TERM PROJECT

Enhancing data hit ratio by using adaptive caching Technique

COEN 283
Operating System
Prepared By:
Darshita Shah
Preethi Yellappa
Nidhi Singh
# Table of Content

<table>
<thead>
<tr>
<th>Topics</th>
<th>Page No</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2. Theoretical background of the problem</td>
<td>3</td>
</tr>
<tr>
<td>3. Hypothesis</td>
<td>5</td>
</tr>
<tr>
<td>4. Methodology</td>
<td>5</td>
</tr>
<tr>
<td>5. Implementation</td>
<td>8</td>
</tr>
<tr>
<td>6. Output Analysis</td>
<td>11</td>
</tr>
<tr>
<td>7. Conclusion</td>
<td>14</td>
</tr>
<tr>
<td>8. Future Enhancement</td>
<td>14</td>
</tr>
<tr>
<td>9. Bibliography</td>
<td>14</td>
</tr>
</tbody>
</table>
Table of Figures

- Simple Scalar Simulator Architecture
- Comparison between once accessed page & multiply accessed page tails
- Hit page access of once accessed page Queue & multiple accessed page Queue
1. Introduction

1.1. **Objective:** We are presenting and implementing an improved version of adaptive cache replacement scheme that is based on the paper “SSARC: the Short-Sighted Adaptive Replacement Cache” with the objective of optimizing the cache system performance by enhancing hit ratio. This replacement scheme assigns a replacement priority value to each cache block according to a set of criteria to decide which block to remove. The goal is to provide an effective utilization of the distributed cache memory and a good application performance.

1.2. **Problem:** Cache miss is the biggest problem while it’s increased distance between main memory & CPU. For some data CPU send data request to nearest cache, if cache has no data it generates Cache miss signal & generate query for main memory. If main memory also does not have data then data request transfer to disk in form of signal. All these procedures increase execution time for CPU or sometimes dismiss the procedure. This problem can be avoided either by reducing cache miss ratio or by increasing cache hit ratio. To improve these functionality couple of methods introduced in recent years like LRU, FIFO, and NRU etc.

1.3. **How to Approach:** We are introducing SSARC using simple scalar simulator. We are calculating histogram of LRU stack positions of cache hit as well as evicted cache lines. We also compare these results with FIFO stack positions of cache hit & we concluded results with bar graph.

1.4. **This approach is better:** We are considering cache hit ratio using simulator. We can compare multiple replacement policies & can generate graphical representation to see better performer on available memory. Lot more enhancements also can be done according to requirement.

1.5. **Scope of Investigation:** In this proposal, we will perform an experiment on a single core machine with a set of programs under certain benchmark. We will simulate a general adaptively scheme based on using knowledge of hit ratio of cache component replacement policy. We can enhance caching replacement algorithm & adaptive cache implementation using stronger implementation than ARC and SSARC.

2. Theoretical background of the problem:

2.1. **Definition of the problem:** The goal of counter based adaptive caching is to evaluate best cache replacement policy for available resource and enhance new algorithm for increasing cache hit for increasing CPU performance.

2.2. **Theoretical background of the problem:** Our counter based cache replacement algorithm approach uses replacement policies LFU and FIFO.

- LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory as good as Adaptive Replacement Cache. The most expensive method in this is the linked list method, which uses a linked list containing all the pages in
memory. At the back of this list is the least recently used page, and at the front is the most recently used page.

- FIFO is low overhead algorithm where the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the earliest arrival in front. When a page needs to be replaced, the page at the front of the queue the oldest page is selected.

2.3. Related Search:

We are enhancing algorithm based on the Short-Sighted Adaptive Replacement Cache using simple scalar tool.

2.3.1. Introduction of Simple Scalar Simulator Tool:
- Simple Scalar is an open source computer tool set that model virtual computer system with CPU, cache & memory. Using this tool set, users can build modeling applications that simulate real programs running on a range of modern processors and systems. This tool set includes dynamically scheduled processor model that supports non-blocking caches, speculative execution, performance visualization tools, statistical analysis resources, and debug and verification infrastructure.
- Here is Simple Scalar Simulator Architecture:

![Simple Scalar Simulator Structure](image)

- The tool set it consists of a collection of microarchitecture simulators that emulate the microprocessor at different levels of detail.

2.3.2. Introduction of SSARC
- As the performance gap between disks and processors continues to increase, dozens of cache replacement policies come up to handle the problem. Unfortunately, most of the policies are
static. A low overhead adaptive policy called ARC which outperforms most of the static policies in most situations. But, ARC adapts itself to the workloads by the feedback of the missed pages. It hasn’t carried out the adaption before missed pages are discovered. SSARC propose a high performance adaptive replacement policy which adapts itself to the workloads by the feedback of the hit pages, so, it is more sensitive to the changes of the workloads than ARC. SSARC endeavors to protect the would-be-reused pages from being replaced aggressively. We compared SSARC with LRU and FIFO. The trace-driven experiments represent that SSARC gains higher performance.

3. Hypothesis

- This paper proposes that use of optimal LRU and FIFO policies to increase the performance of the cache. Our hypothesis is that we can introduce widely used policy – counter based algorithm, along with LRU and FIFO in to make the cache system more robust. The processor will be provided with both policies which would result cache hit rate relative to other cache systems which use both policies.

4. Methodology

4.1. Simple Scalar Simulator Installation Introduction:

A) Download the necessary source code files:
- simpletools-2v0.tgz from http://www.simplescalar.com/tools.html
- simplesim-3v0d-with-cheetah.tar.gz from http://www.ict.kth.se/courses/IS2202/software
- simpleutils-990811.tar.gz from http://www.eecs.umich.edu/mirv/
- gcc-2.7.2.3.ss.tar.gz from http://american.cs.ucdavis.edu/RAD/gcc-2.7.2.3.ss.tar.gz

B) Setup some environment variables (assuming your home directory is /home/matsbror):
- $ export IDIR=~/home/matsbror/simplescalar
- $ export HOST=i686-pc-linux
- $ export TARGET=sslittle-na-ssrix

C) Create the directory “simplescalar” under your home directory, and copy all the four tgz files:
- $ mkdir $IDIR
- $ mv simplesim-3v0d-with-cheetah.tgz $IDIR
- $ mv simpletools-2v0.tgz $IDIR
- $ mv simpleutils-990811.tar.gz $IDIR
- $ mv gcc-2.7.2.3.ss.tar.gz $IDIR

D) Update Ubuntu (or whatever distribution you are using) with the following packages:
- build-essential
- flex
- bison
- yacc (maybe not needed)

E) Just un-pack the package file, and remove the old gcc folder:
- $ cd $IDIR
- $ tar xvfz simpletools-2v0.tgz
- $ rm -rf gcc-2.6.3
4.2. Requirements

4.2.1. Before building code
A) First un-pack the package file.
   • $ cd $IDIR
   • $ tar xvfz simpleutils-990811.tar.gz
   • $ cd simpleutils
B) In directory ld find file ldlex.l and replace all instances of yy_current_buffer with
   • YY_CURRENT_BUFFER
   • $ ./configure --host=$HOST --target=$TARGET --with-gnu-as
   • --with-gnu-ld --prefix=$IDIR
   • $ make
   • $ make install

4.2.2. Building Simulator
A) Un-pack the simulator package.
   • $ cd $IDIR
   • $ tar xvfz simplesim-3v0d.tgz
   • $ cd simplesim-3.0
B) Modify the Makefile by changing “-O0” with “-O” to enable compiler optimizations.
   • $ make config-pisa
   • $ make
C) Test simulator by:
   • $ ./sim-safe tests/bin.little/test-math

4.2.3. Installation of gcc cross-compiler
A) Un-pack the source code and configure the installation:
   • $ cd $IDIR
   • $ tar xvfz gcc-2.7.2.3.ss.tar.gz
   • $ cd gcc-2.7.2.3
   • $ ./configure --host=$HOST --target=$TARGET --with-gnu-as
   • --with-gnu-ld --prefix=$IDIR
   • $ chmod -R +w .
4.2.4. Pseudo Code of implementation

Block A is accessed

If(A is in the once-accessed queue) {
    If(A is in the once-accessed tail) {
        If(E₁ >= 1)
            U₂ = U₀ + E₂;
        If(E₂ >= 1)
            U₀ = U₀ + E₀;
    }
    If(A is a twice accessed page)
        Stamp A with stampₘ and add it to the MRU end of the multi-ply accessed queue.
    Else
        Label A as a twice accessed page, Stamp it with stampₘ and add it to the MRU end of the once-accessed queue.
}Else if(A is in the multi-ply accessed queue) {
    If(A is in the multi-ply accessed tail) {
        If(E₁ >= 1)
            Uₘ = Uₘ + Eₙ;
        If(E₂ >= 1)
            Uₙ = Uₙ + E₂;
    }
    Stamp A with stampₘ and add it to the MRU end of the multi-ply accessed queue.
}Else {
    If(there is no free slot)
        Replace();
    If(A is in the ghost cache)
        Stamp A with stampₘ and add it to the MRU end of the multi-ply accessed queue.
    else
        Stamp A with stampₘ and add it to the MRU end of the once-accessed queue.
}

Subroutine Replace()
{
    Uₘ = Uₘ × C/(Uₘ + U₂)  // C is the size of cache
    U₂ = U₂ × C/(Uₘ + U₂)
    If((|once-accessed queue| >= (int)Uₙ) || (|multi-ply accessed queue| <= (int)Uₘ)) {
        While(true) {
            If(the tail of once-accessed queue is a once-accessed page or a dated multi-ply accessed page) {
                Remove the tail of once-accessed queue;
                Break;
            }Else
                Shift the tail of once-accessed queue to its head, Label it as a dated multi-ply accessed page;
        }
    }Else
        Remove the tail of multi-ply accessed queue;
}
4.3. Methodology Specification

4.3.1. Language Used: C Programming Language
4.3.2. Tool Used: Simple Scalar Simulator Tool Set
4.3.3. Tool set architecture: Alpha instruction set, Pisa Instruction set

4.4. Input: We used standard benchmark from SPEC95 to test our algorithm.

4.5. Output: Our output expresses in terms of Hit pages per thousand instructions.

5. Implementation

5.1. How we approached:

5.1.1. SSARC algorithm:
- SSARC divides the pages into two groups: pages only have been accessed once (which are called once-accessed pages) and pages have been accessed multiple times (which are called multi-ply accessed pages).
- Correspondingly, SSARC builds up two LRU queues: once-accessed queue and multi-ply accessed queue. As the victim of the replacement is either the LRU page of the once-accessed queue or the LRU page of the multi-ply accessed queue, SSARC compares the LRU-ends of the queues (we call the LRU-ends once-accessed tail and multi-ply accessed tail respectively), the victim is taken from the useless tail. That’s to say: if the once-accessed tail is more useful, SSARC replaces the LRU page in the multi-ply accessed queue; otherwise, replaces the LRU page in the once-accessed queue.

5.1.2. SSARC Emergency
- To compare the utility of the tails, we introduce into the concept “Emergency”. When a page of the once-accessed tail is hit, SSARC calculates the “Emergency” for the page, and adds the result to the total “Emergency” of the once-accessed tail. The same work is carried out for multi-ply accessed tail. When a free slot is required, SSARC selects the victim from the very tail that has less total “Emergency”.
- SSARC balances between once-accessed tail and multi-ply accessed tail with the changes of workloads. It is an adaptive policy.
- SSARC emphasizes on the tails of the two queues. It protects the maybe-reused pages in the tails from being replaced, but ignores other pages. It is short-sighted, so, we call it short-sighted policy.
- To compensate for the short-sighted decision, the policy maintains a ghost cache to capture some of the missed pages.
- SSARC maintains three queues: once-accessed queue, multi-ply accessed queue and ghost cache. They are all LRU queues, so the policy has constant-time complexity per request. We simulated the policy with traces and compared it with LRU [1]! FIFO[2]. The simulation shows that SSARC outperforms them with most of the traces.
Figure A: The comparison of the tails, when the once-accessed tail is more valuable, the victim is taken from multi-ply accessed tail, or otherwise. “O” denotes the once-accessed tail; “M” denotes the multi-ply accessed tail.

Figure B: Two hit pages (colored blue) belong to once-accessed tail and multi-ply accessed tail respectively, page A has little “Emergency” because of the longer distance between A and the tail of queue.

Case 1: figure B represents two hit-pages (blue pages), they belong to once-accessed tail and multi-ply accessed tail respectively. The neighbor pages of them are assumed to be requested soon after, and should be protected from being replaced. Compared to the hit-page in the multi-ply accessed tail, the one belonging to the once-accessed tail is more dangerous to be replaced due to the shorter distance between the page and the tail of the queue. That is to say, the hit-page in the once-accessed tail has great “Emergency”. Hit-page in the multi-ply accessed tail A, the distance between page A and the tail of the queue is d, the size of cache is C. The first component is defined as below. \( E_1 = \log_m(C/d) \), \( m \) is a parameter.

Case 2: if the once-accessed queue has fewer pages than the multi-ply accessed queue, the multi-ply accessed tail is more likely to be replaced; or otherwise. Beyond the analysis above, we define the two components of “Emergency” correspondingly. For the same hit-page A, we denote \( \text{SIZE}_m \) and \( \text{SIZE}_o \) as the sum of pages in the multi-ply accessed queue and once-accessed queue respectively. Parameter \( Q \) is defined as \( \text{SIZE}_o/\text{SIZE}_m \) (of course, if the hit-page belongs to the once-accessed tail, \( Q = \text{SIZE}_m/\text{SIZE}_o \)). The second component is defined as \( E_2 = \log_m Q \) \( \wedge m \) is a parameter. **The total “Emergency” of A is:** \( E = E_1 + E_2 \).

5.1.3. SSARC Queues:
- A high performance replacement policy should be scan resistant. An effective way to achieve scan resistance is dividing the pages into once-accessed pages and multi-ply accessed pages. When the scan pattern comes up, only the once-accessed pages can be replaced, the repeated pages are protected.
- To achieve scan-resistance SSARC divides the pages into once-accessed pages and multi-ply accessed pages. Only twice accessed pages take half the sum of the multi-ply accessed pages, or more. when a
once-accessed page is requested, it becomes a twice accessed page, but SSARC inserts it into the once-accessed queue all the same. If it hasn’t been requested before it comes to the tail of the once-accessed queue, it will be taken as a once-accessed page and be replaced ultimately. If it did be reused, SSARC inserts it into multi-ply accessed queue. Now, the once-accessed queue contains some pages that had been accessed twice, but we still call it once-accessed queue. Maybe, the policy is not scan-resistant anymore because of the twice-accessed pages in the once-accessed queue. Fortunately, the ghost cache can compensate for it more or less. When a miss page was captured by the ghost cache, SSARC inserts it into the multi-ply accessed queue directly. The size of the ghost cache is the same with the cache.

5.1.4. SSARC Short-sighted Policy:
- SSARC pays attention to once-accessed queue and multi-ply accessed queue. As the LRU policy does, replacement occurs at the tails of the two queues. SSARC compares the utilities of once-accessed tail and multi-ply accessed tail. If the once-accessed tail is more valuable, the victim is taken from multi-ply accessed tail, or otherwise. The concept “Emergency” is introduced into our policy to explain the utility of each tail. We illustrate the use of “Emergency” as follows. When a page belonging to once-accessed tail is hit, SSARC calculates the “Emergency” for it, then, adds the “Emergency” to the total “Emergency” of the once-accessed tail. The total “Emergency” is the utility of the once-accessed tail. Replacement occurs at the tail with less utility.

5.1.5. SSARC Replacement:
- We denote the utilities of the tails Um and Uo respectively. When a page A in the multi-ply accessed tail is hit, SSARC calculates the “Emergency” EA, update the Um , Um=Um+EA; the same work is carried out for a hit page in the once-accessed tail. When a replacement is on the way, update the Um and Uo·Um=Um*C/(Um+Uo) , Uo=Uo*C/(Um+Uo) If SIZEo>=Uo or SIZEm<=Um, replace the LRU page of the once-accessed queue; otherwise, replace the LRU page of the multi-ply accessed queue.

5.1.6. Three parameters:
- Only the hit-page belongs to the tail that could make contribution to the utility of its queue. To determine whether a hit-page is in the tail or not, We have defined the size of tail as |tail|=min(C/m, SIZEo, SIZEm). If the distance between a hit-page A and the tail of the queue is less than |tail|, the page A is in the tail, SSARC calculates the “Emergency” for it. SSARC maintains stamp for once-accessed queue and multi-ply accessed queue respectively, stampo and stampm. When a page is added to the MRU end of the queue, SSARC labels the page with the stamp of the queue. SSARC calculates the distance for a hit-page in the tail as: SIZE denotes the sum of pages belonging to the queue, stamphead and stamptail denote the stamp of the MRU page and the LRU page respectively. The stamp of hit-page A is stampA. The distance between page A and the LRU page of the queue is approximated as

\[ d=\text{SIZE} \times (\text{stampA} - \text{stamptail}) / (\text{stamphead} - \text{stamptail}) \]

The parameter m is necessary to calculate the “Emergency” and |tail|.
6. Output Analysis
   6.1. Output Comparison
   According to our experimenters, SSARC slightly turned out as better performer than LRU as shown numbers in below screen shots:

Compress95 Results:
Go Results:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttlb.ssarc_pol_misses</td>
<td>22863</td>
</tr>
<tr>
<td>ttlb.ssarc_pol_misses</td>
<td>22846</td>
</tr>
<tr>
<td>ttlb.invitations</td>
<td></td>
</tr>
<tr>
<td>ttlb.miss_rate</td>
<td>0.0000</td>
</tr>
<tr>
<td>ttlb.writeback_rate</td>
<td>0.0000</td>
</tr>
<tr>
<td>ttlb.inv_rate</td>
<td>0.0000</td>
</tr>
<tr>
<td>dtlb.accesses</td>
<td>157196033</td>
</tr>
<tr>
<td>dtlb.hits</td>
<td>157194424</td>
</tr>
<tr>
<td>dtlb.misses</td>
<td>1609</td>
</tr>
<tr>
<td>dtlb.replacements</td>
<td>1487</td>
</tr>
<tr>
<td>dtlb.writebacks</td>
<td>1238</td>
</tr>
<tr>
<td>dtlb.evictions_s</td>
<td>1009</td>
</tr>
<tr>
<td>dtlb.evictions_pol_replace</td>
<td>0</td>
</tr>
<tr>
<td>dtlb.accesses</td>
<td>157196033</td>
</tr>
<tr>
<td>dtlb.ssarc_pol_hit</td>
<td>1571944441</td>
</tr>
<tr>
<td>dtlb.ssarc_pol_misses</td>
<td>1699</td>
</tr>
<tr>
<td>dtlb.invitations</td>
<td></td>
</tr>
<tr>
<td>dtlb.miss_rate</td>
<td>0.0000</td>
</tr>
<tr>
<td>dtlb.writeback_rate</td>
<td>0.0000</td>
</tr>
<tr>
<td>dtlb.inv_rate</td>
<td>0.0000</td>
</tr>
<tr>
<td>ld_text_base</td>
<td>0x00000000</td>
</tr>
<tr>
<td>ld_text_size</td>
<td>642272</td>
</tr>
<tr>
<td>ld_data_base</td>
<td>0x10000000</td>
</tr>
<tr>
<td>ld_data_size</td>
<td>576796</td>
</tr>
<tr>
<td>ld_stack_base</td>
<td>0x7ffffffc00</td>
</tr>
<tr>
<td>ld_stack_size</td>
<td>16384</td>
</tr>
<tr>
<td>ld_entry</td>
<td>0x00000000</td>
</tr>
<tr>
<td>ld_enviror_base</td>
<td>0x7ffffff800</td>
</tr>
<tr>
<td>ld_target_big_endian</td>
<td>0</td>
</tr>
<tr>
<td>mem.page_count</td>
<td>280</td>
</tr>
<tr>
<td>mem.page_mem</td>
<td>1140</td>
</tr>
<tr>
<td>mem.ptab_misses</td>
<td>285</td>
</tr>
<tr>
<td>mem.ptab_accesses</td>
<td>2530832739</td>
</tr>
<tr>
<td>mem.ptab_miss_rate</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
Anagram Results:

6.2. Output Bar Graph
7. Conclusion

- During out testing SSARC protects the would be referenced pages aggressively so it can achieve better performance than LRU and FIFO.

8. Future Enhancement

- After approaching our algorithm we thought it can be more advance while ARC & SSARC can adaptively switch among each other can come up with more reliable, faster & scalable results.

9. Bibliography

9.1. https://mail-attachment.googleusercontent.com/attachment/u/0/?ui=2&ik=25b13401c7&view=att&th=13f2a64d9a186215&attid=0.2&disp=inline&realattid=f_hgfqb1tn1&safe=1&zw&saduie=AG9B_P8eOCHEYJaM-fnZcN36ws1M&sads=PIFqMWN2L6z9SG00VKx7kprL-A0


