

# Shared Resource Control in Multi-Core Processors

Xiao Zhang (student), Sandhya Dwarkadas, and Kai Shen  
Department of Computer Science, University of Rochester

Emerging multi-core processors present new challenges to operating systems in terms of shared resource allocation and control. In particular, a shared on-chip cache can result in sub-optimal performance if simultaneously executing processes compete for capacity or incur conflicts. Without additional hardware support, the control technique available to the operating system in order to effect cache partitioning is page coloring. Page coloring is a classic technique that can be used for software-based manipulation of cache utilization. It has been used to reduce cache misses [5, 4, 1, 2, 6], improve on-chip network traffic [3], and most recently, to demonstrate the viability of cache partitioning in multi-core processors [7]. We build on this prior work to develop a practical and automated page coloring technique for fair and performance-effective multi-core cache partitioning.

We demonstrate the benefits of cache partitioning via page coloring on our Linux-based Xeon Dual-Core platform. Figure 1 compares the performance of the hardware-default L2 cache sharing with that of each application running stand-alone, in addition to using page coloring to partition the cache according to the knee of its miss ratio curve. As shown, page coloring can improve performance by as much as **23%** on average.

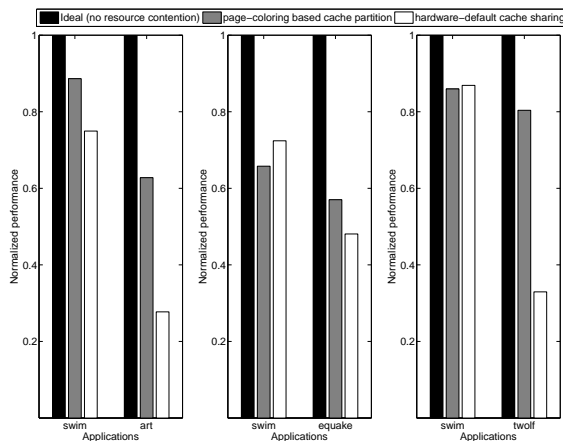


Figure 1. Performance comparison between cache sharing and partition

However, in order for the approach to be practical, several implementation issues arise. The first issue is that of constraining the allocated memory space. Imposing page color restrictions on an application means only a portion of the memory can be allocated to this application. When the OS runs out of pages of a certain color, the application can either evict some of its own pages to secondary storage or steal pages from other page colors. The former can result in dramatic slowdown due to page swapping while the latter may yield negative effects on other applications' performance (e.g. cache pollution).

Another challenge is the high overhead of online recoloring. For example, different pairs of applications on a multi-core platform may need different cache partition allocations. At one time quantum, application A is running together with B and has its working pages all in color *b* at this time; later it might need to recolor its working pages to color *c* to suit a new partner C. Table 1 shows a typical set of CPU-intensive applications and their working set size at time quantum granularity (100 milliseconds on Linux by default). Without

Benchmark	Working Set Size per time quantum	Percentage of full memory footprint
gzip	1.8K pages	4%
wupwise	13.7K pages	31%
swim	27.2K pages	57%
mgrid	11.2K pages	81%
applu	19.4K pages	43%
mesa	1.9K pages	81%

Table 1. Working set size at time quantum granularity.

extra hardware support, recoloring a page means memory copying and takes several microseconds on existing commodity platforms. Recoloring **1K+** pages each time quantum is not affordable.

To solve these problems, we propose hot-page based page coloring, which only recolors those most frequently accessed pages. Our experiments on several SPECcpu2000 applications listed above suggest that more than **80%** of memory references access only **20%** of the address space.

There is a subtle issue about physically contiguous page allocations. If someone grabs all pages of a certain color out of *N* colors, then *N* physically contiguous pages in memory will not be available. In such cases and if such contiguous page allocation is necessary, the OS may need to evoke a page reclamation algorithm to coalesce contiguous pages. However, in our Linux page-allocation trace, contiguous page allocation is not frequent (less than 3%). In particular, we observe very few DMA page requests except during OS booting. This is probably due to pooling of DMA pages in a way similar to the slab allocator. On the other hand, we observe physically-contiguous page allocations for new task structures in 64-bit Linux but they are at a 2-page granularity only. Modern OS memory management schemes (e.g., the buddy list allocator in Linux) are commonly optimized for physically contiguous page allocations, which is inappropriate when most memory allocations are under page coloring. Our preliminary experiments show that by simply keeping a large portion of memory on low order buddy lists, we achieve a **4.4-fold** reduction in page allocation overhead.

We anticipate that once the individual mechanisms are tied together, software-based cache partitioning will achieve improved performance even in dynamic environments where the workload is not known a-priori.

## References

- [1] B. Bershad, D. Lee, T. Romer, and J. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *ASPLOS'94*, pages 158–170, San Jose, CA.
- [2] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. Compiler-directed page coloring for multiprocessors. In *ASPLOS'96*, pages 244–255, Cambridge, MA.
- [3] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *Micro'06*, pages 455–468, Orlando, FL.
- [4] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *TOCS*, 10(4):338 – 359, November 1992.
- [5] T. Romer, D. Lee, B. Bershad, and J. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *OSDI'94*, pages 255–266, Monterey, CA.
- [6] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page replacement. In *ICS'99*, Rhodes, Greece.
- [7] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *WIOSCA'07*, San Diego, CA.