Write Activity Reduction on Flash Main Memory via Smart Victim Cache

Liang Shi1,2,3, Chun Jason Xue2,3, Jingtong Hu4, Wei-Che Tseng4, Xuehai Zhou1,2 and Edwin H.-M. Sha4
1Department of Computer Science and Technology, University of Science and Technology of China, Hefei, China.
2Joint Research Lab of Excellence, CityU-USTC Advanced Research Institute, Suzhou, China.
3Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong.
4Department of Computer Science, University of Texas at Dallas, Richardson, TX, 75080.
shil0704@mail.ustc.edu.cn, jasonxue@cityu.edu.hk, {jxh068100, wxt043000, edsha}@utdallas.edu, xhzhou@ustc.edu.cn

ABSTRACT
Flash Memory is a desirable candidate for main memory replacement in embedded systems due to its low leakage power consumption, higher density and non-volatility characteristics. There are two challenges in applying flash memory as main memory. First, the write operations are much slower than read operations. Second, the lifetime of flash memory depends on the number of the write/erase operations. In this paper, we introduce a smart victim cache architecture to reduce the write activities by exploring the coarse grain accessing character of NAND flash memory. Experimental results show that the proposed approaches can reduce write activities on flash main memory by 65.38% on average compared to traditional architecture.

Categories and Subject Descriptors:B.3.2[Memory Structures]: Design Styles–Cache memories, Primary memory
General Terms:Design, Performance, Experimentation
Keywords:Cache, Victim cache, NAND flash Memory, Main memory

1. INTRODUCTION
Flash memory has the characteristics of low-cost, non-volatility, shock-resistivity and higher density, which are attractive to embedded systems [7, 4, 3, 10, 8, 9]. Recently, many researchers have extend NAND flash memory as main memory in embedded systems [9, 10]. This paper focuses on embedded systems employs flash memory as main memory. We propose a smart victim cache [6] to reduce the number of writes on flash memory in order to extend flash memory’s lifetime.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GLSVLSI’10, May 16–18, 2010, Providence, Rhode Island, USA.
Copyright 2010 ACM 978-1-4503-0012-4/10/06 ...$10.00.

NAND flash memory’s page size is much larger than common cache block size. Most of consecutive write operations take place in a small set of pages, which can be exploited to reduce the write activities. Currently, most computer systems have cache block size of 32 bytes, 64 bytes, or 128 bytes. However, NAND flash memory read and write in the unit of pages, and the page size varies from 512 bytes to 4KB [5], which is large compared to cache block size. By analyzing the write operations of a set of applications, we found that in a set of consecutive write activities, most of the data comes from limited number of pages of flash memory, which is shown in Figure 1. For every 10 consecutive

Figure 1: Number of unique pages in 10,20,30,40 contiguous write operations.
write operations, there are only 5.03 unique pages written on average. For every 40 consecutive write operations, there are only 14.3 unique pages written, which means in average there are 2.8 write operations takes place in the same page.

In this paper, we propose a smart victim cache to achieve the goal of write activity reduction. We efficiently manage the victim cache to write the cache blocks to flash memory in batches. From the experiments showing, the proposed approach is shown to be able to efficiently reduce 65.38% of write activities on flash memory compared to the baseline architecture with traditional victim cache on average.

2. ARCHITECTURE AND MECHANISMS
Figure 2 shows the proposed cache architecture. The smart victim cache architecture consists of 2 parts: victim cache and write buffer. Victim cache is used to keep the dirty data from the last level cache (LLC), which is organized similar to traditional victim cache, but with an
additional bit: \textit{replace}. The proposed cache organization tolerates an order of magnitude slower write latency of flash memory using a write buffer, and overcomes the endurance limit of flash memory using techniques to reduce the number of write activities to the flash memory. The write buffer keeps batches of lines from victim cache in queue based on the address of the lines. These mechanisms are discussed in detail in the next few sections.

2.1 Batch Write

We propose to write a set of cache lines of victim cache to write buffer in batch. All the cache lines in the set come from the same page of flash memory. In order to reduce write activities to main memory, we use a smartly managed small victim cache. In our approach, only dirty blocks are kept in the victim cache, which can fully utilize the dirty blocks before the dirty blocks being written back to the main memory. Smart victim cache also delays the write activities of the dirty blocks, which allows us to exploit the ordering of write activities for reduction.

On a cache eviction in the last level cache (LLC), the evicted line would be written to the victim cache. If the victim cache has free lines, the evicted line would be placed in one of the free lines. In the case that the victim cache is full, a set of cache lines belong to the same page would be selected, which would be written to write buffer in a batch. A set of lines need to be written to write buffer, which is time consuming. We add a tag in each victim cache line: \textit{replace}. This tag shows that this cache line is replaceable. When a set of cache lines need to be written back, the replaceable tag would be set. In this case, when a cache line from LLC need to write to victim cache, if a line in the victim cache is replaceable, we can write it back to write buffer. These replaceable lines can be written back to write buffer once the first line is evicted to write buffer in parallel with the application’s execution. When the whole set of the lines has finished written to the write buffer, these data would be written to flash memory. Smart victim cache also performs the same function of traditional victim cache, which is providing additional hits for the LLC.

2.2 Victim Cache Management

In previous section, batch write back needs to select a set of cache lines from the same page to set the replace tag and write back them to write buffer. In this section, we discuss three management techniques of victim cache on how to efficiently manage the victim cache to reduce write activities by careful selection of the set of cache lines.

A Max lines-based approach This approach is based on the max lines in victim cache, which write the lines to write buffer that belong to the page having the maximum number of lines in the victim cache. This approach needs an array to support, which records the line counts information for each page. When no free or replaceable lines in the buffer, the page number with the max count in the array is identified, and the corresponding lines’ replace tag in victim cache is set. All of these lines would be written to write buffer in a sequence. Figure 3(a) shows the organization of the victim cache and counter array. The counter in the array is organized in descending order. Every time, we select the largest number in the counter array and the corresponding page number in the array. All the cache lines with the same page number would have its replace tag set to 1.

B History-based approach This approach is based on the least recently used (LRU) policy, which writes the lines to write buffer that belong to the page having a cache line in the LRU position of the victim cache. Traditionally, victim cache is managed by LRU policy. LRU can best reflect the character of cache accessing. In the second approach, we also apply LRU as the replacement mechanism. Since we need to select a set of lines in victim cache, which is different from traditional victim cache, we extend LRU policy in this second approach. When selecting the set of lines to write back to the write buffer, we select the cache line in the LRU position first, and then set the replace tag for all the cache lines with the same page number as this selected line. First, the evicted line of LLC would be placed in the LRU position of the victim cache, and then be moved to MRU, followed by all the replaceable lines been written to write buffer sequentially. Note that even the data has been written to write buffer, these data would still be kept in the victim cache until they are replaced by newly evicted lines from LLC. Figure 3(b) shows the organization of the victim cache. The last line of victim cache is selected, the cache lines in victim cache with the same page number would have their replace tag set. This approach can reduce the write activities by only write back data that have been least recently used and does not need a counter array. As applications often exhibit spatial locality, the line in LRU position should correspond to the page data usage pattern.

C Enhanced History-based approach Enhanced approach can further reduce the write activities by potentially writing the evicted line from LLC to write buffer. History-based approach can further reduce the write activities by potentially writing the evicted line from LLC to write buffer. History-based approach can further reduce the write activities by potentially writing the evicted line from LLC to write buffer. History-based approach can further reduce the write activities by potentially writing the evicted line from LLC to write buffer. History-based approach can further reduce the write activities by potentially writing the evicted line from LLC to write buffer.
used line would be selected to be replaced and be written back to write buffer. In this third approach, we enhance the history-based approach by first comparing the page tag of the evicted line from LLC and the selected line in LRU position of victim cache. If the page tag is equal to each other, which means they are coming from the same page of flash memory. Then the evicted line’ replace tag would be set, which would be written to write buffer with current batch, if not, it would follow the history-based approach to replace the lines and move forward to the MRU position.

2.3 Write Back to Flash Memory

Data in the write buffer is flushed to flash memory in batches. A whole page of flash memory is updated once by multiple write back operations, which can reduce the write operations on flash memory. From the last section, dirty data have been placed in the write buffer queue based on the address of page in the flash memory, and then batches of write operation to flash memory are performed.

In order to avoiding write buffer overflow, a signal is needed to indicate whether a batch of data has been finished flushing to flash memory. When writing operation performed, this signal is set, and when the whole batch of data has been flushed to flash memory, this signal is reset. Then the data in the write buffer can be overwritten.

When a set of lines are written to the buffer, it only uses one batch size to keep the data (the batch size is equal to the size of victim cache). Furthermore, the write buffer can provide additional hits for cache miss.

3. EXPERIMENTS AND ANALYSIS

3.1 Experimental Framework

We use the SimpleScalar toolset [2] to perform our experiments under various configurations. We model read and write operation timing characteristics based on SAMSUNG K9F8G08U0M [5] NAND flash memory. We simulate a set of programs from SPEC2000 [1]. Table 2 outlines the benchmarks used for the experiments. In Table 2, Column Pages Written shows how many pages written of the applications running in our experiment.

<table>
<thead>
<tr>
<th>Application</th>
<th>Type</th>
<th>Pages Written</th>
<th>Application Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bzip</td>
<td>Compressor</td>
<td>104</td>
<td>320KB</td>
</tr>
<tr>
<td>Gcc</td>
<td>Compiler</td>
<td>381</td>
<td>3.6MB</td>
</tr>
<tr>
<td>Gzip</td>
<td>Compressor</td>
<td>64</td>
<td>368KB</td>
</tr>
<tr>
<td>Parser</td>
<td>Parser</td>
<td>896</td>
<td>584KB</td>
</tr>
<tr>
<td>Vortex</td>
<td>database</td>
<td>1047</td>
<td>2.23MB</td>
</tr>
<tr>
<td>Vpr</td>
<td>CC/AF Program</td>
<td>554</td>
<td>640KB</td>
</tr>
</tbody>
</table>

3.2 Reduction of write activities

The max lines write approach is shown to have more write activities than the history-based approach. The max lines counter only considers the number of lines in victim cache, neglecting the history information of lines in the victim cache. For the history based approach, which fully considers the history of the lines in victim cache, it only evicts the lines that have been least recently used in the victim cache. In most case, these lines would never be reused again, which can efficiently reduce write activities. Figure 4 shows the write activities reduction compared to the baseline approach. The enhanced history-based approach can further reduce the write activities, because the approach can dynamically write back the newly evicted lines in the current batch of write backs. On average, there are 65.38% of write activities reduced using the proposed approaches. Figure 5 shows the max write count reduction of the page in flash memory. This is common to the reduction of write activities. Our history-based approach can efficiently reduce the max write count: First, batch of write back can significantly reduce the write activities on each page. Second, history based approach only write the page that has been least recently used in victim cache, and free more lines for the newly evicted line. The max written counts are reduced by 70% on average. For Parser, max line-based approach always select the data segment pages to write and neglect the history information of the lines, which make the data segment pages been written more times than the basic configured approach.

Figure 4: Write activities reduction normalized to the baseline.

Figure 5: Max write count reduction normalized to the baseline.

3.3 Impact of the LLC configuration

Last Level Cache (LLC) ways are associated with the speed of cache, cache miss rate and hardware overhead. Figure 6 shows the reduction in write activity by keeping the cache size constant and vary the ways of the cache. From the results, we can see that the number of write activities increases as the number of ways increases. From 1-way to 8-way associated cache, the number of write activities has been increased by 26.2%. Many applications have little conflict misses, for a high way associated configuration, there would be no benefit. Furthermore, we use a victim cache, which can provide additional hits. Because embedded systems have the constraints of speed and size, using a low associated LLC not only provides high speed, but also with small die area.

3.4 Impact of the number of victim cache lines

Figure 7 shows the impact of the number of victim cache lines on the reduction of write activities. The proposed victim cache is used to keep the dirty lines from LLC, and similar to traditional victim cache, the victim cache can provide...
additional hits for LLC, which will improve the performance of the system especially when there are conflict misses in LLC. Another benefit for large line number of victim cache is that victim cache can keep more dirty data with a larger address space, and then the history-based approach can benefit with selecting more lines written to flash memory in a batch. From Figure 7, the number of write activities is significantly reduced when the number of victim cache lines is increases to 32.

![Figure 6: Reduction of Write Activities when the way of LLC is varied.](image1)

![Figure 7: Reduction of Write Activities when the number of victim cache lines is varied.](image2)

### 3.5 Delay and write buffer size analysis

In this section, we analyze how to use the write buffer to reduce the write delay. As shown in Figure 2, the LLC first places the dirty data in victim cache. This operation needs $T_{v\rightarrow w}$ cycles. We assume that a write operation takes place in every $T_w$ cycles (50ns/cycle). Data in victim cache are written to write buffer in batches. We assume there are $N$ lines written to write buffer in a batch. We assume these cache lines coming from $P$ pages of flash memory (Our approach only focuses to optimize 1 page). The delay between victim cache and write buffer is $T_{v\rightarrow w}$. We assume that the write buffer is big enough to keep as many dirty data as possible. When the data are written to write buffer, which are organized as a queue and a write operation between write buffer and flash memory is a page operation. We assume that the delay between write buffer and flash memory is $T_{w\rightarrow f}$ per cache line size and the write delay of flash memory is $T_{f\rightarrow r}$. Before we update the page, we need to read it to the register. The read delay is $T_{r\rightarrow r}$.

Time to fill the replaceable lines in victim cache is:

$$T_{fill_vw} = (T_{v\rightarrow w} + T_w + T_{w\rightarrow w}) \times N$$

Time to write one batch of lines from write buffer to flash memory is:

$$T_{wb\rightarrow flash} = (N \times T_{f\rightarrow r} + T_{w\rightarrow r} + T_{r\rightarrow f}) \times P$$

So the size of write buffer must be:

$$V_{size_{wb}} \geq \left\lceil \frac{T_{wb\rightarrow flash}}{T_{fill_{wb}}} \right\rceil \times 32 \times 64\text{bytes}$$

In our experiments, there is only 1 page written to flash memory for each batch of write operation. Because the data in victim cache is written to write buffer in batches, there are many lines would become replaceable in victim cache. A write to the victim cache only takes place when there is a cache miss in LLC and the evicted data is dirty. Because LLC miss rate is low, and many of evicted data are not dirty. From our experiments, write operation takes place in every 2315 cycles on average. We assume $T_{v\rightarrow w} = 2\text{cycle}, T_{w\rightarrow r} = 4\text{ns}, T_{w\rightarrow w} = 1.56\text{us}$ (We assume delay for a 4KB page is 100us), $T_{f\rightarrow w} = 200\mu\text{s}$ and $T_{w\rightarrow f} = 25\mu\text{s}$. There are 3.9 lines are written in a batch to write buffer on average. The write buffer size of 12KB is enough to reduce the gap.

### 4. CONCLUSION

We have proposed an efficiently managed smart victim cache architecture which can reduce the write activities on flash memory when applying flash memory as the main memory. The proposed approach writes data back to flash memory in batches. This approach can significantly reduce the write activities by exploring the coarse grain accessing characteristic of flash memory. We also extend the traditional LRU method to manage cache blocks in victim cache. We further analyze how to reduce the high latency between cache and flash memory by adding a suitable write buffer. The experimental results show that our proposed architecture can reduce the write activities significantly.

### 5. REFERENCES