A Graph Theoretic Approach to Cache-Conscious Placement of Data for Direct Mapped Caches

Mirza Beg
David R. Cheriton School of Computer Science
University of Waterloo, Canada
mbeg@cs.uwaterloo.ca

Peter van Beek
David R. Cheriton School of Computer Science
University of Waterloo, Canada
vanbeek@cs.uwaterloo.ca

Abstract

Caches were designed to amortize the cost of memory accesses by moving copies of frequently accessed data closer to the processor. Over the years the increasing gap between processor speed and memory access latency has made the cache a bottleneck for program performance. Enhancing cache performance has been instrumental in speeding up programs. For this reason several hardware and software techniques have been proposed by researchers to optimize the cache for minimizing the number of misses. Among these are compile-time data placement techniques in memory which improve cache performance. For the purpose of this work, we concern ourselves with the problem of laying out data in memory given the sequence of accesses on a finite set of data objects such that cache-misses are minimized. The problem has been shown to be hard to solve optimally even if the sequence of data accesses is known at compile time. In this paper we show that given a direct-mapped cache, its size, and the data access sequence, it is possible to identify the instances where there are no conflict misses. We describe an algorithm that can assign the data to cache for minimal number of misses if there exists a way in which conflict misses can be avoided altogether. We also describe the implementation of a heuristic for assigning data to cache for instances where the size of the cache forces conflict misses. Experiments show that our technique results in a 30% reduction in the number of cache misses compared to the original assignment.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Memory Management

General Terms Algorithms, Measurement, Performance

Keywords Offline Algorithms, Memory Management, Cache Consciousness, Cache Optimization, Data Placement in Cache

1. Introduction

Latencies from the memory hierarchy have a significant impact on program performance. Caches were designed to improve memory access times by copying frequently accessed data into relatively smaller on-chip storage that is readily accessible to the processor. As the gap between processor speed and memory access speed widened, caches became the bottleneck for program performance. A significant amount of research has been done to reduce the impact of cache misses on program performance. Both hardware and software techniques have been employed to this end. Hardware enhancements to caches include increased associativity for reducing conflicts between objects mapped to the same block, multibanked caches to increase cache bandwidth and multi-level caches to reduce miss penalty, among others [16].

Software techniques, including compile-time optimizations as well as run-time optimizations, have also been useful in reducing cache misses. Among the most well known ones are prefetching [18], loop interchange [25], code and data rearrangement [12, 22], and blocking [17]. More recent works have focussed on more accurately computing reference locality of objects [6, 13, 15, 23, 26–28] and on reorganizing objects in memory using an intelligent memory manager [7–10].

The number of cache misses depends on how the data in memory is accessed. A cache miss occurs when an object is accessed for the first time by a program and that object had not been previously fetched into the cache. This is called a *compulsory miss*. Prefetching is a technique that is employed to reduce the number of compulsory misses. When an object in the cache is replaced by another object mapped to the same line in the cache and the original object is accessed again, a *conflict miss* occurs. In cases where the objects accessed by the program do not fit in the cache, *capacity misses* are a result. One of the ways in which performance can be improved is to layout the data in memory such that it minimizes conflict misses. The problem of placing data in memory such that conflict misses are minimized has been known to be hard for some time and has been a topic of NP-hardness studies by Thabit [24] and Petrank et al. [20]. Calder et al [4] used profiling to place data in memory via intelligent heuristics to reduce conflict misses in the cache.

In this paper we consider the offline problem of minimizing cache misses in direct mapped caches as formulated in [21]. This is the most interesting and the most difficult case among problems in cache conscious data placement and the results from this problem can be extended to t-way associative caches. The problem for fully-associative caches is known to have a trivial solution. The problem considered in this paper can be formalized as follows:

Let $\mathcal{O} = \{o_1, \cdots, o_m\}$, be the set of objects accessed by a given program where we assume that each object fits in a single cache line. Given a cache of k lines and a data access sequence $\sigma = (\sigma_1, \cdots, \sigma_n)$ where $\sigma_i \in \mathcal{O}$ for all $i \in \{1, \cdots, n\}$, we want to assign each object to a cache line such that misses are minimized. The access sequence can be obtained by profiling the program. The problem can now be considered as a mapping problem for all objects in $\mathcal{O}$ where the domain of $o_i \in \mathcal{O}$ is $\{1, \cdots, k\}$ for all i where $1 \leq i \leq m$. A valid solution is an assignment of values

ISMM'10, June 5–6, 2010, Toronto, Ontario, Canada.

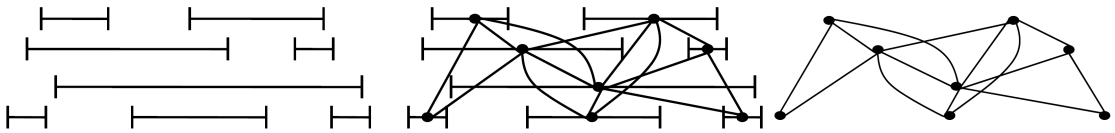

Figure 1. A set of intervals and the interval graph defined by it.

to all the objects in $\mathcal{O}$. An optimal assignment would entail that cache misses are minimum for the given sequence σ. We assume that any data object in memory can be assigned to any one of the k cache lines. The objective is to find a coloring of the data conflict graph such that each object o_i, $i \in \{1, \cdots, m\}$ can be placed in a memory block that maps to a cache line indicated by that color and results in the fewest possible cache misses. The problem can be defined as:

DEFINITION 1 (Minimum k-cache Misses Problem). *Given a set of objects $\mathcal{O}$ and a sequence of accesses σ find a mapping $f : \mathcal{O} \rightarrow \{1, \cdots, k\}$ such that Misses($(\mathcal{O}, \sigma), f$) is minimized.*

In this paper we describe an algorithm for solving the Minimum k-cache Misses Problem. We use results from graph theory to show that for instances where the objects in a given sequence can be laid out in memory without any conflict misses, an optimal solution can be found. For other instances we heuristically reduce the problem and apply the same algorithm to optimize for the number of cache misses. To apply this approach, a program is first profiled to record the order of its data accesses. The profile information thus gathered is used by the algorithm to construct a conflict graph for the program. This conflict graph is then used by the algorithm to determine a cache assignment to all objects which reduces the number of misses. This assignment is then used to guide the memory manager to create a placement for objects in memory such that it complies with the assignment determined by our algorithm.

The rest of the paper is organized as follows. The next section gives an overview of the necessary background material. In Section 3 we describe our solution to the cache conscious data placement problem. In Section 4 we present results on selected benchmarks. In Section 5 we give an overview of some related work. In Section 6 we discuss some of the issues related to the problem and analyze our solution. Finally, in Section 7 we conclude with a summary of our work.

2. Background

This section gives an overview of the relevant results from graph theory which are used in our solution to the data placement algorithm. For more on interval graphs see [14]. Here we also describe the construction of the data conflict graph from a sequence of accesses. The graph representation used in our algorithm is similar to the proximity graph described in [24] and the temporal relationship graph described in [4].

2.1 Graph Theory

Given a set of intervals $\mathcal{I} = \{I_1, \cdots, I_m\}$ on a real line a vertex v_j can be defined for each interval I_j, and an edge (v_j, v_k) exists if and only if the two corresponding intervals intersect, i.e. $I_j \cap I_k \neq \emptyset$.

DEFINITION 2 (Interval Graph). *A graph G is called an interval graph if it is the intersection graph of some intervals.*

Given an undirected graph $G = (V, E)$, where $v_i \in V$, a vertex order $(v_1, \cdots, v_m)$ can be obtained by directing each edge $(v_i, v_j) \in E$ as $v_i \rightarrow v_j$ if $i < j$ and $v_j \rightarrow v_i$ if $i > j$. This means that each edge is directed from left to right in the order $(v_1, \cdots, v_m)$. If $v_i \rightarrow v_j$ is an edge implied by the vertex order, then $v_i \in predecessor(v_j)$.

DEFINITION 3 (Perfect Elimination Order). *A perfect elimination order is a vertex ordering $(v_1, \cdots, v_m)$ such that for all $i \in \{1, \cdots, m\}$, the set $\{v_i\} \cup predecessors(v_i)$ forms a clique.*

THEOREM 1. *Every interval graph has a perfect elimination order.*

Proof: If the vertices of an interval graph are ordered by the left end-point of the intervals then the set $\{v_i\} \cup predecessors(v_i)$ forms a clique for any i. This means that if an interval intersects with v_i and is a predecessor of v_i, it must intersect v_i at the left most endpoint of v_i where it also intersects all the other predecessors of v_i (as illustrated in 2).

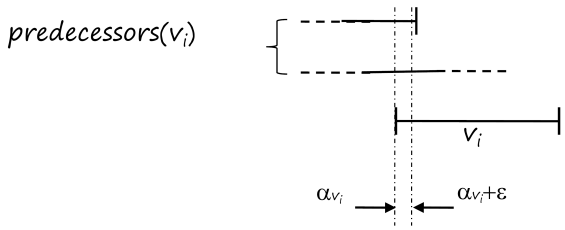

Figure 2. Perfect elimination order of an interval graph. All the predecessors of v_i intersect v_i at its leftmost end-point α_{v_i}. All $v_j \in predecessors(v_i)$ also intersect with each other at the points between α_{v_i} and $\alpha_{v_i} + \epsilon$ where ϵ is smaller than the shortest possible length of an interval.

□

THEOREM 2. *For a graph G where the vertices in G can be ordered into a perfect elimination order, the chromatic number $\chi(G)$ can be determined in linear time.*

Proof: Given the vertex order $(v_1, \cdots, v_m)$, scan the vertices in order and color each vertex v_i with the smallest color not used in $predecessors(v_i)$.

The number of incoming edges incident on vertex v_i are given by $indegree(v_i)$. Since a vertex v_i has $indegree(v_i)$ predecessors, at least one of the colors $\{1, \cdots, indegree(v_i) + 1\}$ is not used among the predecessors. The algorithm finds a coloring with at most $max_i\{indegree(v_i) + 1\}$ colors.

Let v_{i^*} be the vertex with the largest number of incoming edges. So, $\chi(G) \leq indegree(v_{i^*}) + 1$. Since, $(v_1, \cdots, v_m)$ is a perfect elimination order, the set $predecessors(v_{i^*})$ form a clique. All these predecessors are also adjacent to v_{i^*}, so $\{v_{i^*}\} \cup predecessors(v_{i^*})$ forms a clique. If $\omega(G)$is the maximum clique of G then $\omega(G) \geq indegree(v_{i^*}) + 1$. But, $\chi(G) \geq \omega(G)$, so $\chi(G) = \omega(G) = indegree(v_{i^*}) + 1$, which implies that this is an optimal coloring.

□

THEOREM 3. *For a graph G where the vertices in G can be ordered into a perfect elimination order, the maximal cliques can be found in linear time.*

The proof of this theorem is given in [14] which gives an algorithm to find all the maximal cliques in the graph.

DEFINITION 4 (Edge Contraction). *Given an edge $e = (u, v)$ in graph G, contracting the edge e results in an induced subgraph G' in which the edge e is removed and the two vertices u and v are merged. All edges incident to u and v in G become incident to the merged vertex.*

THEOREM 4. *An induced subgraph G' of an interval graph G, resulting from contracting an edge, is also an interval graph.*

Proof: Let (u, v) be the edge to be contracted in G. Consider the interval representation of G where I_u and I_v represent the two intervals corresponding to u and v. Replace the intervals I_u and I_v with I_{uv} where the left end-point of I_{uv} is the left-most point in I_u or I_v and the right-most end point of I_{uv} is the right-most point in I_u or I_v. Now remove I_u and I_v and add I_{uv} to the interval representation. The intersection graph of the new set of intervals is G'.

□

2.2 Conflict Graph

DEFINITION 5 (Data Conflict Graph). *For a given program, let $\mathcal{O} = \{o_1, \cdots, o_m\}$ be the set of objects accessed by it. For a sequence of accesses $\sigma = (\sigma_1, \cdots, \sigma_n)$ where $\sigma_i \in \mathcal{O}$ for all $i \in \{1, \cdots, n\}$, the data conflict graph can be given by $G = (V, E)$ where $|V| = m$ and each vertex $v_i \in V$ represents the memory object o_i and an edge $(v_i, v_j) \in E$ exists if and only if mapping o_i and o_j to the same cache line results in one or more conflict misses.*

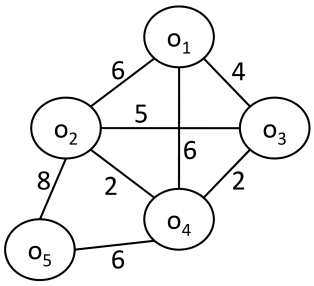

Figure 3. The conflict graph of a program for an access sequence $\sigma = \{ o_1, o_2, o_3, o_1, o_2, o_3, o_1, o_2, o_1, o_4, o_1, o_4, o_1, o_4, o_1, o_4, o_3, o_2, o_5, o_2, o_5, o_2, o_5, o_2, o_5, o_2, o_4, o_5, o_5, o_4, o_5, o_4, o_5 \}$.

An edge between two vertices v_i and v_j means that there is a subsequence of σ of the form $\sigma_i = (o_i, \cdots, o_j, \cdots, o_i)$ or $\sigma_j = (o_j, \cdots, o_i, \cdots, o_j)$.

DEFINITION 6 (Conflict Graph Edge Weight). *Each edge $(v_i, v_j) \in E$ in the data conflict graph can be assigned a weight which is an integer value denoting the number of unique instances of the subsequence $(v_i, \cdots, v_j)$ in the sequence σ.*

The edge weight of (v_i, v_j) represents the number of times the two objects will be swapped out of the cache if they were assigned to the same cache line. These weights can be computed by counting the number of alternate occurrences of the two objects in the sequence. The data conflict graph for a given access sequence can be constructed in time linear in the length of the sequence.

EXAMPLE 1. *Consider a program which accesses objects in the set $\mathcal{O} = \{o_1, \cdots, o_5\}$ where the access sequence is given by σ. The conflict graph is illustrated in Figure 3. The first access of an object results in a compulsory miss and is not represented in the edge weight. After the first access, each alternating access of a conflicting object is counted as a miss.*

Note that the sum of all edges in the graph or even in a subgraph may not represent the total number of misses if all the objects are assigned to the same cache line. Thus if two or more objects are assigned to some cache line the sum of all the edges in between these objects is greater than or equal to the actual number of conflict misses that would result in this case. Consider the example with $\mathcal{O} = \{o_1, \cdots, o_4\}$ and $\sigma = \{o_1, o_2, o_3, o_4, o_1, o_2, o_3, o_4\}$, where the sum of all edge weights in the conflict graph is 12 but the total number of conflict misses is 4 if all the objects are assigned to the same cache line.

It is also worth noting that the sum of all edge weights is an upper-bound on the total number of conflict misses.

3. Data Assignment to Cache

In this section, we describe our solution to the data placement problem for minimizing cache misses. We use a methodology similar to [4] with an application of results from graph theory. Our optimization framework consists of (1) the profiler, (2) the data placement algorithm and (3) a cache simulator to determine the number of misses for a given assignment.

Cache conscious data placement of objects attempts to assign objects to different cache lines if the profile data indicates that there would be a large number of misses if they were assigned to the same cache line. By assigning highly conflicting objects to different cache lines we aim to achieve fewer cache misses leading to improved performance.

3.1 Profiler

The objective of profiling is to develop a data conflict graph for the given program. We developed our profiler to record memory allocations and accesses. When memory is allocated on the stack or the heap, an id is assigned to the object and a record is created for the object location and size. Every access to an object is recorded by the profiler. The profiler also implements certain optimizations to compress the data sequence without losing any information. These include treating consecutive accesses of the same object as a single access.

The program which is to be profiled is instrumented by inserting calls to the profiler at each instance of memory allocation and access. The instrumented program when run generates the sequence of accesses to the data objects.

3.2 Conflict Graph

The conflict graph construction from a given access sequence was discussed in the last section. Here we classify the data conflict graph as an interval graph and use this classification to simplify the problem of data placement.

THEOREM 5. *Given a sequence of data accesses, the data conflict graph of a program is an interval graph.*

Proof: Given that σ is a finite and totally ordered sequence, each object has a well defined first and last occurrence in σ. Also given that exactly one object occupies each position in the sequence σ, each object can be represented by a unique interval from the first to the last occurrence of that object in σ.

Since each object can be represented by an interval given an access sequence σ and a conflict miss only occurs if two intervals

intersect, the data conflict graph is an intersection graph of intervals.

□

Since the data conflict graph is an interval graph, the results which are applicable to interval graphs can also be applied to the data conflict graph. This means that problems like colorability and max-clique can be computed easily for the data conflict graph. The most relevant result which can be applied here is given by the following theorem.

THEOREM 6. *Colorability of a data conflict graph of any program can be determined in linear time.*

Proof: Since we have already established that the data conflict graph is an intersection graph of intervals and interval graphs can be represented by a perfect elimination order and the chromatic number for a graph represented by a perfect elimination order can be determined in linear time. By transitivity the chromatic number of a data conflict graph can be determined in linear time.

□

Using the given results we can find the chromatic number for the conflict graph.

COROLLARY 1 *The chromatic number for the conflict graph gives us the number of cache lines required to achieve zero conflict misses for a given sequence.*

Consider the example in Figure 3 where the chromatic number is four. Thus if we have four lines to assign the five objects a placement can be found which would result in zero conflict misses.

COROLLARY 2 *For a placement that creates no cache conflicts, the sum of edges in the subgraph for the objects assigned to the same line for each cache line is zero.*

This means that if there is a placement that results in zero conflict misses there are no edges between objects which have been assigned to the same cache line.

3.3 Our Approach

The data placement algorithm uses the profiled sequence and the configuration of the cache to determine an assignment for each object to a cache line while minimizing misses. Algorithm 1 gives an outline of our data placement technique. The algorithm returns a mapping for each object to a line within the cache.

Algorithm 1 Data Placement $(\mathcal{O}, \sigma, k)$

1: $\mathcal{G} \leftarrow CreateConflictGraph(\mathcal{O}, \sigma)$
2: **if** $Colorable(\mathcal{G}, k)$ **then**
3: **return** a mapping based on the coloring of $\mathcal{G}$
4: **else**
5: **while** $|MaximumClique(\mathcal{G})| > k$ **do**
6: $\mathcal{C} \leftarrow MaximalClique(\mathcal{G})$ of size $> k$
7: $e \leftarrow e \in \mathcal{C}$, that minimizes $\frac{w(e)}{q_k(e)}$ in $\mathcal{G}$
8: $\mathcal{G} \leftarrow Contract(e)$
9: $\mathcal{G} \leftarrow$ update edge weights in $\mathcal{G}$
10: **end while**
11: **return** a mapping based on the coloring of $\mathcal{G}$
12: **end if**

First we generate a data conflict graph using the sequence from the profiler. The rest of the algorithm has two main phases. We use the classification of the data conflict graph as an interval graph to find the chromatic number for the graph. If the number of lines in the cache, given by k, is atleast the chromatic number then we are done. We simply color the graph (by assigning numbers from $1-k$ as colors) and return the coloring as the mapping to cache lines. This coloring algorithm is linear in the number of vertices in the graph. However, if the chromatic number of the data conflict graph is greater than the number of cache lines available, we heuristically merge vertices in the graph until it becomes colorable. The objective of this exercise is to merge vertices with the smallest number of conflicts among them.

In order to achieve our goal we need to systematically decrease the size of large cliques. This is because the chromatic number of an interval graph is equal to the size of the largest clique in it. Thus we create a list of all the maximal cliques that are of size greater than k (which is the size of the cache) and iteratively merge edges in each one of these cliques until the maximum clique in the reduced graph is of size k. A maximal clique is not the maximum clique in the graph but it is not a part of a larger clique. The maximal cliques in an interval graph can be listed in linear time. In our algorithm we iteratively reduce each large maximal clique to a smaller clique.

Once we have a clique of size greater than k (line 6), the next step is to chose the best possible edge to contract (merge the two vertices connected by it) and reduce the size of the clique by one. To find the edge which would be the overall optimal choice is a hard problem if there are two or more cliques of size greater than k in the data conflict graph and are sharing edges. We chose an edge e which minimizes the fraction $\frac{w(e)}{q_k(e)}$ where $w(e)$ is the weight of the edge and $q_k(e)$ are the number of cliques larger than size k which include e as an edge. Once the edge is contracted the weights of all the edges adjacent to the contracted edges are recomputed. This recomputation reflects the change in the number of misses between the newly combined objects and other objects. The resulting graph is still an interval graph because the contraction of an edge can be seen as merging two intervals. The process is repeated by choosing another edge until the size of the reduced clique is equal to k.

After all the large cliques have been reduced to size k, the resulting graph (which is still an interval graph) can be colored using the interval graph coloring algorithm. In this scenario all the objects represented by merged vertices are given the same color. This coloring is used to generate a mapping for each object to a cache line.

3.4 Cache Simulation

We developed a simple cache simulator which consumes a sequence, cache size and an assignment of objects to cache lines and outputs the number of misses resulting from the given assignment. The simulator computes the misses for each cache line by looking up the objects assigned to the line and traversing the data access sequence. It adds up the misses from all the cache lines to get the total number of misses.

4. Evaluation

In this section we present the performance results from the experiments on a variety of benchmarks.

4.1 Methodology

We evaluate our algorithm for the number of cache misses compared to the original assignment of objects. The original assignment evenly distributes objects over the cache lines simply by a modulo operation on the virtual address of the location of an object in memory. For comparison purposes we assume that objects are evenly assigned to the cache.

We collected profile information from six C benchmarks, for three different inputs each. Two of these benchmarks (*bisort* and *mst*) are part of the Olden benchmark suite which has been popular for data structure layout and data prefetching studies. *fft* is part of the benchFFT benchmarks, *fir* is a part of the trimaran bench-

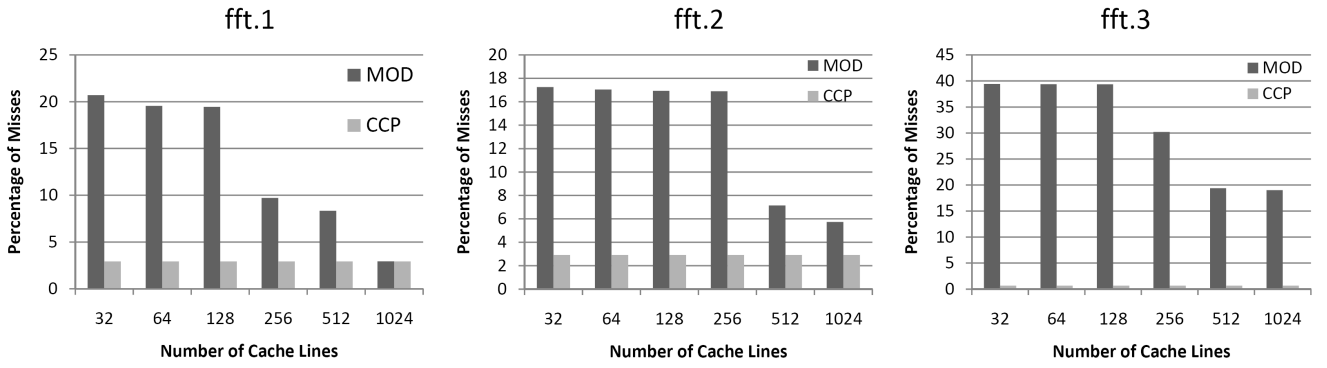

Figure 4. Experimental results for the fft benchmark.

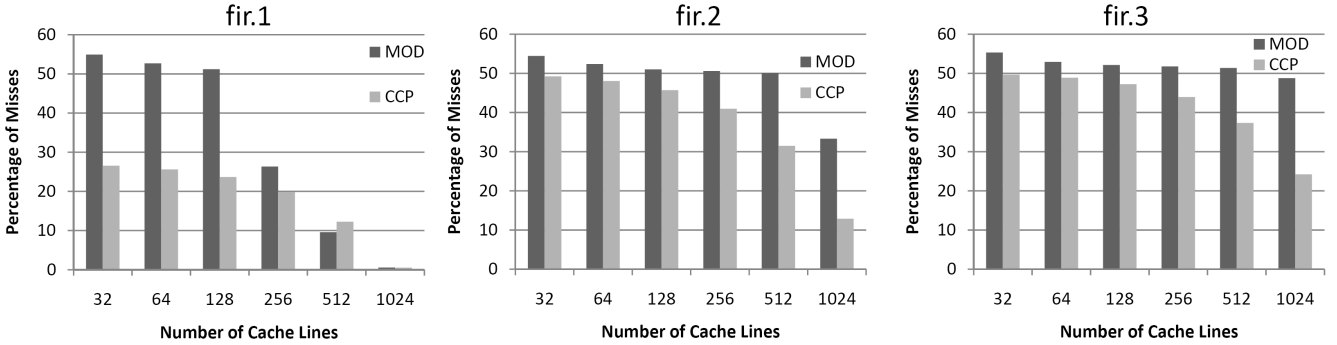

Figure 5. Experimental results for the fir benchmark.

mark suite, whereas *mm* is a matrix multiplication benchmark and *cachekiller* was posted to usenet where it generated some discussion for its effect on cache performance on different machines. These benchmarks were specifically chosen from various sources to test the algorithm on a variety of programs. Table 4.1 gives the size of the instances (number of objects and the size of the access sequence) for each benchmark. A brief description of each benchmark is given below along with the primary data structure used in them. It should also be noted that we do not make any assumptions about the order of access of the array elements, and thus we treat each element in the array as a separate object.

Benchmark	Instance	$\|\mathcal{O}\|$	$\|\sigma\|$
fft	fft.1	1123	38363
	fft.2	2148	73636
	fft.3	2396	360126
fir	fir.1	1030	200784
	fir.2	1518	221851
	fir.3	2054	194425
mm	mm.1	1200	18800
	mm.2	2700	60300
	mm.3	4800	139200
mst	mst.1	1150	159994
	mst.2	1534	280533
	mst.3	2288	630988
bisort	bisort.1	1023	1352132
	bisort.2	2047	307060
	bisort.3	4095	687600
cachekiller	cachekiller.1	811	4846
	cachekiller.2	1579	9454
	cachekiller.3	2603	15598

Table 1. Benchmark instances used for evaluation

fft computes the fourier transform or inverse transform of its complex inputs to produce complex outputs. It uses several floating point arrays for doing fourier transforms and inverse Fourier transforms and optimizes for trignometric calculations.
fir selectively filters an input signal to remove unwanted noise and distortion. The benchmark implements a digital filter using floating point arrays.
mm creates and multiplies two matrices and sums up all the elements of the resulting matrices. This benchmark implements the matrices as list of lists.
mst performs a hash-based search, with the linked lists originating from the indices of the hash table to compute the minimum spanning tree of a graph. It uses an array of singly-linked lists.
bisort conducts a forward and backward sort of integers using two disjoint bitonic sequences which are merged to get the sorted result. The main data structure in bisort is a binary tree.
cachekiller is a 2D image processing program. It reads the pixels of an image, performs a 1D filter and writes to an output image. This benchmark uses two dimensional integer arrays for the images.

4.2 Empirical Analysis

Figures 4 - 9 show the performance of our data placement algorithm compared to the original placement in terms of the cache miss rate. Results are shown for the original cache misses (MOD) and for our cache conscious placement (CCP) algorithm. Cache misses are given as a percentage of the total number of data accesses. Each benchmark instance was run for 32 to 1024 cache lines.

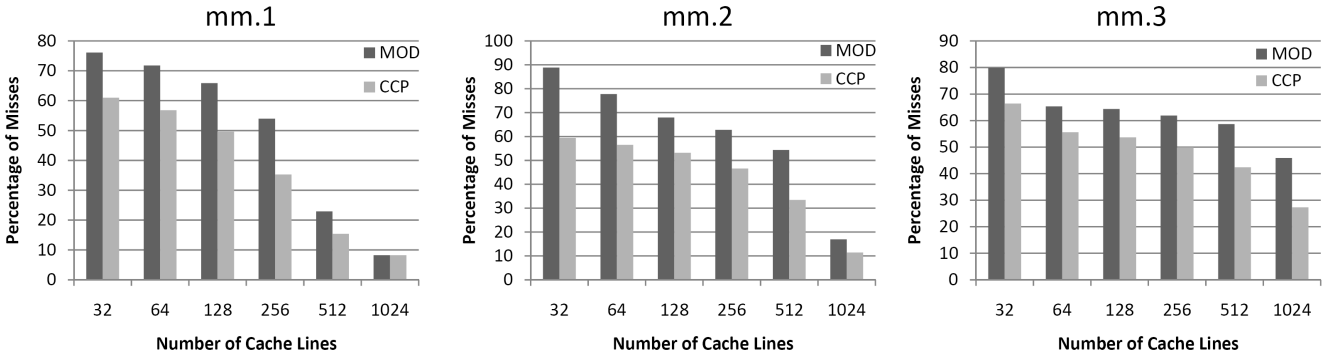

Figure 6. Experimental results for the mm benchmark.

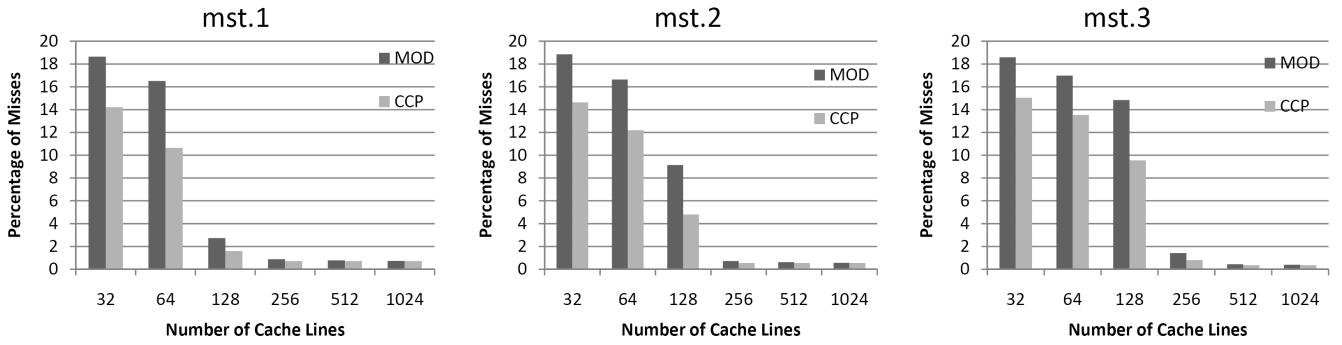

Figure 7. Experimental results for the mst benchmark.

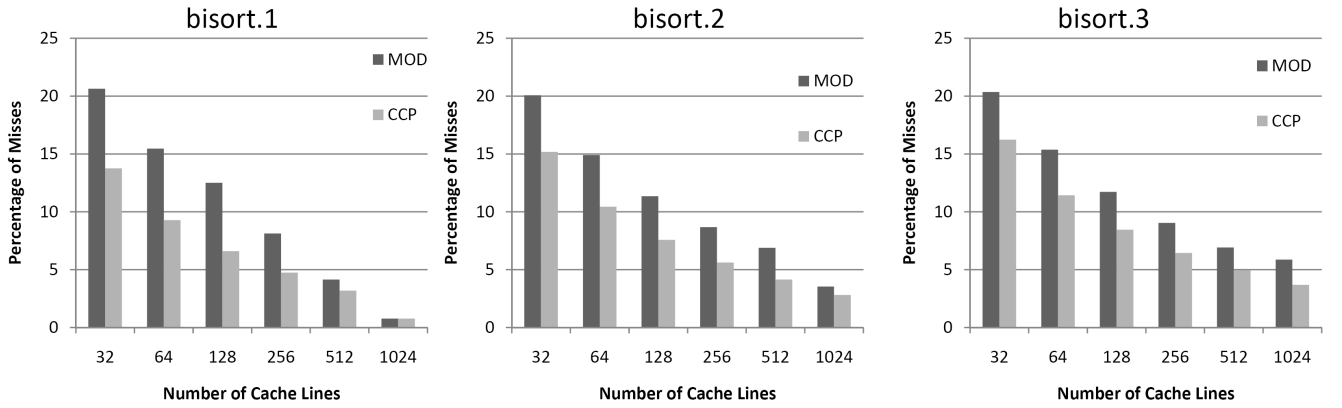

Figure 8. Experimental results for the bisort benchmark.

The fft benchmark results in Figure 4 show that the miss rate achieved by our algorithm for all three instances is the same for 32 cache lines and 1024 cache lines. For this benchmark our algorithm was able to find a placement for all the objects without incurring any conflict misses even for a cache with 32 lines. This shows that the working set of array elements in the fft benchmark does not exceed 32 at any point during its execution. Our algorithm was thus able to achieve an overall 78% reduction in the number of cache misses for the fft benchmark.

Figure 5 shows the results for the fir benchmark. Our algorithm performs relatively better on the small instance (fir.1) compared to the larger instances (fir.2 and fir.3). It is interesting to note that for the smaller instance and with 512 cache lines the original assignment gives lower cache misses than our algorithm. In this case our heuristic performs badly which is likely due to several edges with equally low weights being shared among many cliques. This results in some bad decisions made by the reduction heuristic early on which cannot be taken back in the current implementation. On the larger instances our algorithm performs relatively better than the original placement on caches with more lines rather than fewer lines. In these instances the strategy of evenly distributing objects over the cache lines seems to be ineffective most likely because of the existence of large cliques in the data conflict graph with similar weights on the edges.

The matrix multiplication benchmark is a cache intensive program and has a very high miss rate because of the access pat-

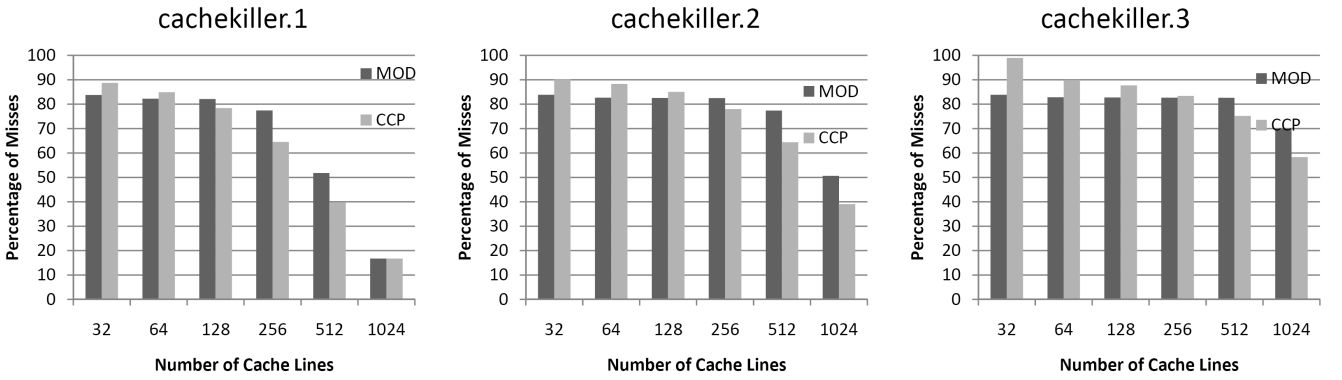

Figure 9. Experimental results for the cachekiller benchmark.

terns resulting from matrix multiplication. As the results in Figure 6 show, our algorithm consistently outperforms the original data placement to reduce the cache misses by 25% on average over all the instances.

The results for the mst benchmark shown in Figure 7 give an interesting picture. For all three instances, our algorithm was able to optimally assign objects to cache lines for more than 256 lines. But, for smaller sized caches it was not able to do so, because there are cliques in the data conflict graph of size close to 300. Still the heuristic performed better than the original placement by 22%. mst has the lowest miss rates among the selected benchmarks which makes it difficult to further reduce the miss rates.

Figure 8 shows the results for the data placement algorithms on bisort instances. The results show a reduction of 29% in the number of cache misses over the original placement and performs better than the original placement in every single instance even though the miss rate is fairly low for this benchmark.

The cachekiller benchmark, for which the results are shown in Figure 9, really tests our algorithm. The miss rates for most instances on this benchmark are extremely high and our algorithm performs poorly when the number of cache lines are few. The original placement gives better miss rates simply by distributing the objects evenly over the cache lines for fewer lines and our algorithm performs better for caches of larger sizes. This is again because of the large working set of objects with approximately the same conflicts between the objects. Overall, for this benchmark our algorithm reduces the number of misses by 3%.

The results show that our data placement algorithm performs well in most cases. Over the given instances of benchmarks it improves upon the original placement by 30% on average. The algorithm does particularly well when the data conflict graph has edges with a diversity in weights rather than homogeneity.

5. Related Work

Cache conscious data placement is known to be a hard problem and has been studied for more than three decades.

Thabit [24] was the first to study the hardness of the problem. He discussed the problem of minimizing cache misses by constructing a *proximity graph* where objects are represented by vertices and the edge weight between two vertices is determined by the number of times the two objects appear adjacent to each other in the access sequence. He formulated the optimal placement as a partitioning problem such that if the proximity graph could be partitioned into subgraphs of equal size while minimizing the edges between the partitions, an optimal placement could be found. Petrank et al. [20] further improved on the theoretical results for the data placement problem by showing that the off-line version of cache conscious optimization cannot be approximated reasonably. Essentially their result shows that cases where there are a small number of misses cannot be distinguished from those where there are a large number of misses. They show that there does not even exist a sub-linear approximation algorithm to solve the problem. We use their formulation to describe the problem in this paper. Furthermore, we do not dispute their results but give an alternate heuristic solution. In terms of finding the best solution, Bixby et al. [3] presented a framework to find a data placement using state-of-the art 0-1 integer programming. Calder et al. [4] used profiling to determine the data access pattern of programs and optimized data placement in memory for improved cache performance. They assumed that the programs generally have similar data access patterns even with varying inputs. The problem is then solved via some smart heuristics. Their approach is similar to ours but lacks formalism. Parts of their solution can be augmented with ours to extend our solution for a profile based approach. Recent attempts at improving cache performance have turned their focus on regrouping and splitting data in objects such that objects with greater affinity appear in the same line of the cache [7–10, 19]. Parallel efforts have been made to improve available information on temporal and spatial locality [6, 13, 15, 23, 26–28].

6. Discussion and Limitations

In this section we would like to discuss some of our design decisions and limitations of our implementation to the data placement problem.

Firstly, the assumption that all objects are of the same size and fit the cache line is not realistic. This problem can be solved in a real data placement framework by integrating existing techniques like the ones given in [7, 8, 11, 13] to utilize cache lines better by coalescing conflicting objects or splitting larger objects to group conflicting fields. Since most objects are much smaller than lines in a cache, the coalescing of objects would dramatically reduce the size of the data conflict graph and hence the complexity of the problem.

Secondly, we do not distinguish between data objects created on the stack and the heap. This was thoroughly discussed by Calder et al. [4] and a detailed solution was given for placement of objects allocated on the heap and handling objects which do not occur in the profile. Our algorithm can be integrated into the framework given by Calder et al. [4] to evaluate the practicality of our algorithm.

Thirdly, a program rarely repeats the same sequence of allocation and accesses twice and profiling for a representative sequence is a challenge in itself. A practical solution would create the conflict graph from a small set of trial executions of the program and

employ some approximation to reasonably handle objects not appearing in the profile.

Lastly, other techniques such as smart prefetching and more accurate calculations for spatial and temporal locality can complement our solution for better miss rates and the solution can easily be extended for associative caches by incorporating a reasonable replacement policy.

In this work we highlight some of the theoretical issues related to cache-conscious data placement. However, we acknowledge that the implementation is not very practical in its current state but we consider it as the first step on the road to practicality.

7. Conclusions

Cache conscious placement of data in memory has been shown to be a difficult problem in the past. Earlier attempts at the problem lack proper formalization and their impact has been limited. In this paper we have shown that given the access sequence and the configuration of the cache we can assign each object to a cache line for minimum number of misses if there exists an assignment with no conflict misses. For other instances we have described a heuristic algorithm based on a graph theoretic solution which has been shown to reduce cache misses of a diverse set of benchmarks by a significant number.

References

[1] A. Aggarwal. Software Caching vs. Prefetching. In ISMM '02: Proceedings of the 3rd International Symposium on Memory Management, pages 157-162, 2002.

[2] A. H. Badawy, A. Aggarwal, D. Yeung, and C. Tseng. Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations. In ICS '01: Proceedings of the 15th International Conference on Supercomputing, pages 486-500, 2001.

[3] R. E. Bixby, K. Kennedy, and U. Kremer. Automatic Data Layout Using 0-1 Integer Programming. In PACT '94: Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, pages 111-122, 1994.

[4] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. In ASPLOS VIII: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 139-149, 1998.

[5] D. Callahan , K. Kennedy, and A. Porterfield. Software Prefetching. In ASPLOS IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 40-52, April 08-11, 1991.

[6] T. M. Chilimbi. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pages 191–202, 2001.

[7] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-Conscious Structure Definition. In PLDI '99: Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation, 1-12, 1999.

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In PLDI '99: Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation, 1-12, 1999.

[9] T. M. Chilimbi, and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, June 17-19, 2002.

[10] T. M. Chilimbi, and J. R. Larus. Using Generational Garbage Collection to Implement Cache-Conscious Data Placement. In ISMM '98: Proceedings of the 1st International Symposium on Memory Management, 37-48, 1998.

[11] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R.Silvera, and R. Archambault. Memory Pooling Assisted Data Splitting (MPADS). Proceedings of the 7th International Symposium on Memory Management, June 07-08, pages 101-110, 2008, Tucson, AZ, USA

[12] C. Ding, and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pages 229-241, May 01-04, 1999.

[13] C. Ding, and Y. Zhong. Predicting Whole-Program Locality through Reuse Distance Analysis. In PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 245-257, 2003.

[14] M. C. Golumbic. Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathe- matics, Vol 57). North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004

[15] X. Gu, I. Christopher, T. Bai, C. Zhang, and C. Ding. A Component Model of Spatial Locality. Proceedings of the 2009 International Symposium on Memory Management, June 19-20, 2009, Dublin, Ireland

[16] J. L. Hennessy, D. A. Patterson, and A. C. Arpaci-Dusseau. Computer Architecture: A Quantitative Approach. Fourth Edition, 2007. Morgan Kaufman Publishers.

[17] G. Jin, J. Mellor-Crummey, and R. Fowler. Increasing Temporal Locality with Skewing and Recursive Blocking. Proceedings of the 2001 ACM/IEEE conference on Supercomputing, pages 43- 43, 2001.

[18] A. Jula, and L. Rauchwerger. Two Memory Allocators that use Hints to Improve Locality. In ISMM '09: Proceedings of the 2009 International Symposium on Memory Management, June 19-20, 2009.

[19] C. Lattner, and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 129-142, 2005.

[20] E. Petrank, and D. Rawitz. The Hardness of Cache Conscious Data Placement. In POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pages 101-112, 2002.

[21] E. Petrank, and D. Rawitz. The Hardness of Cache Conscious Data Placement. Nordic Journal of Computing, vol 12(3), pages 275-307, 2005.

[22] G. B. Prokopski, and C. Verbrugge. Analyzing the Performance of Code-Copying Virtual Machines. In OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pages 403-422, 2008.

[23] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality Approximation Using Time. In POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 55-61, 2007.

[24] K. O.Thabit. Cache Management by the Compiler. Ph.D. thesis, Rice University, 1982.

[25] M. E. Wolf, D. E. Maydan, and D. Chen. Combining Loop Transformations Considering Caches and Scheduling. In the International Journal of Parallel Programming, vol 26(4), pages 479-503 1998.

[26] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu. A Hierarchical Model of Data Locality. In POPL '06: Conference of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 16-29, 2006.

[27] Y. Zhong, amd W. Chang. Sampling-based Program Locality Approximation. Proceedings of the 7th International Symposium on Memory Management, June 07-08, 2008, Tucson, AZ, USA

[28] Y. Zhong, X. Shen, and C. Ding. Program Locality Analysis using Reuse Distance. ACM Transactions on Programming Languages and Systems (TOPLAS), v.31 n.6, p.1-39, August 2009