UC Irvine
ICS Technical Reports

Title
Automated record layout for dynamic data structures

Permalink
https://escholarship.org/uc/item/4sv185xn

Authors
Kistler, Thomas
Franz, Michael

Publication Date
1998-03-14

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ICS

TECHNICAL REPORT

Automated Record Layout for Dynamic Data Structures

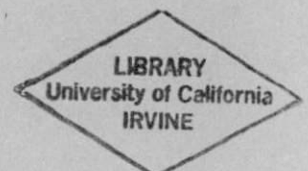
Thomas Kistler
Michael Franz

Technical Report UCI-ICS 98-22
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

March 14, 1998

Information and Computer Science

University of California, Irvine



Automated Record Layout for Dynamic Data Structures

SL BAR
Z
699
C3
no. 98-22

Thomas Kistler and Michael Franz

Department of Information and Computer Science
University of California at Irvine
Irvine, CA 92697–3425

Abstract. As the gap between processor power and memory speed continues to widen, cache performance of modern processors is becoming increasingly important for program performance. Lately, compilers have addressed this problem by prefetching data ahead of time, by changing the memory layout of program data, or by changing the control structure of a program. While most of these techniques work well for array-based numeric programs, their potential for component-based and object-oriented applications that use dynamic pointer-based data structures is fairly limited.

In this paper, we present and evaluate a simple, yet efficient optimization technique that increases cache performance for pointer-based applications. Based on temporal profiling data, our algorithm reorders and aligns individual record fields in dynamically allocated objects to increase spatial and temporal locality. It also takes advantage of modern hardware features such as data cache line-fill buffer forwarding.

Our results demonstrate that the proposed algorithm significantly increases performance of pointer-based applications. It increases program speed by as much as 15% and reduces data cache miss rates by up to 30%.

1 Introduction

As the growth in raw processing power continues to outpace improvements in memory speed, cache performance is increasingly becoming a limiting factor of application performance. To some degree, compilation techniques offer relief, especially in the area of scientific computing. Data-prefetching loads data and instructions ahead of time to reduce memory latency [LM96, MLG92], and cache-blocking, loop-skewing, and loop-tiling increase data locality by transforming the control structure of a program [WL91]. Although these techniques have proven very valuable for array-based programs, their merit for applications that utilize highly dynamic data structures remains widely unexplored. Applications of the latter kind include object-oriented programs and component-based programs. The cache behavior of such programs is extremely difficult to predict at compile time, not only because the size of data structures can usually not be determined ahead of time, but also because of irregular memory access patterns.

In this paper, we present an optimization technique that increases cache performance specifically in the domain of pointer-based applications. It is based upon the observation that most compilers poorly exploit one of the major sources

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

for increasing spatial and temporal locality—the possibility of arranging fields within records (and classes) at will. Instead, most compilers retain the ordering of fields declared in the source code. It does not take much insight to see that this strategy is likely to be sub-optimal. Software designers declare types, variables, and classes to increase readability and maintainability of the source code. They favor source code portability over efficiency and are generally not willing to take into account many architecture-dependent parameters. Also, software engineers often lack principal knowledge about the underlying machine architecture (e.g. the size of the cache, or the size of a cache line) that would make such programmer-directed optimizations feasible in the first place. However, the full achievable performance can be reached only with a very specific ordering of record fields. Not very surprisingly hence, retaining the declared ordering often does not completely exploit the potential of the underlying processor.

Another approach usually taken by commercial compilers is to arrange fields to minimize the space used by dynamically allocated objects [Muc97]. This approach is based on the assumption that a smaller memory footprint leads to faster applications, especially in garbage collected environments. However, as we will see in Section 6, this assumption is misleading. In some cases, increasing the object size uncovers greater flexibility in grouping and aligning record fields, and accelerates applications quite noticeably.

The algorithm that we present in this paper pursues a different approach. It automatically reorders fields within records to increase data cache locality and to take maximum advantage of advanced hardware features (e.g. data cache line-fill buffer forwarding [Mot96]). Also, it does not involve the programmer in the optimization process but elegantly decouples software-engineering concerns from performance issues.

The algorithm is based on a simple strategy that first splits records into partitions the size of a single cache line. Record fields whose individual accesses are close together in time are assigned to the same cache line to maximize data locality and to minimize the number of cache misses. Then, after splitting, all the fields in a single partition are ordered to minimize load latency. Here the algorithm has specifically been designed to exploit data cache line-fill buffer forwarding. In addition, the optimization process is guided by profiling data. The profiler runs prior to program optimization and collects temporal information about access patterns and access frequency of individual record fields.

The remainder of this paper is organized as follows: Sections 2 through 5 discuss different aspects of the algorithm. Section 6 applies the algorithms to several benchmarks. Section 7 discusses related work and Section 8 concludes the paper.

2 An Algorithm for Automatic Record Layout

Traditionally, the main focus in improving data cache performance has been on data locality—to reduce the number of cache misses that occur while executing a program. Our algorithm increases data locality by splitting records into parti-

tions of fields that are temporally related—that is, frequently accessed within a certain period of time.

However, with the ever advancing state-of-the-art of microprocessor architectures, seemingly insignificant hardware features can suddenly have a major effect on cache performance as well. Cache line-fill buffer forwarding is an example of such a hardware feature. It can be found in more and more microprocessors, such as the PowerPC 604e. Traditionally, on processors without this feature, when a load misses the cache, it is placed into the load queue. The critical requested data (*d0*) comes back first and is unconditionally forwarded to the requesting unit. Subsequent requests to the same cache block are required to wait until the remaining double words (*d1, d2, d3*) are transferred from memory—one double word is transferred per cycle from increasing addresses, wrapping back to the beginning of the cache block as required. Not so with line-fill buffer forwarding: if a subsequent load to the same cache block is requested, the data is also immediately forwarded to the load/store unit upon availability. Fig. 1 demonstrates

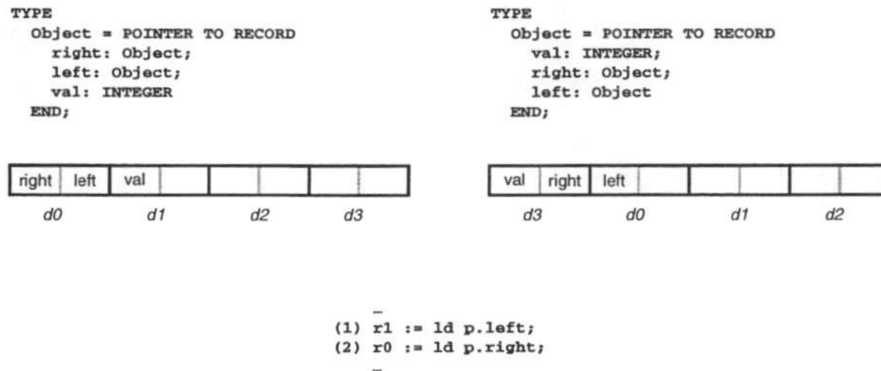


Fig. 1. Post Order Tree Traversal

the impact of line-fill buffer forwarding on cache performance. It depicts a small code sequence for a post-order traversal of a binary tree, with two possible storage layouts for tree nodes illustrated at the top-left and at the top-right. Using the type declaration at the left, and assuming a memory latency of 10 cycles, load instructions (1) and (2) can be executed in 11 cycles; one cycle is required to compute the effective address of instruction (1) and 10 cycles are required to cover the memory latency. Since both instruction (1) and (2) address the critical double word *d0* they can be retired in the same clock cycle.

Not so if we use the type declaration depicted at the right. It requires three additional cycles which is equivalent to a performance loss of 30%. Though the result of (1) is available after 11 cycles, the result of (2) is only available after double word *d1*, *d2*, and *d3* have been transferred from memory.

This example illustrates that a good strategy for rearranging fields not only

considers the placement of fields into individual cache lines to increase data locality, but also models the special capabilities of the underlying processor architecture. This can best be done by carefully ordering record fields within single cache lines.

The following sections present an overview of our algorithm. We first elaborate on the requirements and on implementation issues for the profiler that drives the optimization. We then discuss the partitioning of fields. Finally, we explain the ordering of fields within cache lines. For clearness, a pseudo code outline of the algorithm is given in the appendix of this paper.

3 Creating Profiles

In order to determine how to split records into optimal partitions and how to order fields within partitions, a conflict cost measure is required. An optimal conflict cost measure estimates the execution time penalty caused by a particular arrangement of fields. Unfortunately, such an optimal measure is extremely difficult to find in practice. We therefore fall back on a simplified measure that estimates the number of cache misses, under the assumption that the number of cache misses is related to the execution time penalty.

We compute this conflict cost measure based on a temporal relationship graph that, for a particular record, captures information on how its fields are accessed. Similar relationship graphs have lately been proposed to guide procedure placement [GBS+97] in the context of instruction cache optimization. Vertices in this graph correspond to record fields. Vertices are connected by edges whose weights represent the degree of temporal dependency between the two connected fields. More concretely, the weights reflect the number of times that the two fields are accessed subsequently during a specified time interval. They consequently capture an estimate on the number of cache misses that occur if both fields are placed in different cache lines.

The graph is created by profiling the program and maintaining a stack-like data structure of the least recently accessed fields. For every load/store instruction that references a field F in a dynamic object O we push the pair (O, F) onto the stack and remove an earlier reference to (O, F) from the stack. When the size of all the pairs on the stack exceeds the stack size, pairs are removed from the stack bottom. Next, we traverse the stack top-down and search for pairs (O, F') that reference the same object but a different field F'. For every match, we increment the weight of the edge between F and F'. We so model the fact that field F is used following an access to field F'.

Unfortunately, the two main optimization parts—partitioning fields into different cache lines and ordering fields within cache lines—require slightly different temporal information and parameters. The first dissimilarity is the optimal *stack size*. The stack size limits the number of fields concurrently on the stack and controls the recording of relationships—older objects are displaced from the stack due to capacity constraints. On the one hand, for partitioning, an optimal algorithm records relationships as long as both fields reside in the data cache.

The optimal stack size is therefore equivalent to the size of the data cache, to accurately model the capacity constraints of the data cache. But for ordering, the stack size must be no larger than the number of cycles required to load an entire cache line (i.e. 4 load/store instructions for the PowerPC 604e). This is because accessing a field within an already loaded cache line comes at no additional cost. Only accessing the field while the cache line is being loaded penalizes an inadequate ordering.

The second dissimilarity is the *edge increment* for two related fields on the stack. For partitioning, a unique increment by one is sufficient. Accessing two fields located in different cache lines requires loading both cache lines—independent on how many cycles pass between these two accesses. However, for ordering fields within cache lines, we prefer an increment that is proportional to the number of cycles between the accesses—or that is proportional to the distance between the two pairs on the stack. The reason for this is that accessing two fields in the same cache line within one cycle might penalize more then accessing them within three cycles.

So how can these different requirements be integrated into one basic model? Maintaining two different temporal graphs and two stacks with different sizes would be a naive solution. A more adequate solution maintains two different weights per edge—one that is used for partitioning the graph, and one that is used for ordering fields within partitions. The weights for graph partitioning are computed as described above, using a stack whose size is optimal for graph partitioning, and using a constant edge increment of one. The weights for field ordering are computed using the same stack, and using edge increments that are proportional to the distance between the two pairs on the stack. However, though utilizing the same stack, we only consider items within a certain distance from the top of the stack while traversing the stack top-down. This is equivalent to using a smaller stack. We will subsequently call this distance the *look-ahead* distance.

In order for the cost measure to be accurate, the profiler has to capture all the load/store instructions in the running program. Interpretation of the program is one method to achieve this goal. Unfortunately, interpretation slows down the program by several orders of magnitude, making it a practically useless tool in a production environment. Instrumenting all load/store instructions with calls to special profiling routines alleviates this problem and reduces runtime, but still not by as much as required to make it applicable. A third possibility is to utilize path profiling [BL96]. Instead of creating the temporal relationship graph during the profiling pass, we only measure the execution frequency of paths within single procedures. The temporal relationship graph is then created during a post-profiling pass which, for every procedure, walks through the most frequently executed paths and applies the above technique. For path profiling, Ball and Larus [BL96] have reported a runtime overhead of approximately 10%. Unfortunately, utilizing path profiling also comes at a cost: Since we look at procedures individually we miss important relations between subsequent field accesses from different procedures. The larger the stack the more relations we

miss because of the increased likelihood of field accesses from different procedures being on the stack concurrently. At first sight this is a serious disadvantage for our intuition suggests that a larger stack captures more relations and is thus better suited for our layout technique. However, our results indicate the contrary. Using a smaller stack size achieves better performance throughout all benchmarks (see Section 6). Consequently, path profiling is capable of capturing most of the essential relations and appears to be a good alternative for recording temporal relationships.

4 Graph Partitioning

The first component of our optimization partitions the fields of a record into subsets of fields that fit into a single cache line. In technical terms, it searches for a k-way graph partitioning of the temporal relationship graph G into partitions $P_1, \ldots, P_k$, where $k = \lceil \mathit{recordsize}/\mathit{cachelinesize} \rceil$, $|P_i| \leq \mathit{cachelinesize}$, and the sum of all edges between the partitions is minimized. We solve the k-way partitioning problem by recursive bisection, that is, we first obtain a 2-way partition of the graph that splits the graph into two equal-sized parts. We then further subdivide each part using 2-way partitioning (also called a graph bisection).

For bisecting the graph, we have adopted a variation of the Kernighan-Lin Graph Bipartitioning Algorithm [KL70, Dut93] that can be implemented quite efficiently. It works as follows: Given a graph G with $n = 2m$ vertices, we initially create two arbitrary partitions P_1 and P_2, with $|P_1| = |P_2| = m$. We then start an iterative process that is called a *pass*. A pass can best be summarized as trying to find two equal-sized subsets $S_1 \subset P_1$ and $S_2 \subset P_2$, such that swapping S_1 and S_2 reduces the total cost of edges from P_1 to P_2. This is done by choosing a pair of unmarked vertices $(v_1, v_2) \in P_1 \times P_2$ that, by swapping vertices v_1 and v_2, minimizes the total cost of edges from P_1 to P_2. Both vertex v_1 and v_2 are then marked in P_1 respectively in P_2, but not actually swapped. This procedure is repeated until all the vertices in P_1 and P_2 are marked. At this point we have computed a sequence of pairs $(v_{1,1}, v_{2,1}), \ldots, (v_{1,i}, v_{2,i}), \ldots, (v_{1,m}, v_{2,m})$. For every $i : 1 \leq i \leq m$, we compute the associated gain G_i for swapping vertex pairs $1 \ldots i$. We choose p so that gain G_p is maximal and exchange all vertex pairs $(v_{1,1}, v_{2,1}), \ldots, (v_{1,p}, v_{2,p})$ between partitions P_1 and P_2. A number of passes are made until the maximal gain G_p is 0 and a local minimum is reached. Since this local minimum is highly depended on the choice of the initial partition, we repeat this process for different randomly created initial partitions.

Prior to applying the Kernighan-Lin algorithm, however, the graph requires some minor adjustments. First, in order to end up with k equal-sized partitions of the size of a cache line, the original graph has to be expanded to the nearest $\mathit{cachelinesize} \cdot 2^k$. We do this by inserting additional *fill vertices* into the graph. The fill vertices are weakly connected to all other vertices in the graph with edges of weight zero.

Second, individual fields have different sizes, which prevents the algorithm from swapping any two arbitrary vertices in the graph. In order to maintain

equal-sized partitions the algorithm is allowed to swap fields with equivalent sizes only, such as two one-byte characters fields or two four-byte floating-point fields. However this is too restrictive for we might also want to swap a field of type floating-point with four fields of type character (note that this maintains two equal-sized partitions). To facilitate this, we split every field F of size s into a cluster of s vertices of size one. These vertices are strongly connected by assigning infinite weights to the edges between them. This prevents placing the cluster into different partitions but allows swapping vertices at will.

Finally, the concept of structural type inheritance needs special attention. For saving memory resources, we might consider placing the first field of a derived type into the last partially used cache line of its base type, rather than placing it into a new cache line. We simulate the last partially used cache line by inserting an additional *leader vertex* into the graph. The size of this leader vertex is the used fraction of the last cache line of the base type (size(*leader*) = size(*basetype*) mod *cachelinesize*). The partitioning algorithm is then modified so that the leader vertex is always placed at the beginning of the first cache line. In addition, instead of inserting one individual vertex for every field of the base type, the leader vertex acts as a container that combines all of them into one vertex. The leader vertex is connected to other vertices in the graph by employing one of two different algorithmic variants—subsequently called *memory-conscious* and *latency-conscious*. In the latency-conscious version, all the edge weights between the leader vertex and other vertices are set to zero. This reduces the likelihood that any field is placed into the same partition as the leader and thereby wastes memory resources. However, it increases cache locality and reduces memory latency in the case of minor temporal relationships between fields of the base type and fields of the derived type. In the memory-conscious variant, the edge weights between the leader vertex and other vertices are computed as we compute weights between two regular vertices. Fields are more likely to be placed into the same cache line as the leader vertex which reduces memory expenditures. However, more cache misses might result in the case of minor relations between the base type and the derived type. Section 6 illustrates how these two scenarios affect execution time of optimized programs.

5 Field Ordering

The second component of our algorithm arranges fields within cache lines. It takes advantage of cache line-fill buffer forwarding. The algorithm exhaustively searches for the permutation that minimizes the load latency cost $C(P)$ associated with a particular permutation P. The cost function $C(P)$ for a given permutation of n fields $P = (F_1, F_2, \dots, F_n)$ is depicted below:

$$C(P) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{i,j} d_{i,j} + w_{j,i} d_{j,i}$$

$$d_{i,j} = (F_i.\mathit{adr} \text{ div } \mathit{buswidth}) - (F_j.\mathit{adr} \text{ div } \mathit{buswidth})$$

$$d_{j,i} = (\mathit{linesize} \text{ div } \mathit{buswidth} - d_{i,j}) \bmod (\mathit{linesize} \text{ div } \mathit{buswidth})$$

In this formula $w_{i,j}$ represents the weight of the edge between field F_i and field F_j in the temporal graph. $F_i.adr$ denotes the address of field F_i relative to the cache line, and $d_{i,j}$ represents the number of intervening cycles between accessing field F_i and accessing field F_j in the case of a cache miss. Accordingly, $d_{j,i}$ represents the number of intervening cycles between accessing field F_j and accessing field F_i in the case of a cache miss. As an example in the left type declaration of Fig. 1, if *right* is accessed first and misses the cache, *val* is only available one cycle after *right* is available ($d_{right,val} = 1$). However, in the case of *val* being accessed first, *right* is available at earliest three cycles after *val* is forwarded to the requesting unit ($d_{val,right} = 3$). Although not illustrated in the above formula, fields also have to be aligned properly. As an example, a double precision floating-point value that is not aligned on an 8 byte boundary results in a high cost value.

Although our algorithm uses an exhaustive search technique that has an exponential complexity, runtime is not a major problem in practice—the number of fields in a cache line is usually fairly small. Also, using smarter branch-and-bound versions of the algorithm further reduces the runtime requirements.

6 Results

In the last few months, we have experimented with record layout techniques and implemented a version of the described algorithm that has been integrated into our optimizing compiler. In addition, we have also integrated the technique into our operating system that performs optimizations in the background, while the application is being executed [Kis96, Fra97].

To measure the gains of our automated field layout technique, we selected a suite of seven benchmarks and ran them multiple times in a PowerPC based, garbage-collected environment [WG89]. The suite is briefly summarized in Table 1: *OOO2* is an optimizing compiler for the programming language Oberon [Wir90]. It utilizes large internal data structures, based on static single assignment form [CFR+91]; *Layouts* is an interpreter for a batch-processing language

Benchmark	Program Size (Lines of Code)	Execution Time (Mio. Cycles)	Load/Store Ratio
OOO2	23621	5452	7.49
Layouts	1855	305	1.13
TreeAdd	85	5928	8.56
Bisort	216	1706	3.86
TSP	424	5416	6.75
Numbers	1108	965	8.62
Huffman	554	630	10.38

Table 1. Benchmark Characteristics

that composes interactive graphical user interface components; both *TreeAdd* and *Bisort* are part of the Olden benchmark suite [RCR+95] and, although they are very small in size, they represent frequently used operations on dynamic data structures; *TSP* is an implementation of the traveling salesman problem; *Numbers* is a module for arbitrary precision floating-point number calculation; and *Huffman* is an implementation of a compression/decompression algorithm. Although all these benchmarks considerably vary in size, they represent a variety of applications and programming styles. Also, each of them allocates many megabytes of data and executes billions of instructions.

To evaluate the exact performance of our algorithm in terms of improvement in execution time and cache miss rates we ran the entire benchmark suite on a PowerPC 604e (32-Kbyte, four-way set-associative data cache) utilizing the PowerPC's performance monitor. The performance monitor includes four 32-bit hardware counters that count detailed events during execution, such as instruction dispatches, instruction cycles, misses in the cache, or load/store miss load latencies. Further, in order for the profiler to compute an accurate cost measure, we instrumented all load/store instructions prior to program execution.

Benchmark	Reduction of Execution Time	Reduction of Cache Miss Rate
OOO2	3.4%	4.1%
Layouts	0.1%	1.0%
TreeAdd	16.5%	33.2%
Bisort	2.9%	26.5%
TSP	0.7%	1.0%
Numbers	0.0%	0.0%
Huffman	0.1%	1.6%

Table 2. Overall Performance

Table 2 shows the overall performance results. *TreeAdd* achieves the largest speedup and improves execution time by over 16%. Since *TreeAdd* allocates objects smaller than a single cache line, the entire speedup has to be attributed to the proper exploitation of cache line-fill buffer forwarding. Again, this emphasizes the importance of this seemingly unimportant hardware feature. An algorithm based solely on improving cache locality could simply not achieve this level of performance. The speedups of *OOO2* and *Bisort* are also promising—we recorded an increase in performance by 3%. Not so however for the rest of the programs in the benchmark suite. They do not show any significant increase in performance. The reasons for this are twofold. First, *TSP*, *Numbers*, and *Huffman* do not allocate large objects, hence lack potential for increasing cache locality. Second, the ordering of fields within records is already near optimal leaving very little room for optimizations. The story is different for *Layouts*. *Layouts* is very

memory intensive while composing graphical user interface gadgets. It allocates and initializes numerous objects but makes little reuse of allocated objects. The potential for reducing the cache miss rate is thus minimal.

Table 3 and Fig. 2 show a more distinctive picture of the results by breaking down the results by different input parameters—the *stack size* and the *look-ahead* for profiling, and the choice between *memory-conscious* partitioning (MC) and *latency-conscious* partitioning (LC). They illustrate that LC does clearly better in *OOO2*, although the average data structure is 8% larger than in MC. This can be explained by the fact that *OOO2* is an "allocate once—reference often" type of application. It creates an intermediate data structure early in the compilation process and frequently traverses it during optimization passes. Reducing latency—not memory consumption—is therefore of primary importance. Also, for many object-oriented applications that extensively use type hierarchies, types are usually well encapsulated and do not allow direct access to fields from derived types. This minimizes temporal relations between fields in different derivation levels. It is therefore wiser to arrange fields of single types in separate cache lines rather than mixing them with base types. In contrast to *OOO2, Layouts*

Allocation-method	Stack-size	Look-ahead	Runtime Reduction OOO2	Runtime Reduction Layouts
MC	2	1	1.51%	-0.70%
MC	64	1	1.25%	0.18%
MC	1024	1	0.58%	-0.67%
MC	64	2	-0.50%	-0.65%
MC	1024	2	-0.61%	-0.02%
MC	64	4	0.61%	0.04%
MC	1024	4	-0.72%	0.11%
LC	2	1	1.42%	0.16%
LC	64	1	3.38%	-2.38%
LC	1024	1	3.15%	-0.81%
LC	64	2	2.28%	0.08%
LC	1024	2	2.39%	-0.11%
LC	64	4	3.07%	-0.83%
LC	1024	4	2.77%	-0.83%

Table 3. Method Comparison

does better in MC. The reason for this is that *Layouts* is an application of type "allocate often—reference once". It allocates and connects data structures but rarely reuses them. Especially in a garbage-collected environment, reducing the memory footprint is therefore more important. From our experience, the ratio of load instructions to store instructions is usually a good instrument to classify applications. Applications with a high load/store ratio are very likely to perform better with LC whereas applications with a low ratio usually do better with MC.

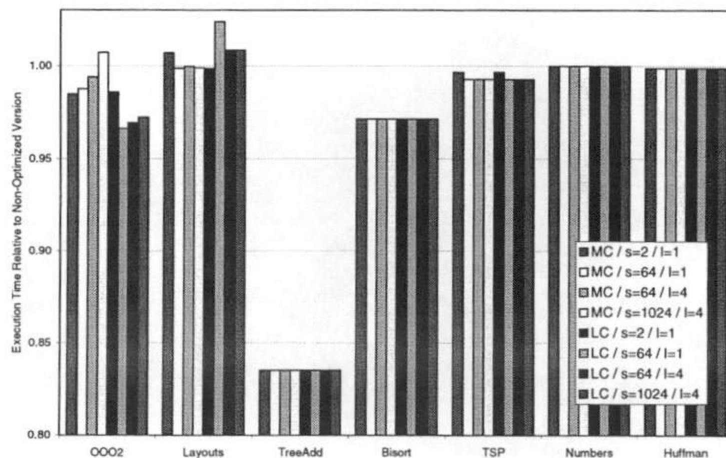


Fig. 2. Method Comparison

Table 3 also shows that using a medium-sized stack does better than using a large stack. This is in surprising contrast to what we would expect in the first place. In all the benchmarks, we found that using a stack size of 64 entries performs best. Similarly, a small look-ahead of one performs clearly better than a larger look-ahead. This supports our theory that the profiler need not necessarily interpret the program or insert instrumentation for every load/store instruction. Since a small stack size considerably reduces the likelihood of two fields accessed from different procedures being on the stack concurrently, path profiling is powerful enough to capture the essential relations.

7 Related Work

To bridge the gap between memory speed and processor speed, several cache optimization techniques have been proposed lately. In the area of scientific, array-based programs, loop-reversal, loop-tiling, loop-skewing, and cache-blocking [WL91] are all popular techniques to increase data locality. They change the algorithmic behavior of the program by reordering the execution sequence of iterations and changing the shape of a loop's iteration space and iteration depth. Another technique proposed by Rivera and Tseng [RT98] further increases cache performance. They propose an algorithm that inserts inter-variable and intra-variable padding to control the placement of arrays in memory and to control the optimal row size of arrays. Their technique can be applied orthogonally to the above-mentioned control-transforming optimizations.

Previous work has also studied the problem of data-prefetching in the context of array-based programs [MLG92]. Prefetching reduces memory latency by load-

ing data values ahead of time into the cache. Primarily for array-based programs that exhibit highly regular data access patterns, prefetching is very promising.

Unfortunately, because of the fundamental differences in program structure, the techniques utilized for scientific programming cannot be applied directly to pointer-based applications. Pointer-based programs usually exhibit much more complex access patterns. Also, the size of data structures can usually not be determined at compile time. It is therefore not very surprising, that only few studies have been published about data prefetching [LM96], automatic placement of data in memory, or automatic data transformations in the context of pointer-based applications. However, with the ever-increasing memory hunger of pointer-based programs, other issues have also become important for data cache performance. Among them, are the effects of memory allocators [GZH93, SZ97, AG98] and garbage collectors [Rei94] in modern operating systems.

Finding an optimal k-way partition for large graphs is an NP-complete problem. As such, there exists no algorithm that solves the problem in polynomial time. However, a wide variety of heuristics-based approaches have been published in the last 30 years. One of the original papers by Kernighan and Lin describes a very efficient algorithm for bipartitioning large graphs [KL70]. Several refinements of this algorithm have been described since, among them the improved version by Dutt [Dut93] that we use for partitioning records. There also exist more advanced algorithms based on multilevel partitioning schemes [KK95]. A multilevel graph partitioning algorithm first coarsens the original graph down to a few hundred vertices. It then bipartitions the coarsened graph and projects the result back onto the original graph. Multilevel partitioning algorithms are usually targeted towards graphs with thousands of vertices—especially in the domain of electronic circuitry layout and telephony network design. However, since our graphs are usually small in size, the use of multilevel-partitioning algorithms does not seem justified.

8 Conclusion

In this paper, we have proposed a simple record layout technique for dynamically allocated objects. Our technique not only increases data cache locality but also exploits advanced hardware features such as cache line-fill buffer forwarding. This is becoming increasingly important to exploit the full potential of modern processor architectures. Moreover, our technique can be implemented as an orthogonal addition to commonly used optimization techniques, such as data-prefetching or optimal data placement.

The algorithm is based on a two-tiered strategy that first assigns fields to single cache lines and then optimizes the order of fields within cache lines. According to the type of the application to be optimized, it can be parameterized to arrange fields in a cache-conscious way, or in a latency-conscious way.

While we have investigated the effect of different data layout techniques for dynamic data structures and have measured performance improvements by as much as 16%, we have not yet studied the effects of placing global variables,

local variables, or procedure parameters. This is an issue that we are planning to address in the near future.

9 Acknowledgement

We would like to thank Peter Fröhlich who provided many helpful comments on an earlier version of this paper. Part of this work is being funded by the National Science Foundation under grant CCR–97014000.

References

[AG98] A. Aiken and D. Gay. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[BL96] T. Ball and J. Larus. Efficient Path Profiling. *In Proceedings of MICRO-29*, Paris, France, December 1996.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently Computing Static Single Assigment Form and the Control Dependence Graph. In *ACM Transactions on Programming Languages and Systems* 13(4):451-490, 1991.

[Dut93] S. Dutt. New Faster Kernighan-Lin-Type Graph-Partitioning Algorithms. In *Proceedings of the IEEE/ACM International Conference on CAD*, November 1993.

[Fra97] M. Franz. Run-Time Code Generation as a Central System Service. In *The Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, IEEE Computer Society Press, pp 112–117, May 1997.

[GBS+97] N. Gloy, T. Blackwell, M. Smith, and B. Calder. Procedure Placement Using Temporal Ordering Information. In *Proceedings of Micro-30*, December 1997.

[GZH93] D. Grunwald, B. Zorn, and R. Henderson. Improving the Cache Locality of Memory Allocation. *In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp 177–186, June 1993.

[Kis96] T. Kistler. Dynamic Runtime Optimization. In *Proceedings of the Joint Modular Languages Conference, JMLC'97*, pp 53-66. Published as Springer Lecture Notes in Computer Science No. 1204, March 1997. Also published as Technical Report No. 96-54, Department of Information and Computer Science, University of California, Irvine, November 1996.

[KK95] G. Karypis and V. Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* Technical Report CORR 95-035, University of Minnesota, Department of Computer Science, Minneapolis, June 1995.

[KL70] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. In *Bell System Tech. Journal*, Vol. C-33, pp 291-307, May 1970.

[LM96] C.-K. Luk and T. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[MLG92] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 76-84, October 1992.

[Mot96] Motorola Inc. *PowerPC 604e Microprocessor Supplement and User's Manual Errata.* Motorola Inc. September 1996.

[Muc97] S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufman, 1997.

[RCR+95] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. In *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995.

[Rei94] M. Reinhold. Cache Performance of Garbage-Collected Programs. In *Proceedings of the 21th Annual Symposium on Principles of Programming Languages,* pp 206–217, June 1994.

[RT98] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the 25th Annual Symposium on Principles of Programming Languages,* June 1998.

[SZ97] M. Seidl and B. Zorn. *Predicting References to Dynamically Allocated Objects.* Technical Report CU-CS-826-97, Department of Computer Science, University of Colorado, Boulder, January 1997.

[WG89] N. Wirth and J. Gutknecht. The Oberon System. In *Software—Practice and Experience,* 19(9): 857–893, September 1989.

[Wir90] N. Wirth. *The Programming Language Oberon.* Technical Report No. 143, Institute for Computersystems, Swiss Federal Institute of Technology, Zurich (ETHZ), November 1990.

[WL91] M. Wolf and M. Lam. A Data Locality Optimization Algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp 30–44. Published as SIGPLAN Notices 26(6), June 1991.

A Outline of Algorithm

```
(* Recursively partition the given graph into
equal-sized partitions of the size of a cache line *)
PROCEDURE Partition(graph: Graph; list: GraphList);
  VAR left_graph, right_graph: Graph;
BEGIN
  IF size(graph) > CacheLineSize THEN
    (* Partition the graph into two equal-sized subsets
    left_graph and right_graph, such that the total cost
    of edges from left_graph to right_graph is minimized *)
    KLBisect(graph, left_graph, right_graph);

    Partition(left_graph); Partition(right_graph)
  ELSE
    Append(list, graph)
  END
END Partition;

(* Order fields within cache lines in such a way that
access costs are minimized *)
PROCEDURE Order(graph: Graph): Permutation;
BEGIN
  Exhaustively search for field-permutation P with minimal
  total cost C:
    let P := (field_0, field_1, ..., field_n)
    d_i,j := field_i.adr DIV CacheBusWidth - field_j.adr DIV CacheBusWidth
    d_j,i := (LineSize DIV BusWidth-d_i,j) MOD LineSize DIV BusWidth
    C := sum_i=1..n-1(sum_j=i+1..n(w_i,j*d_i,j + w_j,i*d_j,i))
  RETURN P
END Order;

(* Main procedure *)
PROCEDURE AlignFields (graph: Graph);
  VAR list: GraphList; perm: Permutation;
BEGIN
  (* Adjust graph *)
  InsertLeader(graph);
  InsertSizeAdjustment(graph);
  InsertFiller(graph);

  (* Split graph into partitions of the size of a cache line *)
  list := {}; Partition(graph, list);

  (* Reorder fields within cache lines (i.e. paritions) *)
  adr := 0;
  FOR all graphs G in list DO
    RemoveSizeAdjustment(G);
    perm := Order(G);

    (* Assign addresses to fields in permutation *)
    AssignAddr(perm, adr);
    adr := adr + size(perm)
  END
END AlignFields;
```