of problem. The



(a) A block with three tuple updates in IPL



(b) Three updated tuples in a log-structured file system

Fig. 2. Example showing the difference between the log-structure database IPL and a log-structured file system.

approach discriminates between data pages with a different number of associated log pages and the dirty log pages in order to maintain a set of separate *local buffers* of varying sizes. Data pages are stored in the local buffers that have a matching number of associated log pages. Dirty log pages are placed in its own separate local buffer. The *global* buffer consists of all the local buffers. The problem is to find the optimal maximum buffer sizes for the local buffer so that the total expected energy cost of future references is minimized and the global buffer does not exceed a certain total size limit.

We formally define the problem as follows. Let the workload, W, be defined as a set of data and log pages along with their associated stationary probability of being referenced (either updated or read). In addition, for the data pages we also include the number of associated log pages. Therefore, W = {<page id, stationary probability of reference, the number of associated log pages if page is a data page >}. For example, W = $\{< dp_1, 0.6, 3 >, < dp_2, 0.2, 5 >, < dp_3, 0.1, 5 >, < lp_4, 0.1, - >\}$. In this example, W consists of the data page dp_1 with a stationary probability of 0.6 and has 3 log pages associated with it. The example also contains the log page lp_4 with stationary probability of 0.1. The stationary probability for a page p can be computed using the following ratio: number of references to page p / total number of references for all pages. These workload statistics are updated continuously during the on-line phase so the most up-to-date information can be used to find the optimal maximum local buffer sizes.

The total expected energy cost per page reference in W, $TEC(W)$, is defined as:

$$\begin{aligned} TEC(W) \;=\; & \textstyle\sum_{p \in D(W)} (Pr_{miss}(p) \times cost_{miss}(p) \times Pr_{read_{ref}}(p)) \\ & + \textstyle\sum_{p \in L(W)} (Pr_{flush}(p) \times cost_{flush}(p) \times Pr_{update_{ref}}(p)) \end{aligned} \tag{1}$$

where $D(W)$ is the set of data pages in W, $L(W)$ is the set of log pages in W, $Pr_{miss}(p)$ is the stationary probability of a page fault occurring for the data page p, $cost_{miss}(p)$ is the cost for loading a data page p, $Pr_{read_{ref}}(p)$ is the stationary probability of a read reference for data page p, $Pr_{flush}(p)$ is the stationary probability of a page flush occurring for the log page p, $cost_{flush}(p)$ is the cost for flushing a log page p to the flash memory, and $Pr_{update_{ref}}(p)$ is the stationary probability of an update reference for log page p.

Note in the above expect energy cost formulation we assume that the energy consumption for page loads and writes are constant and we do not factor in the energy cost of accessing RAM.

The problem is then defined as finding the maximum local buffer sizes such that the expected energy cost per page reference, $TEC(W)$ for a given workload W is minimized while conforming to the *global* total size constraint.

The global total size constraint states that the total maximum local buffer sizes must be equal to or less than the total RAM available to the global buffer. It is defined as:

$$\sum_{B \in G} (size(B)) - ML = 0 \tag{2}$$

where $size(B)$ is the maximum buffer size of local buffer B and ML is the global memory limit and G is the set of all local buffers. Note $size(B)$ and ML are in terms of number of pages.

In Section 5.4, we minimize Equation 1 under the global total size constraint (2) with re-

spect to energy to derive a *resize formula* that enables us to compute the optimal maximum local buffer sizes.

5. MULTI-BUFFER MANAGER DESIGN

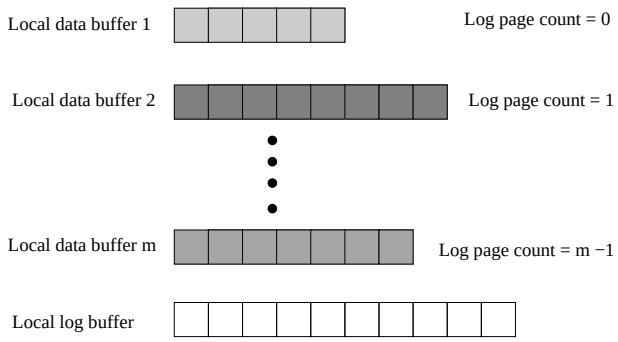


Fig. 3. Multi-Buffer with m local data buffers and one local log buffer

In this paper, we propose the Multi-Buffer manager to solve the problem defined in the previous section, which means it is a buffer manager that is designed to minimize energy usage for log based flash databases such as IPL[Lee and Moon 2007]. To fully understand this section, it is important the reader understands how IPL works (see the seventh paragraph of Section 3 and the first two paragraphs of Section 4). The Multi-Buffer Manager has a *global buffer* which consists of $m+1$ *local buffers*. The first m local buffers are *local data buffers* and the last one is a *local log buffer*. The local buffers can have different maximum sizes (see Figure 3). Having different local buffers with different sizes allows the buffer manager to discriminate between pages based on their associated loading and eviction costs. Pages in smaller local buffers will be evicted more aggressively. The optimal sizes for the local buffers is determined by the resize formula derived in Section 5.4. We define the *maximum global buffer size constraint* which states that the sum of all the pages that reside in all the local buffers is no greater than the maximum global buffer size.

Each *local data buffer* is associated with an attribute called the *log page count*. The log page count is the number of log page loads that are required to load a data page into a particular data local buffer. The cost of loading a data page into a particular local data buffer is proportional to the log page count attribute of the local data buffer. The *local log buffer* stores all log pages that have been updated in RAM and are yet to be flushed into flash memory.

The number of local buffers and the maximum sizes of each local buffer are computed from the reference behaviour of the workload. First, an off-line component of the Multi-Buffer Manager is run by using a previously collected workload in order to generate the necessary statistics to compute the optimal number of buffers and the maximum local buffer sizes. During on-line operation, the local buffers are periodically resized using off-line generated statistics that are partially updated during normal operation. This allows the buffer manager to adapt to changing workloads. The reason that not all the statistics

can be updated dynamically is that some of the statistics require too much computation to update on-line.

A global page replacement algorithm is developed to evict pages from the buffer. It tries to keep the local buffers from overflowing their computed optimal maximum sizes by evicting pages from local buffer that are overflowing. By keeping the number of pages in each local buffer close to the computed optimal maximum size, the total energy cost of reading and writing flash pages is minimized.

5.1 Global Page Replacement Principles

In this paper, the Least-Recently Used (LRU) page replacement policy is used to decide which data page within a local buffer to evict. Note that the Multi-Buffer approach can be used with any existing buffer replacement policy internally within local buffers.

In addition to the internal page replacement within the local buffers, we introduce a *global page replacement policy* (see Section 5.2) which follows the following four principles:

Principle 1. The sum of all the pages across all the local buffers cannot exceed the maximum global buffer size.

Principle 2. The number of pages in a local buffer can exceed the local buffer's maximum size as long as principle 1 holds.

Principle 3. Data pages should always reside in the local data buffer associated with its current log page count.

Principle 4. Log page evictions are triggered solely based on minimizing IO costs considerations.

Principle 1 is a consequence of the maximum global buffer size constraint. The intuition behind this principle is that it allows the Multi-Buffer Manager to resize the maximum sizes for each local buffer, as long as the maximum global buffer size constraint is not violated. The fundamental idea behind the Multi-Buffer Manager is to set the local buffer sizes such that the total energy consumption is minimized. The local buffer sizes are computed by the resize formula which will be described in detail in Section 5.4.

Principle 2 states that local buffers are allowed to overflow. Although overflow is allowed, it is not desirable since it means the maximum buffer sizes that are computed are not being obeyed. So overflow is avoided as much as possible. However, it is not a good idea to avoid overflow in all cases. An example is when a data page is loaded into a local data buffer that is full but the global buffer is not full yet. In such a situation, it would be sub-optimal to evict a data page from the local data buffer when there is still enough RAM globally available for the data page. When the number of pages in a local buffer exceeds its maximum size, then it is said that the local buffer has a *Local Buffer Maximum Size (LMBS) Violation* . In effect, principle 2 states that LMBS violations are allowed as long as the maximum global buffer size is not violated.

Principle 3 states that when a data page's log page count is increased (caused by updates) or decreased (as a result of merging data and log pages for a block), it must be moved to a local data buffer that has a matching log page count. This is important since, if we allow data pages to reside in local data buffers that have a different log page count, then LBMS violation would not be accurate. Note moving pages into different local buffers could cause the target local data buffer to overflow. The overflow is allowed according to principle 2.

Principle 4 states that when deciding whether a log page should be evicted, we only consider IO costs. This gives us full control on when the log pages are flushed which, in turn allows our buffer replacement algorithm the maximum flexibility in reducing IO costs. This principle differs from both the basic and no-force log page eviction policies described in IPL[Lee and Moon 2007]. In the basic policy, log pages must be flushed when its associated data page is flushed. This policy does not allow for recovery since no updated data is forced onto flash when a transaction commits. However, by linking flushing of log pages with data pages we lose the flexibility of flushing log pages at the optimal time in order to reduce IO costs. The no-force log page eviction policy ensures recoverability of the data by flushing all updated log pages when a transaction commits. This, however, generates a lot of write IO since many log pages are flushed when they are still close to empty.

5.2 Global Page Replacement Algorithm

performEviction(G : the global buffer, B: local buffer that the requested page will be placed in)

1. Let the local buffer chosen for eviction be E
2. **if** (at least one local data buffer in G has a LBMS violation)
3. E = local data buffer in G that has a LBMS violation and has the smallest log page count
4. **else if** (local log buffer has a LBMS violation)
5. E = the local log buffer
6. **else if** (B is not empty)
7. $E = B$
8. **else if** there is at least one non-empty local data buffer
9. E = the non-empty local data buffer with the smallest log page count
10. **else**
11. E = the local log buffer
12. **end if**
13. evict a page from E according to the local buffer replacement algorithm
14. **if** E is the local log buffer
// let the evicted log page be e
// let p be the data page associated with log page e
15. flush e into flash memory
16. // let fb be the block that e was flushed into
17. **if** the flushing caused fb to merge its log and data pages
18. place all RAM resident data pages in block fb into local data buffer with log page count 0
19. **else**
20. // let lc be the original log page count of p
21. promote p into the local data buffer with log page count $lc + 1$
22. **end if**
23. **end if**

Fig. 4. Algorithm for evicting a page.

The global page replacement algorithm trims back local buffers that have LMBS violations in order to maintain optimal performance. When the global buffer is full, there may

be one or more local buffers that have more pages than their maximum buffer size (LMBS). The remaining local buffers will be under-utilised. The under-utilised local buffers contain fewer pages than the optimal number as calculated by the resize formula. This has the consequence that the Multi-Buffer Manager would experience sub-optimal performance.

The global page replacement algorithm is presented in Figure 4. It follows the four principles described in Section 5.1. The algorithm is triggered when a page needs to be placed in the buffer. The algorithm first tries to evict pages from the local buffer that have LBMS violation (lines 2 to 5). Among the local buffers that have a LBMS violation, it tries to evict from the one with the lowest cost. Hence, the order it uses to pick the local buffer to evict from is the local data buffer with the lowest log page count followed by the local log buffer. The local log buffer has the highest cost since it requires flushing a page into flash memory. If no page has LBMS violation (lines 6 - 12), then we try to evict from the buffer that the next page will be placed into B (lines 6 - 7). However, if B is empty, then it will try to evict from the lowest cost local buffer, where the cost is again ranked according to the ranking policy used for lines 1 to 4 of the algorithm. When a local buffer is chosen for eviction, a page in the local buffer in evicted according to the local buffer replacement algorithm (line 13). In our experiments, we use the least recently used (LRU) as the local buffer replacement algorithm. Note as a consequence of principle 4, data page evictions do not trigger any log page evictions. Lines 13 - 22 of the algorithm describe what happens if a log page is evicted. When a log page is evicted the log page count of various pages in the buffer may have changed. Therefore, according to principle 3, the affected data pages that are currently RAM resident need to be moved into their new corresponding local data buffers (lines 16 - 21).

In our experiments, the local buffers are resized after every 10000 evictions. Although this is very infrequent, it still generates very good results. The resize is done according to the optimal local buffer resize formula described in Section 5.4. The computation of the resize formula requires a few basic arithmetic operations which, given modern computers have much faster CPU compared to disk, is not a large overhead. Section 7.4 shows an experiment in which we vary the rate at which the local buffers are resized.

5.3 Read and Write Operations

In this section, we describe the algorithm used to handle a data page read and data page write respectively. All the algorithms follow the three principles described in Section 5.1. The subscripts i and j used throughout Figures 5 and 6 denote the page ID and the log page count respectively. This notation will become more clear in 5.4 when we discuss the workload.

Figure 5 shows the algorithm used to handle a data page read. The algorithm is straightforward (lines 1 - 8). At the end of the algorithm (line 9), the statistics used by the resize formula are recomputed.

Figure 6 shows the algorithm for updating a data page. First, the algorithm checks if the data page to be updated resides in RAM. If not, it is first loaded up (lines 1 - 3). When a data page is updated, a log record is generated. The log record needs to be placed into the RAM resident log page allocated to the data page. All data pages that reside in the same flash block have the same allocated log page. This is because all data pages of the same block share the same set of log pages. If there are currently no log pages in the global buffer for the updated data page, then a new one is created (lines 4 - 9). When a log record is written, it is possible that an old log page becomes full and needs to be flushed (lines 13

readPage($p_{i,j}$: the requested data page, B_{D_j} : the local data buffer for the requested data page, G : the global buffer)

1. **if** (data page $p_{i,j}$ is not in G)
2. **if** (G is full)
3. performEviction(G, B_{D_j})
4. **end if**
5. Load data page $p_{i,j}$ from flash memory
6. Load and apply any non RAM resident log pages associated to data page $p_{i,j}$
7. Insert data page $p_{i,j}$ into local buffer B_{D_j}
8. **end if**
9. Update resize statistics // Described in Section 5.4.6

Fig. 5. Algorithm for reading a data page.

- 22). When a log page is flushed, some data pages may need to be moved into a new local data buffer that corresponds to its new log page count. This is done for the same reasons as for lines 14 - 23 of Figure 4. Lastly, the statistics needed for resizing the local buffers are updated.

5.4 Resize Formula

This section presents the resize formula and its derivation. The resize formula is used to set the optimal maximum size of the local buffers. The resize formula is a mathematically optimal solution to the problem defined in Section 4. The formula considers a range of factors such as the probability of page references to local buffers, the probability of cache miss as a function of local buffer size, the total number of pages (either in RAM or in flash) that is associated with the local buffer, the cost of loading or evicting pages from the local buffer, etc. Intuitively, a local buffer will be set a smaller maximum size if the local buffer is not used much, has few data pages associated to it, has the property that a larger size does not bring significant reduction in cache miss rate, and the energy cost of loading or evicting a page associated with the local buffer is small.

We now formally describe the derivation of the resize formula. Let the global buffer, G, be a set of m local data buffers and one local log buffer. Let each local data buffer in G be denoted by B_{D_j}, where the subscript j denotes the log page count associated with the local buffer. Let the local log buffer be denoted by B_L. A data page can only be inserted into a local data buffer if its log page count matches the log page count of the local data buffer. All RAM resident log pages are placed in B_L.

To compute the optimal maximum local buffers sizes, we proceed to reformulate the total expected cost formula (Equation 1) in terms of the local buffers. Before we define the reformulated cost formula we need to provide a new definition of workload W called W' in which page reference statistics are grouped into local buffers. Each data page $p_{i,j} \in W$ belongs to some local data buffer B_{D_j} and each log page p_{L_i} belongs to the local log buffer, so the pages in the ordered set W can be rearranged into groups of pages, W', where each member of the same group in W' belongs to the same local buffer. W' is formally defined as follows:

DEFINITION 1. : *Workload W' consists of a set of m local buffer workloads $\{w'_0, w'_1, ..., w'_{m-1}\}$.*

updatePage($dp_{i,j}$: the data page to update, lp : the log page the update will be stored in,
lr: the update log record, B_{D_j} : the local data buffer for the updated data page,
G : the global buffer)

1. **if** ($p_{i,j}$ is not in G)
2. readPage($p_{i,j}$, B_{D_j}, G)
3. **end if**
4. **if** lp does not exist in G
5. **if** (G is full)
6. performEviction(G, local log buffer)
7. **end if**
8. create a new log page and place it into G, and let lp be a reference to the new log page.
9. **end if**
10. **if** (space left in log page $lp >=$ size of the update log record lr)
11. write log record lr into log page lp
12. **else**
13. flush log page lp into flash memory
14. // let BL($dp_{i,j}$) be the block that page $dp_{i,j}$ resides in
15. **if** flushing log page lp caused block BL($dp_{i,j}$) to merge its data and log pages
16. place all RAM resident data pages in block BL($dp_{i,j}$) into local buffer B_{D_0}
17. **else**
18. promote data page $dp_{i,j}$ into local buffer $B_{D_{j+1}}$
19. **end if**
20. place a new log page nlp into the local log buffer
21. write log record lr into new log page nlp
22. **end if**
23. Update resize statistics // Described in Section 5.4.6

Fig. 6. Algorithm for updating a data page.

Where m is the total number of local buffers in the global buffer and each local buffer workload w_j consists of a set of page reference statistics for pages that belong to the j^{th} local buffer. The first $m-1$ workloads belong to the data local buffers and the final one belong to the local log buffer. $w_j = \{< p_{1,j}, s_{p_{1,j}} >, < p_{2,j}, s_{p_{2,j}} >\}$. Where $p_{i,j}$ is the i^{th} page in local buffer j and $s_{p_{i,j}}$ is the stationary reference probability of referencing page $p_{i,j}$.

For example, suppose the workload is as follows: $W = \{< p_{1,1}, 0.1, 1 >, < p_{2,0}, 0.2, 0 >, < p_{3,1}, 0.1, 1 >, < p_{L_1}, 0.2, - >, < p_{4,1}, 0.1, 1 >, < p_{L_2}, 0.1, - >, < p_{5,0}, 0.2, 0 >\}$. Where W is as defined in Section 4. W can be rearranged into three workload subsets $W' = \{w'_0 = \{< p_{2,0}, 0.2 >, < p_{5,0}, 0.2 >\}, w'_1 = \{< p_{1,1}, 0.1 >, < p_{3,1}, 0.1 >, < p_{4,1}, 0.1 >\}, w'_2 = \{< p_{L_1}, 0.2 >, < p_{L_2}, 0.1 >\}\}$, each of the subsets belongs to a different local buffer. The first subset w'_0 has data page references that belong to buffer B_{D_0}, the second subset w'_0 has data page references that belong to buffer B_{D_1} and the log pages in the third subset w_2 belong to B_L. After the reformulation, Equation 1 becomes:

$$TEC(W') = \sum_{B_{D_j} \in D(W')} (Pr_{miss}(B_{D_j}) \times \sum_{p_{i,j} \in B_{D_j}} cost_{miss}(p_{i,j}) \times Pr_{read_{ref}}(p_{i,j})) + Pr_{flush}(B_L) \times \sum_{p_{L_i} \in B_L} cost_{flush}(p_{L_i}) \times Pr_{update_{ref}}(p_{L_i}) \quad (3)$$

where the symbols used are the same as those for Equation 1 and above. Observe that the loading cost for all data pages in the same local data buffer is the same and the probability of reading from a buffer is equal to the probability of reading all the pages in the buffer. Hence, the second summation that appears in Equation 3 becomes:

$$\sum_{p_{i,j} \in B_{D_j}} cost_{miss}(p_{i,j}) \times Pr_{read_{ref}}(p_{i,j}) = cost_{miss}(B_{D_j}) \times Pr_{read_{ref}}(B_{D_j}) \quad (4)$$

Similarly, all flushing costs and update reference probability for the local log buffer are the same. Hence, the third summation in Equation 3 becomes:

$$\sum_{p_{L,i} \in B_L} cost_{flush}(p_{L,i}) \times Pr_{update_{ref}}(p_{L,i}) = cost_{flush}(B_L) \times Pr_{update_{ref}}(B_L) \quad (5)$$

Substituting Equation 4 and 5 into Equation 3, the reformulated total expected cost formula (TEC') becomes:

$$\begin{aligned} TEC'(W') \; = \; & \textstyle\sum_{B_{D_j} \in W'} (Pr_{miss}(B_{D_j}) \times cost_{miss}(B_{D_j}) \times Pr_{read_{ref}}(B_{D_j})) \\ & + Pr_{flush}(B_L) \times cost_{flush}(B_L) \times Pr_{update_{ref}}(B_L) \end{aligned} \quad (6)$$

The computation of the terms $Pr_{miss}(B_{D_j})$, $cost_{miss}(B_{D_j})$, $Pr_{flush}(B_L)$ and $cost_{flush}(B_L)$ in Equation 6 are each described in the following four sections.

5.4.1 *Computing $Pr_{miss}(B_{D_j})$.* Since the ultimate aim of this section is to set the optimal maximum local buffer sizes, we need to know how different maximum local buffer sizes affect the buffer miss probability (page fault rate). Therefore, we need a miss probability that is a function of local buffer size. In order to produce a function that is bounded on the x-axis between 0 and 1, we set the x-axis as the fraction of data pages in the local buffer (buffer size / total number of data pages).

Figure 7 shows two example miss probability functions. Function a. of Figure 7 shows a typical miss probability function. Observe that initially the miss probability rapidly drops as the buffer size increases, then levels out. This is a result of temporal locality in the reference pattern. However, function b. shows a linear miss probability function which is the case when a random reference pattern is encountered (no temporal locality in the reference pattern). In this paper, we use a miss probability equation that can be adjusted to model both reference patterns of high and low temporal locality. Equation 7 shows the general form of the miss probability function used in this paper. However, it is important to note that this is only an example of a function that can be used. For future work, we plan to explore the suitability of other functions.

$$Pr_{miss}(x_j) = \frac{c_j(1 - x_{D_j})}{a_j x_j + c_j} \quad (7)$$

where a and b are constants which are obtained from off-line training using a training workload. x_j is the fraction of data pages that reside in buffer B_{D_j} and it is given by

$$x_j = \frac{s_j}{t_j} \quad (8)$$

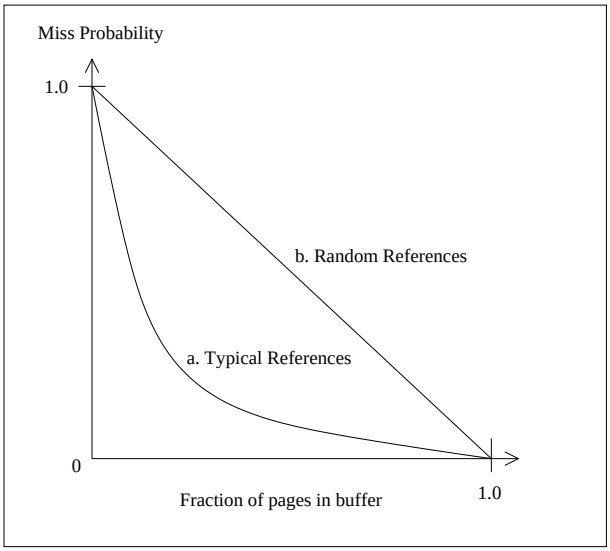


Fig. 7. Typical miss probability vs. fraction of data pages in the buffer function.

where s_j is the maximum size of the buffer B_{D_j}, and t_j is the total number of distinct data page references that can reside in a buffer B_{D_j}. The miss probability function was derived from the general form of the equilateral hyperbola[Middlemin 1955] and solved so that it is constrained to pass through the points (0, 1) and (1, 0). The point (0,1) reflects the fact that when a buffer is empty, there will be a miss probability of 1, and the point (1,0) reflects the fact that when the buffer size is equal to the number of distinct data pages that are loaded into the buffer, then the miss probability falls to zero. Different forms of the miss probability function can be expressed by varying the parameters a_j and c_j in Equation 7. The values of a_j and c_j will depend on the system behaviour and reference patterns. Hence we determine the values of a_j and c_j during the off-line training which is described in Section 5.6.

Since x_j is a function of B_{D_j}, we can express Equation 7 in terms of B_{D_j} instead of x_j as follows:

$$Pr_{miss}(B_{D_j}) = \frac{c_j(x_j - 1)}{a_j x_j + c_j} \tag{9}$$

5.4.2 *Computing* $Pr_{flush}(B_L)$. $Pr_{flush}(B_L)$ is the probability that an update into the local log buffer will trigger a log page flush. This probability depends on size of the local log buffer. A larger log buffer will decrease the probability that a flush will be triggered as a result of a log page update. This is similar to the behaviour of the system in response to cache misses. Hence, we use the miss probability function (Equation 7) to also model this function.

5.4.3 *Computing* $cost_{miss}(B_{D_j})$. $cost_{miss}(B_{D_j})$ is the cost of cache miss when referencing data page $p_{i,j}$ into local buffer B_{D_j}. This cost equals the cost of loading data page $p_{i,j}$ and all of its associated log pages. We express the cost in terms of data buffer B_{D_j} since all data pages in the same data buffer incur the same load cost. The cost is defined

by the following equation:

$$cost_{miss}(B_{D_j} \in W') = (1 + j) \times cost_{read} \tag{10}$$

where j is the number of log pages associated with any data page in buffer B_{D_j} (note this is the same j as that used in the subscript of B_{D_j}), and $cost_{read}$ is the cost of reading a page from flash to RAM. Note that the constant of 1 accounts for loading the data page itself, which must always be done upon a page miss.

5.4.4 *Computing $cost_{flush}(B_L)$.* $cost_{flush}(B_L)$ is the cost of flushing a log page p_{L_i} from the local log buffer B_L. $cost_{flush}(B_L)$ incorporates the cost of a merge. This is because flushing a log page may cause the destination flash block to overflow, which in turn, triggers data and log pages in the destination block to be merged. $cost_{flush}(B_L)$ is expressed as follows:

$$cost_{flush}(B_L \in W') = cost_{write} + \frac{cost_{merge}}{n} \tag{11}$$

where $cost_{write}$ is the cost of writing a page from RAM to flash memory and n is the number of log pages per block. $cost_{merge}$ is the cost of merging the data and log pages in a block. Note the merge cost includes the cost of erasing the block. We divide the merge cost by the number of log pages n since this apportions the merge cost evenly across the log pages of the block.

5.4.5 *Computing $Pr_{read_{ref}}(B_{D_j})$ and $Pr_{update_{ref}}(B_L)$.* Computing $Pr_{read_{ref}}(B_{D_j})$ and $Pr_{update_{ref}}(B_L)$ is straightforward. $Pr_{read_{ref}}(B_{D_j})$ equals the total number of read references to the local data buffer B_{D_j} divided by the total number of read and update references to any page in the entire global buffer. Similarly, we compute $Pr_{update_{ref}}(B_L)$ by dividing the total number of update references to the local log buffer B_L by the total number of read and update references to any page in the entire global buffer. We keep track of the number of read and update references by counting them during the online operation of the buffer manager.

5.4.6 *Online update of statistics .* The following statistics are updated online: $Pr_{miss}(B_{D_j})$, $Pr_{flush}(B_L)$, $Pr_{read_{ref}}(B_{D_j})$ and $Pr_{update_{ref}}(B_L)$. Note when updating $Pr_{miss}(B_{D_j})$ and $Pr_{flush}(B_L)$ we do not recompute c_j and a_j since that would require expensive curve fitting, hence they are computed in the offline training as described in Section 5.6. To update the statistics above, we just need to increment the following three counters: the total number of references, the number of references to each local buffer and the number of distinct pages in each local buffer.

5.5 Computing Optimal Local Buffer Sizes

Before describing the solution for finding the optimal maximum local buffer sizes, we would like to simplify Equation 6. The simplification involves using the same symbols to describe the terms for both local data and local log buffers. After simplification, Equation 6 becomes the following:

$$TEC'(W') = \sum_{B_{A_k} \in W'} Pr_{IO}(B_{A_k}) \times cost_{IO}(B_{A_k}) \times Pr_{ref}(B_{A_k}) \tag{12}$$

where B_{A_k} is the k^{th} local buffer of any type (either local data buffer or local log buffer), $Pr_{IO}(B_{A_k})$, $cost_{IO}(B_{A_k})$, $Pr_{ref}(B_{A_k})$ are defined as follows:

$$Pr_{IO}(B_{A_k}) = \begin{cases} Pr_{miss}(B_{A_k}) & \text{if } B_{A_k} \text{ is a local data buffer} \\ Pr_{flush}(B_{A_k}) & \text{if } B_{A_k} \text{ is a local log buffer} \end{cases}$$

$$cost_{IO}(B_{A_k}) = \begin{cases} cost_{miss}(B_{A_k}) & \text{if } B_{A_k} \text{ is a local data buffer} \\ cost_{flush}(B_{A_k}) & \text{if } B_{A_k} \text{ is a local log buffer} \end{cases}$$

$$Pr_{ref}(B_{A_k}) = \begin{cases} Pr_{read_{ref}}(B_{A_k}) & \text{if } B_{A_k} \text{ is a local data buffer} \\ Pr_{update_{ref}}(B_{A_k}) & \text{if } B_{A_k} \text{ is a local log buffer} \end{cases}$$

Observe that $cost_{IO}(B_{A_k})$, $Pr_{ref}(B_{A_k})$ are constant for each local buffer $B_{A_k} \in W'$. By letting $cost_{IO}(B_{A_k}) \times Pr_{ref}(B_{A_k})$ equal U_k, equation 12 becomes:

$$TEC'(W') = \sum_{B_{A_k} \in W'} Pr_{IO}(B_{A_k}) \times U_k \tag{13}$$

By substituting the miss probability function given in Equation 9 as Pr_{IO} into Equation 13 (since both $Pr_{miss}(B_{A_k})$ and $Pr_{flush}(B_{A_k})$ use the same general probability function), the total expected cost formula is re-expressed as:

$$\sum_{B_{A_k} \in W'} \frac{c_k(1 - \frac{s_k}{t_k})}{a_k \frac{s_k}{t_k} + c_k} U_k \tag{14}$$

To compute the optimal buffer sizes ($\{s_1, s_2, ..\}$) for each local buffer we need to minimize the total expected cost as follows:

$$minimize \left[\sum_{B_{A_k} \in W'} \frac{c_k(1 - \frac{s_k}{t_k})}{a_k \frac{s_k}{t_k} + c_k} U_k \right] \tag{15}$$

Equation 15 is a function in more than one variable $\{s_1, s_2, ..\}$ (note the other terms are constants) and it is to be minimized under the constraint given in Equation 2. Minimising a function $f(\mathbf{x})$ under the constraint $g(\mathbf{x}) = 0$ is done by applying the technique of Lagrange Multipliers. The function f takes its maximum (or minimum) value at a point x_0 if there is a number λ such that:

$$\nabla f(x_0) + \lambda \nabla g(x_0) = 0 \tag{16}$$

The Lagrange Multiplier functions for this problem are as follows:

$$f(s_k) = \sum_{B_{A_k} \in W'} \frac{c_k(1 - \frac{s_k}{t_k})}{a_k \frac{s_k}{t_k} + c_k} U_k \tag{17}$$

$$g(s_k) = \sum_{B_{A_k} \in W'} (s_k) - ML = 0 \tag{18}$$

where s_k is the size of the buffer B_{A_k}.

$\nabla f(s_k)$ yields $m+1$ partial derivatives (m local data buffers and 1 local log buffer) of the form:

$$\frac{\delta f}{\delta s_k} = \frac{\frac{-c_k}{t_k}(c_k + a_k)}{(\frac{a_k}{t_k}s_k + c_k)^2}U_k \tag{19}$$

$\nabla g(s_k)$ yields $m+1$ partial derivatives of the form:

$$\frac{\delta g}{\delta s_k} = 1 \tag{20}$$

Combining Equations 16, 19 and 20 yields $m+1$ simultaneous equations of the form:

$$\frac{\frac{-c_k}{t_k}(c_k + a_k)}{(\frac{a_k}{t_k}s_k + c_k)^2}U_k + \lambda = 0 \tag{21}$$

Solving Equation 21 for s_k yields:

$$s_k = \frac{t_k}{a_k}\left[\pm\sqrt{\frac{\frac{c_k}{t_k}(c_k + a_k)}{\lambda}U_k} - c_k\right] \tag{22}$$

After solving Equation 22 for λ and substituting the result into the constraint (Equation 18), the buffer resize formula becomes the following:

$$s_k = \frac{t_k}{a_k}\left[\frac{\sqrt{\frac{c_k}{t_k}(c_k + a_k)U_k}\left(\sum_{B_{A_k}\in W'}^{W'}(\frac{t_k c_k}{a_k}) + ML\right)}{\sum_{B_{A_k}\in W'}^{W'}\frac{t_k}{a_k}\sqrt{\frac{c_k}{t_k}(c_k + a_k)U_k}} - c_k\right] \tag{23}$$

During normal operation, the multi-buffer manager uses Equation 23 to determine the optimal size of the local buffers. It is important to note that Equation 23 can be computed very fast since the two summations can be computed just once and reused for setting each of the $m+1$ buffer sizes s_k.

The values computed by the resize formula are real numbers, so they are rounded to the nearest whole number. Note that the resize formula may compute negative values for the buffer sizes for some of the local buffers. This means that no data pages should be put into these local buffers. To avoid setting buffer sizes to negative values, we set the maximum buffer size for these local buffers to zero, and re-compute the maximum buffer sizes of the remaining local buffers. However, even after this resize, there is still a possibility of obtaining negative values. So, we resize the local buffers recursively in this manner until each local buffer has either a zero or positive value for its maximum buffer size.

5.6 off-line Training

The resize formula has parameters that need to be generated by performing off-line training. In off-line training, the Multi-Buffer Manager is run with one local buffer using some traditional buffer replacement algorithms such as LRU (or any other page replacement policy that is chosen to be used internally for each local buffer). The single local buffer is

Parameter	Value
Page size	2 KB
Block size	128 KB
Flash memory size	187.5 MB
RAM size	500 KB
Number of transactions	10000

Table II. Simulation parameters used in the experiments

repeatedly run with different buffer sizes to record the miss and flush probabilities as a function of buffer size.

The parameters that are collected during off-line training are as follows:

—The constants a_k and c_k for the IO probability function $Pr_{IO}(B_{A_k})$ for each local buffer.

—The total number of distinct data pages for each local buffer (t_k).

—The probability of buffer references ($Pr_{ref}(B_{A_k})$) for each local buffer.

To compute the constants a_k and c_k, a weighted least-squares fit of the non-linear rational IO probability function (IO probability used to denote both miss and flush probabilities) is used. In order to compute the IO probability for each discrete point on the x-axis, the Multi-Buffer Manager sampled the IO probability at fixed intervals of buffer sizes. Once the data points are computed, the least-squares fit is performed to compute the constants a_k and c_k.

All the parameters used in the resize formula, with the exception of the constants a_k and c_k, are dynamically readjusted during on-line operation. This allows the resize formula to use more up-to-date statistics.

6. EXPERIMENTAL SETUP

This section describes the experimental setup that was used to conduct the experiments for this paper.

6.1 Simulation and benchmark

The experiments were conducted on a simulation of the NAND flash memory device. The simulation was written in C++. The simulation modeled the NAND flash memory characteristics described in Section 2. The flash memory's energy costs of a read, a write and an erase are taken from Table I. The other parameters of the simulation are described in Table II. Unless otherwise specified, the parameter values described in Table II are those used in the experiments.

The experiments were conducted using the TPC-C benchmark. We used the initial database size specified by the benchmark. We simulated all five transactions specified by the benchmark. All experiments involved running 10000 transactions where each transaction is one among the five specified by the benchmark (chosen using uniform random distribution).

6.2 Buffer Manager Settings

All buffer managers tested are built within a slightly modified IPL flash database management system from Lee et. al.[Lee and Moon 2007]. The default parameters specified in the IPL paper were used except we set the default number of log page slots per block to 8. The reason is we found this makes IPL perform much better for our workload than

when the number of log page slots per block was set to 4 (the default IPL setting). We only simulated the non-recovery version of IPL since the recovery version of IPL is highly inefficient for the reasons described in Section 3. As also mentioned in Section 3, our algorithm can still work in recovery mode since we can still control the buffering of the data pages. However, we expect the performance benefits of our algorithm for recovery mode to be significantly smaller since we would have no control on when log pages are flushed. The slight modification we made to the basic IPL algorithm is that instead of allowing a log page to only store log records for a single data page, we allow log pages to store a mixture of log records from different data pages as long as all the data pages belong to the same block. The benefit of this slight modification of IPL is log pages become more full before being flushed, which, in turn, means better cache utilization and less expensive merges. However, the down side is there is potentially more reads since updates for a data page may be spread across more log pages. We believe our modification is an improvement on IPL since page writes cost a lot more than page reads.

The IPL paper does not specify how insertion operations should be handled. We handle insertions as follows. We wait and accumulate inserted tuples until we fill up an entire data page and then write that data page directly into the flash. Therefore, newly inserted tuples are not kept in our buffer. After the data page, has been written into the flash as a new data page we can handle it like any other data page.

We used the greedy garbage collection algorithm proposed by Wu and Willy Zwaenepoel[Wu and Zwaenepoel 1994b]. The greedy algorithm chooses the block with the most number of invalid pages to recycle first. This algorithm was found to perform very well for all but highly skewed data distributions. Testing the effect of different garbage collection algorithms on the performance of our system is an area for future work.

The experiments compare the results of three buffer managers and are described as follows:

Multi-Buffer. The Multi-Buffer Manager proposed by this paper. By default, the local buffers are resized after every 10000 evictions. Each of the local buffers use LRU internally as the buffer replacement algorithm.

LRU. the least recently used buffer replacement algorithm managing a single global buffer.

CFLRU. This is the clean first least recently used (CFLRU) buffer replacement algorithm proposed by Park et. al. [Park et al. 2006]. The algorithm defines a window of the least recently referenced pages. When a page needs to be evicted, the least recently accessed clean page in the window is selected to be evicted. If no clean page exists in the window, then the least recently accessed dirty page is evicted. This policy results in reduced page flushes since clean pages are preferred when making eviction choices. Park et. al. propose both a static and dynamic methodology of defining the window size. Their dynamic algorithm uses a very ad hoc reactive approach which consistently performed worse than their static approach which is trained off-line. We have chosen to implement their static algorithm since we want to compare our algorithm against the better performing variant of CFLRU.

LLCCFLRU. This is an algorithm we have created called the lowest log count clean first LRU (LLCCFLRU). It is a simple variant of CFLRU. Like CFLRU, LLCCFLRU also evicts clean pages before dirty pages from the window of pages. However, it picks the data page with the lowest log count within the least recently used window to evict first. In this

way, data pages that can be reloaded cheaply are evicted before data pages that are more costly to reload. This algorithm was introduced to enable us to compare a simple algorithm that distinguishes between data pages with a different number of log counts over our more complex multi-buffer algorithm.

The off-line training for Multi-Buffer and CFLRU was done using the same workload generator as the on-line testing but a different random seed was used. The different random seed means the workload of the off-line training and on-line testing contains transactions with different parameters and occurring in different orders. In a real system, typically the previous day's transactions would be used training for the next day. Typically, the workload for consecutive days would be similar.

6.3 Off-line curve fitting

This section compares how well the fitting curve (Equation 9) matches the collected training points. Figure 8 shows the results for the first and second local data buffers. The graphs for the other local buffers show similar trends and therefore are not shown. The miss probability for each local buffer was sampled at 100 intervals.

In general, the fitting curve fits well with the training points. The miss probability start with a miss probability of 1 for an empty buffer size is expected. However, the miss probability curves drop dramatically when the buffer size is just a little over 0 fraction of pages in the buffer. The reason for this is that a page can contain a large number of tuples as soon as one or two pages fit in the buffer, hence the relatively large number of buffer hits. Although the fitting curve does not fit the training points perfectly, it does model the training points well, which contributes to the good performance of the Multi-Buffer described in later sections. Finding a better fitting curve is an area of future research.

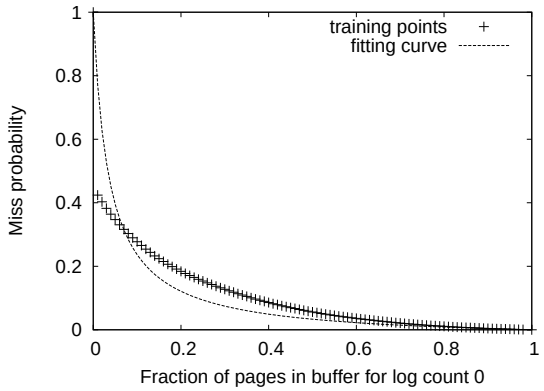
6.4 Hardware

The hardware used was a DELL M1530 notebook with a core 2 duo T9300 2.5 GHz CPU with 4 MB of RAM and a 320GB HDD. The operating system used was Windows Vista.

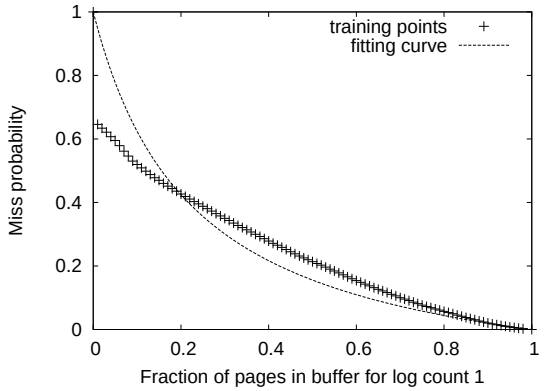
7. EXPERIMENTAL RESULTS

Nine experiments were conducted to compare the performance of multi-buffer versus LRU, CFLRU and LLCCFLRU. The first set of experiments varied the RAM size. The second set of experiments varied the maximum number of log pages per block. The third set of experiments varied the number of transactions. The fourth experiment varied the rate at which the local buffers were resized. The fifth set of experiments measured the fraction of local buffer maximum size (LBMS) violations as the number of transactions varied. The sixth set of experiments measured the number of data page loads based on the number of associated log pages. The seventh set of experiments varied the percentage of training skew. The eighth set of experiments tested the algorithm under read intensive and write intensive workloads. Finally, in the ninth set of experiments, we compared different buffer replacement algorithms in terms of their overhead.

In the experiments, we report the results in terms of the following metrics: number of page reads, page writes and erases; and energy consumption. We do not report execution time results since the read/write/erase cost ratios are almost the same for both execution time and energy consumption, therefore the relative performance of the different algorithms would be almost identical whether we report energy consumption or execution time



(a) Local Buffer B_0



(b) Local Buffer B_1

Fig. 8. off-line training miss probability curve fit for the first two local buffers

results. We choose to report energy consumption because in embedded devices, energy consumption is typically a higher concern for users than execution time.

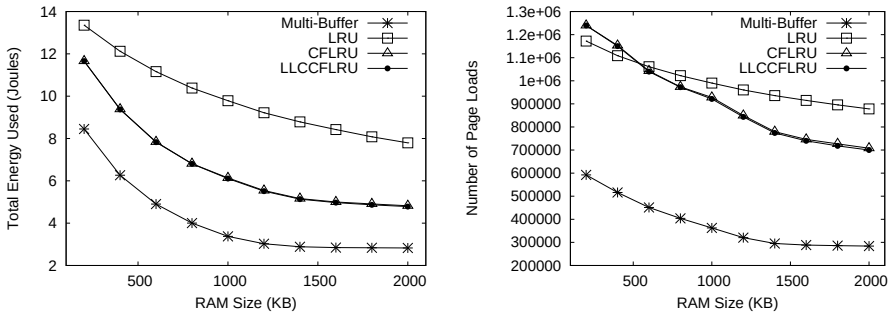
7.1 Varying RAM size Experiment

Figure 9 reports the results of the varying RAM size experiment. The Multi-Buffer manager outperforms the LRU, CFLRU, LLCCFLRU algorithm for all RAM sizes. The Multi-Buffer manager consumes up to 63%, 40% and 40% less energy than LRU, CFLRU and LLCCFLRU respectively.

The Multi-Buffer outperforms LRU, CFLRU and LLCCFLRU in terms of the number of page loads (Figure 9 (c)). The reason Multi-Buffer generates less page loads is that unlike LRU, CFLRU and LLCCFLRU, it has an eviction policy that both tries to keep the

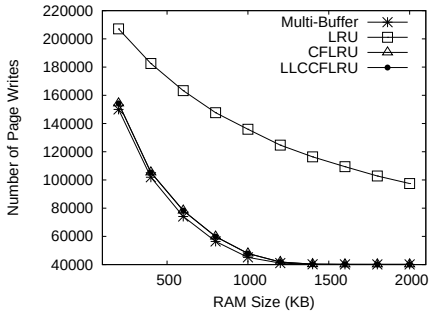
optimal ratio of clean pages versus dirty pages and also the optimal ratio of data pages with small versus large associated number of log pages. In contrast, LRU does not distinguish between clean and dirty pages or data pages with smaller or large log count when choosing pages for eviction. CFLRU and LLCCFLRU are too biased toward evicting clean pages and hence do not retain clean pages in RAM for long enough. Intuitively, LLCCFLRU should outperform CFLRU for the number of page loads since LLCCFLRU tries to retain the data pages with the smallest number of log pages. However, LLCCFLRU performs about the same as CFLRU. This is because both LLCCFLRU and CFLRU are so focused on keeping the dirty pages in RAM that it leaves very little room for clean data pages. The result, is during eviction, there are usually only one or two clean data pages to choose from in the LRU eviction window. Since there is so little choice as to which data page to evict, LLCCFLRU and CFLRU end up performing about the same.

Figure 9 (c) and (d) shows that Multi-Buffer generates a lower number of page writes and erases compared to LRU. This is because of multi-buffer ability to distinguish between the high cost of flushing the log pages and the relatively lower cost of loading a data page from a low ranking data buffer. In contrast, LRU makes no such distinction.

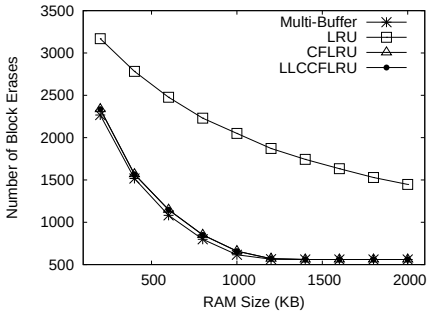


(a) Energy Consumption

(b) Number of Page Loads



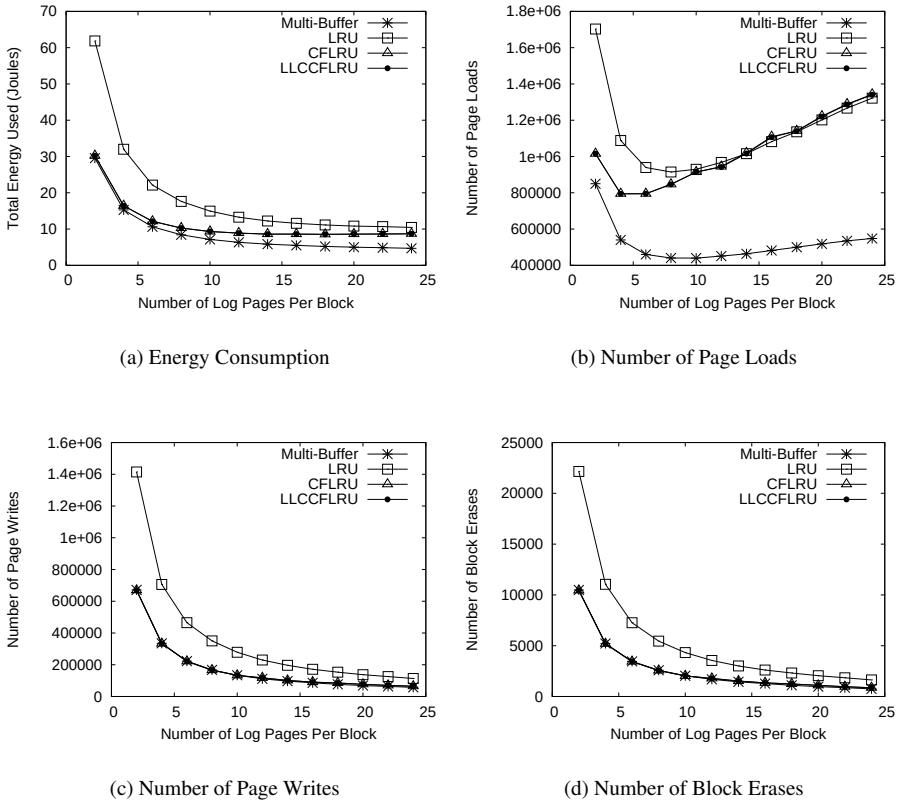
(c) Number of Page Writes



(d) Number of Block Erases

Fig. 9. Varying buffer size results.

7.2 Varying Maximum Number of Log Pages Per Block Experiment



(a) Energy Consumption

(b) Number of Page Loads

(c) Number of Page Writes

(d) Number of Block Erases

Fig. 10. Varying number of log pages results.

Figure 10 reports the results of varying the number of log pages per block. The Multi-Buffer manager outperforms LRU, CFLRU and LLCCFLRU algorithms for the entire range of number of log pages per block. The Multi-Buffer manager consumes up to 50%, 40% and 40% less energy than LRU, CFLRU and LLCCFLRU respectively.

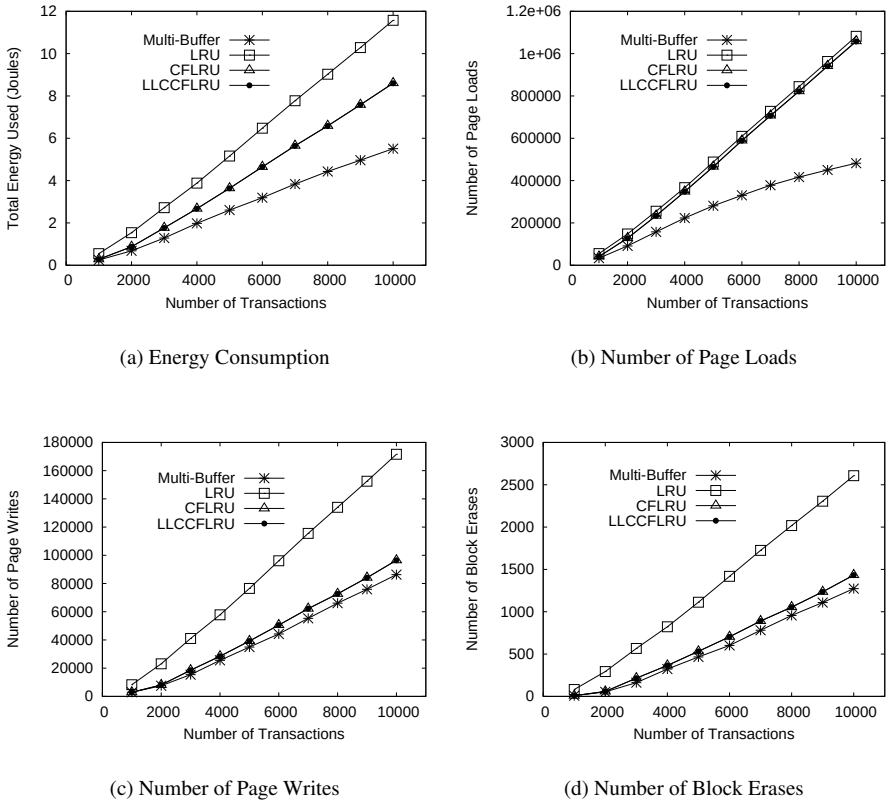
The reasons for the superior performance of the Multi-Buffer algorithms over LRU, CFLRU and LLCCFLRU are the same as the previous experiment. Namely, Multi-buffer tries to keep the optimal ratio of clean pages versus dirty pages and also the optimal ratio of data pages with small versus large associated number of log pages. Also, the factor by which Multi-Buffer outperforms LRU stays relatively equal when varying number of log pages. This shows the superior performance of Multi-Buffer over LRU is robust with respect to different values for the number of log pages per block.

As the number of log pages per block increases, the amount by which Multi-buffer outperforms CFLRU and LLCCFLRU increases. This is because as the number of log pages per block increases, it becomes increasingly more important to favor retaining data pages

in RAM since the cost of loading data pages increases. However, CFLRU and LLCCFLRU evict the data pages before log pages and hence incurs the high cost of reloading the data pages with large log counts. This is evidenced by the increase in read cost as the number of log pages per block increases as shown in Figure 10 (b).

The reason all measure metrics become lower as the number of log pages per block increases is that more log pages per block means less merges of data and log pages in blocks. Merges incur a large amount of page loads, writes and block erases.

7.3 Varying Number of Transactions Experiment



(a) Energy Consumption

(b) Number of Page Loads

(c) Number of Page Writes

(d) Number of Block Erases

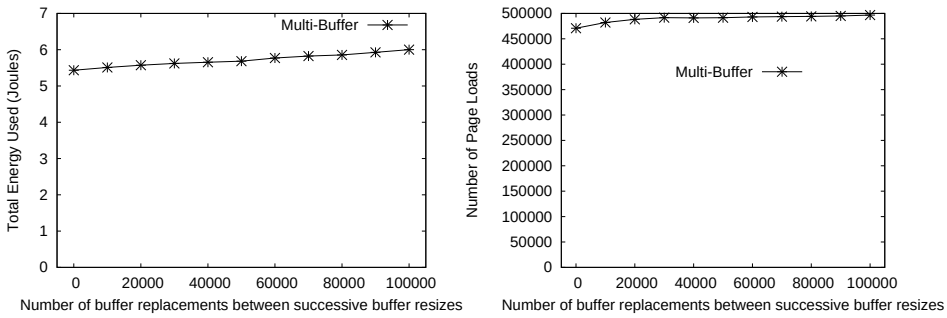
Fig. 11. Varying number of transactions results.

Figure 11 reports the results of varying the number of transactions. The Multi-Buffer manager outperforms the LRU, CFLRU and LLCCFLRU algorithms for the entire range of number of transactions tested. The Multi-Buffer manager consumes up to 50%, 38% and 38% less energy than LRU, CFLRU and LLCCFLRU respectively.

Observe that when there are few transactions, all four buffer managers experience similar performance. However, as the the number of transactions increases, the Multi-Buffer

manager outperforms its competitors at a higher rate. The reason for this is that in the beginning, when the Multi-Buffer has seen few transactions, there will be few log pages. So effectively, the Multi-Buffer runs as if it has a single buffer and so gives similar results to LRU, CFLRU LLCCFLRU. When the Multi-Buffer sees more transactions, data pages will have more associated log pages which, in turn, expands the number of local buffers. So only after a sufficient number of transactions will the Multi-Buffer have enough buffers in order to distinguish between data pages with a different number of associated log pages and dirty log pages. The effect of the increase in the number of local data buffers is most evident in the number of page loads as shown in Figure 11 (b).

7.4 Varying the Resize Rate of Local Buffers



(a) Energy Consumption

(b) Number of Page Loads

(c) Number of Page Writes

(d) Number of Block Erases

Fig. 12. Varying the Resize Rate of Local Buffers. The resize rate is presented on the x-axis as the number of buffer replacements between successive buffer resizes.

Figure 11 reports the results of varying the rate at which Multi-buffer resizes the local buffer. A higher resize rate can be computationally expensive so we wanted to see what happens if we do not resize very often. The x-axis shows the number of buffer replacements between successive buffer resizes.

The results show that performance deteriorates very slowly with the rapid decrease in the rate at which the buffers are resized. In fact, the total energy consumption metric deteriorates by only 10% when we resize after every 100000 buffer replacements. This is because the buffer sizes are initialized to the optimal buffer sizes using the off-line training. The off-line training is very effective at determining the optimal buffer size for a typical workload. As mentioned before, typically database workloads do not change much between successive days. The results are very encouraging since it shows our Multi-buffer replacement algorithm can perform very well with very few resizes. This reduces the computational overhead of our algorithm.

7.5 Fraction of Local Buffer Maximum Size (LBMS) Violations

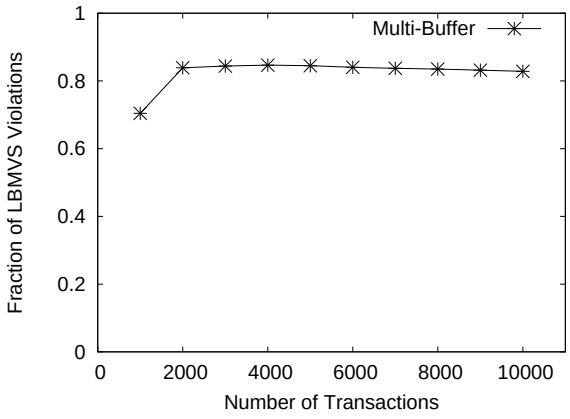


Fig. 13. Fraction of LBMS violations as the number of transactions was varied.

Figure 13 reports the change in the fraction of local buffer maximum size (LBMS) violations as the number of transactions increases. The results show that LBMS violation occurred during a large fraction (0.68 - 0.82) of evictions. This is not surprising since LBMS violations occur when the local buffers fill up at rates different from their maximum buffer sizes as a proportion of the global buffer size. In addition, when a data page's log page count is changed, it is moved into a different local buffer, thus potentially causing an LBMS violation.

7.6 Measure number of data pages loads based on log count

Figure 14 shows the number of data page loads for each category of data page, where category is determined by the number of associated log pages. The results show for data pages with zero log count Multi-buffer performs the same or worse than the other algorithms. However, for data pages with larger log counts, Multi-buffer performs much better compared to the other algorithms. This is due to the use of the cost formula by Multi-buffer to determine the size of the local buffers. Local buffers with lower log counts have low log costs and hence, will be assigned smaller buffer sizes. Data pages belonging to smaller buffers will be evicted and consequently loaded more often. LLCCFLRU performs almost the same as CFLRU. Intuitively, LLCCFLRU should evict more data pages with a smaller

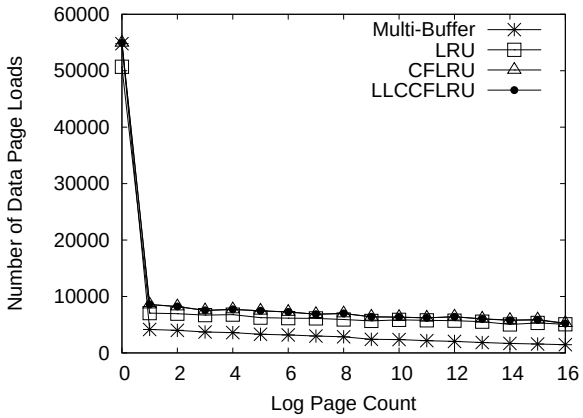


Fig. 14. The number of data pages loaded based on the log count of the data pages.

log count than CFLRU, since it is designed to prefer evicting data pages with lower log counts. Although LLCCFLRU also tries to evict data pages with a lower log count, it tries too hard to keep the dirty pages (log pages) in RAM for as long as possible, due to the use of the clean first policy. This results in a very small portion of clean data pages in the tail LRU window, often just one or two data pages. Since there is so little choice as to which data page to evict, LLCCFLRU performs about the same as CFLRU.

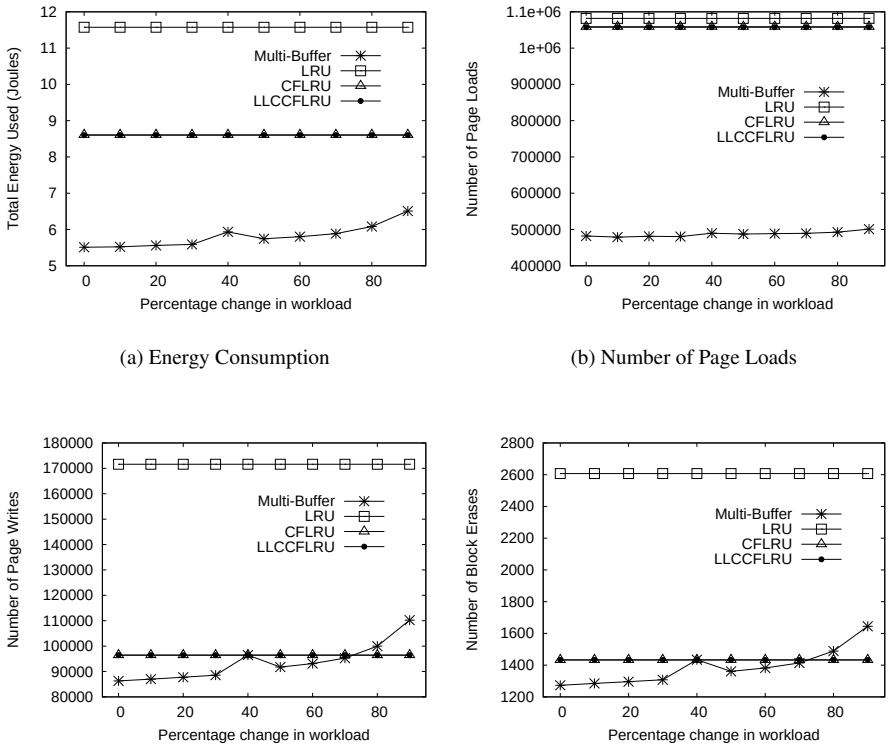
7.7 Varying Training Skew

In this experiment, we varied the percentage of training skew by varying the percentage difference between the transaction mix used for training versus testing. For example, when the percentage difference is at 10%, there was 10% more update transactions in the training workload compared to the testing workload. The results are shown in Figure 15. The results show that Multi-buffer significantly outperforms its counterparts even when there is 90% training skew. This is because the dynamic re-adjustment of the resize statistics used by Multi-buffer is effective at correcting the incorrect statistics obtained from the training phase.

The results show that the training skew affects the amount of writes more than reads for Multi-buffer. When the training skew is greater than 70%, Multi-buffer performs more writes than CFLRU and LLCCFLRU. This is because when the percentage of training skew is high, the training data set has a much higher ratio of read transactions compared to the testing data set. This results in the trained statistics heavily underestimating the probability of flushes due to write versus probability of cache misses due to read.

7.8 Different Read/Write Ratio

In this experiment, we compared the algorithm in two scenarios with contrasting read versus write ratios. Figure 16 shows the results of varying the number of transactions when the workload has 80% read-only transactions and 20% update transactions. Figure 17 shows the results of varying the number of transactions when the workload has 80% update transactions and 20% read-only transactions. The results show that Multi-buffer outperform its counterparts for both update and read intensive workloads. This is an encouraging result



(a) Energy Consumption

(b) Number of Page Loads

(c) Number of Page Writes

(d) Number of Block Erases

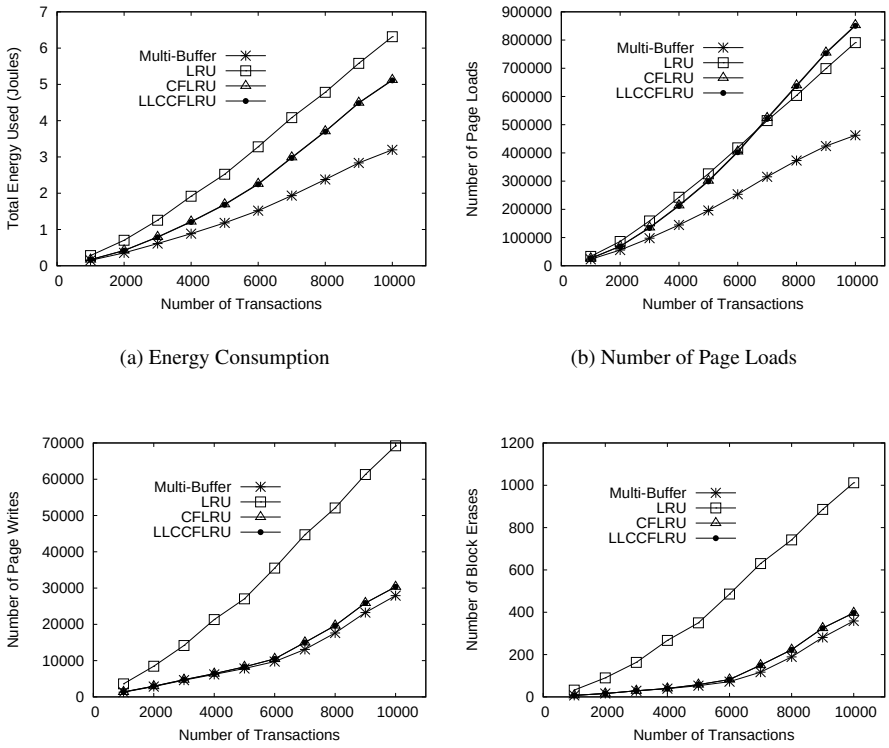
Fig. 15. Varying the percentage of training skew.

since it tells us that Multi-buffer is flexible enough to handle both read intensive and write intensive workloads well.

7.9 Buffer Replacement Overhead

In this experiment, we measured the buffer replacement overhead of the various algorithms. The results are shown in Figure 18. Unsurprisingly, the results show that Multi-buffer has significantly larger overhead compared to its simpler counterparts. However, we argue this extra overhead is a small price to pay for the significant energy savings for flash memory access. Although the overhead of Multi-buffer is around a factor of 10 compared to LRU, this overhead is quite small in absolute terms, since LRU is a very simple algorithm which uses very little computation. This additional overhead of Multi-buffer is easily offset by approximately halving the energy consumption used for flash memory access compared to LRU. Similar arguments can be made when comparing Multi-buffer against the other buffer replacement algorithms.

Figure 18 (a) shows that the relative difference between the total overhead of the different buffer replacement algorithms stay about the same as the number of transactions in-



(a) Energy Consumption

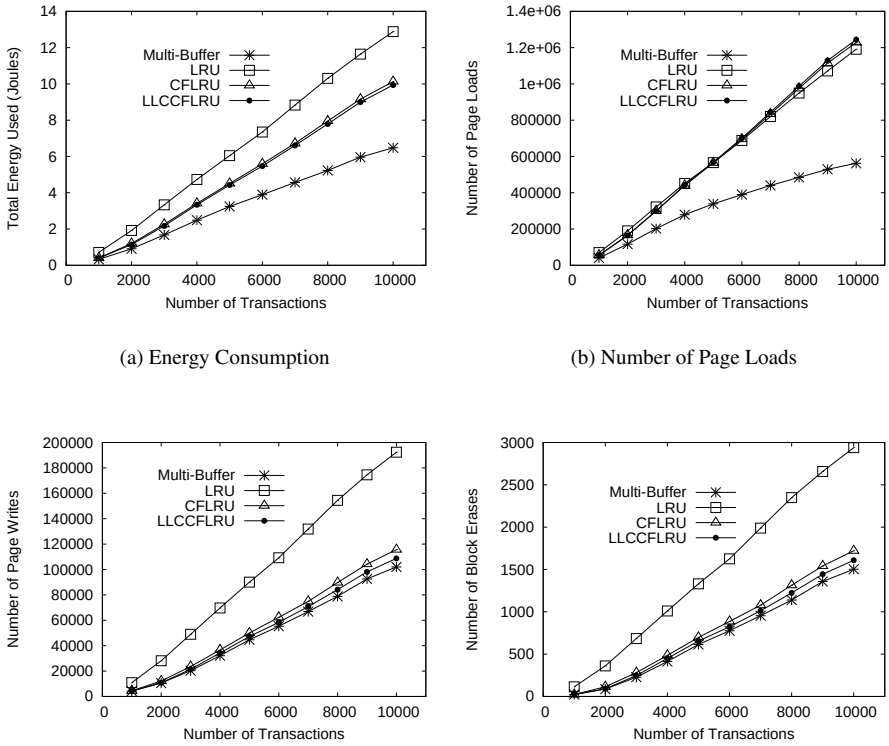
(b) Number of Page Loads

(c) Number of Page Writes

(d) Number of Block Erases

Fig. 16. The results of varying the number of transactions when the workload has 80% read-only transactions and 20% update transactions.

creases. However, Figure 18 (b) shows that the average overhead per miss of Multi-buffer (total overhead divided by the number of misses) increases with the number of transactions whereas the others stay relatively flat. The reason for the discrepancy between these two graphs is that the majority of the overhead incurred by Multi-buffer occurs when pages are moved from one local buffer to another, this is not dependent on the number of buffer misses. As the number of transactions increases the number of buffer misses decreases since the buffer becomes populated with pages that are likely to be reused in the future. Hence as the number of transactions increases the overhead per transaction remains constant because the number of misses decreases and the overhead per miss increases. At the end what really matters is Figure 18 (a), because it shows the total overhead as the number of transactions increases. For that graph the relative difference between the different algorithm stay about the same as the number of transactions increases, hence our algorithm do not become increasingly worse compared to the other buffer replacement algorithms as the number of transactions increases.



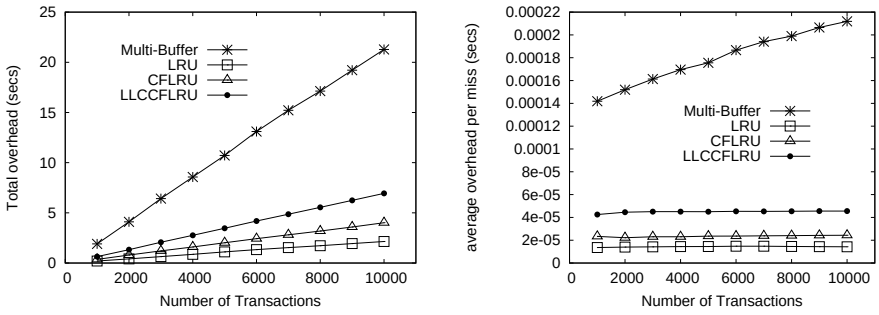
(a) Energy Consumption

(b) Number of Page Loads

(c) Number of Page Writes

(d) Number of Block Erases

Fig. 17. The results of varying the number of transactions when the workload has 80% update transactions and 20% read-only transactions.



(a) Total overhead

(b) Average overhead per miss

Fig. 18. The buffer replacement overhead as the number of transactions is varied.

8. CONCLUSION

This is the first paper that is focused specifically on addressing the buffer management problem for logging based databases that run on flash memory. The experimental results show the large potential gains in performance when a customized buffer manager for flash memory is used. The novelty in our approach stems from the fact we recognize that different pages in the buffer have different read and write costs. The difference in the costs stems from the logging nature of the flash databases. In order to incorporate the different costs of reading and writing pages, we divide the global buffer into a set of local buffers where each local buffer contains pages of the same cost. We then give preference to some local buffers with respect to others by adjusting the maximum size of local buffers according to an optimal local buffer size formula. The optimal local buffer size formula is derived from a sound mathematical analysis of the total read and write costs for a workload that represents the recent behavior of the system.

The Multi-Buffer Manager's performance was compared to CFLRU, LRU and LLC-CFLRU (log count sensitive variant of CFLRU). All these algorithms were running on top of IPL. Extensive experiments have demonstrated that the Multi-Buffer Manager performs the same or better than CFLRU, LLCCFLRU and LRU for all settings tested by up to 40%, 40% and 63% respectively for energy consumption. Although Multi-buffer imposed significantly more buffer replacement overhead than its counterparts, we argue that the large energy savings from flash memory access far outweigh the additional buffer replacement overheads. We have presented the evidence on the relative merits of Multi-buffer versus other simpler buffer replacement algorithms; it is up to the reader to decide if this algorithm will be the best for their application.

An area of future work, as mentioned earlier, is to find an even better miss probability fitting curve. Using higher-order polynomial functions to fit the curve may produce a better fit. Another area of future work is to adopt Multi-buffer to work for a logging-based database designed to use a combination of flash memory and hard disk drive. Currently, multi-buffer uses LRU within each local buffer to choose the particular page to evict when a particular local buffer is chosen for eviction. Another area of future work is to use different buffer replacement algorithms within the local buffers. Finally, testing the effect of different garbage collection algorithms on the performance of our system is an area for future work.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and suggestions which have greatly improved the quality of this paper. This work is supported under the Australian Research Council's Discovery funding scheme (project number DP0985451).

REFERENCES

CHIANG, M. L. AND CHANG, R. C. 1999. Cleaning policies in mobile computers using flash memory. *The Journal of Systems and Software 48*, 213–231.

CHIANG, M.-L., LEE, P. C. H., AND CHANG, R.-C. 1997. Managing flash memory in personal communication devices. In *Proceedings of 1997 IEEE International Symposium on Consumer Electronics*. Seattle, WA, USA, 177–182.

CHOI, J., NOH, S. H., MIN, S. L., AND CHO, Y. 1998. An adaptive block management scheme using on-line detection of block reference patterns. In *Multi-Media Database Management Systems, 1998. Proceedings. International Workshop*. 172–179.

CORNELL, D. W. AND YU, P. S. 1989. Integration of buffer management and query optimization in relational database environment. In *Proceedings of the Fifteenth International Conference on Very Large Databases*. Amsterdam, The Netherlands, 247–255.

GLASS, G. AND CAO, P. 1997. Adaptive page replacement based on memory reference behaviour. In *ACM SIGMETRICS Conference*. 115–126.

HEESEUNG JO, JEONG-UK KANG, S.-Y. P. J.-S. K. AND LEE, J. FAB: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics 52,* 2.

JIANG, S. AND ZHANG, X. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Conference*.

LEE, D., JONGMOO CHOI, J.-H. K., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, S. 1999. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS Conference*. Atlanta, Georgia, USA, 134–143.

LEE, D., JONGMOO CHOI, J.-H. K., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, S. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers 50,* 2, 1352–1361.

LEE, S. AND MOON, B. 2007. Design of flash-based dbms: An in-page logging approach. In *SIGMOD*.

MANNING, C. 2002. *YAFFS (Yet Another Flash File System)*. Aleph One Ltd.

MEGIDDO, N. AND MODHA, D. S. 2003. ARC: A self-tuning, low overhead replacement cache. In *USENIX File and Storage Technologies Conference (FAST)*.

MIDDLEMIN, R. R. 1955. *Analytic Geometry*, 2 ed. McGraw Hill Book Company, Inc.

O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. 1993. The LRU-K Page replacement algorithm for database disk buffering. In *Proceedings ACM SIGMOD Conference*. 297–306.

PARK, C., KANG, J.-U., PARK, S.-Y., AND KIM, J.-S. 2004. Energy-aware demand paging on NAND flash-based embedded storages. In *Proceedings of the 2004 international symposium on low power electronics and design*. 338–343.

PARK, S., JUNG, D., KANG, J., KIM, J., AND LEE, J. 2006. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on compilers, architecture and synthesis for embedded systems*. 234–241.

ROBINSON, J. T. AND DEVARAKONDA, M. V. 1990. Data cache management using frequency-based replacement. In *Proceeding of 1990 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*. 134–142.

SACCO, G. M. AND SCHKOLNICK, M. 1982. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the Eigth International Conference on Very Large Databases*. 257–261.

SACCO, G. M. AND SCHKOLNICK, M. 1986. Buffer management in relational database systems. *ACM Transactions on Database Systems 11,* 4 (December), 473–498.

SMARAGDAKIS, Y., KAPLAN, S., AND WILSON, P. 1999. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 122–133.

THEODORE JOHNSON, D. S. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*. 439–450.

TSENG, H., LI, H., AND YANG, C. 2006. An energy-efficient virtual memory system with flash memory as the secondary storage. In *Proceedings of the 2006 international symposium on low power electronics and design*. 418–423.

WOODHOUSE, D. 2001. JFFS: The journaling flash file system. *Ottawa Linux Symposium*.

WU, M. AND ZWAENEPOEL, W. 1994a. eNVy: A non-volatile, main memory storage system. In *Proceedings of the sixth international conference on architectural support for programming languages and operating systems*. 86–97.

WU, M. AND ZWAENEPOEL, W. 1994b. eNVy: A non-volatile, main memory storage system. In *Proceedings of ASPLOS*. 86–97.