Improving Cache Performance by Structure Reordering* (Extended Abstract)

Kevin Zatloukal† Adrian Corduneanu‡ Richard E. Ladner†
Vinod Grover§ Simon Meacham§

November 8, 1999

Abstract

Reordering of the members of structures is used to improve the cache performance of C and C++ programs. A new analytical model, the membership transition graph, is developed to model access behavior to structures for different orderings. The model is used to define an optimization problem whose solution guarantees a minimum number of cache misses. The model is applied to Microsoft SQL Server 7.0 to yield improvements in cache performance and overall performance.

1 Introduction

In this paper, we consider whether reordering members of C and C++ structures can improve the data cache performance and overall performance of programs. Our focus is on C and C++ programs because many major pieces software, like Microsoft SQL Server, are written in C and C++ and improving their performance is of critical importance. Because memory latency continues to grow relative to processor speed, the cache performance of programs has become an important consideration for system designers, compiler designers, and application programmers alike. All of us must be aware of and try to reduce processor stalls caused by data cache misses in order to achieve maximum performance from our programs.

Memory is organized into lines, where a line is the unit of memory that is moved into the cache as the result of a memory access. Typically, lines are 32 or 64 bytes, so several data items can be placed on the same line. It has been proposed that a good way to reduce cache misses is to place data that are often accessed together on the same line [1, 2, 3, 4, 5, 6, 7, 10]. If one datum is accessed shortly after another and neither

*Richard Ladner's research was supported by NSF grant CCR-9732828, Microsoft, and AT&T. Kevin Zatloukal and Adrian Corduneanu were interns at Microsoft Research while this research was conducted.

†Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195. {kevinz,lander}@cs.washington.edu

‡Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A1. a.corduneanu@utoronto.ca

§Microsoft Research, One Microsoft Way, Redmond, WA 98052. {vinodgro,simonme}@microsoft.com

is in the cache, then two cache misses can be reduced to one if the data are relocated to the same line. In C and C++, compilers layout the members of structures in the order given in the declaration, respecting the alignment constraints of the members. If the structure spans more than one cache line, then it may be beneficial to attempt to reorder its members such that those members that are accessed together in time are located on the same line.

This paper presents two main results. First, we present a new model for analyzing the cache performance of programs. This model allows the structure reordering problem to be represented as a specific optimization problem. Theoretically, the optimization problem is NP-hard, but various heuristic algorithms can be used to solve the optimization problem approximately. Second, we demonstrate that optimizing with respect to this model can improve the cache performance and overall performance of real industrial programs such as Microsoft's SQL Server 7.0.

The idea of reordering the members of structures is not a new idea and has been considered by others [2, 5, 6, 10]. Chilimbi, Davidson, and Larus [2] define a *field affinity graph* and Kistler and Franz [6] define a *temporal relationship graph* for a structure, both with a similar objectives. The nodes of the graph are labeled with the members of the structure and the edges are weighted with nonnegative numbers, where a large weight indicates that the members at the ends of the edge are often accessed closely in time. As a result, an ordering of the members of the structure corresponds to a clustering of the nodes of the graph where the members in each cluster are placed on the same line. The clusters are constrained so the members of each cluster can actually fit on one line. A good ordering then corresponds to a clustering where the sums of the weights of edges within clusters is maximized or, equivalently, the sums of the weights of edges between clusters is minimized. The field affinity graph and temporal relationship graph differ in how the weights are determined. The clustering algorithm of Chilimbi, Davidson and Larus and the algorithm of Kistler and Franz differ also: the former using a greedy bottom-up approach and, the latter using a top-down global optimization approach. Neither algorithm optimally solves the problem, but both approaches yield good results.

Ours is quite different from these previous approaches. We do not try to model the temporal affinity of members of a structure with weights because it is not clear how these weights formally relate to minimizing the number of cache misses, which is our real objective. Instead, we directly model the behavior of the program's accesses to members of a structure and how the ordering affects the number of cache misses. To model the behavior of the program's accesses to a structure, we define the *member transition graph* (MTG). The MTG of a structure is a Markov model, augmented to account for cache behavior. In this model, we can define precisely the miss rate for an ordering of the structure by solving a system of linear equations. With this, we have an optimization problem that we can solve in a number of ways. Most importantly, we know that a good solution to the optimization problem must yield an ordering which does a good job of minimizing the miss rate. A major contribution of this paper is the analytical model that provides a direct connection between optimization in the model and minimizing the miss rate. We discuss our model in detail in section 2.

For the MTG model to be useful, we must be able to gather trace data from which we can derive the parameters of the model. Microsoft has developed DLP (Data Locality Profiler) for gathering such data. DLP has access to Microsoft's C++ compiler's

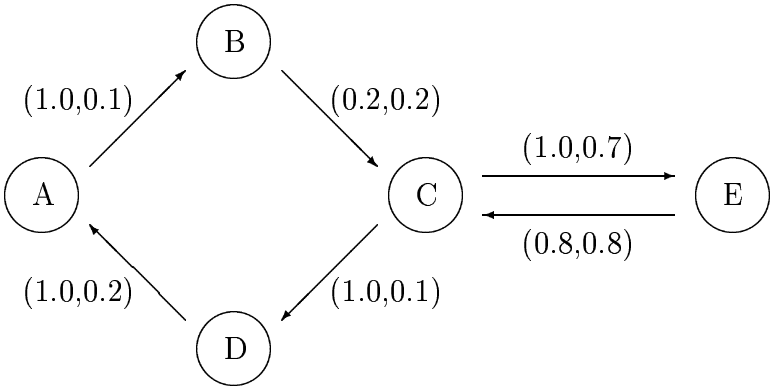

Figure 1: An example of a member transition graph (MTG).

type information which allows us to gather traces that include each memory address accessed, and for each access to a structure member, the name of the structure and its member. In addition, we need the ability to simulate the cache behavior when the structure members are reordered. The simulation step is crucial in assessing the potential benefit of a structure reordering. DLP provides the capability of simulating a structure reordering to determine the number of cache misses that would occur. We discuss DLP in detail in section 3.

To determine whether structure reordering is effective in practice, we tested our methodology on a large industrial program, namely, Microsoft's SQL Server 7.0. In section 4, we present our results, which show improvements in both cache performance and running time. In this preliminary study, we were able to reorder only seven structures. The reorderings produced by our heuristic algorithm reduced the number of cache misses for all seven structures. SQL Server's running time improved by about 1.3%. This result is less than the 2-3% improvement for SQL Server 7.0 reported by Chilimbi *et al.* [2] using their approach. However we should note that their improvement came from reordering the five most frequently accessed structures while we did not reorder any of those. This fact was due to limitations in our early versions of DLP, which will shortly be remedied. In the final version of this paper, we will include the results of reordering more than 100 SQL Server structures. We expect to see performance improvements of at least 7-10%. These results would be impressive when we consider that the original orderings of these structures were produced by the hand-tuning of software professionals. In section 5, we give our conclusions and directions for future research.

2 Modeling the Cache Behavior of Structures

To model the cache behavior of a program's accesses to the members of a structure, we build a *member transition graph* (MTG) like the one shown in Figure 1. The nodes of this graph represent the members of the structure. An edge between any members i and j contains a pair of probabilities (p_{ij}, q_{ij}). The *transition probability*, p_{ij}, is

the probability that i was the member accessed immediately before j. The *cache line survival probability*, q_{ij}, is the probability that a cache line that was in the cache when i was accessed is still in the cache when j was accessed. These two values should be independent of the ordering. In our example in figure 1, the edge from A to B is labeled $(1.0, 0.1)$ meaning that member A is always accessed immediately before member B, but the probability at that a line survives in the cache that long is only 0.1.

Our model is essentially an augmented Markov model. As such, it makes the assumption that the probability that a particular member will be accessed next depends only on which member was accessed last. Most real programs do not strictly satisfy this Markov assumption. Thus, our model is just an approximation of program behavior. However, we have found that the approximation works well in this application for two reasons: the probabilities can be measured by frequency counting from trace data, and the model provides a principled basis for reordering algorithms that improve cache performance. For some programs, the Markov model seems to be a very close approximation to the actual behavior.

With this model and its assumptions, it is possible to compute the expected cache performance of any reordering. We start by expressing the probability of a cache hit as a conditional sum over the member that is being accessed:

$$\Pr(\text{hit}) = \sum_i \Pr(\text{hit}|\text{accessing } i)\Pr(\text{accessing } i). \tag{1}$$

The probabilities $p_i = \Pr(\text{accessing } i)$ are easily computed from the p_{ij}'s by the system of linear equations:

$$\begin{aligned} p_i &= \sum_j p_{ji}\, p_j, \\ 1 &= \sum_i p_i. \end{aligned}$$

The p_i's are also easily computed by frequency counts in the trace data, so we may instead choose to consider them as inputs.

It remains to determine how we can compute $\Pr(\text{hit}|\text{accessing } i)$. This can be done by solving a system of linear equations. Let L be a collection of members on one cache line. We indicate that member i is on line L with the notation $i \in L$. Define X_i^L to be the probability that L is in the cache when i is accessed. We can determine the probabilities X_i^L by solving the following system of equations.

$$X_i^L = \sum_{j \in L} p_{ji}\, q_{ji} + \sum_{j \notin L} p_{ji}\, q_{ji}\, X_j^L, \quad \text{if } i \in L \tag{2}$$

$$X_i^L = \sum_j p_{ji}\, q_{ji}\, X_j^L, \quad \text{if } i \notin L. \tag{3}$$

To understand these equations let us first consider the case when i is on line L, equation (2). The index j represents the previous member accessed. If j is also on line L, then the probability that i's line is still in the cache when accessing i is simply q_{ji}. If j is not on line L, then L can only be in the cache when accessing i if it was in the cache when accessing j (which occurs with probability X_j^L) and it survived the transition from j to i (which occurs with probability q_{ji}). In the case when i is not on line L, equation (3), the probability that L is in the cache when i is accessed simplifies because it only

depends on which member j was previously accessed, whether L was in the cache when j was accessed (X_j^L), and whether L survived (q_{ji}). The system represented in (2–3) consists of mn equations, where n is the number of members in the structure and m is the number of cache lines spanned by the structure.

Define $L(i)$ to be the line containing the member i. For each member i, the value $X_i^{L(i)}$ is exactly the quantity Pr(hit|accessing i). Thus, we can express the probability of a hit simply as:

$$\Pr(\text{hit}) = \sum_i p_i \, X_i^{L(i)}. \tag{4}$$

The goal of finding the ordering that minimizes cache misses is now equivalent to finding the assignment of members to lines that maximizes the sum in equation (4) subject to capacity constraints for the lines and alignment constraints for the members. There is one word of caution however. It is not hard to think of an example of a structure whose members fit on 2 lines, but placing the members on 3 lines has a much better miss rate. In terms of our equations, this means that care has to be taken to choose the number of lines m to be large enough to include the optimal placement of members.

There are two problems left for DLP to solve. First, it must compute the p_{ij}'s and q_{ij}'s from trace data. Second, it must determine how to place the members onto cache lines such that (4) is maximized while respecting capacity and alignment constraints. Although we will not prove it here, this latter problem is NP-Hard. Nonetheless, there are a number of approaches to approximately solving NP-Hard problems like this one. We will describe in section 3 how DLP approximately solves this problem.

It should be apparent by now how different our approach to the structure ordering problem is from previous approaches. In previous methods [2, 6], a weighted graph was formed to model the temporal affinity between members and clustering done to find a good ordering. In our approach, we statistically model accesses to members directly. From this we construct an optimization problem using the solutions to a set of linear equations. An optimal solution to the optimization problem is one that minimizes the miss rate. In the previous methods, there is no such guarantee.

3 Data Locality Profiler

The Data Locality Profiler (DLP) consists of two distinct pieces. The first piece is the *logging engine*, which has the job of recording all of the memory access information while the program is running. In doing so, it must have the minimum possible impact on the performance of the program. This is quite a challenging task, and is outside the scope of this paper. One major difference between DLP and other profiling tools such as ATOM [9] and Etch [8] is that DLP has access to compiler tables which enables it to identify which members of which structures are accessed in the memory access stream.

The second piece is the *analysis tool*, called `dlreport`, which has the job of analyzing the trace produced by the logging engine, and from that information, finding the optimal reordering for each structure according to the model described in section 2. The analysis tool goes through three phases, as depicted in Figure 2. We will now describe each of these phases in detail.

During the scanning phase, `dlreport` examines the entire trace, and from the information therein, computes the p_{ij}'s and q_{ij}'s described above. Since a large percentage of a program's instructions are memory accesses, the logging engine can produce several

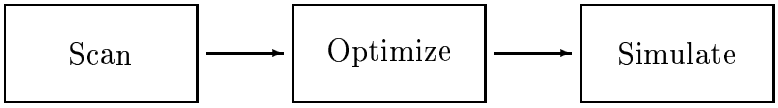

Figure 2: The three phases of the DLP analysis tool.

gigabytes of data from only a few minutes of program execution. Scanning through the trace is computationally expensive, as each memory access may require the fetching of type information, the determination of which member of the structure instance was accessed previously, and the updating of statistical information. As a result, it is not unusual for the scanning phase to require over an hour to complete for traces that are several gigabytes in length.

The technique used by `dlreport` to compute the p_{ij}'s is straightforward. During the scanning, it counts how many times each transition is taken. If we let C_{ij} denote the number of times an access to member j is immediately preceded by an access to member i, then p_{ij} is simply $C_{ij} / \left(\sum_k C_{kj}\right)$. The technique used by `dlreport` to compute the q_{ij}'s is also quite simple. During the scanning, it simulates the memory addresses in the trace on a typical cache. When member j is accessed, it simply checks whether i's line (which was brought into the cache when i was accessed) is still in the cache. The resulting probability q_{ij} is simply the number of times that this occured divided by the total number of times the transition occured. Technically, the computed q_{ij}'s are dependant on the intial ordering. However, if we assume that the size of a structure instance is small relative to the size of the cache, then this dependency disappears. The computed q_{ij}'s are also dependant on the cache parameters used. However, we have seen that in most cases changing the cache parameters used only modifies the computed q_{ij}'s by a scale factor and does not change which ordering is optimal.

During the optimization phase, `dlreport` determines, for each structure, how to partition the members into cache lines so as to optimize (4). It includes two algorithms for searching through potential orderings. The first is a Branch and Bound algorithm that considers all possibilities. Since its running time is exponential in the number of members, it can only be used for relatively small structures.

For larger structures, we use a Local Search algorithm, which iteratively improves the initial ordering. A single iteration of Local Search proceeds as follows: for each pair of members, we remove them from their respective lines and then consider placing them on every other line. The number of orderings considered on each iteration is $O(m^2n^2)$ in the worst case, where n is the number of members and m is the number of cache lines used. On all examples we have seen thus far, the Local Search algorithm has worked very well. In addition, it has the property that the final ordering is guaranteed to be no worse than the initial ordering in our model.

During the simulation phase, `dlreport` makes a second pass over the trace. During this pass, it simulates in the cache the memory references that would occur using the new orderings. The miss rate of each structure is recorded and, at the end, displayed. These miss rates can be compared with the miss rates from the original orderings, which were displayed out at the end of the scanning phase. If `dlreport` worked well,

Struct #	*Miss Rate Decrease (%)*
1	30.1
2	6.0
3	33.5
4	27.7
5	26.5
6	5.5
7	42.4

Figure 3: The percentage decreases achieved by DLP on the seven structures chosen from SQL Server 7.0.

then there should be decreases in the miss rates of each structure.

4 Results

We began by validating DLP on collection of six small example programs. These example programs were created so that we could test DLP on inputs for which we knew the optimal reordering. Some of the examples satisfied the Markov assumption; however, most did not. On all of the examples we tried, DLP produced the optimal ordering with both the Branch and Bound and Local Search algorithms.

Our main preliminary study was to use DLP to reorder the structures of Microsoft's SQL Server 7.0, a large and finely-tuned industrial application. SQL Server has more than 200 structures that are longer than one cache line in length. However, limitations in the early version of DLP — dealing with bit fields and inheritance — allowed only 43 structures to be eligible for reordering. At that point, the optimization algorithms in `dlreport` were still under development, so the optimization was done by hand; however, the optimization methodology was still that described in section 2 above. We could only deal with relatively small structures, which left about 15 eligible. Of those, we picked seven that were frequently accessed and for which we saw strong miss rate improvements in the simulator. The decrease in cache miss rate for those seven structures ranged from 5-45% and are shown in Figure 3.

A version of SQL Server with the reordering in place was run on a standard benchmark. It showed about a 1.3% improvement over the baseline measurement. In particular, the reordered version maintained a more than 1% speedup relative to the baseline for more than 2 minutes.

The structures reordered in our preliminary study were chosen for convenience. We know that these did not include any of the most-referenced SQL Server structures. In particular, we have not yet attacked the five most-referenced SQL Server structures that were reordered by Chilimbi *et al.* [2], from which they achieved a 2-3% performance improvement. In our upcoming work, we will test a fully-functional version of DLP. This test will include many more structures (at least 10 times as many) and will use the superior, automated algorithms. We expect to see speed improvements of at least 7-10%.

5 Conclusion

We have shown that we can model access behavior to members in structure as an augmented Markov model. The problem of finding the structure ordering that minimizes cache misses in the model is equivalent to maximizing a sum that depends on the solution to a set of linear equations. This model can be applied to improve the cache performance of programs by using trace data to supply the parameters of the model. Although one can criticize the use of a Markov model that simplifies program behavior to statistics, it is often still useful to use these models since they do work well in practice.

Kistler and Franz [6] point out that it may be advantageous to reorder structures dynamically depending on the data that is being supplied to the program. Our framework could be used as replacement for their TRG approach for dynamic reordering. In the dynamic setting the needed probabilities for the model change over time. Our Local Search algorithm might work well because it iterates from known solutions to betters ones. The previously best solution will likely not be best when the parameters of the model change. The Local Search algorithm will search for a better solution "near" the previous one. In essence, it will adapt to the changes in the model.

6 Acknowledgments

We are grateful to the Performance Tools Team in Microsoft Research for sponsoring our work. Specifically, we would like to thank David Erb and Michael Parkes for their ideas, discussions, support, and encouragement.

References

[1] Brad Calder, C. Krintz, S. John, and T. Austin, Cache-conscious data placement, Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 139-149, October, 1998.

[2] T.M. Chilimbi, B. Davidson, and J.R. Larus. Cache-conscious structure definition. Proceedings of the ACM SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI), pp. 13-24, May, 1999.

[3] T.M. Chilimbi, M.D. Hill, and J.R. Larus. Cache-conscious structure layout. Proceedings of the ACM SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI), pp. 1-12, May, 1999.

[4] Dirk Grunwald, Ben Zorn, Robert Henderson, Improving the cache locality of memory allocation. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation(PLDI), pp. 177-186, 1993.

[5] T. Kistler and M. Franz. Automated layout of data members for type-safe Languages, Technical Report No. 98-22, Department of Information and Computer Science, University of California, Irvine; May 1998, revised September, 1998

[6] T. Kistler and M. Franz. The case for dynamic optimization. Technical Report No. 99-21, Department of Information and Computer Science, University of California, Irvine, May, 1999.

[7] P.R. Panda, N.D. Dutt and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 2, No. 4, pp. 384-409, October, 1997.

[8] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad. Instrumentation and optimization of Win32/Intel executables using Etch. USENIX Assoc. 1997.

[9] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI), pp. 196-205, 1994.

[10] Dan Truong, Francois Bodin, and Andre Zeznec. Improving cache behavior of dynamically allocated data structures. Proceedings of the Conference on Parallel Architectures and Compilation Techniques, October 1998.