Colorable Memory

Jochen Liedtke
IBM T. J. Watson Research Center
jochen@watson.ibm.com

November 10, 1996

Abstract

Recent research reported page coloring to influence system performance. Application-driven coloring techniques are restricted on conventional memory systems by two facts: (a) there are equally many page frames per color and this number is static; (b) for recoloring a page, it must be copied to a different page frame. This paper describes a scheme permitting to change the color of any page frame dynamically. Thus the number of frames per color can also be changed dynamically. Furthermore, cache-clean pages can frequently be recolored without copying. Cache flushing is never required. The scheme is simple and affects solely main-memory address decoding, not the cache system. For small and medium-sized memories, no additional hardware is needed. Very large main memories require to extend the physical address bus what is probably too expensive.

1 Rationale

Due to their size, second-level caches cannot be addressed solely by the page-offset bits of an address. The additionally required bits are taken from the physical page number part. These bits define the *color* of a physical page. In a virtual memory system, the mentioned (physical) color of any virtual page is determined by the operating system's page allocation. Therefore, the operating system has some control over the second-level cache by coloring strategies.

Recent research reported page coloring to influence second-level cache miss rates and thus overall system performance. Kessler and Hill [1992] investigated static page coloring and dynamic bin hopping techniques to avoid color conflicts in the second-level cache. Bugnion et al. [1996] got even better results on multiprocessors by compiler-directed page coloring. Bershad et al. [1994] (see also [Romer et al. 1994]) improved cache performance by monitoring cache misses and dynamically recoloring pages. Liedtke et al. [1996] used coloring techniques to partition the second-level cache for higher predictability in mixed real-time and timesharing systems.

Here, we simply take the results mentioned above as given and try to find a *no-cost* hardware mechanism which overcomes the space restrictions of coloring and avoids in some cases the copying or cache-flushing costs of dynamic recoloring.

Space restrictions: The number of available page frames per color is fixed for all colors. This restricts in particular small memories. For example, in a 4 Mbyte system with 4-K pages and a 128-K cache, coloring conflict must be expected as soon as 2 Mbyte of memory are used (assuming a random equal distribution of requested colors). Partitioning caches for better real-time predictability puts this to an extreme. The technique frequently permits only to use one or two pages of a certain color while all further pages of the color have to remain unused.

Time restrictions: Dynamic recoloring of a used page requires to copy the complete page.

Presumably, the mentioned space restrictions are much harder than the time restrictions.

2 Colorable Memory

The coloring-related space restrictions would diappear, if all page frames in main memory were usable for arbitrary physical colors. If the color of a page frame could also be changed dynamically without flushing the cache or copying the page, the costs of dynamic recoloring would disappear as well.

Assume a hardware memory of 2^k pages, a page size of 2^p and 2^c colors. The total size of hardware memory is then 2^{k+p} bytes. The second-level cache has a size of $n \times 2^{c+p}$ bytes, where n is its associativity.

Conventionally, an equally sized and contiguous part of the physical address space is mapped 1:1 to this hardware memory; physical memory basically *is* hardware memory.

Instead, we introduce an address level below the physical addresses, the *hardware address space*. Address translation then maps virtual $\rightarrow$ physical $\rightarrow$ hardware. The physical address space is divided into aligned *boxes* of size 2^{c+p} (see figure 1). We use a physical address

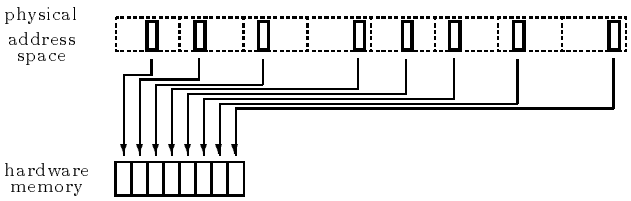

Figure 1: *Colorable Physical Address Space.*

space which is *color*-times (2^c) larger than the real hardware memory.

> *The central idea is that all physical pages of a box are mapped to the same hardware page frame (see figure 2).*

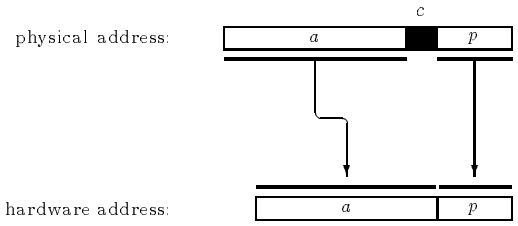

Figure 2: *Free Coloring.*

The physical-to-hardware translation is very simple: the c color bits of the physical address are ignored when the memory hardware is accessed. Basically, compared to a conventional memory system, "only some address lines are rewired".

The effect is that all pages inside a box are aliases of the same hardware page. By appropriate construction of the virtual-to-physical mapping, the operating system can ensure that never more than one physical page of a box is used simultaneously. Then the system behaves like a conventional memory system, except that the operating system can choose arbitrary values for the c color bits of the physical addresses. The memory can be recolored like in figure 3 so that, for example, no page frames are blocked unusable by cache partitioning.

The mentioned mechanism can be implemented without changing the secondary cache architecture and with standard memory chips. However, c-bits wider second-level tags are required for accessing the same amount of memory as in a conventional system. For the same reason, main memory is limited to 2^{n-c} bytes, where n is the physical-address width. For example, a PentiumPro with its 4-way set-associative 256-Kbyte second-level cache (16 colors) needs 4 of its 36 physical address bits for the color so that up to 1 Gbyte main memory can be supported. An Alpha with its 40

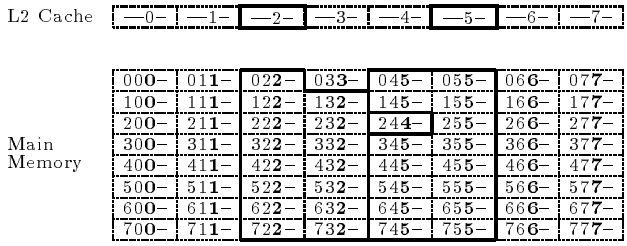

Figure 3: *Colorable Memory.*

physical address bits, 8-K pages and a 2-way cache of 4 Mbyte (256 colors) would be restricted to 4 Gbyte of main memory.

The mentioned scheme does neither affect the cache logic nor the physical address bus. Therefore, it is transparent for all multiprocessor cache-coherence protocols.

Coloration, the above described transformation of physical to hardware addresses, substantially differs from virtual to physical address translation. Coloration occurs after the virtual address was translated into the associated physical address. Coloration is not required for each data access, only at cache miss, write back or when accessing uncached memory regions. Since the transformation can be implemented by appropriate wiring the address lines, it needs no additional time.

3 Recoloring Costs

In general, recoloring a page makes the cache inconsistent. Flushing the cache prior to any recoloring is clearly unacceptable. Instead, we need solutions which in most cases impose no costs to the system and are never more expensive than the conventional recoloring by copying. Although a page-selective cache-flush operation might solve the problem, we look for purely software-based methods which do not need special cache hardware. For reasons of simplicity, we first describe solutions for write-through cache systems and extend the methods later to the more complicated write-back caches.

We differentiate between *allocation recoloring* and *on-the-fly recoloring*. The first operation occurs when a hardware page frame is used with a new color for a newly allocated page. Allocation recoloring has two characteristic features: (a) the content of the old page is discarded, perhaps after saving it to disk/network; (b) the content of the new page is explicitly written to memory, either by reading it from disk/network or by copying a source page[1] or by writing an initial pattern[2]

[1] if the operating system uses copy-on-write techniques

[2] Usually, operating systems cannot tolerate undefined memory for security reasons. Initializing freshly allocated pages is there-

On-the-fly recoloring occurs when a used paged gets a new color without loosing its content.

3.1 Write-Through Caches

Memory writes are never delayed in a write-through cache. Therefore it suffices to ensure that stale data is never read from the cache after recoloring. Fortunately, allocation recoloring automatically invalidates all cache entries related to the old page: since the new page has a different physical address and the TLB will never deliver the old one, cache tags corresponding to the old physical address will never match. If the page frame is re-used again with the first color, initializing the page frame (or reading it from disk) overwrites cache entries which belonged to the very first page and are not replaced in the meantime. *Concluding:* allocation recoloring is for free.

On-the-fly recoloring is as simple as allocation recoloring, if the new color was not used before for this hardware page frame. As well, read-only pages can be recolored for free, since they do not change their content and "old" cache entries therefore always hold correct values. A simple algorithm is therefore to copy only read/write-mapped pages prior to recoloring from the old to the new color so that all new-color-related cache entries are updated. Since conventional memory requires always to copy the page, this algorithm performs never worse but sometimes better than in the conventional case.

A closer analysis shows that the above algorithm can be improved. Potentially stale cache entries can only exists if the new color was already used before *and the page was modified in the meantime.* We introduce a field version per frame and per color, initially set to $+\infty$. To keep the algorithm simple, we define $+\infty + 1 = 0$. Furthermore, we assume that the processor and/or operating system supports a modified-bit per page:

```
recolor on-the-fly:
    if is modified_frame
        then version_frame,oldcolor INCR 1 ;
             is modified_frame := false
    fi ;
    if version_frame,newcolor < version_frame,oldcolor
        then map (dest, frame, new color) ;
             map (source, frame, old color) ;
             disable cache fills (dest) ;
             disable cache fills (source) ;
             copy page (source, dest)
    fi ;
    version_frame,newcolor := version_frame,oldcolor .
```

We assume that cache fills can be enabled/disabled on a per page basis. Disabling cache fills for source and destination (which is only relevant for write-allocate caches) avoids unnecessary cache flooding. Depending on the characteristics of the concrete memory system and operating system, enabling cache fills might sometimes be the better solution.

fore required.

3.2 Write-Back Caches

Write-back caches contain dirty cache lines which can be written back to memory arbitrarily delayed. We call a page *cache-clean* if the cache contains no dirty lines related to this page. Cache-clean pages can be handled exactly in the same way as described for write-through caches.

The basic task is to find out whether a page is cache-clean and to make it cache-clean otherwise. Fortunately, many cache-coherency protocols and DMA hardware ensure that writing a page to disk or network makes it cache-clean. As well reading a page into memory invalidates all priorily corresponding entries. Therefore, only discarding a modified page (*) requires additional overhead:

```
recolor allocate:
    if old page must be swapped out
        then write page to disk (frame, old color)
    elif is modified_frame {not cache-clean}
        then map (source, frame, old color) ;
             disable cache write back (source) ;
             initialize page (source) (*)
    fi ; {old page is cache-clean}
    if new page must be swapped in
        then read page from disk (old color) ;
             is modified_frame := false
             {new page is cache-clean}
        else map (dest, frame, new color) ;
             initialize page (dest)
             is modified_frame := true
             {not new page is cache-clean}
    fi .
```

The on-the-fly recoloring algorithm of section 3.1 can be modified accordingly. Not cache-clean pages are made cache-clean by the same technique. Slightly improved and more complicated algorithms, in particular such co-operating with on-the-fly recoloring, are analyzed in [Xxxxxxx 19xx].

Concluding: Write-back caches permit in many cases as cheap recoloring as write-through caches. Sometimes an additional page-copy operation is required to make a page cache-clean. However, highly expensive operations like a complete cash flush are never required.

4 Variants

4.1 Partially-Colorable Memory

To weaken main-memory restrictions, colorable memory can be used like an overflow or victim cache: the main part of the physical address space is mapped 1:1 to the hardware memory; only a smaller part of the

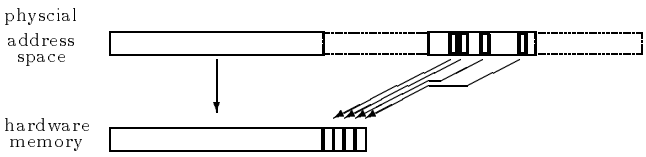

Figure 4: *Physical Address Space with Colorable Region.*

hardware memory is used as colorable memory (see figure 4). Then the mentioned restrictions apply only to the colorable region.

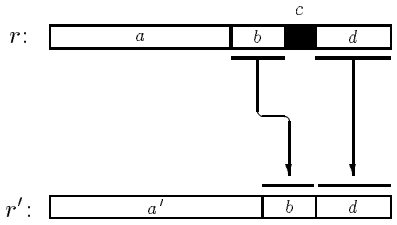

Figure 5: *Physical Address → Hardware Address.*

The transformation should be applied only to a specific *colorable region* of the physical address space. The region is identified by the higher address bits a. It starts at physical address $2^{c+p}a$ and has a size of 2^{k+c+p}. The corresponding hardware memory begins at hardware address $2^p a'$ and has a size of 2^{k+p}.

A physical address r which is subject to this special physical → hardware translation, consists of higher value bits a, an k-bit field b which determines the actual box, an c-bit color field which holds the physical color and the offset field d. The hardware address r' then is formed by the unchanged offset d, the box field b shifted right by c bits, replacing a by a'.

4.2 Multiple Page Sizes

Colorable memory cam also be used for multiple page sizes. The basic mechanism is implemented per page size: for each page size 2^{p-i}, there is a dedicated region of the physical address space with the appropriate color field ($c+i$ bits). All these regions can share the same hardware memory area, e.g. like in figure 6.

Assume that page sizes $2^p, 2^{p-1}\ldots$ should be supported and that 2^k pages of size 2^p should be freely colorable. Then the size of region 0 is 2^{k+c+p}. Region i handles 2^{p-i}-byte pages, has a region size of $2^{k+c+i+p}$ bytes and contains 2^{k+i} boxes. (The box size remains constant over all regions: $2^{c+i} \cdot 2^{p-i} = 2^{c+p}$.) Therefore, each region of the physical address space needs a hardware memory area of $2^{k+i} \cdot 2^{p-i} = 2^{k+p}$ bytes.

If all physical regions share the same hardware memory area, we have 2^{k+p}-bytes of main memory which can

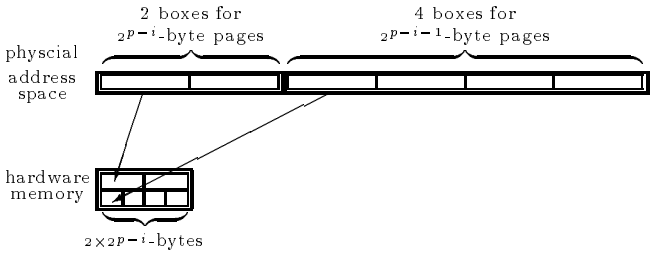

Figure 6: *Multiple Colorable Regions.*

be occupied by arbitrarily colored pages of freely mixed sizes 2^{p-i}.

5 Conclusion

If page coloring and recoloring is relevant for operating systems, it can be easily supported by no-cost hardware. The scheme basically ignores color-related address bits for main-memory access. Therefore, the basic scheme can only be used for small and medium-sized systems whose memory is at most $1/colors$ of the physical address space. However combining conventional memory regions and colorable regions weakens this restriction significantly.

References

Bershad, B. N., Lee, D., Romer, T., and Chen, B. 1994. Avoiding conflict misses dynamically in large direct-mapped caches. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, pp. 158–170.

Bugnion, E., Anderson, J. M., Mowry, T. C., Rosenblum, M., and Lam, M. S. 1996. Compiler-directed page coloring for multiprocessors. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, pp. 244–255.

Kessler, R. and Hill, M. D. 1992. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems 10,* 4 (Nov.), 11–22.

Liedtke, J., Härtig, H., and Hohmuth, M. 1996. Predictable caches in real-time systems. SFB-Bericht 358–G2–1/96 (March), SFB 358 at TU Dresden, Dresden.

Romer, T. H., Lee, D. L., Bershad, B. N., and Chen, B. 1994. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, pp. 255–266.

Xxxxxxx, X. 19xx. xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx. In *xxxx xxxxxx xxxxx xxx xxxx xxxx xxxxx xxxxx xxx*, xxxxxxxxxx, pp. xx–xx.