A Project Report

On

Improving Cache Performance by Mitigating Last Level Cache Pollution

By

Nikitha Shenoy Karkala
Eunice Naswali
Najuka Prakash Sankhe

Under the guidance of,
Dr Ming-Hwa Wang

Spring Quarter 2014
(COEN 283)
Table of Contents

<table>
<thead>
<tr>
<th>Topics</th>
<th>Page No</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>4</td>
</tr>
<tr>
<td>2. Theoretical background of the problem</td>
<td>5</td>
</tr>
<tr>
<td>3. Hypothesis</td>
<td>10</td>
</tr>
<tr>
<td>4. Methodology</td>
<td>10</td>
</tr>
<tr>
<td>5. Implementation</td>
<td>10</td>
</tr>
<tr>
<td>6. Analysis and Discussion</td>
<td>14</td>
</tr>
<tr>
<td>7. Conclusion</td>
<td>16</td>
</tr>
<tr>
<td>8. Bibliography</td>
<td>16</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENT

We would like to express our special appreciation and thanks to our Professor, Dr. Ming-Hwa Wang for giving us this opportunity and encouraging our research.

We would also like to thank the Santa Clara University library for providing us with group discussion rooms in the college, which helped us in concentrating on the task.

Lastly, we would like to thank our family and friends who supported us.
1. **Introduction**

1.1. **Objective**
To improve overall cache efficiency and utilization by reducing cache pollution.

1.2. **Why this is a project related this class**
The project deals with memory management which is an important part of Operating Systems. Long-latency memory access is one of the major performance bottlenecks of modern computing platforms. The behavior of cache determines system performance due to its ability to bridge the speed gap between the processor and main memory. The focus of this project is to increase the cache utilization so that it leads to better performance.

1.3. **Why other approach is no good**
- Require additional hardware.
- Consider pollution that occurs only due to prefetched data.
- Predict behavior of a cache block from the group behavior of multiple cache blocks. In case of low correlation between the cache block and the group, false prediction results.
- Due to the complexity of other approaches, additional accesses and processing may reduce effective access time of the cache.

1.4. **Why you think your approach is better**
Our approach considers cache pollution not only due to data that’s never being accessed but also data that have short/long access time which can pollute the Last Level Cache (LLC). The algorithm is simple and has very less overhead.

1.5. **Statement of the problem**
In operating systems, page caches generally have weaker locality than virtual memory pages. Most OS manage unified page caches to minimize disk I/O operations by keeping the data in the memory from disks. The page caches, which are large and less reusable data, usually degrade the cache efficiency by replacing cache-sensitive data; this inefficient behavior is called cache pollution.

1.6. **Area or scope of investigation**
1. Implement mechanism to handle cache pollution in LLC.
2. Simulate and generate results showing cache statistics such as miss rates.
3. Investigate to what extent the mechanism tackles cache pollution.
4. Explore the role of cache replacement algorithms when it comes to reducing the amount of pollution.
5. Summarize the performance, practicality and efficiency of our implementation.

2. Theoretical bases and literature review

2.1. Definition of the problem
With the prevalence of multicore, managing shared resources, such as CPU cache or memory bus, is an emerging issue in computer system. Contention caused by multiple computing threads upon the shared resources is a source of performance degradation of individual threads and the entire system. Among many resources, managing the LLC shared by multiple cores on a single chip or die is an important issue since the contention of the shared cache adversely affects the throughput of individual workloads due to performance interference. One major problem hindering its performance is cache pollution, where non-useful data occupies the precious cache space thus increasing the miss rate.

2.2. Theoretical background of the problem
Processor performance is significantly impacted by the behavior of the memory hierarchy. Main memory access time accounts for several hundreds of cycles. To avoid such a huge penalty, modern processors feature a complete memory hierarchy including small L1 caches with 1–3 cycles access time, larger L2 caches with 8–15 cycles access time and often (shared) very large multi-megabytes L3 caches with 20–30 cycles access time. Processors also resort to prefetch mechanisms to avoid cache misses whenever possible.

In general cache replacement policies do not take into account the fact that blocks have very different usages in applications. In particular, in some cases a block stored in the cache after a miss is not accessed again before eviction. Thus causing other needed data to be evicted from the cache into lower levels of the memory hierarchy, potentially all the way down to main memory, degrading performance. This problem is referred to as cache pollution.

Figure 1 shows the fraction of LLC blocks that never get re-accessed (re-referenced) while residing in the cache between the time the block is brought into the cache and the time it is evicted from the cache. In other words, these never re-accessed blocks do not receive any hits while in the LLC. Such blocks unnecessarily occupy valuable cache space and cause cache pollution and thrashing if they had replaced more useful cache blocks when they were brought into the cache.

Many approaches have been shown to be effective in mitigating negative performance due to cache pollution. However, this approaches have not altered LLC pollution significantly.
Fig. 1. Fraction of LLC blocks that are never re-accessed after being brought into the cache

2.3. Related research to solve the problem

2.3.1. Achieving Non-polluting Cache Accessing by Using Data Valid Tag Splitting

- This paper proposes a non-polluting cache accessing technique based on data tag valid-bit splitting (RVB and WVB), which is called **Pease**. Then Cache is accessed according to the different combinations of RVB and WVB.

- Firstly, Pease can distinguish the cache data lines written by speculative path instructions and cache data lines written by right path instructions.

- Secondly, Pease can preserve and make use of the data pre-fetching character of speculative executing effectively.

- Finally, a simple mechanism is used to wipe off the negative impact of polluting data to cache performance.

- When a new data needs to be written into cache, so far as polluting cache data lines exist, the new data can be written into the cache line that contains polluting cache data, which means that the polluting cache data line can be overlaid directly without performing cache replacement.
2.3.2. **SCIP: Selective Cache Insertion and Bypassing to Improve the Performance of Last Level Caches**
- SCIP proposes a bypassing algorithm that aims at inserting only non-polluted data into the cache.
- Cache blocks are categorized based on the access time as short, medium and long. Only the blocks that are predicted to have medium access times are inserted into the cache.

2.3.3. **The Evicted Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing**
- To mitigate cache pollution, the work in this paper predicts the reuse behavior of a missed block based on its own past behavior.
- To implement this prediction scheme, this mechanism keeps track of a small set of recently evicted blocks in a hardware structure known as the Evicted-Address Filter (EAF).
- The operation of a cache augmented with an EAF is such that when a block is evicted from the cache, the block’s address is inserted into the EAF. On a cache miss, the cache tests whether the missed block address is present in the EAF. If yes, the block is inserted into the EAF (the premise is that this block was prematurely evicted from the cache). Otherwise, the block is predicted to have low reuse and inserted with the bimodal insertion policy. When EAF becomes full, it is completely cleared.

2.3.4. **Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level Software-Only Pollute Buffer**
- Attacking last level cache pollution is the focus in this paper. This is achieved through identifying the cache pollution from observed miss rates of an application’s address space.
- To implement this solution, hardware performance counters are used to classify memory pages with respect to pollution. A pollute buffer is then used to host cache lines of pages with little or no reuse. Since the pollute buffer is implemented with software-based cache partitioning; a small partition of the last level cache is dedicated as the pollute buffer. The results is less competition between pages that pollute the cache and those that benefit the cache.

2.3.5. **Explicit Non-reusable Page Cache Management to Minimize Last Level Cache Pollution**
- A part of cache sets is used as a pollute buffer to hold non-reusable page caches.
- Page coloring is used to confine the non-reusable page caches into the pollute buffer.
The users or applications inform the OS kernel which pages are unlikely to be used.

2.3.6. **Mitigating the Cache Data Pollution by Using Branch Path Tracking**

- This paper proposes a cache pollution control technique by using branch paths tracking, which is called Contra.
- Firstly, we construct a branch paths tracking table to follow the cache data written by memory accessing instructions in speculative branch paths.
- Then, processing of storage, accessing and replacement to these data are particularly controlled in order to mitigate influence of data pollution to cache system and processor performance.

2.4. **Advantage/disadvantage of those research**

2.4.1. **Achieving Non-polluting Cache Accessing by Using Valid Tag Splitting**

- Pease technique treats polluting cache blocks as blocks that are never accessed (i.e. only due to prefetching), but in reality cache pollution can also occur for blocks that have only single usage.
- Cache replacement policy (PLRU in this paper) needs to be partially modified.

2.4.2. **SCIP : Selective Cache Insertion and Bypassing**

- Though SCIP improves overall performance, the main disadvantage is it is applicable only for non-inclusive caches. Exclusive caches are not favored due to the enormous bandwidth they impose.

2.4.3. **The Evicted Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing**

- The EAF-cache is implemented with a Bloom filter and a counter, the Bloom filter requires low hardware overhead and complexity.
- No modifications are made to the cache design leading to low design and verification costs.
- EAF only operates on a cache miss leaving the cache hit operation unaltered. Therefore this mechanism can be combined with cache replacement policies for better performance.
- Additional hardware costs are incurred in implementing a bit array for the bloom filter, a counter that counts the number of blocks in the cache and logic structure for the insertion policy.

2.4.4. **Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level Software-Only Pollute Buffer**
A shortcoming of this technique is that it has a high storage overhead.
- Cache line behavior is based on page level behavior therefore false predictions due to low correlation is unaccounted for.
- Compared to cache partitioning that focuses on isolating cache usage of different applications, this work focuses on isolating cache usage within the same application.

2.4.5. **Mitigating the Cache Data Pollution by Using Branch Path Tracking**
- Contra technique treats polluting cache blocks as blocks that are never accessed (i.e. only due to prefetching), but in reality cache pollution can also occur for blocks that have only single usage.
- Cache data structure should be modified for contra technique.

2.4.6. **Explicit Non-reusable Page Cache Management to Minimize Last Level Cache Pollution.**
- Advantage; no complex monitoring or hardware extension.
- Disadvantage; requires the users/application to explicitly specify the non-reusable page caches.

2.5. **Your solution to solve this problem**
- Cache blocks are divided into 3 categories based on their access time i.e short, long or medium access time.
- Blocks that are initially predicted to have short/long access time are inserted into that LLC with a low priority.
- Blocks predicted to have medium access time are inserted into LLC with a high priority.
- If a block was falsely predicted to have short/long access time but is later predicted to have medium access time, then the priority of such a block is increased.
- During replacement, blocks with low priority are replaced. In case of no such blocks then LRU is called.

2.6. **Where your solution different from others**
We differ from the others in the following ways;
- We are not bypassing blocks with short/long access time like in case of SCIP since there is a possibility of false prediction.
- Unlike Data Valid tag splitting, we are not just concentrating on eliminating pollution from blocks that are never accessed but are focusing on cache blocks that have very little usage before eviction.
- Our solution unlike in the EAF-cache doesn’t require a bloom filter which adds extra hardware.
3. **Hypothesis**

Our goal is to improve the performance of the SCIP algorithm further to mitigate cache pollution. We propose that SCIP has three additional memory accesses when writing to the LLC that could be eliminated. Thus increasing hit rate for the LLC and further reducing LLC cache pollution.

4. **Methodology**

4.1. **How to generate/collect input data**

SPEC95 benchmark will be used as input to our project.

4.2. **How to solve the problem**

4.2.1. **Language used**: C programming

4.2.2. **Tools used**: SimpleScalar simulator

4.3. **How to generate output**

The expected output are the cache statistics such as cache hit rate and miss rate which will help us assess how well our algorithm is working.

5. **Implementation**

5.1. **Code**

The code consists of the source code to implement SimpleScalar. The following files were modified to run and test the algorithm on this tool;

- cache.c
- cache.h
- sim-cache.c

5.2. **Design**

The design process commenced with modifying the SCIP algorithm, flowcharts in figure 1 and 2 illustrate the sub algorithms for handling hit and miss on the LLC given a block B for the proposed solution. The algorithm was tested with the simple scalar simulator; more details about the simple scalar simulator are given in the appendix. The memory hierarchy that was modeled to work with simple scalar is shown in table 1.

**Pseudo Algorithm:**

**Used Bit** indicates whether the block was accessed while residing in the cache or not.

```plaintext
used = 0  // not accessed.
```
**Polluted Bit** indicates whether the block is polluted or not.
polluted = 0 // not polluted.

**Conditions for Replacement (LRU):**
Replacement algorithm is called only if there is no polluted block in cache.

**On LLC Miss on a block B:**
If ( LLC full) {
    Find victim block V using LRU;
bypassBuffer[V] = V.used * 3;
} else {
    if (bypassBuffer[B] > 3) {
        B.polluted = 0;
    }
    else {
        B.polluted = 1;
bypassBuffer[B]++;
    }
}

**On LLC Hit on a block B:**
if (bypassBuffer[B] > 3) {
    if (B.polluted == 0) {
        B.used = 1;
    }
    else {
        B.polluted = 0;
    }
} else {
    bypassBuffer[B]++;
}
Figure 2: flow chart representing algorithm for LLC hit
This algorithm was modified after output analysis showed poor performance to handle the case during insertion when there are no polluted blocks in LLC to prevent useful blocks from being evicted.
Table 1: Memory hierarchy for simulation

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>nsets</th>
<th>bsize</th>
<th>Associativity</th>
<th>Replacement Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction</td>
<td>64B Lines</td>
<td>32KB</td>
<td>4-way</td>
<td>LRU</td>
</tr>
<tr>
<td>L1 Data</td>
<td>64B Lines</td>
<td>32KB</td>
<td>4-way</td>
<td>LRU</td>
</tr>
<tr>
<td>L2 Instruction</td>
<td>128B Lines</td>
<td>64KB</td>
<td>8-way</td>
<td>LRU</td>
</tr>
<tr>
<td>L2 Data</td>
<td>128B Lines</td>
<td>64KB</td>
<td>8-way</td>
<td>LRU</td>
</tr>
<tr>
<td>LLC</td>
<td>256B Lines</td>
<td>64KB</td>
<td>16-way</td>
<td>LRU</td>
</tr>
</tbody>
</table>

6. Analysis and Discussion

6.1. Results

Figure 3 shows simulation results for a 2-level memory hierarchy. SCIP algorithm is shown to perform worse than the reference algorithm with LRU. On the other hand, the proposed solution performs comparably with the reference algorithm with LRU. The basic idea behind the bypass algorithm is to accommodate the cache blocks with a short access time in lower level caches (L1 and L2). Since there exists 1 lower level cache (L2) bypassing gives worse performance. Therefore, adding L3 to the hierarchy shows better performance as shown in figure 4.

Figure 3: Miss rate with 2-level cache hierarchy
Figure 4 shows fewer replacements for cache accesses for both the SCIP algorithm and the proposed solution. More importantly, figure 5 proves that the algorithm for the proposed solution improves cache pollution mitigation further. Our proposed solution aimed to reduce the 3 additional memory accesses incurred by SCIP algorithm by inserting the LLC rather than bypassing the LLC till the number of references is 3. Fewer replacements implies a higher hit rate for all cache references hence higher chances of presence of needed cache blocks/lines in the LLC.
6.2. Challenges
One of the challenges encountered in this project was a lack of proper input files to test the algorithm. The required input files included SPEC benchmarks and trace files which are all licensed and therefore not easily available. Therefore our input files attempted to model the required input files; this accounts for a significant percent deviation in our results.

7. Conclusion
The work in SCIP algorithm has been reproduced and validated using Simplescalar and SPEC95 benchmarks. The results show that with a memory hierarchy containing only L1 and L2 SCIP algorithm does not reduce cache pollution however with a memory hierarchy containing L1, L2 and L3, SCIP algorithm does reduce LLC miss rates and hence perform better. In addition, our proposed algorithm that modifies the SCIP algorithm outperforms both SCIP and LRU algorithms. On an average it reduces miss rates by 22% compared to LRU and by 12% compared to SCIP.

Future work includes exploring the role of cache replacement algorithm augmented with the SCIP algorithm in reducing cache pollution. In addition, extensive testing of the SCIP algorithm with different benchmarks such as SPEC 2000 or SPEC 20006 could be explored. Finally, the SCIP algorithm could be tested with a multiprocessor architecture.

8. Bibliography

a) http://en.wikipedia.org/wiki/Cache_pollution
b) http://www.simplescalar.com
c) http://www.ece.umass.edu/ece/koren/architecture/Simplescalar/SimpleScalar_introduction.htm
References:


j) https://github.com/jsantangelo/sim-cache-mod

k) http://techtitude.blogspot.com/2011/12/how-to-execute-spec95-benchmarks-in.html

l) http://www.burningcutlery.com/derek/bargraph/


n) http://simon.denel.fr/Articles/How-To-Install-SimpleScalar/

o) http://www.simplescalar.com/docs/hack_guide_v2.pdf

9. Appendices

9.1. Input/Output listing

Sample Input command:

```
```

Sample Output for LRU:

```
ul3.accesses 1029848 # total number of accesses
ul3.hits 284948  # total number of hits
ul3.misses 744900  # total number of misses
ul3.replacements 740804 # total number of replacements
ul3.writebacks 48019  # total number of writebacks
ul3.invalidations 0  # total number of invalidations
ul3.miss_rate 0.7233 # miss rate (i.e., misses/ref)
ul3.repl_rate 0.7193 # replacement rate (i.e., repls/ref)
```
Sample Output for our solution:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ul3.accesses</td>
<td>1029848 # total number of accesses</td>
</tr>
<tr>
<td>ul3.hits</td>
<td>403968 # total number of hits</td>
</tr>
<tr>
<td>ul3.misses</td>
<td>625880 # total number of misses</td>
</tr>
<tr>
<td>ul3.replacements</td>
<td>103655 # total number of replacements</td>
</tr>
<tr>
<td>ul3.writebacks</td>
<td>15944 # total number of writebacks</td>
</tr>
<tr>
<td>ul3.invalidations</td>
<td>0 # total number of invalidations</td>
</tr>
<tr>
<td>ul3.miss_rate</td>
<td>0.6077 # miss rate (i.e., misses/ref)</td>
</tr>
<tr>
<td>ul3.repl_rate</td>
<td>0.1007 # replacement rate (i.e., repls/ref)</td>
</tr>
</tbody>
</table>

9.2. Simplescalar

The SimpleScalar tool set is a system software infrastructure used to build modeling applications for program performance analysis, detailed microarchitectural modeling, and hardware-software co-verification. Using the SimpleScalar tools, users can build modeling applications that simulate real programs running on a range of modern processors and systems.

SimpleScalar tools overview

SimpleScalar suite overview

| sim-fast | fast instruction interpreter, optimized for speed. This simulator does not account for the behavior of pipelines, caches, or any other part of the |

Figure 1. SimpleScalar tool set overview
<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim-safe</td>
<td>Slightly slower instruction interpreter, as it checks for memory alignment and memory access permission on all memory operations. This simulator can be used if the simulated program causes sim-fast to crash without explanation.</td>
</tr>
<tr>
<td>sim-profile</td>
<td>Instruction interpreter and profiler. This simulator keeps track of and reports dynamic instruction counts, instruction class counts, usage of address modes, and profiles of the text and data segments.</td>
</tr>
<tr>
<td>sim-cache</td>
<td>Memory system simulator. This simulator can emulate a system with multiple levels of instruction and data caches, each of which can be configured for different sizes and organizations. This simulator is ideal for fast cache simulation if the effect of cache performance on execution time is not needed.</td>
</tr>
<tr>
<td>sim-bpred</td>
<td>Branch predictor simulator. This tool can simulate different branch prediction schemes and reports results such as prediction hit and miss rates. Like sim-cache, this does not simulate accurately the effect of branch prediction on execution time.</td>
</tr>
<tr>
<td>sim-outorder</td>
<td>Detailed microarchitectural simulator. This tool models in detail and out-of-order microprocessor with all of the bells and whistles, including branch prediction, caches, and external memory. This simulator is highly parameterized and can emulate machines of varying numbers of execution units.</td>
</tr>
</tbody>
</table>