Page-Mapping Techniques to Reduce Cache Conflicts on CC-NUMA Multiprocessors

Zhiyuan Li

Department of Computer Sciences

Purdue University, West Lafayette, IN 47907,USA

E-mail:li@cs.purdue.edu

Jian Huang

Department of Computer Science and Engineering

University of Minnesota, Minneapolis, MN 55455, USA

E-mail: huangj@cs.umn.edu

Guohua Jin

Department of Computer Science

Rice University, Houston, TX 77005

E-mail: jin@cs.rice.edu

Abstract Page-coloring and bin-hopping are two well-known page mapping schemes for reducing cache conflicts. Previous work found bin-hopping to have 4% less cache miss rate than page-coloring on uniprocessor machines. Using execution-driven simulations, we find that bin-hopping significantly outperforms simplistic page-coloring on CC-NUMA multiprocessors. In certain cases, bin-hopping has 32% to 58% less execution time and over 60% fewer cache misses. By using part of the memory ID bits to hash the page color during page-mapping, we improve the performance of page-coloring to match that of bin-hopping on CC-NUMA multiprocessors.

Keywords: CC-NUMA multiprocessor, data-task affinity, page mapping, private cache

1 Introduction

Cache-Coherent Non-Uniform Memory Access (CC-NUMA) multiprocessors become increasingly attractive as an architecture which provides a transparent access to local and remote memories and a good scalability. Stanford DASH [10] and FLASH [9], MIT Alewife [1], University of Toronto NUMAchine [19], and Sun's S3.mp [13], as well as commercial products including the Sequent STiNG [12], Hewlett-Packard SPP, and Silicon Graphics Origin 2000 [16].

A CC-NUMA machine has a number of nodes connected by an interconnection network. Each node consists of one or a few processors, a private cache hierarchy, and a local memory module. Each node has a Node ID (NID), each processor has a Processor ID (PID), and each memory module a Module ID (MID). The local memory modules of all nodes compose of a shared, contiguous physical memory space used by the operating system (OS). References to a local memory module avoid the network latency while the references to a non-local memory module (remote references) may experience a two-hop or three-hop network delay. Although this is transparent to the programmer, the OS and the compiler must reduce the average memory reference latency by reducing coherence actions, conflict misses and remote memory references.

To reduce coherence actions, a good compiler should try to schedule those parallel tasks which share the same data to the same processor. To reduce remote memory references in the event of cache misses, the compiler should also try to align data allocation with the tasks. These two issues have been studied extensively recently [2, 3, 7, 11, 14]. The success of such attempts by the compiler depends on the degree of data-task affinity that can be found in a given program. Obviously, the OS must cooperate with the compiler by honoring the MID designation and hence preserving the data-task affinity cultivated by the compiler.

This paper focuses on the issue of reducing cache set conflicts by properly mapping a virtual page to a physical page. Previous works examine this issue primarily for uniprocessors and multiprocessors with uniform memory access (UMA) and they show that careful page placement by the OS can reduce cache set conflicts considerably [4, 8]. This paper examines the page placement issue in the context of

CC-NUMA machines, a relatively new multiprocessor architecture, where the MID decision comes into play.

Various page-mapping techniques have been proposed in the past, including page-coloring, bin-hopping, best-bin, hierarchical method [8], compiler-assisted page-coloring [4] and dynamic re-mapping [15]. Page-coloring and bin-hopping are simple and hence the most popular ones. Silicon Graphics Inc., adopts page-coloring scheme in its products, while DEC ships OSF/1 with bin-hopping. Previous experiments find these two schemes perform comparably on uniprocessor machines, with bin-hopping delivering 4% fewer cache misses than page-coloring [8]. We make two contributions studying these two schemes in this paper. First, using an execution-driven CC-NUMA machine simulator, we find that bin-hopping can outperform page-coloring scheme by over 70% in terms of miss rates and by 10 to 45% in terms of execution time. Second, we find that, by hashing the page color with part of the MID bits, we can improve the performance of page-coloring to closely approach that of bin-hopping. Four popular SPEC Floating Point benchmark programs were parallelized for experiments. The simulator used for all experiments simulates a CC-NUMA machine with a 2-MB secondary cache and a 16-KB primary cache on each of the 16 four-issue super-scalar processors.

The rest of the paper is organized as follows: Section 2 discuses the extensions of page coloring and bin-hopping in CC-NUMA environment. It also discusses the issue of unnecessary cache misses in page-coloring. Section 3 describes the experimental set-up. Section 4 analyzes the results and section 5 concludes the paper.

2 Careful Page Mapping in CC-NUMA Machines

On a paging-based virtual memory system, the OS translates a virtual address to a real address by placing a faulting virtual page in a physical page frame. Assuming a real-indexed cache, which is common today, the set index (SI) in the real address determines which cache-set will cache the content at that address (Figure 1). Part of the SI is inside the page-offset and it remains the same before and after the address

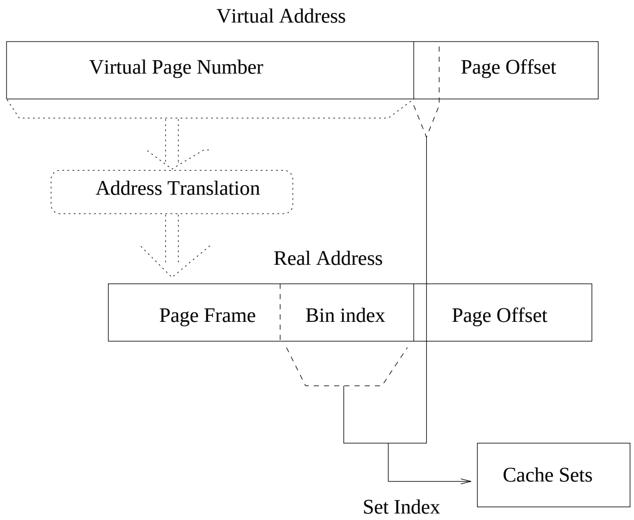

Figure 1: Description of Cache-Bin

translation. The rest of the SI is part of the physical page number and it defines a number of cache-sets which can cache the contents of the given page. These cache-sets together are called a *cache-bin* and the part of SI which defines the cache-bin is considered as the *color*, or the *cache-bin ID* of the page [8]. Obviously, there exists a certain number of physical pages which have the same color in a given system. If a particular color is overly used, then excessive cache-set conflicts may result.

A high performance computer, especially a modern shared-memory multiprocessor, usually has a large physical memory. As a result, a typical page replacement scheme, e.g. an *approximate LRU scheme*, can often find a pool of physical page frames, e.g. the n least-recently used frames, which are equally favorable, as far as future page-faults are concerned, as the store for the next faulting virtual page. This fact leaves a considerable freedom for the page mapping scheme to select a color. Given a virtual address, one may call those bits at locations corresponding to the color bits as the *virtual color*. In the *page-coloring* mapping scheme, the OS tries to select a physical page, from the available pool, that has the same color as the virtual color, while in *bin-hopping*, a physical page with a cache-bin ID which follows that of the previously allocated cache-bin is assigned. The page-coloring scheme picks a page randomly from the available pool if the desired color is unavailable, while bin-hopping resorts to

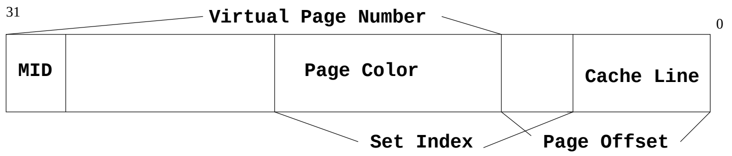

Figure 2: A Virtual-Address Format

the next cache-bin in the row which is available from the pool. There is no apparent advantage to either scheme as far as the page fault rate is concerned, since both schemes will pick an arbitrary frame from the pool when a desired color is unavailable.

On CC-NUMA machines, the page mapping scheme must perform yet another important step, namely, to choose the physical memory module for the faulting virtual page. This is usually done by designating a set of $n = log_2(p)$bits in the physical address to identify the MID. Here p stands for the number of nodes in the whole system. For convenience, we assume that each node has only one processor. Hence, MID and NID are the same for each individual node.

The MID and SI bits are usually separated so that all physical addresses from different memory modules could be mapped to the whole cache, giving the OS the greatest flexibility to influence cache mapping. (If n MID and SI bits were overlapped in hardware, the cache would in effect be divided into 2^nportions, and each memory module can only be cached to one of those 2^nportions. This would limit the utilization of the cache.) We assume that the MID bits are also separated from the page offset such that one page resides in one processor's memory in its entirety. Other than these constraints, the MID positions in the physical address and the virtual address is quite flexible, although it must be fixed on the given hardware and the OS. Current CC-NUMA multiprocessors, such as SGI's Origin 2000 [16] and Toronto's NUMAchine [19], place the MID at the highest bits in the physical address.

For certain applications with regular data sets, the compiler can often find opportunities to allocate the data in such a way that the processors tend to find the required data in their respective local memories [2, 3, 7, 11, 14]. In order for such opportunities to materialize, the compiler must be able to present its MID decision to the OS and the OS must honor such a decision. An easy way is to designate MID bits in a virtual address and leave them unchanged during address translation. Figure 2 illustrates the virtual-address format adopted in this paper. In irregular applications and in certain regular applications, quite often the compiler cannot find a favorable allocation for a particular set of data, e.g. an array. In such cases, the compiler will usually distribute the particular data set evenly across the memory modules. For example, it may partition an array into blocks of the size b and distribute the blocks across the modules, where b is equal to the array size divided by the number of nodes in the system.

Without proper page-mapping, an array evenly distributed on CC-NUMA can have many array elements, apart by a constant stride, whose cache set indices are identical. If a processor makes a sequence of references to these elements across the memory modules, cache conflicts can be severe. The page-coloring scheme behaves poorly in this regard. Consider an example from program Ora in the SPEC benchmarks [17]. Ora has a parallel loop at the outmost level whose iterations can be executed by multiple processors simultaneously. Several arrays are referenced in that loop with subscripted-subscripts. It is difficult for compilers to allocate these arrays in such a way that each processor executing the parallel loop will find its array operands in the local memory. On the other hand, when these arrays are evenly distributed across memory modules, individual processors will access array elements from different modules. The first column of Table 1 shows a sub-sequence of virtual pages referenced by one of the processors. (The virtual address trace shown here is generated by an execution-driven simulator, described in Section 3, which simulates the execution of the machine code generated by a Silicon Graphics f77 compiler for an Origin 2000 machine.)

Assuming sixteen processors, a cache-line size of 32 bytes, and a page size of four kilobytes, the second column of Table 1 lists the MID which is the top hexadecimal digit of the virtual page number, and the third column lists the virtual color which is the last two digits. The page-coloring scheme will map most of these virtual pages to physical frames of the color "00". In contrast, bin-hopping, using a globally managed cache-bin ID to allocate pages, can spread the virtual pages with the same color to physical frames of different colors.

The above problem with page-coloring can be solved by a technique which hashes the virtual color

Table 1: A Sequence of Virtual Page Numbers (in Hexadecimal) from Ora.

Virtual Page Number	MID	Virtual Color	Replacing Bits (Binary)	Physical Color after Replacing
00fff	0	ff	00	3f
e1000	e	00	11	c0
11000	1	00	00	00
d1000	d	00	11	c0
21000	2	00	00	00
b1000	b	00	10	80
c1000	c	00	11	c0
01000	0	00	00	00

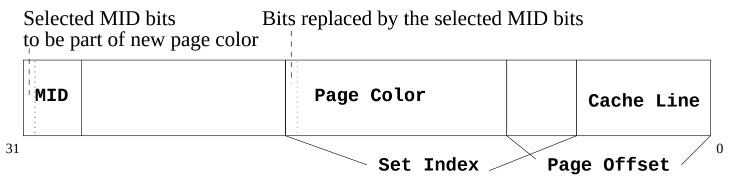

Figure 3: Part of Page Color Bits are replaced by selected MID bits

with part of the MID bits, resulting possibly different physical colors. Suppose we have n bits for the page color, we can take k MID bits and treat them as part of the color of a page and then carry out page-coloring based on the modified color. We call these k bits the *replacing bits*. This is explained in Figure 3. When we select one MID bit as part of the color, we essentially divide the cache-sets into halves. Suppose the MID bit we pick is the highest bit and the number of MID bits is n, then the addresses with a MID of $2^{(n-1)} - 1$ or smaller will map to the first half of the cache, and the addresses with a MID of $2^{(n-1)}$or greater will occupy the second half of the cache. It may happen that we would fail to find a wanted page, in which case a page is picked randomly from the free-page pool. Table 1 illustrates the case of using two replacing bits (shown in Column 4) and the resulting physical colors. Four physical colors are referenced now instead of two. We have performed a number of experiments to examine the behavior of page-coloring and bin-hopping, as well as the effectiveness of hashing the page-color with MID bits, on a CC-NUMA machine.

3 Experimental Setup

Each program in our experiments is parallelized and instrumented by the Panorama interprocedural parallelizing compiler [6]. The output program is then compiled by SGI's *f77* compiler for an Origin 2000 machine running the IRIX 6.2 OS. The -$O3$ optimization flag is set for the compiler. The executable object code is then fed to our multi-processor simulator (NUMAsim). We select Block Scheduling, also known as Simple Scheduling on SGI multiprocessors, to be the default scheduling technique. This technique divides an n-iteration parallel loop into chunks of n/p iterations for p processors. Each chunk is then assigned to a processor. Our experiments showed that other scheduling schemes are inferior to simple scheduling for the selected benchmark programs.

NUMAsim is an execution-driven multi-processor simulator based on MINT [18]. However, modifications are made to support multiple-issue, out-of-order execution and weak-ordered memory consistency. The key parameters of our processor model and component latency are summarized in Table 2. The bandwidth of each memory port and each network interface is assumed to be one word per CPU cycle.

These parameters are selected based on those of several existing and developing commercial systems. The retirement of instructions is in-order. We simulate a CC-NUMA system of 16 nodes. Each node has two levels of non-blocking cache, the on-chip 16-KB level-one (L1) cache, and the off-chip 2-MB level-two (L2) cache. Level-two caches of different nodes are kept coherent based on a write-invalidate policy. Inclusion property [5] is maintained and all writes to the L1 cache are written through to the L2 cache at the same time. An eight-entry write-buffer is used to hold the write-contents if the corresponding write-operation is a miss in L2 cache, and program control is then returned to the CPU. The content will be written to the cache asynchronously. Sixteen outstanding misses can be tolerated.

Table 2: Latency of Components (in CPU Cycles)

Parameter	L1 Cache	L2 Cache	Interconnection	Memory
Latency	1	9	$100/hop$	40

All page mapping techniques are applied based on the level-two cache. The associativity of the level-one cache is fixed as 2-way in our experiments. The Panorama compiler uses a data-task co-allocation scheme [14] to align the data with tasks. (We inspect the alignment decisions to see their qualities.) The Panorama compiler then instruments the FORTRAN source code by inserting directives to identify the starting and ending address of an array and to specify the data allocation decisions. The simulator uses the inserted information and re-maps addresses dynamically during the simulation.

Table 3: Speed Up and Miss Ratio Summary

Program	Speed-Up	L2 Cache Miss	L1 Cache Miss	Remote Ref.	Coherence Miss
Swm256	9.5 – 12.5	0.41 – 1.86%	9.9 – 10.3%	0.16 – 0.30%	0.17 – 0.72%
Ora	14.5 – 16.5	0.007 – 2.22%	2.01 – 2.21%	0.006 – 3.88%	0.0001 – 0.001%
Mgrid	4.3 – 6.5	0.84 – 1.94%	7.50 – 16.55%	0.33 – 0.79%	0.16 – 0.43%
Tomcatv	2.5 – 4.0	1.53 – 7.69%	13.52 – 22.62%	0.24 – 0.99%	0.18 – 1.07%

3.1 Test Programs

Four numerical FORTRAN programs are used in the experiments. They are Swm256 and Tomcatv from SPEC 92 benchmarks and Ora and Mgrid from SPEC 95 benchmarks. In order to shorten the simulation time, we reduce the iterative time steps in some programs. The number of steps is reduced from 1200 to 60 in Swm256, 100 to 5 in Tomcatv. We believe these settings are long-enough for the programs to demonstrate corresponding characteristics. All data sizes are not changed. In Swm256, 14 arrays have 257 * 257 words each. Tomcatv has 7 arrays of 257*257 double words each. Mgrid uses 3 arrays of 333,944 bytes each.

The whole program of Ora is mainly one parallel loop. It makes the majority of its references to private scalars, which makes the references to the few arrays particularly important because the rest of the references are extremely fast. We discussed the characteristics of these arrays in Section 2. What follows next briefly describes the other three programs.

Swm256 (shallow water model) is a weather prediction program. Most of the parallel loop nests in this program are two-level deep, with each level containing 256 iterations. The data-task co-allocation algorithm parallelized the outer level loop nest. Since the outer loop index is the column index of each array, the second dimension of each array is distributed across the processors. There are a few arrays which are swept in columns in certain parallel loops but in rows in other parallel loops.

Tomcatv is a mesh generation solver. All of the major parallel loops are two-level deep and each level has 256 iterations. Three of the major parallel loops access the arrays by columns, while the middle two sweep by row. To compromise, the second dimension of each array is distributed across the nodes. In each iteration, one neighboring column from each direction and one neighboring row from each direction are accessed as well as the current column or row. In Tomcatv, among the nine major arrays, two of them (RXM and RYM) have the same virtual color.

Mgrid is a 3-dimensional potential field computation program. The largest three shared arrays are 326KB each, competing for six cache bins. Because the most frequently executed loops access the arrays by different different dimensions, the Panorama compiler chooses to distribute the arrays along the first dimension in equally-sized blocks. Many of the cache-bin conflicts come from data in different memory modules, but with the same cache-bin ID.

Various statistics are collected. The execution time is measured in CPU cycles. Since not all references go through the level-two cache, we define the *extended cache miss* as the number of misses of the level-two cache over total number of references. We also define *memory miss rate* to be the number of remote memory accesses over the total number of cache misses and write-backs. Coherence misses refer to the misses caused by invalidation. Table 3 lists the speed-up and cache miss information for the benchmarks tested. The data summarizes the statistics we collected over a variety of simulation setup. It also shows the parallelism and memory reference characteristics for these programs. The next section presents and discusses data collected from the experiments.

4 Data and Discussions

We first want to verify that the simplistic page-coloring scheme indeed results in over-used colors and that the technique of hashing the virtual color with MID bits indeed helps correct the problem. We show three figures which depict the dynamic memory reference patterns in program Ora, under simplistic page-coloring (Figure 4), bin-hopping (figure 5) and page-coloring with 2 replacing bits (Figure 6), respectively. The level-2 cache is 4-way set-associative in all three cases. In these figures, the "Time" axis is the program execution time progressing from time quantum 0 to time quantum 400 (each quantum is 35,000 CPU cycles). The "Page Color" axis marks individual physical colors of the physical addresses of the referenced data and the "Reference Count" axis marks the number of times a distinct physical page of a particular color is referenced within a particular time quantum. Note that references to different physical pages of the same color are marked separately. Hence, denser dots at a particular time indicates a greater number of different physical pages of the same color.

In all three figures, the sparse bands with high reference counts represent references to private scalar variables. Despite their high counts, most of these references are primary-cache hits and they are extremely fast. References to the shared arrays in Ora are represented by the dense stripes near the

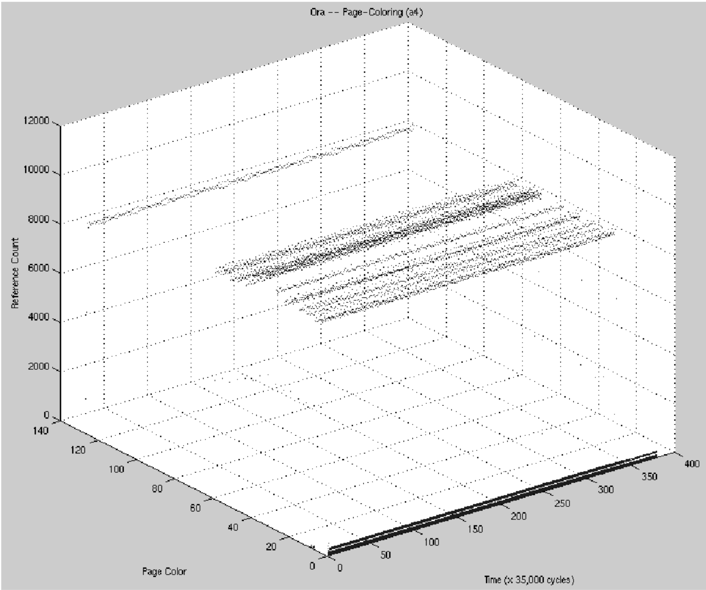

Figure 4: Reference Pattern of Ora under Simplistic Page-Coloring.

bottom. As we discussed in Section 2, these references are much more sensitive to cache conflicts, because in case of cache misses, they will likely have to visit remote memories and incur heavy latency.

As Figure 4 shows, under simplistic page coloring, all physical pages of the shared arrays concentrate on two colors. The chance for cache conflicts is extremely high. In contrast, Figure 5 shows that bin-hopping scheme spreads the colors much better for those shared arrays. Figure 6 shows that a result between simplistic page coloring and bin-hopping can be achieved by hashing the virtual colors with two MID bits. Although not shown here, using four MID bits to replace part of the virtual color bits would spread the physical color nearly as well as bin-hopping in the case of Ora. The other three programs in our experiments exhibit similar behavior.

Figure 7 to Figure 14 graph the execution time and the extended cache miss rate over different associativities of the level-two cache. In the figures, PC stands for page-coloring and p_x means that

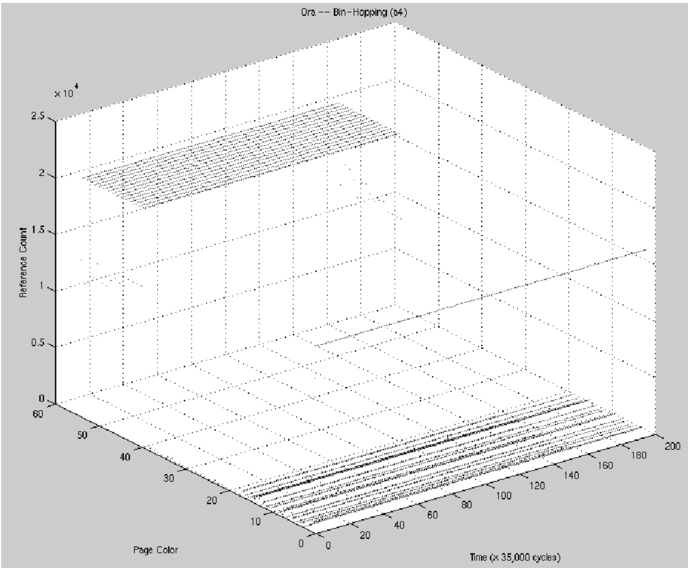

Figure 5: Reference Pattern of Ora under Bin-Hopping.

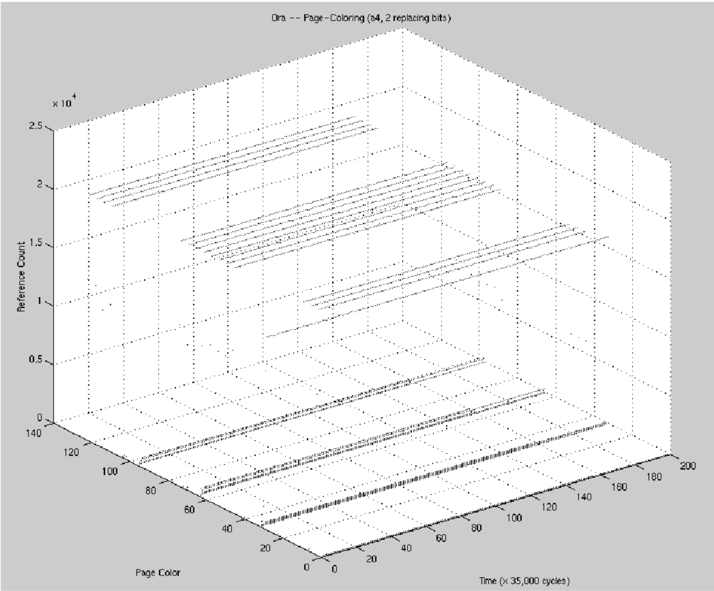

Figure 6: Reference Pattern of Ora Using Two Replacing Bits.

x MID bits are used to replace the highest x page-color bits. *BH* refers to bin-hopping with globally managed cache-bin ID's.

From the data, we observe that page-coloring with zero replacing bits has an inferior performance to all other schemes. In almost all cases, bin-hopping with a global bin-ID delivers the best performance among the studied alternative schemes. Compared with previous work [8] which studies careful page mapping on uniprocessors, the difference between page-coloring and bin-hopping on CC-NUMA machines is quite striking, even though different test programs are used. On uniprocessors, the difference in the miss rates are reported to be no more than 4% for the two mapping schemes. On CC-NUMA, however, our data show that, in the case of two-way set-associative caches, bin-hopping has over 60% lower cache miss rate and between 32% and 58% less execution time than page-coloring. Our discussions in previous sections explained the main reason.

From the same data, we observe a general trend that the introduction of more replacing bits tends to improve the performance further. One of the most important findings from the data, however, is that two replacing bits are often sufficient to bring the performance to the level of bin-hopping. Using too many replacing bits will exhaust the available page frames of certain colors faster, which can potentially lead to higher page faults. (Like previous works, we do not study page faults in this paper.)

The performance of the test programs under different mapping schemes and using different cache associativities can be observed in many obvious ways. Since the data generally fits expectations and the figures are quite clear, we omit more detailed discussions.

5 Conclusion

It has long been recognized that the bin-hopping page-mapping scheme has certain inherent advantages over page-coloring in its effect on reducing cache conflicts. However, the performance difference between these two observed on uniprocessors was within a few percents. The work in this paper finds a much greater performance difference between the two schemes when they are applied to CC-NUMA multiprocessors, due to the situations in which a processor may consecutively address data from different memory modules. Such situations are quite common in irregular applications. In certain regular applications, computation on transposed matrices can also create such undesirable situations.

We find a simple technique which can effectively overcome the difficulty of page-coloring, bringing the program performance almost to the level of that of bin-hopping. This technique simply replaces part of the virtual color bits with part of the memory ID bits when the OS performs page mapping. Future work needs to be done in order to see how our observations stand on other programs and other system configurations, including programs with a larger data set running on a larger system.

Like previous works, we have mainly focused on the cache performance and the execution speed without taking into account context-switching time and page-fault penalty. This is because, when considering a single program's execution, the SPEC benchmarks produce few page faults on a system with a large storage. On the other hand, simulating a meaningful multiprogramming environment involves many complex issues which are good topics for future studies.

Acknowledgement

This research was supported in part by a National Science Foundation CAREER Award under grant CCR-950254 and by a gift from Cray Research, Inc., a Silicon Graphics Company. Florian Mueller at Purdue University assisted in collecting part of the experimental results.

References

[1] Agarwal A *et al.*, The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor, Technical Report 454, MIT/LCS, 1991.

[2] Agarwal A, Kranz D and Natarajan V, Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors, in: *Proc. International Conference on Parallel Processing*, Volume I: Architecture, St. Charles, IL, 1993, 2–11.

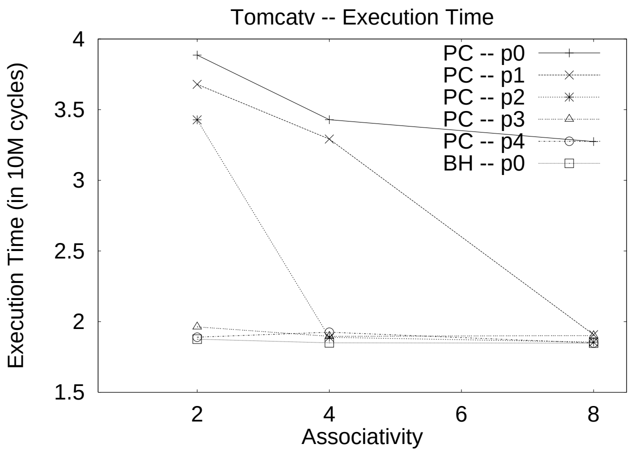

Figure 7: Execution Time Curves for Tomcatv

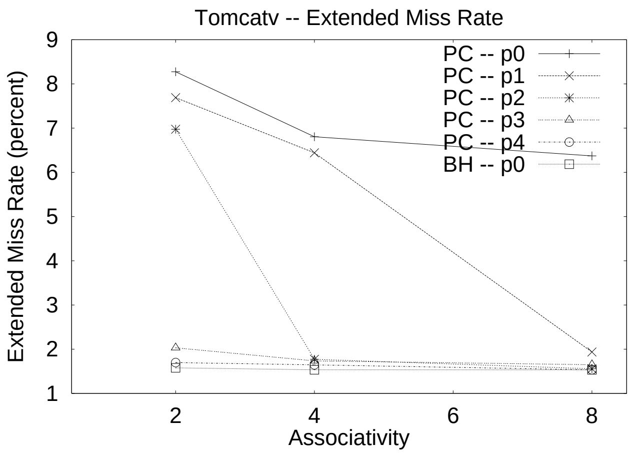

Figure 8: Extended Cache Miss Rate for Tomcatv

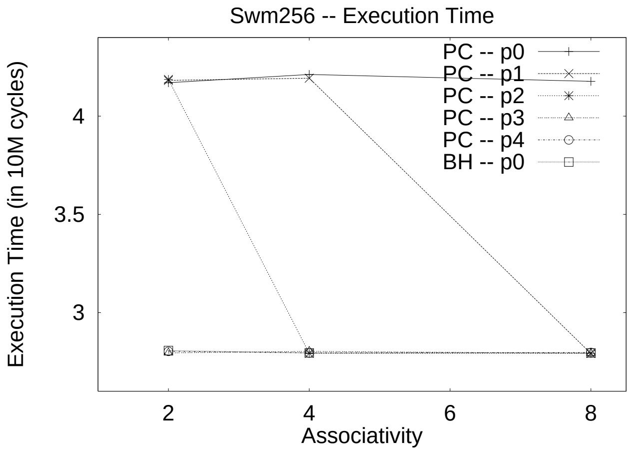

Figure 9: Execution Time Curves for Swm256

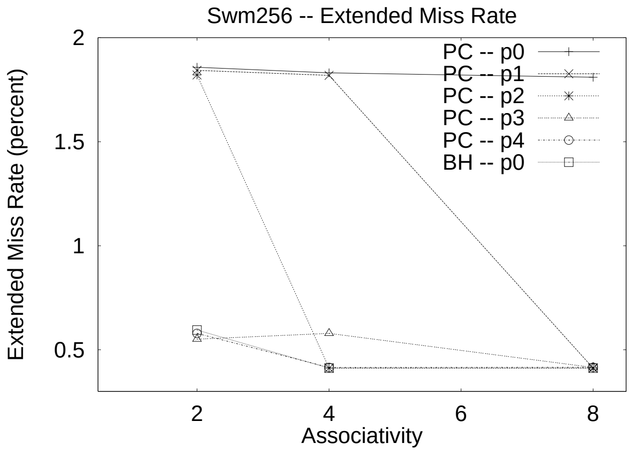

Figure 10: Extended Cache Miss Rate for Swm256

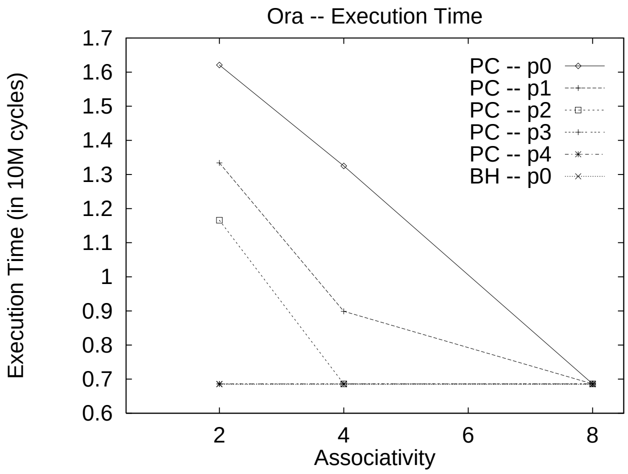

Figure 11: Execution Time Curves for Ora

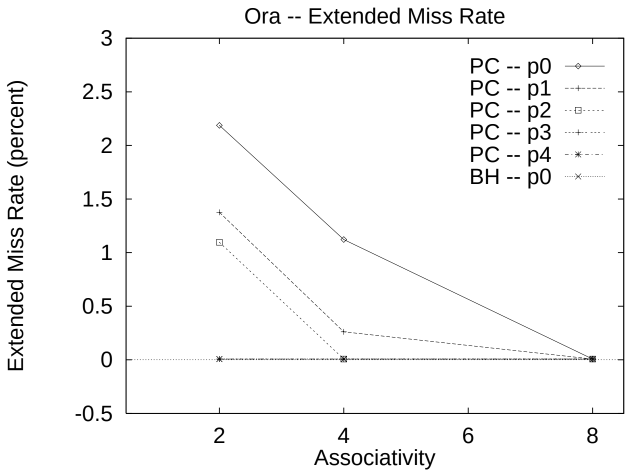

Figure 12: Extended Cache Miss Rate for Ora

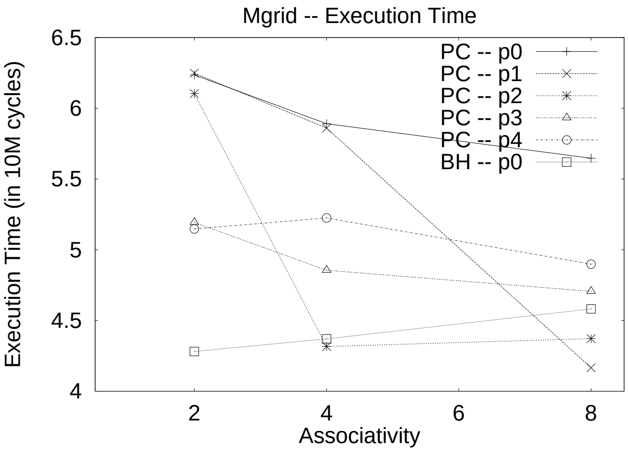

Figure 13: Execution Time Curves for Mgrid

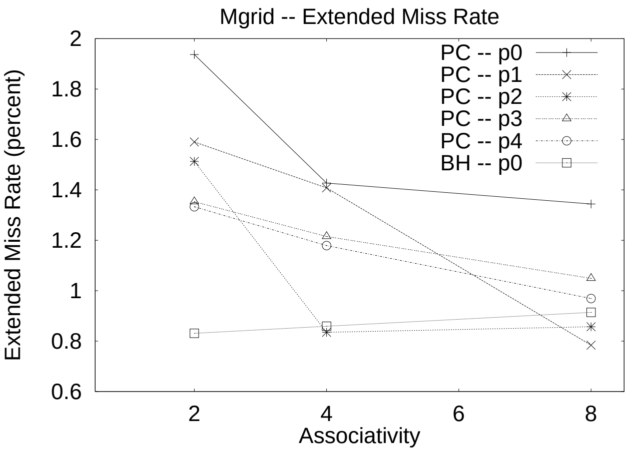

Figure 14: Extended Cache Miss Rate for Mgrid

[3] Anderson J and Lam M S, Global optimizations for parallelism and locality on scalable parallel machines, in: *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Imp.*, June 1993, 112–125, Cambridge, Massachusetts.

[4] Bugnion E, Anderson J, Mowry T, Rosenblum M and Lam M S, Compiler-directed page coloring for multiprocessors, in: *Proc. of the 7th Int. Sym. on Architectural Support for Programming Languages and Operating Systems*, October 1996, Cambridge, Massachusetts.

[5] Chen T and Baer J L, Reducing Memory Latency via Non-blocking and Prefetching Caches, in: *Proc. of the Fifth Int'l Symposium on Architectural Support for Programming Languages and Operating System*, October 1992, Boston, Massachusetts.

[6] Gu J, Li Z and Lee G, Experience with efficient array data-flow analysis for array privatization, in: *Proc. of Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, June 1997*, 157–167, Las Vagas, Nevada.

[7] Kennedy K and Kremer U, Automatic data layout for High Performance Fortran, in: *Proc. Supercomputing '95*, December 1995, San Diego, California.

[8] Kessler R E and Hill M D, Page Placement Algorithms for Large Real-Indexed Caches, *ACM Trans. on Computer Systems*, 10(4), November 1992, 338–359.

[9] Kuskin J *et al.*, The Stanford FLASH multiprocessor, in: *Proc. Int. Sym. on Computer Architecture*, 1994, 302–313, Chicago, Illinois.

[10] Lenoski D *et al*, The Stanford DASH Multiprocessor, *Computer*, 25(3), March 1992, 63–79.

[11] Li W and Pingali K, Access Normalization: Loop Restructuring for NUMA Computers, *ACM Trans. on Computer Systems*, 11(4), November 1993.

[12] Lovette T and Clapp R, STiNG: A CC-NUMA computer system for the commercial marketplace, in: *Proc. Int. Sym. on Computer Architecture*, 1996, 308–317, Philadelphia, Pennsylvania.

[13] Nowatzyk A *et al.*, The S3.mp scalable shared memory multiprocessor, in: *Proc. Int. Sym. on Computer Architecture*, 1995, Santa Margherita Ligure, Italy.

[14] Nguyen T N and Li Z, Interprocedural Analysis for Loops Scheduling and Data Allocation, *Parallel Computing, Special Issue on Languages and Compilers for Parallel Computers*, 24(3), 1998, pp. 477–504.

[15] Romer T, Lee D, Bershad B and Chen J, Dynamic page-mapping policies for cache conflict resolution on standard hardware, in: *Proc. of the First Symposium on Operating System Design and Implementation*, Nov. 94, Monterey, California.

[16] Silicon Graphics, Inc., Origin and Onyx2 Programmer's Reference Manual. Document Number: 007-3410-001, 1996.

[17] Standard Performance Evaluation Corporation, *SPEC Newsletter*, Vols. 1–9, 1989–1997.

[18] Veenstra J E and Fowler R J, MINT Tutorial and User Manual, Technical Report 452, Dep. of Computer Science, University of Rochester, June 1993.

[19] Vranesic Z *et al.*, The NUMAchine Multiprocessor. Technical Report CSRI-324, Computer Systems Research Institute, University of Toronto, 1995.