AMSQM: adaptive multiple super-page queue management

Moshe Itshak and Yair Wiseman*

Department of Computer Science,
Bar-Ilan University,
Ramat Gan 52900, Israel
E-mail: izmo@cs.biu.ac.il
E-mail: wiseman@cs.huji.ac.il
*Corresponding author

Abstract: Super-pages have been wandering around for more than a decade. There are some particular operating systems that support super-paging and there are some recent research papers that show interesting ideas about how to intelligently integrate them. However, nowadays operating system's page replacement mechanism still uses the old CLOCK algorithm which gives the same priority to small and large pages. In this paper, we show a technique that enhances the page replacement mechanism to an algorithm based on more parameters and is suitable for a super-paging environment.

Keywords: virtual memory; super-paging; page replacement algorithms; page fault ratio.

Reference to this paper should be made as follows: Itshak, M. and Wiseman, Y. (2009) 'AMSQM: adaptive multiple super-page queue management', *Int. J. Information and Decision Sciences*, Vol. 1, No. 3, pp.323–341.

Biographical notes: Moshe Itshak got his MSc from Bar-Ilan University. He is a Memory Management Expert. Currently, he is with Radware.

Yair Wiseman got his PhD from Bar-Ilan University and did two Post-Doc – one at the Hebrew University of Jerusalem and one in Georgia Institute of Technology. Currently, he is with the Computer Science Department of Bar-Ilan University. His research interests are process scheduling, hardware-software codesign, memory management, asymmetric operating systems and computer clusters.

1 Introduction

Super-pages are an enhancement for the well-known paging concept. Super-pages are larger pages that are pointed to by the TLB (Khalidi et al., 1993). The internal memory of modern computers has been significantly increased during the last decades. However, the TLB coverage (i.e. the size of the memory that can be pointed to directly by the TLB) has been increased by a much lower factor during the same period (Navarro, 2004). Therefore, several new architectures like Itanium, MIPS R4x00, Alpha, SPARC and HP PA RISC support multiple page size of the frames pointed to by the TLB. In that way the memory size pointed to directly by the TLB is higher and the overhead of the page table access time is reduced. There are also some particular operating systems that support super-paging, for example, Ganapathy and Schimmel (1998) and Subramanian et al. (1998).

Multimedia applications typically have large portions of memory that are clustered in few areas. Such applications can benefit super-paging enormously (Abouaissa et al., 1999). Also, nowadays computers usually have large memories (Geppert, 2003; Wallace et al., 2006); hence, larger pages can be used; however using larger pages can apparently cause a higher page fault rate. This is a well-known flaw of the super-paging mechanism; however the algorithm suggested in this paper does not suffer from this flaw and even utilises the usual behaviour of the paging mechanism to reduce the page fault rate. The algorithm actually makes use of the locality principle to pre-fetch base-pages that are a part of heavy used super-pages and the results show that this pre-fetching makes the memory hit percents better.

We also aim at developing a good technique that finds the best page to be taken out when the page fault mechanism requires this in a super-paging environment based on all the available parameters. The locality principle that the Super-paging environment induces will help the operating system to select the "victim" page better. This selection will be better because if page's neighbors are accessed, this can imply that the page itself might be accessed as well and it may not be a good choice to swap the page out as the common base-page algorithms would have done.

The dilemma of which page should be taken out also occurs in higher levels as well, that is, What should be in the cache and what should be pointed by the TLB. Our algorithm can also be a good alternative for CLOCK in these decisions.

2 Page replacement algorithms

Over the years many replacement algorithms have been published, for example, O'Neil et al. (1993), Johnson and Shasha (1994), Lee et al. (2001), Kim et al. (2000), Jiang and Zhang (2002), Smaragdakis et al. (2003), Zhou et al. (2004) and Wiseman (2005); however, over the last decades, CLOCK (Corbato, 1968) has been dominated by page replacement algorithms.

2.1 CLOCK

The CLOCK algorithm looks at the memory pages as a circular linked list and moves around the pages like a clock hand. Each page is associated with a reference bit. This bit is set to 1 when the page is referenced. When a page fault occurs, the page which is pointed by the hand is checked. If its reference bit is unset, it will be swap out; otherwise, its reference bit is unset and the hand moves to the subsequent page. Research and experiences have shown that CLOCK is a close approximation of LRU and thus suffers from the same problems of LRU. Nevertheless, CLOCK is still dominating the vast majority of OS including UNIX, Linux and Windows (Friedman, 1999).

Some variant of CLOCK have been suggested over the years. GCLOCK (Corbato, 1968) was published at 1992 as an expansion to CLOCK. This algorithm contains a counter to each page (instead of a reference bit), which is increased in each reference. The clock's hand checks the pages and decrements their counter value, until it finds a page with a zero value. This page is swapped out. Unlike CLOCK, GCLOCK is taking into account the frequency, thus achieves better performance.

CLOCK-Pro (Jiang et al., 2005) counts for each page the number of other distinct pages accesses since its last access. This number is called 'reuse distance' and a page with a larger 'reuse distance' will be considered as a colder page and will be swap out before a page with a smaller 'reuse distance'.

2.2 ARC

In the above section, we focused on the CLOCK, because CLOCK dominates the operating systems market; however, some other methods seem to suffer from two acute problems:

1 the need for parameters tuning (e.g. 2*Q* (Johnson and Shasha, 1994) and LRFU (Lee et al., 2001))

2 non-constant complexity (e.g. LRU-K (O'Neil et al., 1993), LRFU (Lee et al., 2001), CLOCK and GCLOCK (Corbato, 1968)).

CLOCK also has a non-constant complexity, so we prefer to adapt more modern algorithm to the super-paging environment. Recently, Megiddo and Modha proposed a new 'online' tunable algorithm called adaptive replacement cache (ARC) (Megiddo and Modha, 2003a,b, 2004). The unique capability of this algorithm is its ability to adapt itself 'online' according to the systems properties, for example, from the stack depth distribution (SDD) model to the independence reference model (IRM) and vice versa.

The main concept of ARC is having two lists of active pages (one for the frequently used pages and other one for the most recent pages) and to endow the list that is performing the best with a larger memory space. The two lists that ARC maintains are variably sized lists called L_1 and L_2. L_1 contains the pages that have been accessed only once and L_2 contains the pages that have been accessed twice or more. The algorithm always holds that $0 \leq L_1 + L_2 \leq 2C$, where C is the number of pages in the memory. L_1 consists of two buffers – T_1 which consists of the most recent pages in the memory and B_1 which consists of the history of the most recent pages that were in the memory. Similarly, L_2 is partitioned into T_2 and B_2. In addition p which always holds $p \leq c$, is the automatic adaptive parameter of the algorithm which sets the target size for T_1.

The algorithm in a simplify version is for any page request:

- If the requested page is in T_1 or in T_2:
  - Move the page to the MRU of T_2.
- If the requested page is in B_1:
  - If $|B_1| \geq |B_2|$
    - $\delta_1 = 1$
  - Else
    - $\delta_1 = |B_2| / |B_1|$
  - $P = \text{Min}(P + \delta_1, C)$
  - Move the page from B_1 to be the LRU of T_2 (swap out page according to P).

- If the requested page is in B_2:
  - If $|B_2| \geq |B_1|$
    - $\delta_2 = 1$
  - Else
    - $\delta_2 = |B_1| / |B_2|$
  - $P = \text{Max}(P - \delta_2, 0)$
- Move the page from B_2 to be the LRU of T_2 (swap out page according to P).
- If the requested page is not in $T_1 \cup T_2 \cup B_1 \cup B_2$:
  - Move the new page to be the MRU of T_1 (swap out page according to P).

As we mentioned above, CLOCK can move its clock hand over many pages, until a page with an unset bit is found. Unlike CLOCK, ARC has a constant complexity – $O(1)$. In addition, ARC is tunable, that is, ARC can adapt itself according to the characteristics of the data that the processes use. These are reasons why we chose to adapt ARC to the super-paging mechanism.

3 AMSQM

We used ARC to develop a new algorithm – adaptive multiple super-pages queues management (AMSQM) (Itshak and Wiseman, 2008) which is an expansion of the ARC algorithm that supports super-paging. AMSQM algorithm has two levels – the high level manages the different super-page queues (sizes and allocations); whereas the low level is the internal management of each super-page's queue. In addition, there is a special buffer for each super-page size that collects fractions of bigger super-pages. The purpose of these buffers is giving the demoted super-pages a chance to get a better priority if they are hot pages.

The suggested algorithm uses a reservation-based scheme, in which region is reserved for a super-page at the page fault time and the promotion is done when the number of the super-page's populated base-pages gets to a promotion threshold. Since we would like a partially populated super-page to have the opportunity of being promoted, the decision for pre-empting reservation of a super-page candidate or swapping out its base-pages is taken based on the super-page 'recency' in the page lists and not based on the number of currently resident base-pages that the super-page consists of. Actually, this is a known technique of information filtering in order to achieve a better decision (Wang, 2008).

Hardware maintains only a single reference bit; thus it is difficult to decide whether all (or at least most) of the base-pages that the super-page consists of are actually in use. Sometimes, only a small percentage of the base-pages should be in the memory. Therefore, AMSQM manages several queues for each super-page size, preventing from cold super-pages to be retained in the cache occupying the space of some potential hotter smaller super-pages or base-pages.

Finally, in order to wisely balance the different queues length, the algorithm counts the number of times that each page has been referenced and checks the relative 'recency' of each super-page's queue.

Similarly to ARC, AMSQM has *B* and *T* lists, but AMSQM has *T* and *B* list for each super-page size that is denoted as T_1^i, T_2^i, B_1^i and B_2^i where *i* is the super-page size. Therefore, the pseudo-code should be briefly:

- find the super-page that contains the requested page
- if the page is in T_1^i or T_2^i, the size of lists is good and no need to change it
- if the page is in B_1^i, the size of L_1^i should be increased
- if the page is in B_2^i, the size of L_2^i should be increased
- if the page is not in the memory, the size of lists is good and no need to change it.

The detailed AMSQM algorithm in pseudo-code is written herein below:

Let us define:

C: the memory size

c_i: physical size of the super (and base) pages buffers. $\Sigma\, c_i \leq C$

s_i: target size of each buffer

Q_i: queue (FCFS) that saves demoted super-pages (or base-pages), which are a fraction of bigger super-pages

T_1^i: the most recent pages in the memory of every super (or base) page, which were accessed only once.

B_1^i: the most recent pages in the history of every super (or base) page, which were accessed only once.

T_2^i: the most recent pages in the memory of every super (or base) page, which were accessed more than once.

B_2^i: the most recent pages in the history of every super (or base-page), which were accessed more than once.

P_i: tunable parameter – the recommended size of T_1^i.

size$_i$: super-page size in base-pages.

bound$_i = \beta\cdot$size$_i$ / size$_1$

count(x): the number of times that super-page x was referenced.

rank$_i$: determines which queue removes an entry. rank$_i$ = $\alpha\cdot$dif$_i$ + $(1 - \alpha)\cdot$rec$_i$, where dif$_i$ is the difference between s_i and c_i; that is, max(0, size$_i\cdot(c_i - s_i)$) and rec$_i$ is the relative recency of the LRU of super-page *i* among the LRU of the other super-pages.

threshold: threshold for promoting a partially occupied (candidate) super-page to a fully occupied super-page.

SP(x_j): the super-page which the base-page x_j belongs to. x_j = SP(x_j) iff x_j does not belong to any super-page (a solitary base-page).

$\omega(x)$: the number of occupied base-pages in super-page x.

α,β,γ: parameters that should be set according to the data characteristic; where $0 \le \alpha \le 1$, $\beta \ge 1$ and $0 \le \gamma \le ½$.

The algorithm AMSQM is:

```
AMSQM($c$, stream of base-pages requests: $x_1,x_2, \ldots, x_n$)
  • $c_1 = c_2 = \ldots = c_k = 0$
  • For each $x_j$
      o Call HandleSuperPage($x_j$, |SP($x_j$)|)
      o If $\omega$(SP($x_j$)) $\ge$ threshold·size$_{|SP(x_j)|}$
          ▪ Promote SP($x_j$)
  • If the access type is 'write', recursively demote SP($x_j$) to clean base/super-pages
    and move them to the suitable $Q$ lists.

HandleSuperPage($x_j$,$i$)
  • If SP($x_j$) is in $T_1^i$,
      o If $x_j$ is valid
          ▪ Move SP($x_j$) to be the MRU of $T_2^i$
      o Else
          ▪ Fetch $x_j$ to the cache
          ▪ Move SP($x_j$) to be the MRU of $T_1^i$
      o If (count(SP($x_j$)) = bound$_i$)
          ▪ count(SP($x_j$)) = $\gamma$· bound$_i$
      o Else
          ▪ count(SP($x_j$)) = count(SP($x_j$)) + 1
  • If SP($x_j$) is in $T_2^i$ or $Q_i$
      o If $x_j$ is invalid
          ▪ Fetch $x_j$ to the cache.
      o Move SP($x_j$) to be the MRU of $T_2^i$
      o count(SP($x_j$)) = count(SP($x_j$)) + 1
```

```
• If SP($x_j$) is in $B_1^i$
    o If the size of $B_1^i$ is at least the size of $B_2^i$
        ▪ $\delta = 1$
    o Else
        ▪ $\delta = |B_2^i| / |B_1^i|$
    o $P_i = \min(P_i + \delta, c_i)$
    o Call Release ($x_j$, $i$)
    o Fetch $x_j$ to the cache
    o Move SP($x_j$) to be the MRU of $T_2^i$
    o count(SP($x_j$)) = count(SP($x_j$)) + 1
• If SP($x_j$) is in $B_2^i$
    o If the size of $B_2^i$ is at least the size of $B_1^i$
        ▪ $\delta = 1$
    o Else
        ▪ $\delta = |B_1^i| / |B_2^i|$
    o $P_i = \max(P_i - \delta, 0)$
    o If count(SP($x_j$)) $\leq 2 \cdot \gamma \cdot \text{bound}_i$
        ▪ Call Release ($x_j$,i)
        ▪ Fetch $x_j$ to the cache.
        ▪ Move SP($x_j$) to be the MRU of $T_2^i$
    o If count(SP($x_j$)) $> 2 \cdot \gamma \cdot \text{bound}_i$
        ▪ If $0 \leq C - \sum c_i < \text{size}_i$
            • Call IncreaseBuffer ($x_j$,$i$)
            • If we could not allocate a continuous space of $\text{size}_i$
                • Call Release ($x_j$,$i$)
        ▪ Else
                • Call Allocate ($x_j$,$i$)
            ▪ Count($x_j$) = $\gamma\cdot$bound$_i$
            ▪ $s_i = s_i + 1$
    • If SP($x_j$) is not in $T_1^i$, $T_2^i$, $B_1^i$ or $B_2^i$
        o If ($i > 1$) and (SP($x_j$) has ever been in lists $B_1^i$ or $B_2^i$)
            ▪ Call Demote ($x_j$,$i$)
        o Else
            ▪ If $0 \le C - \sum c_i <$ size$_i$
                • If ($|Q_i| + |T_1^i| + |B_1^i| = c_i$)
                    o If ($|Q_i| + |T_1^i| < c_i$)
                        ▪ Remove the LRU of $B_1^i$
                        ▪ Call Release ($x_j$,i)
                    o Else
                        ▪ Remove the LRU among $Q_i$ and $T_1^i$.
                • Else
                    o If ($|Q_i| + |T_1^i| + |B_1^i| + |T_2^i| + |B_2^i| > c_i$)
                        ▪ If ($|Q_i| + |T_1^i| + |B_1^i| + |T_2^i| + |B_2^i| = 2\cdot c_i$)
                            ▪ Remove the LRU of $B_2^i$
                        ▪ Call *Release* ($x_j$, $i$)
                • Fetch $x_j$ to the cache.
                • Move SP($x_j$) to be the MRU of $T_1^i$
            ▪ Else
                • Call *Allocate* ($x_j$, $i$)
*IncreaseBuffer* ($x_j$, $i$)
    • Do until size$_i$ base-pages are released:
        o $r$ = max rank$_i$
        o Remove LRU among $T_1^r$, $T_2^r$ and $Q_r$
        o $c_r = c_r - 1$
        o If ($c_r < s_r$)
            ▪ $s_r = s_r - 1$
    • Call *Allocate* ($x_j$, $i$)
```

```
Release ($x_j$, $i$)
    • If (($|T_1^i| > P_i$) or ($|T_1^i| = P_i$ and $x_j$ is in $B_2^i$)
        o Take the LRU page between the LRU of $T_1^i$ and the LRU of $Q_i$ and put it as
          the MRU of $B_1^i$
    • Else
        o Take the LRU page between the LRU of $T_2^i$ and the LRU of $Q_i$ and put it as
          the MRU of $B_2^i$
Allocate ($x_n$, $i$)
    • If there is a contiguous empty space of size$_i$ in the cache
        o Fetch $x_j$ to the cache
        o Move SP($x_j$) to be the MRU of $T_2^i$
        o $c_i = c_i + 1$
Demote ($x_n$, $i$)
    • Cancel SP($x_j$)
    • If($i > 1$)
        o Dsize = size$_{i-1}$
    • Else
        o Dsize = 1
    • free = The biggest available continuous empty space of maximum Dsize.
    • if (free > 0)
        o Create super-page $x'_j$ of size free which must contain $x_j$
        o Move $x'_j$ to the MRU of $Q_{\text{free}}$.
    • Else
        • Call Release($x_j$, 1)
        • Fetch $x_j$ to the cache
        • move $x_j$ to the MRU of $Q_1$.
```

4 Results

4.1 Testbed

We implemented the standard CLOCK algorithm, the ARC algorithm and the AMSQM algorithm. We used Valgrind (Nethercote and Seward, 2007) to capture the pages that were used by some of the SPEC – cpu2000 (SPEC, 2000). The SPEC manual explicitly notes that attempting to run the suite with less than 256 MB of memory will cause a measuring of the paging system speed instead of the CPU speed. This suits us well, because our aim is to measure the paging system speed precisely; hence, we simulated a machine with just 128 MB of RAM, although it is obviously a very small memory.

The sizes of the super-pages that we used were 8, 16, 32, 64, 128 and 256 kB. We assumed a tagged TLB of 32 entries for instructions and 64 entries for data.

Both AMSQM and ARC outperform CLOCK by all the parameters in our simulation, so we found no point in presenting the results of CLOCK; therefore, the results presented here are only the ratio between strict ARC and AMSQM.

Let us define:

n: number of memory requests by the benchmark.

p: number of pages that the benchmark accesses.

tm_{ARC}: number of TLB misses when ARC is the replacement algorithm.

tm_{AMSQM}: number of TLB misses when AMSQM is the replacement algorithm.

pf_{ARC}: number of the benchmark's page faults when ARC is the replacement algorithm.

pf_{AMSQM}: number of the benchmark's page faults when AMSQM is the replacement algorithm.

$$\text{tm\_ratio} = 1 - \left(\frac{\text{tm}_{\text{AMSQM}} - p}{\text{tm}_{\text{ARC}}} \right)$$

$$\text{pf\_ratio} = 1 - \left(\frac{\text{pf}_{\text{AMSQM}} - p}{\text{pf}_{\text{ARC}} - p} \right)$$

The TLB misses are shown as the ratio between the TLB misses that AMSQM produces and the TLB misses that ARC produces. When a page is accessed at the first time, any algorithm will have to induce a TLB miss and obviously there is no way to eliminate this TLB miss, so we calculated only the TLB misses of the pages just from the second time they are accessed. The page faults are also shown as the ratio between the page faults that AMSQM produces and the page faults that ARC produces counting for each page only the second and further accesses.

tm_ratio and pf_ratio are the values that represent the calculation of the TLB miss ratio and the page fault ratio, respectively.

4.2 SPEC-2000 results

Figures 1 and 2 show the extra overhead of ARC over AMSQM. Figure 1 shows the tm_ratio of several selected SPEC2000 benchmarks whereas Figure 2 shows the pf_ratio of the same SPEC2000 benchmarks. It can be clearly seen in Figure 1 that AMSQM achieves a higher TLB ratio because of the super-pages usage.

Figure 1 The TLB miss reduction of AMSQM

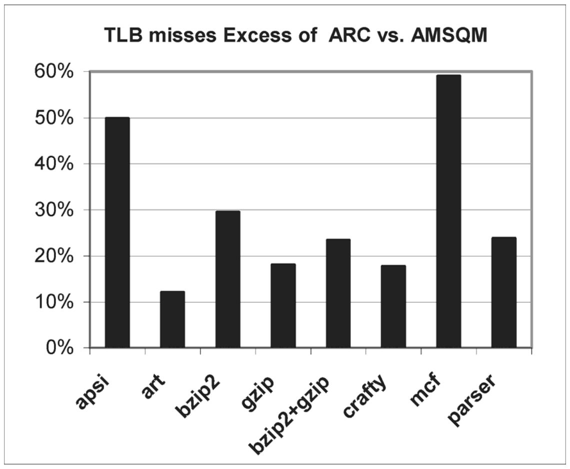
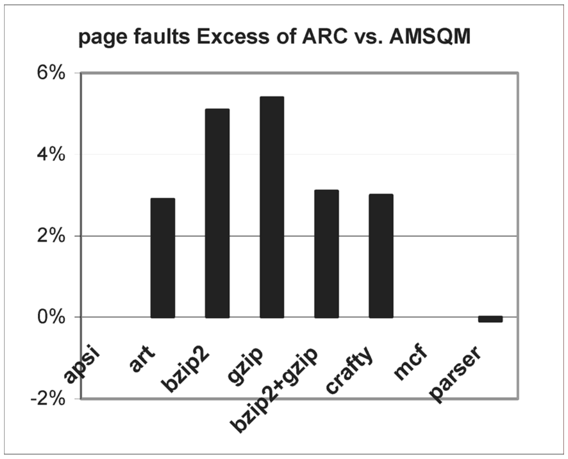


Figure 2 The page fault reduction of AMSQM



Furthermore, AMSQM memory hit ratio is also higher than ARC memory hit ratio in most of the benchmarks as can be noticed in Figure 2. The improvement of the memory hit ratio is because AMSQM takes advantage of the locality principle as it is mentioned above in the introduction section. The other SPEC benchmarks show similar results, so we do not include these benchmarks in this paper.

4.3 Setting the threshold

Figures 3 and 4 show the influence of *threshold* on the system performance. Too high *threshold* harms the TLB hit ratio, whereas too low *threshold* harms the page fault ratio; hence, it can be concluded from Figures 3 and 4 that the best balance of the TLB ratio requirements and the page faults requirements is setting *threshold* to 0.5. On one hand choosing a *threshold* less than 0.5 will yield a good TLB miss ratio, but on the other hand choosing a *threshold* more than 0.5 will yield a good page fault ratio. Setting *threshold* to exactly 0.5 will produce a reasonable result for both the TLB ratio and the page fault ratio.

Figure 3 Influence of *threshold* on TLB misses

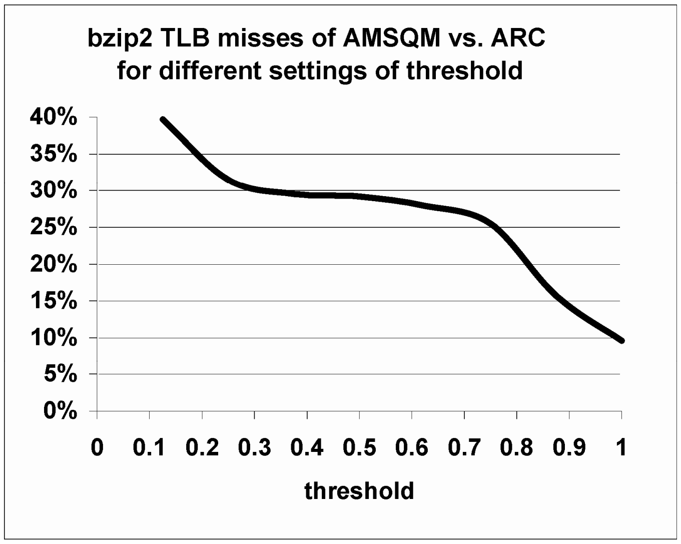
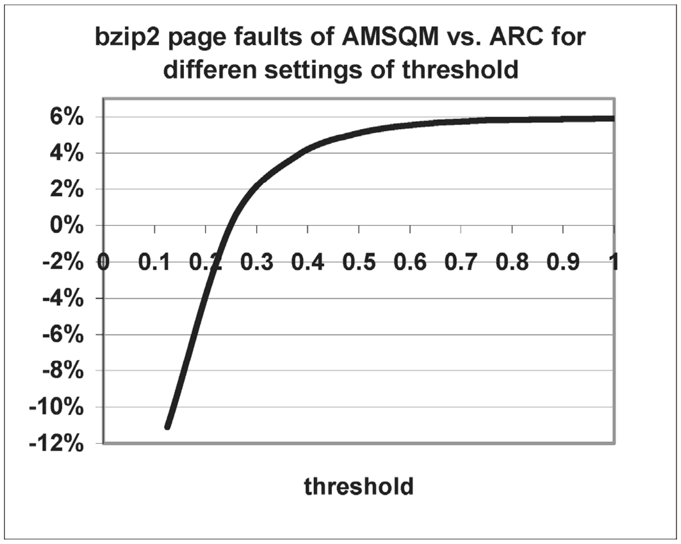


Figure 4 Influence of *threshold* on page faults



We also tested the running of both of the algorithms using the subroutine "clock()" in "time.h" of GNU C compiler. We found the results are quite similar, so we do not include these results in this paper as well.

4.4 Setting β

According to the experiments, we found that AMSQM gives the best results if its parameters are set to the following values:

- $\alpha = 0.5$
- $\beta = 4$
- $\gamma = 0.25$.

Figures 5 and 6 show the effect of different β values on the TLB miss ratio and the number of page faults. It can be clearly concluded from these figures that setting β to 4 gives the best performance in terms of TLB hit ratio and the number of page faults. It can be noticed as well that setting β with low values or alternatively with big values causes a poorer performance of the algorithm. Similar tests were taken to determine the best value of α and γ. The conclusion was that the best value for α is 0.5 and the best value for γ is 0.25.

Figure 5 The influence of β on TLB miss ratio

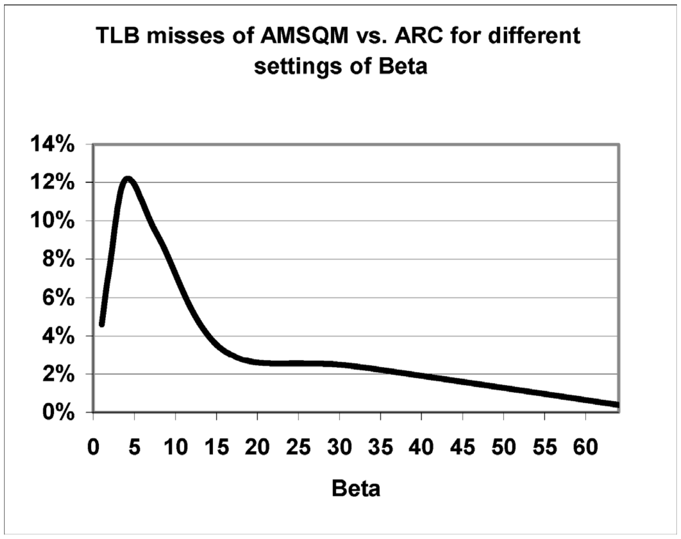
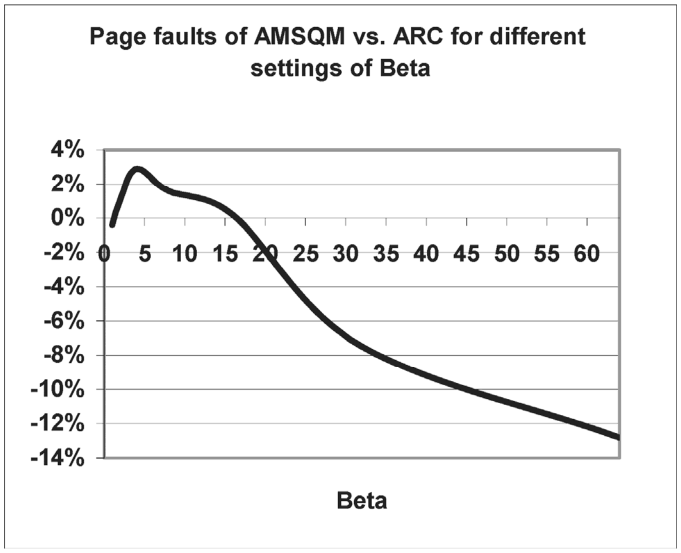


Figure 6 The influence of β on page faults



4.5 Heavy memory consuming benchmarks

As we have mentioned above, during last years, the TLB size has been increased slowly when comparing to the memory increasing rate; hence the TLB coverage has been dramatically reduced. We find it very commonsensical to assume that in the coming years the ratio between the memory size and the TLB size will be even smaller than the current ratio.

It is very uncommon to publish nowadays a new memory management technique or system that is not tested by a heavy workload benchmarking system (Hristea et al., 1997), because the future anticipates a significant increase in the memory usage of the applications; hence we also checked the heavy memory workload scenario.

With the aim of simulated this scenario, we modelled a machine with a TLB coverage that is even smaller than the one we have simulated above. For this purpose, we simulated a machine with 512 MB of RAM and a tagged TLB consists of 32 entries for instructions and 64 entries for data.

Consequently, we had to create new benchmarks that will request for many pages that a machine with 512 MB of RAM cannot handle without causing a thrashing. With the purpose of overloading the memory, we have chosen the heaviest memory consuming benchmarks among the SPEC-CPU2000 benchmarks. The applications which were selected are: apsi, crafty, bzip2 and gzip.

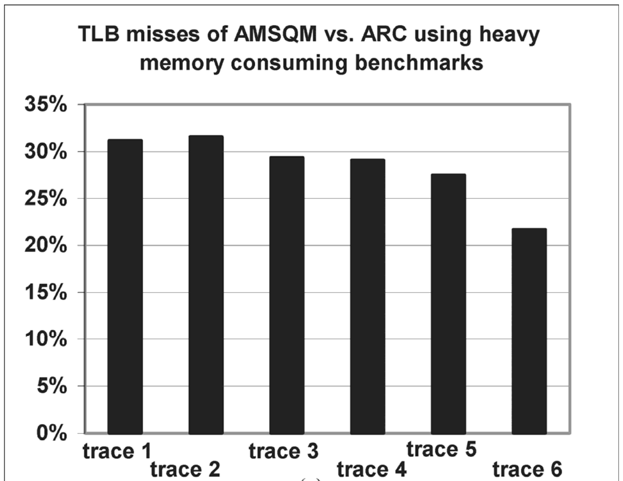
The new traces were created by executing instances of these applications in parallel and merging them into one trace by using the timestamps which we have added to each memory access.

The benchmarks which we have built are defined herein below:

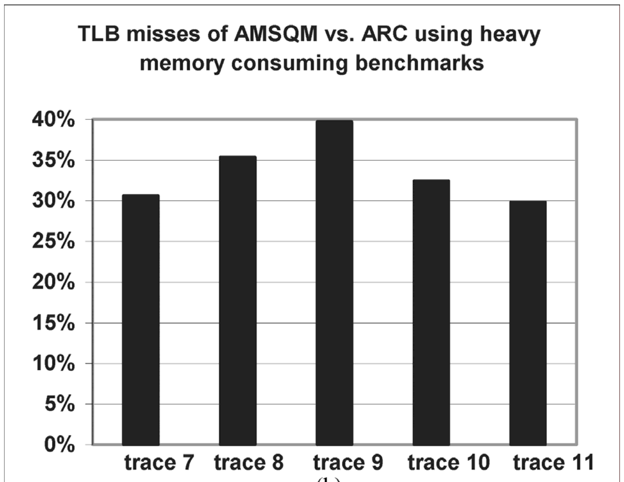
- *Trace 1*: composed of four instances of the application bzip2 executed in parallel.
- *Trace 2*: composed of four instances of the application gzip executed in parallel.
- *Trace 3*: composed of four instances of the application apsi executed in parallel.
- *Trace 4*: composed of four instances of the application crafty executed in parallel.
- *Trace 5*: composed of two instances of the application bzip2 and two instances of the application gzip, executed in parallel.
- *Trace 6*: composed of two instances of the application bzip2 and two instances of the application apsi, executed in parallel.
- *Trace 7*: composed of two instances of the application bzip2 and two instances of the application crafty, executed in parallel.
- *Trace 8*: composed of two instances of the application gzip and two instances of the application apsi, executed in parallel.
- *Trace 9*: composed of two instances of the application crafty and two instances of the application apsi, executed in parallel.
- *Trace 10*: composed of two instances of the application gzip and two instances of the application crafty, executed in parallel.
- *Trace 11*: composed of the instances of apsi, crafty, bzip2 and gzip, executed in parallel.

Figure 7(a)and (b) shows the TLB miss ratio of AMSQM versus ARC. It can be easily seen that AMSQM TLB misses are significantly fewer than ARC TLB misses. It can be noticed that trace 2 achieves a higher TLB hit ratio comparing to strict gzip. This can be explained as a result of the TLB coverage in this experiment which is significantly smaller than the TLB coverage in the previous experiment; thus a base-page replacing algorithm such as ARC will experience enormous number of TLB misses, whereas an algorithm such as AMSQM that utilises wisely the super-paging mechanism will gain a higher TLB coverage and hence will produce relatively less TLB misses comparing to ARC. The significant improvement in the TLB ratio of AMSQM comparing to ARC can be similarly explained in the other traces.

Figure 7 (a) First group of heavy traces TLB misses and (b) second group of heavy traces TLB misses



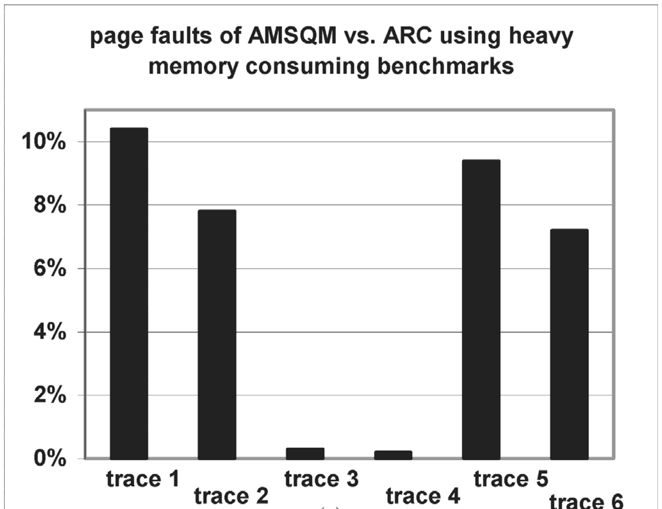
(a)



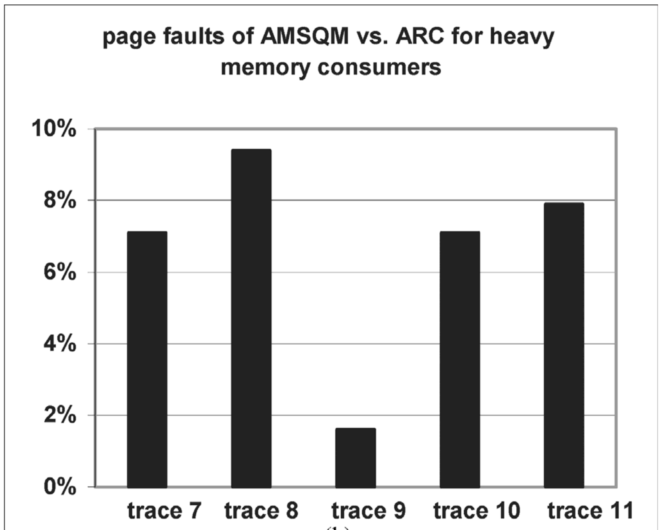
(b)

Figure 8(a)and (b) shows the page faults of AMSQM versus ARC. It can be clearly seen that AMSQM achieves a higher memory hit ratio in all the benchmarks because of a good utilisation of super-pages and based on the locality principle. However, the improvements vary from 0.2% for trace 4 up to 10.4% for trace 1. We found out that ARC performs efficiently in trace 4 (and trace 3), that is, does not produce many page faults because there is enough space in the main memory, and therefore AMSQM's improvement is relatively small.

Figure 8 (a) First group of heavy traces page faults and (b) second group of heavy traces page faults



(a)



(b)

Yet, we found it very encouraging that for an extreme heavy memory consumer benchmarks (such as: trace 1, trace 2, traces 5–7, trace 8 and trace 11), AMSQM achieves a notably higher memory hit ratio, since contemporary applications require a big portion of the memory and reducing the number of the page faults in such applications can significantly improve the overall performance.

5 Conclusions and future work

The new adaptive super-page replacement algorithm AMSQM has been presented. We have shown that AMSQM usually achieves a higher TLB coverage than ARC and also a better page fault ratio in most of the benchmarks we have used.

This paper shows another important aspect of the super-paging environment. We believe operating systems have an improper attitude towards the super-page replacement algorithm selection. They usually just copy the old algorithms of the traditional paging mechanism with no attention to the new super-paging environment. This brings about an improvement of the hardware support for a smaller TLB miss ratio, but the software support for a smaller TLB miss ratio is considerably poorer.

We show a way to adapt one of the most recent algorithms to the super-paging environment with the aim of obtaining a better TLB hit ratio.

In the future, we would like to find ways to set the AMSQM parameters (α, β, γ) dynamically. In our experiments, we have found that the values we used for these parameters are best for most of the benchmarks; however, there is a minority of benchmarks that have a preference of other values and there are also few benchmarks that will have a preference of adaptively modified values. Therefore, we believe that adaptively modified values can improve the performance of several benchmarks. Another issue that should be addressed as well is the mutual influence of the processes scheduled together (Wiseman and Feitelson, 2003).

In addition, we would like to find a pattern for super-pages reoccurrence. Such a pattern can improve the efficiency of the super-page promotion decisions. The traditional threshold parameter seems to be not enough for taking the most beneficial decision. Some applications such as Wiseman et al. (2004), Wisemanm (2001), and Klein and Wiseman (2003) have a pattern of supper-pages reoccurrence and the operating system can take an advantage of it.

We would also like to integrate the suggested patch into the Linux kernel. The current results are encouraging and they support our belief that the new replacement algorithm can significantly enhance the memory management mechanism in the above mentioned two manners: better TLB hit ratio and fewer page faults.

6 Acknowledgement

The authors would like to thank SUN Microsystems for their donation.

References

Abouaissa, H., Delpeyroux, E., Wack, M. and Deschizeaux, P. (1999) 'Modelling and integration of resource communication in multimedia applications with high constraints using hierarchical Petri nets', *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics (SMC-99)*, Vol. 5, pp.220–225, Tokyo, Japan.

Corbato, F.J. (1968) 'A paging experiment with the multics system', *MIT Project MAC Report*, *MAC-M-384*, May.

Friedman, M.B. (1999) 'Windows NT page replacement policies', *Proceedings of 25th International Computer Measurement Group Conference*, pp.234–244, December.

Ganapathy, N. and Schimmel, C. (1998) 'General purpose operating system support for multiple page sizes', *Proceedings of the USENIX Annual Technical Conference*, New Orleans.

Geppert, L. (2003) 'The new indelible memories', *IEEE Spectrum*, Vol. 40, Part 3, pp.48–54.

Hristea, C., Lenoski, D. and Keen, J. (1997) 'Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks', *Supercomputing, ACM/IEEE 1997 Conference*, p.45, November.

Itshak, M. and Wiseman, Y. (2008) 'AMSQM: adaptive multiple superpage queue management', *Proceedings of IEEE Conference on Information Reuse and Integration (IEEE IRI-2008)*, Las Vegas, Nevada, pp.52–57.

Jiang, S., Chen, F. and Zhang, X. (2005) 'CLOCK-Pro: an effective improvement of the CLOCK replacement', *Proceedings of 2005 USENIX Annual Technical Conference*, Anaheim, CA, pp.323–336, April.

Jiang, S. and Zhang, X. (2002) 'LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance', *Proceeding of 2002 ACM SIGMETRICS*, Marina Del Rey, CA, pp.31–42, 15–19 June.

Johnson, T. and Shasha, D. (1994) '2Q: a low overhead high performance buffer management replacement algorithm', *Proceedings of the Twentieth International Conference on Very Large Databases, VLDB'94*, Santiago, Chile, pp.439–450, September.

Khalidi, Y.A., Talluri, M., Nelson, M.N. and Williams, D. (1993) 'Virtual memory support for multiple page sizes', *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, Napa, California, October.

Kim, J., Choi, J., Kim, J., Noh, S., Min, S., Cho, Y. and Kim, C.A. (2000) 'Low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references', *4th Symposium on Operating System Design and Implementation*, San Diego, CA, pp.119–134, 23–25 October.

Klein, S.T. and Wiseman, Y. (2003) 'Parallel huffman decoding with applications to JPEG files', *The Computer Journal*, Swindon, Vol. 46, No. 5, pp.487–497.

Lee, D., Choi, J., Kim, J-H., Noh, S.H., Min, S.L., Cho, Y. and Kim, C.S. (2001) 'LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies', *IEEE Transactions on Computers*, Vol. 50, No. 12, pp.1352–1360.

Megiddo, N. and Modha, D.S. (2003a) 'ARC: a self-tuning, low overhead replacement cache', *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'2003)*, San Francisco, pp.115–130, 31 March–2 April.

Megiddo, N. and Modha, D.S. (2003b) 'One Up on LRU', *Login: The Magazine of the USENIX Association*, Vol. 28, No. 4, pp.7–11, August.

Megiddo, N. and Modha, D.S. (2004) 'Outperforming LRU with an adaptive replacement cache algorithm', *IEEE Computer*, pp.4–11, April.

Navarro, J. (2004) 'Transparent operating system support for superpages', PhD Thesis, Department of Computer Science, Rice University, April.

Nethercote, N. and Seward, J. (2007) 'Valgrind: a framework for heavyweight dynamic binary instrumentation', *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, CA, USA, June.

O'Neil, E., O'Neil, P. and Weikum, G. (1993) 'The LRU-K page replacement algorithm for database disk buffering', *Proceedings of SIGMOD '93*, Washington, DC, May.

Smaragdakis, Y., Kaplan, S. and Wilson, P. (2003) 'The EELRU adaptive replacement algorithm', *Performance Evaluation*, Vol. 53, No. 2, pp.93–123, July.

SPEC (2000) *CPU-2000*. Standard Performance Evaluation Corporation, Warrenton, VA, Available at: http://www.spec.org/.

Subramanian, I., Mather, C., Peterson, K. and Raghunath, B. (1998) 'Implementation of multiple page size support in HP-UX', *Proceedings of the USENIX Annual Technical Conference*, New Orleans.

Wallace, R.F., Norman, R.D. and Harari, E. (2006) 'Computer memory cards using flash EEPROM integrated circuit chips and memory-controller systems', *US Patent No. 7106609*.

Wang, J. (2008) 'Improving decision-making practices through information filtering', *Int. J. Information and Decision Sciences*, Vol. 1, No. 1, pp.1–4.

Wiseman, Y. (2001) 'A pipeline chip for quasi arithmetic coding', *IEICE Journal – Transactions on Fundamentals*, Tokyo, Japan, Vol. E84-A, No. 4, pp.1034–1041.

Wiseman, Y. (2005) 'ARC based superpaging', *Operating Systems Review*, Vol. 39, No. 2, pp.74–78.

Wiseman, Y. and Feitelson, D.G. (2003) 'Paired gang scheduling', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 6, pp.581–592.

Wiseman, Y., Schwan, K. and Widener, P. (2004) 'Efficient end to end data exchange using configurable compression', *Proceedings of the 24th IEEE Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, pp.228–235.

Zhou, Y., Chen, Z. and Li, K. (2004) 'Second-level buffer cache management', *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, Vol. 15, No. 7, pp.505–519.