Near-Optimal Page Placement based on Application Memory References

Zhe Wang, Pei Cao and Shankar Pasupathy
Department of Computer Science
University of Wisconsin-Madison
{zhe,cao,shankar}@cs.wisc.edu

Abstract

As microprocessor technology evolves, effective use of hardware caches becomes crucial of application performance. For physically-indexed caches with limited associativity, operating system's page placement can affect cache utilization significantly. In this paper, we study the technique of using application memory references to derive a nearly optimal page placement and its performance benefits. We describe an algorithm called "Min-Weight", which builds a graph capturing the conflicts between virtual pages and uses graph coloring techniques to improve page placement. We show that Min-Weight can reduce 2nd-level cache miss ratios by 10% to 60% compared with existing page placement strategies such as Bin-Hopping. Furthermore, the algorithm works well in the context of page placement training and can be used for profile-based optimization.

1 Introduction

As the performance gap between memory system and hardware caches increases, effective use of the hardware caches is crucial to application performance. For physically-indexed caches with limited associativity, page placement strategies (which determines which physical page a virtual page is mapped to) has a large impact on cache hit ratio [2]. As a result, many research studies have investigated heuristics for careful page placement [10] and online adjustment of page placement based on various monitoring information [3, 11]. In this study, we investigate techniques that yield nearly optimal page placement given sufficient information on application memory references.

Three factors motivated our study. As the latency difference between main memory and hardware caches approaches a factor of 100 to 1000, it is very important to improve the hit ratio in the second level caches. Meanwhile, the increasing amount of DRAM in modern computers means that page placement is less constrained by the operating system's page replacement policies — in other words, it is likely that the free-list of physical pages will contain all cache colors. Third, improvement in profiling technologies will generate more information on data access patterns, making it possible to perform optimizations based on application memory references.

Our main result is a page placement algorithm called "min-weight," which reduces cache miss ratio by 10% to 60% using information on application memory reference traces. We show that the theoretical problem of computing optimal page placement is computationally infeasible (NP-Complete), and develop the heuristic "min-weight" based on graph coloring methods. The algorithm extracts information of potential conflicts between pages from memory reference traces, and uses graph coloring to group pages into different cache bins. Using a variety of applications including database applications, typical PC applications and SPEC95 benchmarks, we show that the page placement under min-weight can reduce cache misses by up to 60placement strategies (such as bin-hopping) in 4-way set associative caches. Furthermore, the cache misses under the page placement is very similar to those under fully-associative caches.

We examine the use of the min-weight algorithm as a way to obtain application-specific page placement through training data. We experiment with the database application Postgres, and find that the page placement obtained from one query works reasonably well for other queries, reducing the cache miss ratio by up to 30% compared with the existing page placement heuristics. We also examine how well the algorithm performs when only limited sampling of application memory reference traces is available. We find that the performance of the algorithm degrades gradually as the sampling rate decreases, and at the sampling rate of 1%, the algorithm still outperforms many existing page coloring strategies.

Though the algorithm is not yet feasible in an online setting, our study demonstrates the following:

- For a considerable portion of applications, more careful page placement, which takes into account the global reference patterns in an application, can reduce cache misses significantly, even for 4-way set associative caches;
- The essential information that optimal page placements need is the "interference" graph, which captures the conflicts between virtual pages at run-time. Compile time analysis that yields similar information can use the min-weight algorithm to derive application-specific page placement.
- For applications who access similar data across across input data sets, the min-weight algorithm can be used to obtain the page placement best for the application from a training data set, and to advise future page placement for the application.

Below, we first discuss the previous work in this area, and then describe the algorithm.

2 Related Work

Many research studies investigated using careful page placement to improve cache hit ratio. Kessler and Hill [10] proposed and evaluated several page placement strategies using a collection of memory reference traces from various systems. Bershad et al [3, 11] developed run-time collision-detection techniques via hardware and software means, and evaluated strategies for online adjustment of page placement to reduce collisions. Our study differs from these efforts in that our goal is to find optimal or nearly optimal page placement algorithms, understand what information about application memory references these algorithms need, and evaluate the potential of using optimal page placement in maximizing cache utilization.

As a result of these early studies, most operating systems today use some form of careful page placement, using either the bin-hopping algorithm or the virtual coloring algorithm [10]. Recent efforts at continuous profiling demonstrated the potential of performance optimization based on profile data gathered at run-time [2, 13, 6]. The performance optimization, however, has been mostly focused on code layout and rearrangement. Profile-based page placement would be a natural next step to improve operating system's page placement strategy. On the other hand, there is little understanding of how an application's page placement should be adjusted if some information about its data accesses is available. The lack of knowledge on this subject thus motivated our study.

Improving cache utilization has been a key research problem as current technologies continue to increase the latency gap of accessing main memory and accessing hardware caches. A large number of research projects are investigating various ways to improve cache hit ratio, for example, skewing the indices used to probe the cache or selectively bypassing the cache; it is impossible to list all of them here. Our study differs from these studies by assuming that the hardware mechanism for caching is fixed, and instead investigating what can be done through OS mapping mechanisms.

Several recent studies also point out the potential and importance of page placement in the performance of modern computer systems. For example, Sites el al [2] shows that the difference in page placement can lead to a 12% difference in completion time for some applications. As microprocessor technology evolves, page placement will play an even greater role at affecting application performance.

A number of compiler research projects have been focusing on improving cache performance. Some have been looking at changing the layout of data structures to improve cache utilization. Our heuristics for optimal page placement offer insights on what information a compiler can gather to improve page placement.

Our study borrows results from studies on graph algorithms [8, 1]. Compiler writers face a similar problem in allocating registers to variables. Chaitin [5] proposed a heuristic for graph coloring which eliminates nodes and edges from conflicting graph until all nodes have less edges than the total number of colors available. Then the graph can be trivially colored. We investigated similar algorithms (i.e., Edge-Collapse) in this study.

3 Heuristics for Near-Optimal Page Coloring

Page placement is a special form of caching optimization problem. Instead of a fully associative cache in typical caching study (such as virtual memory paging), the cache here consists of bins. A fixed number of pages, corresponding to the cache's associativity, can coexist in the same bin; however, once the number of pages exceeds the associativity, some page must be evicted. Given an application, the goal is to optimally partition the virtual pages into different bins, such that cache misses resulted from collisions in the bins are minimized.

Theoretical results on this problem is surprisingly few. In fact, we were not able to find any result on either off-line or online version of the problem. Thus, we decide to focus on the theoretical aspects of the problem first, investigating whether the problem is tractable.

We find that in the simple case of direct-mapped cache (that is, cache associativity is 1), the off-line optimization problem is NP-Complete. That is, no computationally feasible algorithm exists. We also find that no online algorithm is competitive, that is, for any online page placement algorithm, there always exists a sequence of memory references that results in arbitrarily larger cache misses than the off-line optimal algorithm. The results are detailed in Appendix A.

Thus, we are forced to search for heuristics. Below, we first describe the concept of "interference graph," then describe our algorithm "min-weight".

3.1 Interference Graph

The concept of interference graph is motivated by the special case of direct-mapped cache. In a direct-mapped cache, a memory reference to page a at cache line l is a cache miss if and only if some reference to another page b at cache line l occurs since the last reference to page a at cache line l, or there is no last reference to page a at cache line l. In other words, if a reference to cache line l of page b is between two references to cache line l of page a, and a and b are in the same bin, the second reference to a misses in the cache.

Thus, we can build an interference graph from the memory reference traces of an application. Nodes in the graph are virtual pages of the application. Each edge has a weight, initially 0. We scan the memory references, and for every reference to page b that is between two references to page a at the same cache line, we increase the weight for the edge (a, b) by 1. Clearly, the weight for each edge represents the potential conflicts if the two nodes of the edge are put in the same color. Therefore, we can expect that a good coloring assignment is a coloring of the nodes so that the sum of the weights of edges connecting nodes in the same color is minimized, or alternatively, the sum of the weights of edges connecting nodes in different colors are maximized.

The proof that optimal page placement is NP-Complete is derived from the fact that, given an arbitrary graph G, one can construct a memory reference string whose interference-graph is G, and the Max k-Cut for G corresponds to the optimal placement for direct-mapped cache with k cache lines. The Max k-Cut, which is finding a coloring of the nodes in G in k colors so that the sum of the weights of edges connecting nodes in different colors is maximized, is NP-complete. It follows then that the optimal place placement for direct-mapped cache is NP-Complete.

Note that it is unknown whether page placement under 2-way or 4-way set-associate cache is NP-Complete, and we are currently investigating the issue.

3.2 Algorithm "Min-Weight"

An obvious candidate for page placement heuristics is thus graph-coloring algorithms for the interference graph. That is, an algorithm that colors the vertices in the graphs into k colors effectively place the pages into k cache bins. We looked into the following two heuristics:

- **Edge-Collapse** algorithm is very similar to algorithms used in register allocation in compilers [5]. It starts the edge with the lowest weight, say edge $< a, b >$, marks page a and page b as in the same bin, then merges the two vertices into one. The merging replaces vertices a, b and edge $< a, b >$ with a single vertice c, and replaces every edge of a and b with an edge of c to the same vertice with the same weight. If two edges from a and b connect to the same vertice (say d), the weight of the new edge $< c, d >$ is the sum of the weights of $< a, d >$ and $< b, d >$. The process then repeats, until only k vertices are left.
- **Min-Weight** algorithm involves three steps. First, nodes are sorted in terms of the total weight of its edges, from the largest to the smallest. Then, starting with nodes with the highest total edge weights (that is, those that conflict heavily with other nodes), the algorithm colors the nodes according to the following simple rules:
  - **rule one** if it is possible to use a color that has already been used and yet avoid having the node in the same color with any of its neighbors, then use the color.
  - **rule two** if the above rule fails, and there is a color that has not been used by any node, then use the new color.
  - **rule three** if the above rules fail, assign the color that causes the fewest conflicts, that is, assign the color such that the sum of the weights of the edges between the node and the nodes already in that color is the least.

Our experiments found that the min-weight algorithm significantly outperforms edge-collapse in most traces. Part of the reason is that the min-weight algorithm colors the heaviest nodes first, while the edge collapse algorithm groups the lightest node first. The results indicate that it is more important to separate the heavy nodes into different colors than to reduce the overall weights of edges crossing colors.

We have also evaluated a few variants of the min-weight algorithm. For example, instead of a single initial sort, we implemented a multi-sort variant that performs sorting after each node is colored and the weights of its adjacent nodes are updated. We find the improvement of multi-sort is minimal. To find out the importance of the order of the coloring, we also implemented a **Reverse-Sort** variant, which is identical to the **Min-Weight** algorithm except that it colors the lightest node first. We found this variant performs substantially worse than the original min-weight algorithms, confirming the importance of the order of the coloring.

Computational Complexity The computational complexity of the min-weight algorithm is low. The algorithm involves two steps: building the interference graph, and coloring the nodes. Building the interference graph requires a linear scan of the memory reference traces. Using a hash table and a "least-recently-used" list, the graph can be built in time nearly linear to the length of the trace. Coloring the nodes involves using a priority queue to find the heaviest nodes, and choosing among the k colors for each node. The complexity is at worst $O(k * V * logV)$, where k is the number of cache bins (i.e. colors), and V is the number of virtual pages in the application.

4 Comparison of Page Coloring Strategies

Using trace-driven simulation, we compare the min-weight algorithm with the following commonly used page coloring strategies:

- **Random** assigns a cache bin for each virtual page in a random fashion, simulating the systems where no careful page placement takes place.
- **Virtual-Bit** assigns a physical page whose page number has the same lowest bits as the virtual page number. In other words, the physical page has the same "color" as the virtual page. The algorithm thus makes sure that adjacent virtual page, which might be used concurrently, are not mapped to the same cache bin. It is also called "page coloring" in [10]. We call it "Virtual-Bit" since "page coloring" often refers to the general method of careful page placement.
- **Bin-Hopping** maps sequential virtual pages to sequential cache bins, as described in [10]. Temporal locality is exploited because pages that are accessed close to each other are placed in different bins, thus avoiding cache conflicts.

We compare the cache miss ratios of the page coloring strategies for a cache with limited associativity to those under a fully-associative cache with the same cache size and line size. In other words, we use the fully-associative cache as a baseline in our comparison.

We use three sets of traces:

- *Postgres*: traces of memory references of the database application Postgres [12] running the Wisconsin Benchmark [9]. The Wisconsin Benchmark consists of 32 queries that select, join or aggregate two relations. We present results on 20 of the 32 queries here; the other 12 queries have very similar behavior to the ones presented here. The database consists of a large relation with 1,000,000 tuples and a small relation with 100,000 tuples. The traces are generated by the binary instrumentation tool Shade [7] on a SPARC20 workstation running Solaris.
- *SPEC95*: traces of memory references from 13 of the 18 SPEC95 integer and floating point benchmarks. The traces are generated by the SimpleScalar tool [4]. We currently miss vortex, ijpeg, applu, turb3d and wave5 due to difficulty at trace generation; we expect to include results on them soon.
- *Intel PC applications*: traces of memory references of typical PC applications on Intel PCs running Windows NT. We obtained the traces from Intel Corporation. The traces include both user-level and kernel-level memory references. The current simulation involves only user-level references; however, we are working on a more complete simulation and will include the results in the final version of the paper.

Our simulator is a modification of SimpleScalar to include virtual-to-physical translation mechanism and use traces as input to drive the simulation. The page mapping module implements the above page coloring strategies and user-specified page placement, i.e. it can take as input a file specifying the placement of virtual pages to cache bins and allocate physical pages accordingly. For min-weight, we precompute the page placement and then use the "user-specified page placement" option in SimpleScalar to simulate it.

The simulation assumes a microprocessor with two 32KB direct-mapped 1st-level cache with 64-byte cache lines, one for instructions and one for data. It measures the cache miss ratio under a instruction-data combined, physically-indexed, 4-way set associative 2nd-level cache with 128-byte cache lines. Two sizes for the 2nd-level cache are experimented: 256KB and 512KB.

Figure 1 shows, for the *Postgres* queries, the cache miss ratios under a 256KB 2nd-level cache. The miss ratios of the page placement schemes are normalized to the miss ratios under the Fully-Associative cache. Figure 2 shows the corresponding results for a 512KB 2nd-level cache. The actual cache miss ratios under the Fully-Associative cache for both cache sizes are listed in Table 4.

The graphs show that Min-Weight reduces the cache miss ratios significantly compared to the existing strategies. Of the existing mechanisms, Bin-Hopping works the best for Postgres; both Random and Virtual-Bit can encounter serious collisions across the queries. However, Bin-Hopping is far from optimal. On query 10, 12, 15, 16 and 17 under 256KB caches, min-weight is able to reduce the cache miss ratio by one third to nearly two third. The results clearly demonstrate the performance potential of nearly optimal page placement, even for 4-way set associative caches. In addition, across the queries and cache sizes, min-weight results in a page placement that has similar cache miss ratios to the Fully-Associative cache.

Figure 3 and Figure 4 show the corresponding results for the SPEC95 benchmarks, for the 256KB and 512KB caches. The cache miss ratios under the fully associative caches are reported in Table 4.

The data show similar results to the Postgres traces. Min-Weight generally reduce cache miss ratios compared to Virtual-Bit and Bin-Hopping, and the cache miss ratios under min-weight is similar to those under Fully-Associative cache. For m88ksim, cc1, apsi and go, min-weight performs particularly better than Bin-Hopping. However, for fppp, min-weight failed to outperform Bin-Hopping, showing that improvement to min-weight is still needed.

Figure 5 and Figure 6 show the corresponding results for the Intel PC application traces, for the 256KB and 512KB caches. In addition to the page placement schemes listed above, we also simulated the cache miss ratios under the virtual-to-physical mapping shown in the traces; the traces include the virtual-to-physical page translation information. The results are marked as "Original Mapping" in Figure 6. The figures show the normalized results, while the cache miss ratios under the fully associative caches are reported in Table 4.

The data again show similar results to the other two traces. Min-Weight can reduce cache misses significant for some applications, for example, the application go under 256KB cache, and the application draw under 512KB cache. Even though Bin-Hopping is the best existing strategy, its cache miss ratio is almost always higher than Min-Weight. The data also show that the system where the traces are taken seem to use random page placement (the miss ratios under the page mapping shown in the trace are similar to those under random page placement), and the placement can be improved significantly in a number of applications, for example, draw, go and pcdb.

Combine these results, we see that Min-Weight, which takes the information of application memory references and avoids collision as much as possible, significantly reduces cache hit ratio across the application and cache sizes. The results clearly demonstrate the potential and importance of using more careful page placement to improve cache utilization.

5 Min-Weight with Imprecise Information

Though the above simulations assume that Min-Weight has the full trace of memory accesses for the application, we envision that the algorithm can used in at least the following two scenarios:

- *Page placement training*: a typical data set can be used to drive the application, outputting the traces of its memory reference. The min-weight algorithm then use the trace to generate a page placement that is specific to the application. This trained page placement then acts as a user-level advisory to the OS for future runs of the application. We expect this method to be particularly useful for some scientific applications and database applications.

- *Optimization based on continuous profiling*: in the case where hardware mechanisms can produce samples of memory references online, the algorithm can be run periodically to adjust the page placement and reduce cache conflicts. Though purely software methods for adjusting page placement (that is, copying to a new page) is costly, the overhead for the adjustment can

Queries	1	2	7	9	10	11	12	15	16	17
256KB cache	1.9%	1.1%	2.9%	.67%	.41%	2.2%	.70%	.65%	.58%	.40%
512KB cache	1.5%	.85%	2.6%	.52%	.33%	1.7%	.54%	.49%	.38%	.30%
Queries	18	19	20	26	27	28	29	30	31	32
256KB cache	.27%	1.4%	2.9%	12%	3.0%	2.5%	12%	3.1%	2.6%	2.6%
512KB cache	.26%	1.1%	2.7%	11%	2.7%	2.2%	11%	2.6%	2.0%	2.0%

Table 1: Cache miss ratios under fully-associative 2nd-level caches for Postgres queries. The queries are Wisconsin Benchmark queries.

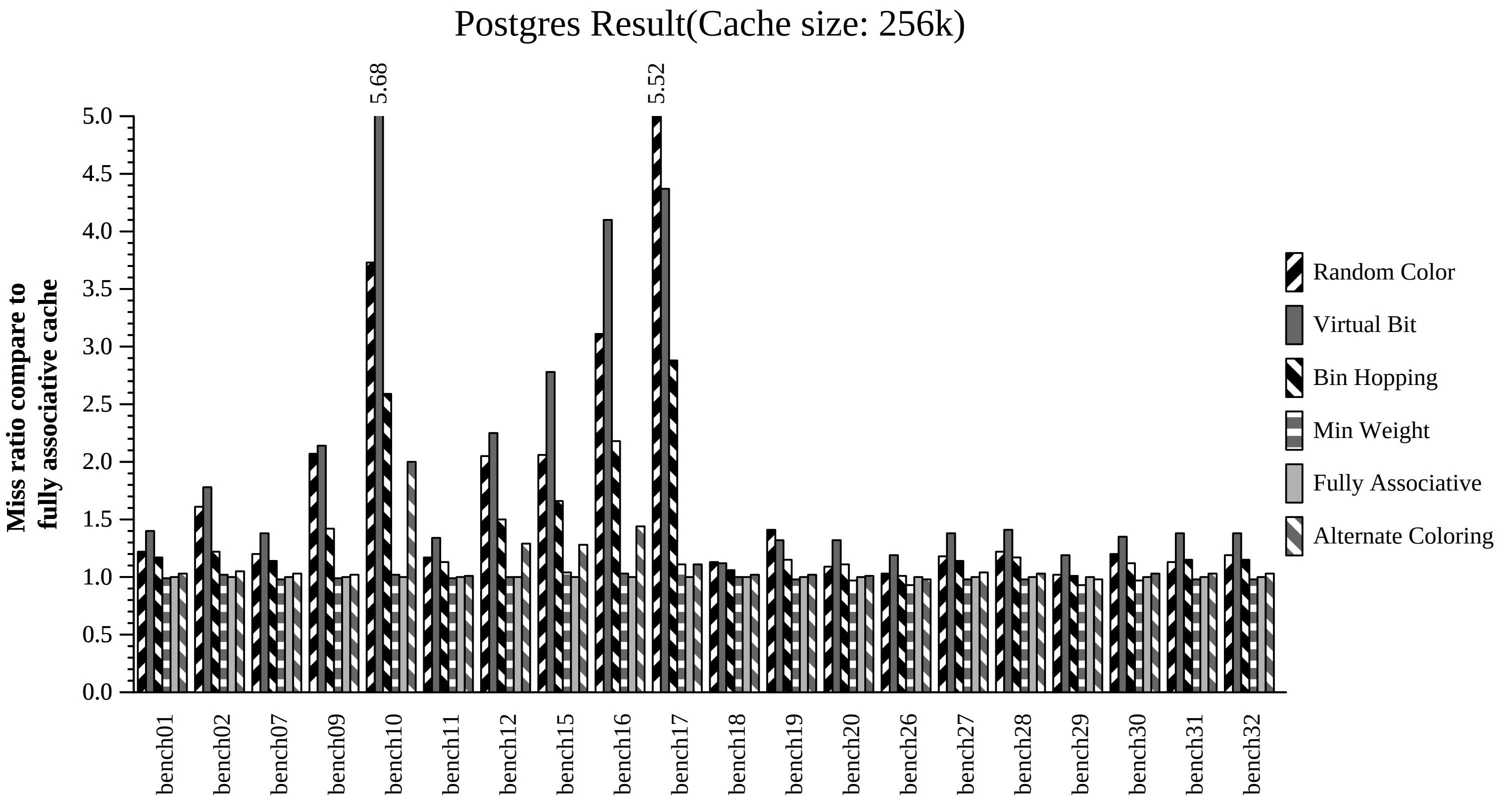

Figure 1: Postgres queries: normalized cache miss ratios for Random, Virtual-Bit, Bin-Hopping and Min-Weight, for a 256KB 4-way set associative 2nd-level cache. The bar named "Alternate Coloring" is explained in the next section.

Applications	apsi	cc1	compress95	fppp	go	hydro2d	li
256KB cache	1.4%	1.4%	60%	.0012%	1.5%	18%	.0024%
512KB cache	.43%	.77%	4.1%	.0012%	.06%	17%	.0024%
Applications	m88ksim	mgrid	perl	su2cor	swim	tomcatv	
256KB cache	.01%	6.2%	.92%	1.4%	1.6%	.77%	
512KB cache	.01%	6.0%	.57%	.79%	1.6%	.76%	

Table 2: Cache miss ratios under fully-associative 2nd-level caches for SPEC95 benchmarks.

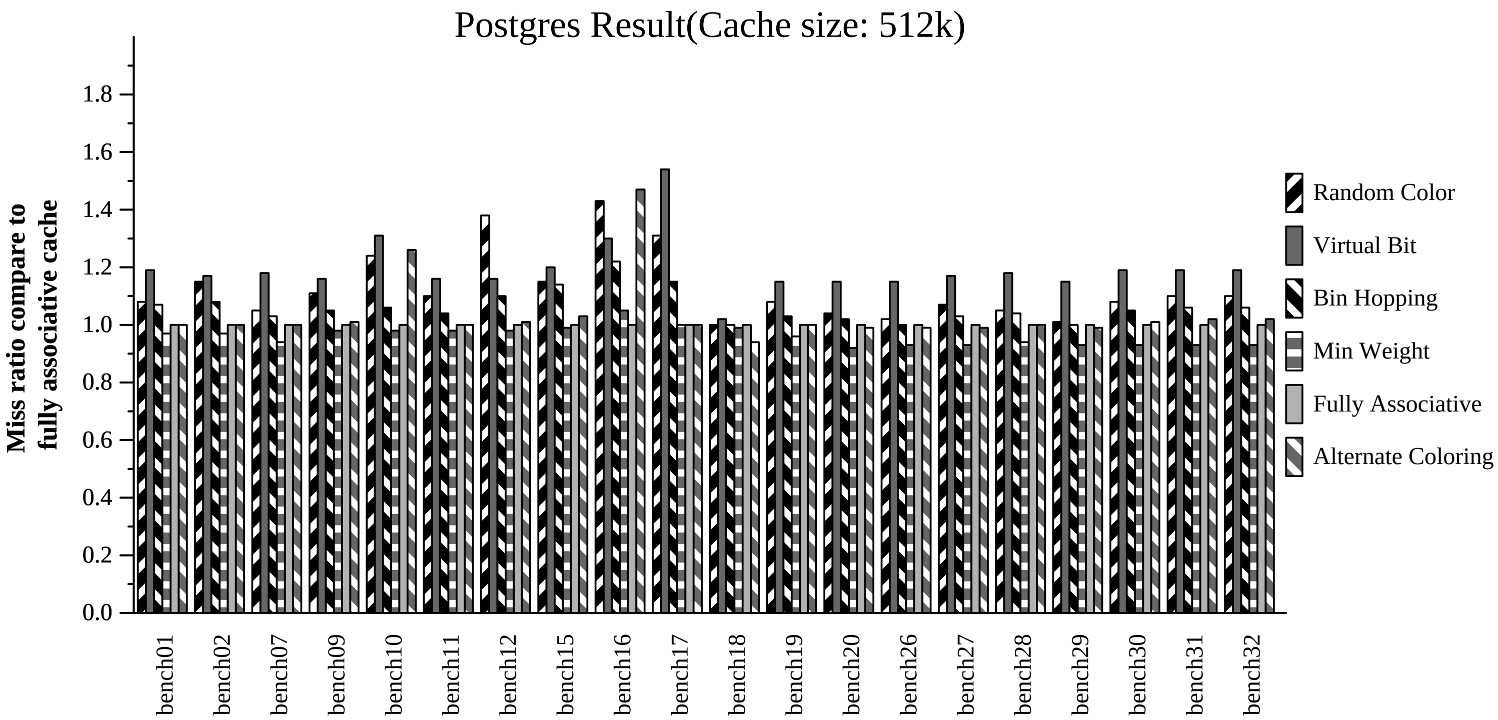

Figure 2: Postgres queries: normalized cache miss ratios for Random, Virtual-Bit, Bin-Hopping and Min-Weight, for a 512KB 4-way set associative 2nd-level cache. The bar named "Alternate Coloring" is explained in the next section.

Applications	draw	gcc	go	gcdb	pres1	pres2	specweb	sprdsht	wdproc1	wdproc
256KB cache	1.1%	.03%	.009%	.25%	.09%	.1%	.20%	.20%	.18%	.18%
512KB cache	.05%	.02%	.008%	.07%	.06%	.05%	.10%	.09%	.05%	.06%

Table 3: Cache miss ratios under fully-associative 2nd-level caches for Intel PC applications.

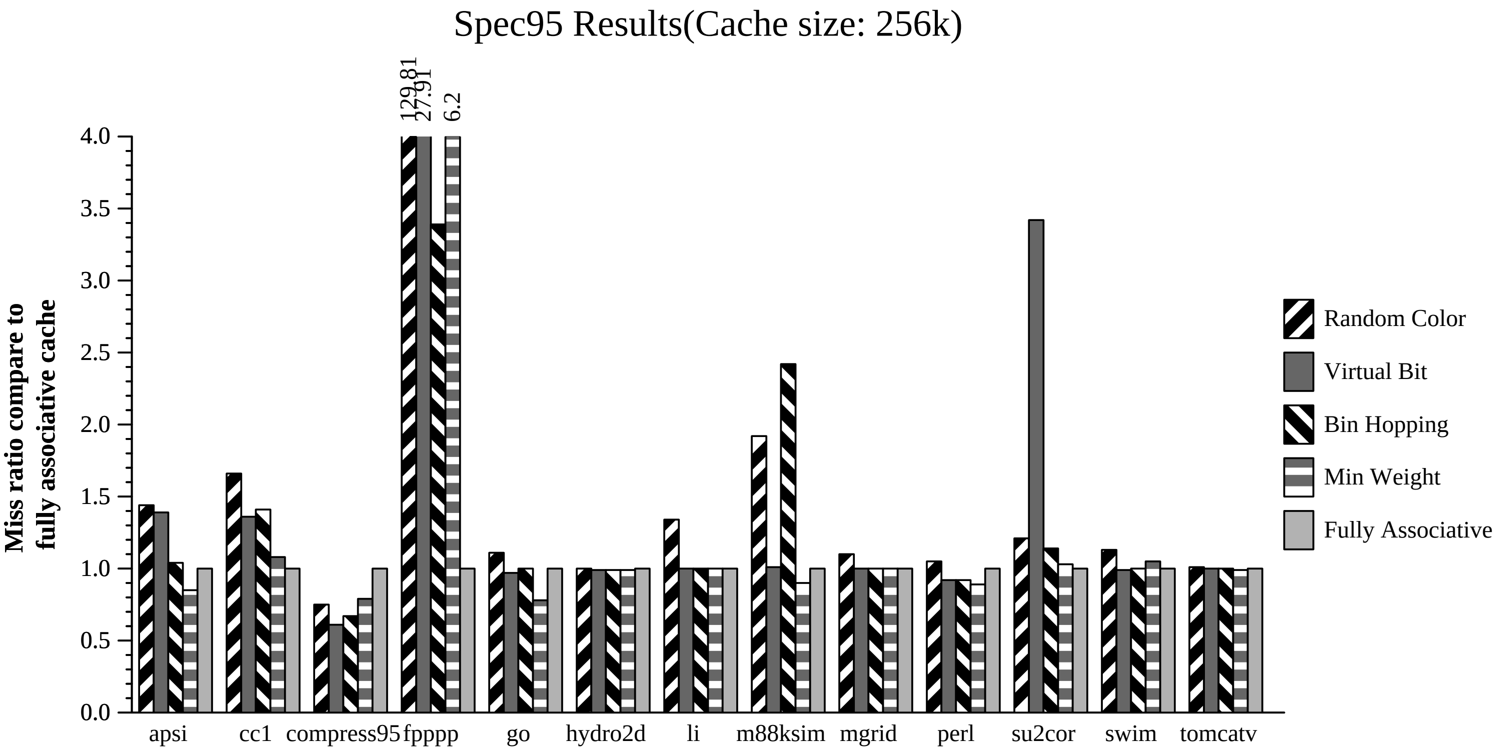

Figure 3: SPEC95 benchmarks: normalized cache miss ratios for page placement strategies under a 256KB 4-way set associative 2nd-level cache.

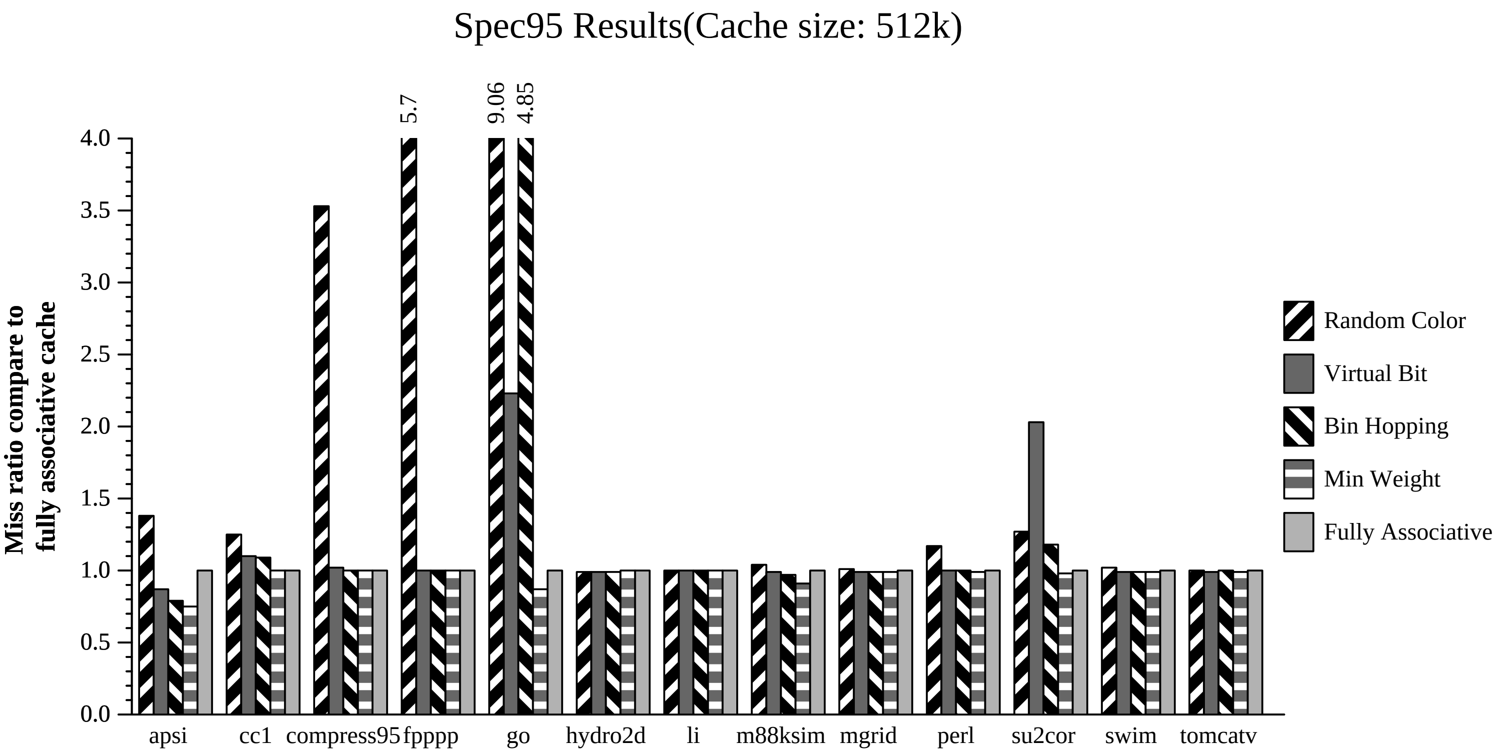

Figure 4: SPEC95 benchmarks: normalized cache miss ratios for page placement strategies, under a 512KB 4-way set associative 2nd-level cache.

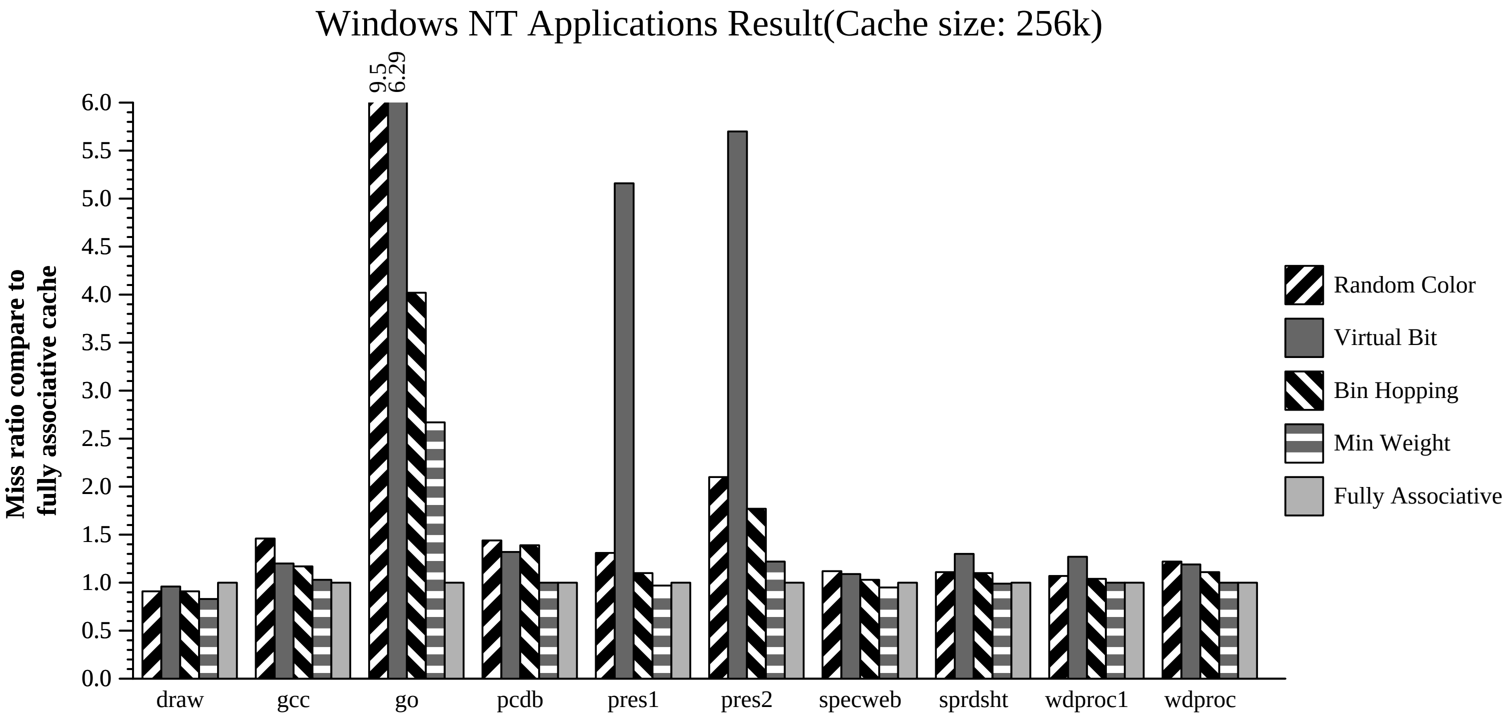

Figure 5: Intel PC applications: normalized cache miss ratios for Random, Virtual-Bit, Bin-Hopping and Min-Weight, for a 256KB 4-way set associative 2nd-level cache.

be reduced significantly with moderate hardware help (for example, extra bits in the indexing logic).

To evaluate the potential of training for application-specific page placement, we experiment with the Postgres application, since it is one application with many different inputs (queries). We take query17 as the trainer, because it is a relatively complex query. Using the traces of Postgres running query17, we apply the min-weight algorithm and yield a placement for each virtual page. We then fix the page placement for Postgres, and examine how well it performs under other queries. For virtual pages that are not specified in the placement, we use bin-hopping. The performance of this approach is shown in Figure 1 and Figure 2, under the bar named "Alternate Coloring" (the rightmost bar in each group).

The results show that the approach performs reasonably well, quite close to min-weight with complete information in most cases. Thus, we believe this is a promising approach. We are in the process of examining more applications for page placement training.

To evaluate the potential of using min-weight in a continuous profiling setting, we first examine the performance of min-weight under limited sampling. Taking the Intel PC application traces, we take 1 out of every 100 memory references and feed the 1% sample to min-weight. The performance of the resultant page placement is shown in Figure 6 under the bar named "Profile MinWeight." As the figure shows, even with 1% samples, the min-weight still performs relatively well, improving the page placement significantly for applications such as "draw" and "pres2." We also experimented with different sampling methods, including sampling a continuous 100 references out of every 10000 references, and sampling the first 1% of the trace. We find that sampling 1 reference out of every 100 references yield the best performance.

We are still in the process of investigating the use of min-weight in continuous profile-based optimization, including the frequency of its invocation and dynamic adjustment of page placement. We expect to obtain more results on this subject soon.

6 Summary and Ongoing Work

We have described a new page placement algorithm "min-weight", demonstrated its performance com-

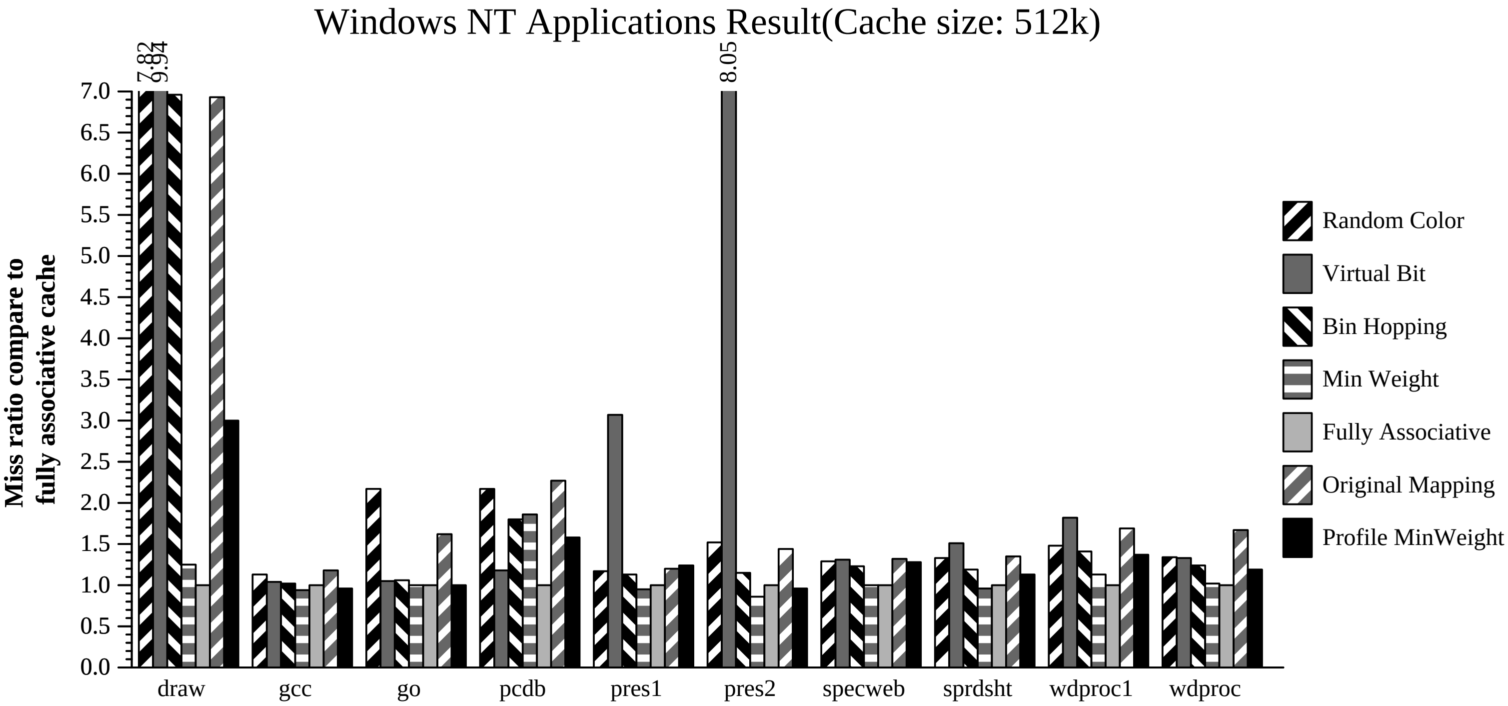

Figure 6: Intel PC applications: normalized cache miss ratios for Random, Virtual-Bit, Bin-Hopping and Min-Weight, for a 512KB 4-way set associative 2nd-level cache. "Original Mapping" is the cache miss ratio under the virtual-to-physical translation shown in the trace. "Profile Min-Weight" is discussed in the next section.

pared with existing page placement strategies, and examined its performance in practical settings. We show that sophisticated page placement that takes application's memory reference patterns into account can reduce miss ratios in physically-index caches significantly (up to 60%).

Our current results have many limitations. We have only considered application memory references, whereas the OS kernel memory references also have a significant impact on cache performance. We are investigating careful page placement for both user and kernel pages using the Intel PC traces, and plan to include results in the final version of the paper. Also, we have only considered static page placement so far, and did not address how min-weight can be used with dynamic adjustment of page placement in an online setting. We are currently working on this issue. Our results on page placement training is still preliminary, and we are using more applications to further explore the method and we expect to have more results soon.

Our future work includes finding better and cheaper graph coloring algorithms and implementing page placement training in an operating system. Min-Weight is only one of many graph coloring algorithms. In addition to simplifying the algorithm and reducing its complexity for online use, we are looking for better graph coloring methods. We have implemented application-specific page placement in the Linux system, and we are evaluating the impact of application-specific page placement (obtained through the training process) on application performance.

References

[1] Aho, Hopcroft, and Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

[3] Brian Bershad, Dennis Lee, Theodore H. Romer, and Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the ASPLOS VI*, 1995.

[4] Doug Burger. Tutorial for simplescalar. Technical report, Computer Science Department, University of Wisconsin-Madison, 1994. See URL http://www.cs.wisc.edu/ arch/.

[5] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices (Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Mass.)*, 17(6):98–105, June 1982.

[6] Bradley Chen and Bradley D. D. Leupen. Improving instruction locality with just-in-time code layout. In *The USENIX Windows NT Workshop Proceedings*, pages 25–32, August 1997.

[7] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, Sun Microsystems Labs, 1993.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the theory of NP-Completelness.* W. H. Freeman and Company, 1979.

[9] Jim Gray. *The Benchmark Handbook.* Morgan-Kaufman, San Mateo, Ca., 1991.

[10] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, November 1992.

[11] Theodore H. Romer, Dennis Lee, Brian Bershad, and Bradley Chen. Dynamic page mapping policies for cache conflict resolution in standard hardware. In *Proceedings of OSDI'94*, November 1994.

[12] The Postgres Group. POSTGRES version 4.1 release notes. Technical report, Electronics Research Lab, University of California, Berkeley, February 1993.

[13] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

Appendix A: NP-Completeness Proof of Optimal Page Placement

We first present a formal definition of the page placement problem. We consider page coloring for a simple cache with configuration $< S, L, k >$. By a simple cache, we mean a cache whose index is the lower bits of the address of the referenced cache line. A cache configuration $< S, L, k >$ means that the cache size is S, the cache line size is L, and the cache is k-way set associative.

We assume that the string of memory references, $R =< r_1, r_2, r_3, ..., r_n >$, is known, where $r_i = (p_i, l_i)$, p_i being the virtual page number, and l_i being the cache line in the page. There are $M = 4096/L$ cache lines in a page, assuming a page size of 4096. Thus, we can decompose R to M substrings, each substring $R(m) =< (q_1, m), (q_2, m), ... >$ consisting of references to the same cache line in the pages. This captures the effect that no matter what the page coloring is, references to different cache lines do not interfere with one another.

By a color c, we mean the collection of physical pages whose data map to the same group of cache sets. For example, in a $< 128K, 32, 4 >$ cache (i.e., 128K-byte cache with 32-byte cache line and 4-way set associative), color 0 includes physical pages $0, 32, 64, ...$ whose data map to cache sets 0 through 127, color 1 includes pages $1, 33, 65, ...$, and so on. Clearly, a $< S, L, k >$ cache has $C = S/(k * 4096)$ colors.

A page coloring assignment is a mapping from the set of virtual pages to the colors, $P_i \rightarrow C_i$. Given a mapping, we can decompose $R(m)$ into C substrings, $R(m, c) =< (v_1, m), (v_2, m), ... >$ where v_i are virtual pages that map to color c. We call the sum of the number of cache misses generated by $R(m, c)$ under a k-entry cache *the number of cache misses under the mapping* $P_i \rightarrow C_i$.

Problem Statement: given a cache configuration $< S, L, k >$, and a reference string $R =< r_1, r_2, r_3, ..., r_n >$, find a mapping $P_i \rightarrow C_i$ such that the number of cache misses under the mapping is the lowest among all mappings.

Theorem: Calculating optimal page coloring is NP-complete for $k = 1$.

Proof. Given a weighted graph G and C colors, the Max-Cut problem is finding a coloring of the nodes in G so that the sum of the weights of edges connecting nodes in different colors is maximized. The Max-Cut problem is NP-Complete [8]. We can construct a reduction from Max-Cut to optimal page coloring in the following way.

For each edge (a, b) with weight e in G, we construct a substring $< (a, 1), (b, 1), (a, 1), (b, 1), ... >$ where $(a, 1), (b, 1)$ is repeated e times. We then construct a reference string R which is a concatenation of all such substrings. Clearly, under cache configuration $< C*4096, 4096, 1 >$, the number of cache misses under a page coloring is exactly two times the sum of the weights of edges connecting nodes in the same color. Thus, an optimal coloring of the pages directly corresponds to a max-cut coloring of the nodes.

Thus, the page placement problem is NP-complete for the case of $k = 1$, i.e. direct-mapped caches. We do not yet know whether the problem is NP-complete for $k > 1$.