

The Case for Dynamic Optimization

Improving Memory-Hierarchy Performance by Continuously Adapting
the Internal Storage Layout of Heap Objects at Run-Time

THOMAS KISTLER and MICHAEL FRANZ

University of California, Irvine

We present and evaluate a simple, yet efficient dynamic optimization technique that increases memory-hierarchy performance for pointer-centric applications by up to 24% and reduces cache misses by up to 35%. Based on temporal profiling information, our algorithm reorders individual data members in dynamically allocated objects to increase spatial locality. Our optimization is applicable to all type-safe programming languages that completely abstract from physical storage layout; examples of such languages are Java and Oberon.

In our implementation, the optimization is fully automatic and operates at run-time on live data structures, guided by dynamic profiling data. Whenever the results of profiling suggest that a running program could benefit from data-member reordering, optimized versions of the affected procedures are constructed on-the-fly in the background. As soon as it is safe to do so, the dynamically generated code is substituted in place of the previously executing version and all affected live data objects are simultaneously transformed to the new storage layout. The program then continues its execution using the improved data arrangement, until profiling again suggests that re-optimization would be beneficial. Hence, storage layouts in our system are continuously adapted to reflect current access profiles.

Our results indicate that it is often worthwhile to re-optimize an already executing and optimized program all over again when the user's behavior changes. The main beneficiaries of such re-optimizations are shared libraries, which at different times can be optimized in the context of the currently dominant client application. In our experiments, we optimized a system library in the context of four different usage patterns and then correlated each of these specialized libraries across all four of the usage patterns. In some contexts, the specialized library performed 13% better than libraries optimized for another access pattern, and also 7% better than the original library that wasn't optimized for any particular access pattern. Hence, in systems where such re-optimizations can be executed rapidly, it becomes worthwhile to construct specialized versions at run-time.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*; D.3.4 [**Programming Languages**]: Processors—*Run-Time Environments*

General Terms: Memory-Hierarchy Optimization, Dynamic Re-Optimization, Dynamic Data Structures

1. INTRODUCTION

As the growth in raw processing power continues to outpace improvements in the storage hierarchy, memory performance is increasingly becoming a limiting factor of application speed. In recent years, compilers have begun to address this issue. For example, techniques have been developed to mask memory latency by fetching data

Authors' address: Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92697–3425. Part of this work is funded by the National Science Foundation under grant CCR–97014000.

ahead of time [Mowry et al. 1992], and program transformations such as cache-blocking, loop-skewing, and loop-tiling have been invented to increase data locality [Wolf and Lam 1991]. All of these optimizations are particularly effective in the domain of scientific computing, in which programs operate extensively on arrays. Unfortunately, they fare considerably worse in application domains in which most data structures are dynamically allocated and accessed via pointers. Applications of the latter kind include object-oriented and component-based programs.

In this paper, we present an optimization technique that increases memory performance specifically for pointer-centric applications. Our optimization is based on determining the best internal storage layout for dynamically allocated data structures. It applies to programming languages that are fully type-safe, such as Java [Gosling et al. 1996] and Oberon [Wirth 1988]. These languages do not attach a semantic meaning to the declaration order of data members and do not expose the actual physical layout to the programmer; as a consequence, choosing an internal layout lies completely in the domain of the compiler.

Our technique strives to *maximize spatial locality* of individual data members and hence is markedly different from traditional data-layout strategies that attempt to minimize the total space requirements of compound data structures [Muchnick 1997]. The traditional strategy is based on the assumption that a smaller memory footprint leads to faster applications, especially in garbage-collected environments. However, our work suggests that this assumption is misleading. In some cases, increasing an object’s size leads to a greater flexibility in placing data members, and thereby facilitates better cache performance. Our algorithm also specifically addresses the fact that there is a preferred bank-access ordering that needs to be observed to obtain optimum performance from interleaved memory.

The algorithm we present in this paper automatically selects an ordering for the data members of each pointer-based data structure. Our technique does not involve programmers in the optimization process, but leaves them free to declare data members in any order whatsoever. It thereby elegantly de-couples software-engineering concerns from performance issues.

Our algorithm is based on a simple strategy that first partitions the individual data members of a dynamically allocated data structure into aggregates that each fit into a single cache line. In this process, data members whose individual accesses are close together in time are assigned to the same cache line to maximize data locality. Then, after partitioning, the data members that have already been mapped to a single cache line are ordered to minimize load latency in case of a cache miss. During this ordering, a distinction is made between fields that are less likely to cause cache misses and those that are more likely to do so; the latter are placed at addresses mapped to the preferred bank of interleaved memory. The entire optimization process is guided by temporal profiling information that captures which paths through the program are taken with what frequency.

In the following, we will use the terminology of the Java programming language and refer to data members as “fields”. The remainder of this paper is organized as follows: Section 2 through Section 4 discuss different aspects of the algorithm. Section 5 describes our particular implementation context. Section 6 applies our technique to several benchmarks. Section 7 discusses related work and Section 8 concludes the paper.

2. AUTOMATED FIELD LAYOUT

Our primary objective is to increase data locality by identifying fields that are frequently accessed within a certain time interval and co-locating them on the same cache line. A secondary objective is to place individual fields on a single cache line in a spatial order that minimizes the performance penalty to be paid in the case of a cache miss. Two specific hardware characteristics of modern memory subsystems cause the ordering of fields on a line to be relevant, namely *memory interleaving* and *cache line-fill buffer forwarding*.

Interleaving has an influence because it partitions the memory into banks that cannot be accessed equally efficiently. Modern memory controllers deliver data from a single row of memory in bursts and use a fixed sequence in which they distribute column addresses to the memory banks (two such banks in our example). There is a preferred memory bank that always receives the first column address cycle. If the read starts with a column address that is mapped to a different bank, then this first cycle is wasted. Hence, in order to achieve optimum performance, fields that have a high probability of causing a cache miss should come to lie at addresses that are mapped to the preferred memory bank.

The second reason why the ordering of fields on a cache line has an influence on performance is related to the way that the cache is filled from memory. On most processors, the words on a cache line do not become simultaneously available after a cache miss has been serviced from memory. Rather, the cache line is filled in ascending memory address order, starting at the location that caused the cache miss, and “wrapping around” at the end of the cache line to load the remaining words. For example, consider a system in which the data bus is one word wide and a cache line holds eight such words. Now imagine that a read from address 003 causes a cache miss, resulting in a cache line being filled with the data stored in locations 000 through 007. The cache line would actually be filled in the order 003, 004, 005, 006, 007, 000, 001, 002; i.e., it would take at least seven additional cycles from the time at which the contents of location 003 become available until the contents of 002 become available also. On processors such as the PowerPC 604e [Motorola Inc. 1996] that forward the contents of the cache line-fill buffer to a requesting load unit immediately upon availability, it can hence make a difference whether the predominant memory access pattern is 003 followed by 002, or vice versa.

For our algorithm, we assume that memory-access instructions are executed in-order, even on processors that generally provide out-of-order execution. This seems reasonable for current architectures. For example, if two load instructions are both ready to execute, then the processor will issue them in order. Our optimization applies only to subsequent loads of *different* fields of the *same* object. Hence, presumably, the same base register would be used for addressing both fields, albeit with different offsets. Consequently, the situation will never arise that an earlier load needs to wait because its address calculation is not yet complete while a later load for the same object can proceed. Note that this underlying assumption may no longer be true if load value prediction [Gabbay 1996; Lipasti et al. 1996; Burtcher and Zorn 1999] becomes commonplace at some point in the future. We also assume that heap objects are aligned to a cache-line boundary. This is not unreasonably

difficult to accomplish and a reality in many existing systems.

In order to determine how to best partition the fields of an object into cache-line sized aggregates and how then to further order the individual fields within partitions, a cost model is required. An optimal cost model would reflect the execution time penalty caused by a particular arrangement of fields. Unfortunately, such an optimal model is very difficult to find in practice. We therefore fall back to a simplified model that estimates the number of cache misses under the assumption that the number of cache misses is proportional to an execution time penalty.

We compute this cost model based on a *temporal relationship graph* (TRG) that, for a particular data object, captures information on how its fields are accessed. Similar graphs have been used in a variety of contexts. In their work on cache-conscious data structures, Chilimbi et al. [Chilimbi et al. 1999] use an “affinity graph” to record field accesses within a certain interval, but without recording the order in which these accesses occur. To aid procedure placement in the context of instruction cache optimization, Gloy et al. [Gloy et al. 1997] use a directed graph that also captures this finer-grained information. Our approach is more similar to the latter than the former, in that we consider each pair of accesses individually rather than grouping them into clusters that are considered simultaneous. In our model, vertices correspond to fields, and they 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 within a specific time interval. The weight is roughly proportional to the benefit of co-locating both fields on the same cache line, as this increases the probability that one field will already be in the cache as a result of accessing the other¹.

The TRG is created by collecting path profiling information and then stepping through each program path returned by the profiler. A stack-like auxiliary data structure of the most recently accessed fields is maintained to simulate the effects of the cache. For every load or 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. The stack is finite; when it reaches its limit, the oldest (bottommost) element is removed to create room at the top. Next, we traverse the stack top-down and search for pairs (O, F') that reference the identical object O 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' (F does not necessarily have to directly follow F').

Unfortunately, the two main objectives of our algorithm—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 under observation and controls the recording of relationships: older objects are displaced from the stack due to capacity constraints. For partitioning, an optimal algorithm records relationships as long as both fields reside in the data cache. Intuitively, to accurately

¹We assume a write-allocate, write-through cache in which a write to a memory location not in the cache will bring the corresponding block into the cache. A write to a location already in the cache will “write through” without stalling the processor.

model the capacity constraints of the data cache, the optimal stack size should be proportional to the size 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., eight memory cycles in the example above). This is because accessing an additional field within an already loaded cache line comes at no extra cost.

The second dissimilarity is the *edge increment* for two related fields on the stack. For partitioning, an increment of one is sufficient. Accessing two fields located in distinct cache lines requires loading both cache lines, independently of the number of cycles 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, i.e., 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 incur a larger execution penalty than accessing them within three cycles.

So how can these different requirements be integrated into one basic model? Maintaining two different TRGs and two stacks with different sizes would be a naive solution. A more refined solution maintains two separate weights per edge: one for partitioning the graph and one for ordering fields within partitions. The weights for graph partitioning are computed as described above, 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. In the field ordering algorithm, we consider only 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.

3. GRAPH PARTITIONING

The first phase of our optimization partitions the fields of a dynamic data structure into aggregates 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, \dots, P_k , where $k = \lceil \text{objectsize} / \text{cachelinesize} \rceil$, $|P_i| \leq \text{cachelinesize}$, 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 equally-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 [Kernighan and Lin 1970; Dutt 1993] that can be implemented 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 equally-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 and in P_2 , respectively, 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 $(v_{1,1}, v_{2,1}), \dots, (v_{1,i}, v_{2,i}), \dots, (v_{1,m}, v_{2,m})$. For every i , $1 \leq i \leq m$, we compute the associated gain G_i for swapping vertex pairs

1... i . We choose p such that gain G_p is maximal and exchange all vertex pairs $(v_{1,1}, v_{2,1}), \dots, (v_{1,p}, v_{2,p})$ 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 dependent on the choice of the initial partition, we repeat this process for different randomly created initial partitions.

Finding an optimal k -way partition for large graphs is an NP-complete problem. As such, there exists no known 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 [Kernighan and Lin 1970]. Several refinements of this algorithm have been described since, among them the improved version by Dutt [Dutt 1993], which our own implementation is based on. There also exist more advanced algorithms based on multilevel partitioning schemes [Karypis and Kumar 1999]. However, since our graphs are usually small in size, the use of multilevel-partitioning algorithms does not seem justified.

The graph requires some minor adjustments prior to bisection. First of all, in order to end up with k equally-sized partitions that each have the size of a cache line, the original graph has to be expanded to the nearest $cachelinesize \cdot 2^k$. We do this by inserting additional *fill vertices* into the graph. The fill vertices are connected to all other vertices in the graph with edges of weight zero.

Secondly, individual fields have different sizes, which prevents the algorithm from swapping any two arbitrary vertices in the graph. In order to maintain equally-sized partitions, the algorithm is allowed to swap only field combinations that leave the size unchanged. Since we might also want to swap a field of type `float` with four fields of type `byte`, we divide 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 a division of the cluster into different partitions but makes it possible to swap vertices arbitrarily.

Finally, the concept of *structural type inheritance* needs special attention. The internal field layout of a derived type is constrained by the layout of its supertype. However, in order to minimize memory consumption, some of the fields of a derived type may be placed into the last partially used cache line of the supertype, rather than placing them onto 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 supertype ($size(leader) = size(supertype) \bmod cachelinesize$). The partitioning algorithm always places the leader at the beginning of the first cache line. Further, the leader vertex acts as a placeholder for the constrained inherited field layout of the supertype.

We have experimented with two different variations of our partitioning algorithm that differ in how the leader vertex is connected to other vertices in the graph. In the first variant that we call *latency-conscious*, all edge weights between the leader and other vertices are set to zero. This reduces the likelihood that any field of a derived type is placed into the same partition as the leader and thereby potentially wastes storage. However, it can increase cache locality if the temporal relationship among the added fields of the derived type is greater than that between the added fields and the inherited fields. In the second variant that we call *memory-conscious*, the leader is treated as a normal vertex in the computation of edge weights. Consequently,

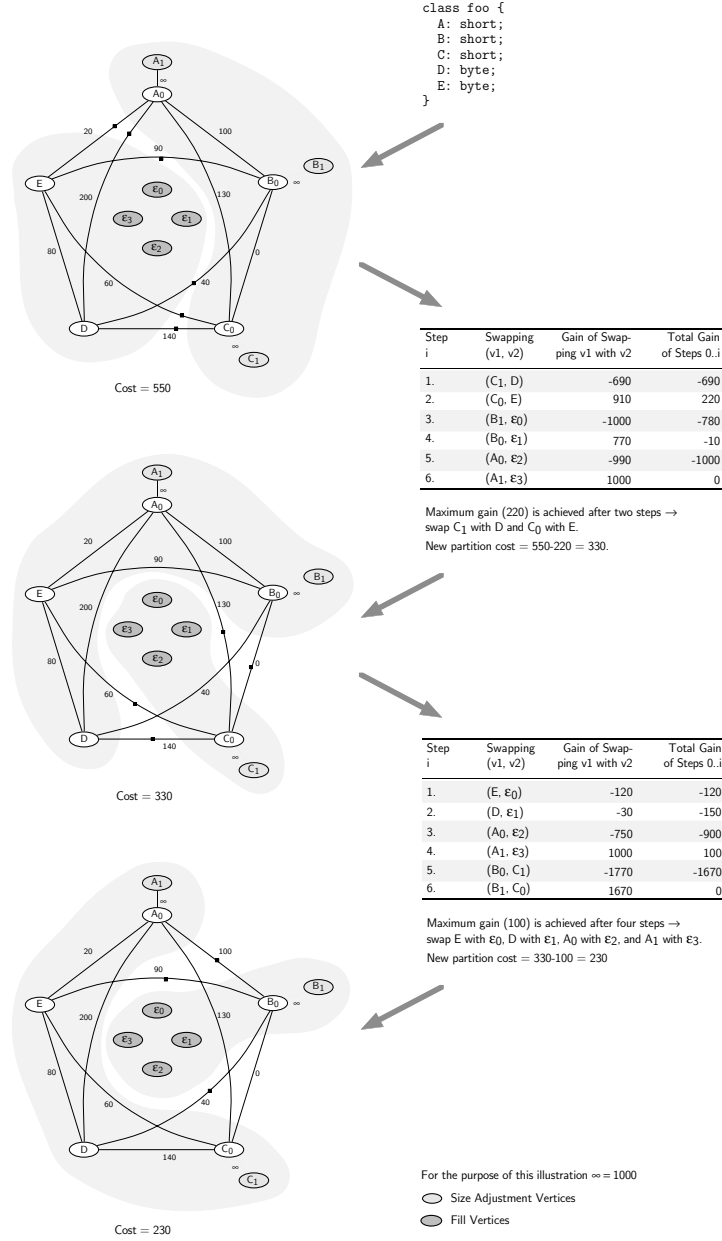


Fig. 1. Graph Partitioning: A small data structure comprising of three **shorts** (2 bytes) and two **bytes** (1 byte) is partitioned for an illustrative cache line size of six bytes. To make the total number of bytes a multiple of the cache line size, four bytes of padding are added (Fill Vertices $\epsilon_0 \dots \epsilon_3$). The algorithm iterates until no further gain can be found. For a graph with N vertices, each iteration comprises of N/2 steps. In each step, we apply the vertex swap that achieves the maximum gain; the corresponding two vertices are then not considered again for subsequent swaps. After executing all N/2 steps (i.e., swapping each vertex exactly once), we choose the step configuration that resulted in the largest overall gain.

fields are more likely to be placed into the same cache line as the leader, which reduces storage consumption.

Our results indicate that applications with a large load/store ratio are likely to perform better with latency-conscious partitioning, whereas applications with a small such ratio usually do better with memory-conscious partitioning. A large load/store ratio suggests an “allocate once, reference often” type of program, for which the benefit of reducing latency usually outweighs the advantage of reducing memory consumption. In contrast, for applications that are characterized by a smaller load/store ratio and that hence intersperse allocation and use more evenly, reducing the memory footprint is more important, especially in a garbage-collected environment. In our experience, the ratio of load instructions to store instructions is a good classification instrument for choosing one of the two partitioning variants.

4. FIELD ORDERING

The second major component of our algorithm arranges fields within cache lines to optimize access patterns for memory architectures offering interleaving and cache line-fill buffer forwarding. Each cache line is subdivided into partitions equal to the data bus width. We optimize for an architecture in which the assignment of partitions to two memory banks alternates between a single preferred and a single non-preferred bank, with the beginning of each cache line being mapped to the preferred memory bank. Further characteristics of our target hardware are that it is able to forward the contents of the cache line-fill buffer to a requesting load unit immediately upon availability, and that it does not re-order load instructions that are mapped to the same cache line.

Our 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 given below:

$$\begin{aligned}
 C(P) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n w_{i,j} (mic_{i,j} + flbfc_{i,j}) + w_{j,i} (mic_{j,i} + flbfc_{j,i}) \\
 mic_{i,j} &= oddbankpenalty ((F_i.adr \div bankwidth) \bmod 2) \\
 mic_{j,i} &= oddbankpenalty ((F_j.adr \div bankwidth) \bmod 2) \\
 flbfc_{i,j} &= (F_i.adr \div buswidth) - (F_j.adr \div buswidth) \\
 flbfc_{j,i} &= (linesize \div buswidth - flbfc_{i,j}) \bmod (linesize \div buswidth)
 \end{aligned}$$

In this formula, $w_{i,j}$ represents the weight of the edge between fields F_i and F_j in the TRG. $F_i.adr$ denotes the offset of field F_i relative to the beginning of its cache line, $mic_{i,j}$ represents the penalty for accessing the second non-preferred memory bank first, and $flbfc_{i,j}$ represents the number of intervening cycles between the availability of field F_i and that of field F_j in the case of a cache miss. In the example given on page 3, if i represents address 003 and j represents address 004, then $flbfc_{i,j}$ corresponds to one cycle and $flbfc_{j,i}$ to seven cycles.

Although not illustrated in the formula, we also require fields to be aligned properly. As an example, a double precision floating-point value that is not aligned to

an 8-byte boundary results in a high cost value. Our algorithm uses an exhaustive search technique that has exponential complexity. Nevertheless, runtime is not a major problem in practice because the number of fields in a cache line is fairly small. Moreover, we use a smart branch-and-bound variant of the algorithm that is an order of magnitude more efficient than a naive implementation.

5. IMPLEMENTATION CONTEXT

We have implemented a version of the described algorithm and integrated it into our dynamic code generation infrastructure [Kistler 1997; Franz 1997; Kistler and Franz 1999]. This infrastructure consists of a system that continually profiles all executing code, dynamically generates globally optimized versions of the same code in the background and then hot-swaps the optimized code image in place of the previously executing one. The architecture of this system is modular, allowing it to be readily extended to support the data layout optimization described here.

Although we anticipate that future generations of out-of-order processors will eventually provide hardware-profiling support for dynamic-optimization systems [Dean et al. 1997], our current implementation uses local path profiling based on dynamically inserted instrumentation code. Our choice of path profiling leads to a comparatively small run-time overhead, but comes at the expense of only being able to look at each procedure in isolation. Consequently, we cannot capture relations between subsequent field accesses occurring in different procedures. The larger the stack, the more such relations we miss because of the increased likelihood of field accesses from different procedures being on the stack simultaneously. At first sight, this is a serious disadvantage. Intuition would suggest that a larger stack captures more interdependencies and is therefore better suited for our algorithm. Luckily, however, our results indicate the contrary. When experimenting with different heuristics (including capturing interprocedural field accesses), using a smaller stack actually yielded better performance. It appears that path profiling is capable of capturing most of the essential relations; hence it seems to be a good solution for recording temporal relationships.

Further, as has been documented elsewhere [Kistler and Franz 1998; 1999], our dynamic re-optimizer doesn't simply haphazardly keep recompiling the whole system. Instead, it uses a history of past profiling data (which periodically undergoes an "aging" function) to constantly monitor for changes in the system's behavior. When a significant change in behavior is observed, the system attempts to re-generate those parts of the system that actually warrant re-optimization (on a procedure-by-procedure basis). In the case of certain optimizations, such as the data-member layout optimization described here, it is not possible to predict inexpensively whether a specific change will yield a significant speed-up or not—a good prediction heuristic would require almost as much work as doing the actual optimization itself. In these cases, optimization decisions are made based on static program analysis and other available information.

With regard to the specific optimization described here, we compute a coarse-grained estimate of the cache-miss penalty from the available profiling data and current storage layout. The heuristic used to compute this estimate is pessimistic. When the estimated cache-miss penalty becomes large and the optimizer determines that the storage layout of some data type T could be improved by data-member

re-ordering, it automatically generates new versions of all the affected procedures. A procedure is affected if its code contains at least one modified field-offset as a literal. Hence, a procedure can be affected by a re-ordering of T 's fields only if it references at least one of the fields declared directly in T —fields inherited from supertypes of T or added in subtypes of T need not be considered because their offsets do not change if only T is optimized.

Precisely at the same time that the new versions of the affected procedures are swapped into the executable state, all existing data structures that include instances of T need to be translated to the new storage layout. The process of translating live data structures into a new format has many similarities with garbage collection and is in fact implemented as an extension to our garbage collector. Structure translation must be precise; as a consequence, detailed pointer maps need to be available for all the points in the program at which such a “hot-swap” would be possible. Our current implementation limits the substitution of live code to certain predefined system states; this is unproblematic in our particular implementation since our system is centered around a main “event loop” in which such a state is entered frequently enough. For a more general implementation, recent work by Stichnoth *et al.* [Stichnoth et al. 1999] reports the encouraging finding that the overhead of keeping detailed pointer maps for every instruction in the program may be far smaller than previously assumed.

Our system supports the dynamic loading of additional program modules at runtime, including modules that make use of, or declare extensions of, data types whose fields have been re-ordered dynamically. Consequently, the existing state of all re-ordering optimizations needs to be applied to each of these modules upon loading. Since no data transformations need to be applied, dynamically adding a module is actually simpler than replacing procedures of an already loaded one.

6. RESULTS

Data layout optimizations are intended to provide performance gains on applications with poor data locality and large working sets. As others have noted [Truong et al. 1998], benchmarks such as *SpecInt95* do not exhibit this behavior. Consequently, to test our results, we have used a nonstandard suite of programs, all of which make extensive use of dynamically allocated data structures. Our benchmark suite includes our optimizing compiler as an example of a non-trivial program. This optimizing compiler (for the programming language Oberon [Wirth 1988] and the PowerPC target architecture) allocates large dynamic data structures that model the program being compiled in a variant of static single assignment form [Brandis 1995; Cytron et al. 1991]. For the compiler benchmark, we only give “ideal” speedup results. Our optimizer processes code that is wholly type-safe and portable. The compiler, on the other hand, uses several constructs that are not portable and makes use of explicit type-casts. With some modifications, we were able to obtain valid results for optimal storage layout, but at present, the compiler cannot be continuously re-optimized.

The other benchmarks are *TreeAdd* and *Bisort* from the Olden benchmark suite [Rogers et al. 1995], *Jigsaw* from the WPI benchmark suite [Finkel et al. 1992], and *BTrees* and *Texts* from the Oberon System 3 [Wirth and Gutknecht 1992; Gutknecht 1994; Gutknecht and Franz 1999]. These benchmarks represent a variety of applications and programming styles but have in common that each of them

Benchmark	Program Size (Lines of Code)	Number of Al- located Objects	Number of Al- located Bytes	Average Object Size (bytes)
TreeAdd	512	200,001	6,400,032	32
Bisort	229	262,143	8,388,576	32
Jigsaw	317	22,501	1,800,032	80
BTrees	1,343	5,788	463,040	80
Texts	961	18,640	1,192,800	64
Compiler	23,621	1,173,369	66,512,064	57

Table I. Benchmark Characteristics

allocates many megabytes of data, executes billions of instructions, and represents frequently used operations on dynamic data structures². Table I summarizes some important characteristics of our benchmark programs. Of particular interest in the following will be *Texts*, which is a fundamental shared library of the Oberon System and is simultaneously accessed by virtually every program running as part of any Oberon session. As can be expected, the different clients of this shared library use the various services offered by the library in different ways. An interesting consequence that will be explored below is that the optimal internal structure for the library's data types depends on the predominant client currently using it.

Our benchmark results were obtained by executing the programs in question multiple times on a PowerPC 604e (32Kbyte, four-way set-associative first-level data cache, 1Mbyte second-level cache, both caches are 32 bytes wide), utilizing the PowerPC's performance monitor. The performance monitor includes four 32-bit hardware counters that record detailed events during execution, such as instruction dispatches, instruction cycles, misses in the cache, and load/store miss latencies. In order to obtain realistic performance data that is applicable to hardware configurations commonplace today, the benchmarks were executed on a machine with 64 megabytes of physical memory, in contrast to many published results based on more uncommon hardware configurations. Hence, our performance data implicitly include the cost of garbage collection, which in our system is implemented as mark-and-sweep collector [Wirth and Gutknecht 1992]. The Oberon system has a document-centric architecture and runs completely in a single address space, rather than one address space per application.

Our first set of benchmarks (Table II and Figure 2) present an idealistic situation in which we compare a program with no data-layout optimizations (field ordering corresponds to programmer's declarations) with the same program after automated data-member reordering, without taking into account the cost of optimization and profiling. A more realistic picture that accounts for these costs is then shown in Figure 3. In our system, the very first instance of a program to be executed in a user session is always created by a fast code-generating loader, from a machine-

²Although our system and these benchmarks are based on Oberon, we are not aware of any particular characteristics of the Oberon language that inhibit a simple mapping into Java. We expect that these results will apply to Java without restrictions. For obvious reasons, they cannot be replicated in languages in which programmers make explicit decisions about storage layouts and then use type-cast operations based on those decisions.

Benchmark	Execution Time		Speedup
	Original Layout	Optimized Layout	
	(Mio Cycles)	(Mio Cycles)	
TreeAdd	2,612	2,273	1.15
BTrees	8,582	4,381	1.96
Bisort	1,701	1,642	1.04
Jigsaw	1,954	1,508	1.30
Texts	2,504	1,479	1.69
Compiler	4,663	4,506	1.03

Table II. Ideal Performance Speedup

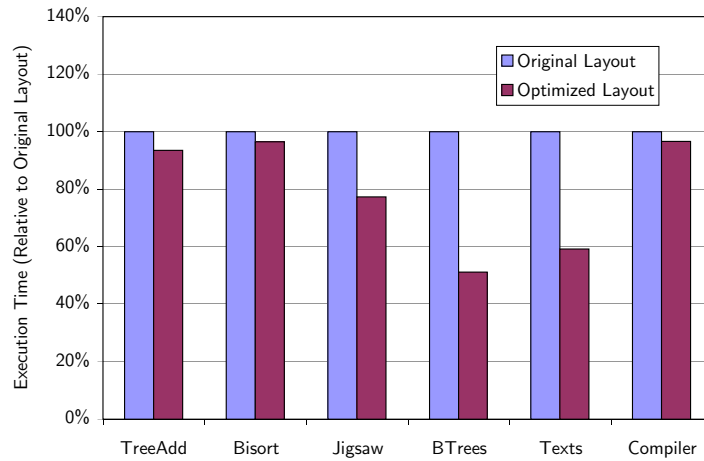


Fig. 2. Ideal Performance Speedup

independent software distribution format³. Because this is an interactive process (a user is waiting for the program to start), and because an optimized version of the same program will be available soon afterwards, the code generating loader performs few optimizations. We call the resulting base-line executable the *unoptimized* code.

When the unoptimized code has started executing, the background re-optimizer commences its task. It performs a series of static optimizations on critical procedures and instruments them with path profiling code. The performance of the resulting executable is presented in Figure 3 under the heading *original layout with path profiling*. Note that in the majority of benchmarks, performance is lowered

³Our particular implementation uses the Slim Binary representation [Franz and Kistler 1997], but our solution does not depend on this fact. Our system could be adapted to use programs represented as class files for the Java Virtual Machine, or even native code for some specific processor. This would merely have an effect on the pre-processing effort required to extract information relevant to the optimizer, such as control flow and data flow information.

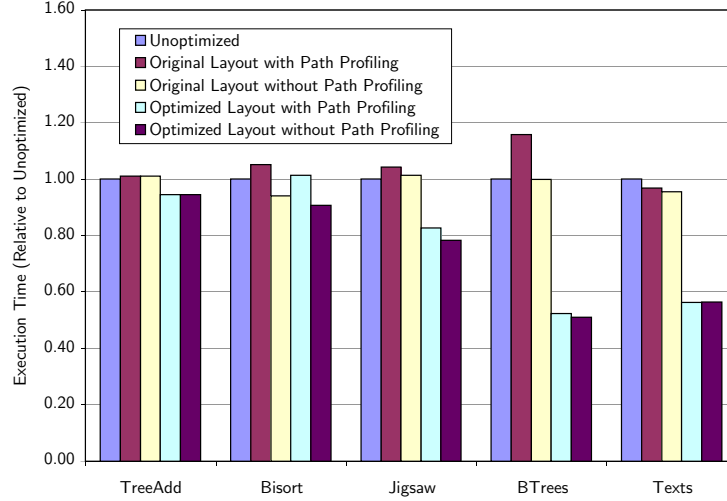


Fig. 3. Profiling Costs: At load-time, the code-generating loader creates a first unoptimized version of the application. At runtime, the background optimizer performs basic code optimizations and instruments applications with path profiling—the original data layout is retained. If profiling information suggests that a different memory layout might increase performance, the storage layout of live data objects is modified.

due to the cost of profiling, which in these cases cannot be offset by static optimizations. In order to show the cost of the instrumentation, we also present the performance that this first optimized code image would have if there were no path profiling code. In actuality, the system never removes the profiling code.

After a while, the system has gathered enough path profiling information to be able to optimize the layouts of critical data structures. The resulting performance is presented as *optimized layout with path profiling*. Again, these figures include the overhead of profiling (but not the overhead of re-optimization itself), which is why we add separate numbers for timings without this overhead. In actual use, the profiling instrumentation is not removed because optimization is continuous: the system will keep monitoring the profiling data for apparent changes in system behavior, and if such a behavior is observed, will re-optimize the affected procedures. Note that this doesn’t invalidate the original goals of optimization: as can be seen in Figure 3, even with the profiling code left in, the optimized program is still faster than the optimized initial program. Table III breaks down the various costs associated with optimization by the different phases in the optimization cycle at which they occur.

So is dynamic optimization of data layouts worth the considerable effort? In order to answer this question, we first need to study how the “break-even” point is reached in a system such as ours—that is, if it is ever reached at all. As Figure 4 illustrates, the benefit of re-optimization is not simply the ratio between the resulting speedup and the combined overheads of profiling and code re-generation. This is because the speedup itself is achieved only after the re-optimization phase has completed:

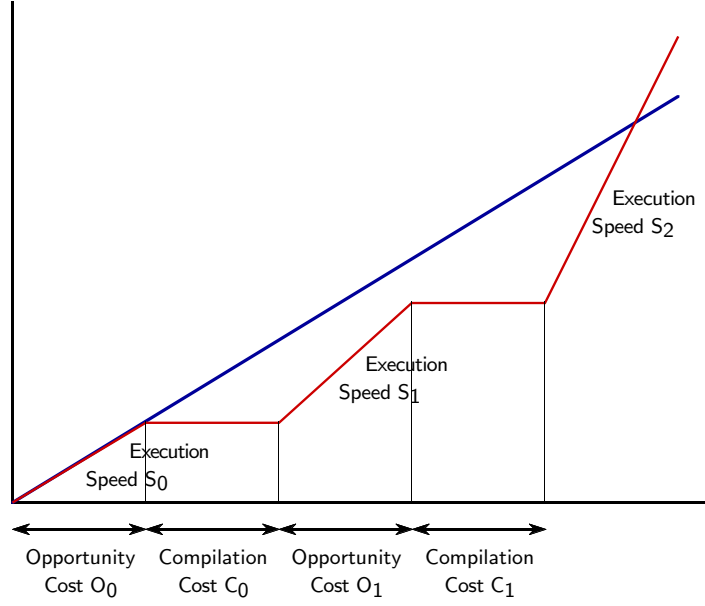


Fig. 4. Computing the break-even point

if the dynamic optimization completed halfway through execution, then only half of its potential benefit could be realized. As shown in Figure 4, the first part of this cost is related to the fact that profile information isn’t immediately available; we cannot circumvent having to execute the unoptimized version of the program for a while first to detect the hot spots in the program (O_0). This period of time is commonly referred to as “opportunity cost”. Once hot spots are detected, there is a further price to pay for re-generating the code and inserting additional path profiling instrumentation for these hot spots (C_0). Again, we have to run the new version of the program for a while until path profiling information becomes available (O_1). Finally, we have to pay the optimization costs for creating the temporal relationship graph, computing the new memory layout, changing the layout of all live objects, and generating code for the new memory layout (C_1). Hopefully, i.e. if the program’s overall runtime is sufficiently long, this cost is eventually recouped because the resulting program is significantly faster than the original.

Also note that the cost for the first optimization cycle is higher than the cost for subsequent optimization cycles. Since profiling instrumentation is never actually removed, subsequent optimization cycles do not have to pay the price for finding hot spots and inserting path profiling instrumentation any more (O_0 and C_0). In addition, the opportunity cost O_1 can partially be overlaid with the time it takes for the previous optimization cycle to pay off.

Hence, the “break-even” point of such a multi-phase optimization process can be represented by the following generic formula:

$$\text{break-even point} = \frac{S_n \sum_{i=0}^{n-1} (C_i + O_i) - \sum_{i=0}^{n-1} O_i S_i}{S_n - 1}$$

In the above formula, O_i denotes the opportunity cost at phase i (i.e., the time required to detect hot spots in the program), C_i denotes the optimization costs at phase i (i.e., the time required to optimize a hot spot), and S_i denotes the execution time ratio of the unoptimized unprofiled program over the current version of the program at phase i .

Based on this formula, Table IV attempts to answer the question how long optimized code needs to run so that the investment of optimization is fully recouped. For example, assuming that the system requires one minute to collect enough profiling information before it can start the optimization, optimizing the storage layout of the Oberon shared text subsystem pays off after invoking text services for a total of 3.3 minutes⁴.

This brings us to the second important insight that our work has yielded, namely that continuous, rather than do-it-once, optimization yields an added benefit. This is because a single piece of code is often put to several quite distinct uses over the course of a single user session lasting several hours, while at each point in time the user’s attention is usually focused on a relatively small number of current tasks. As an example, in the Oberon system, the text library is shared by the Web browser, by the program editor, and by the compiler. Clearly, the text editor that is used to create new text documents and to modify existing ones places very different demands on the texts library than the compiler, which only reads text documents, and does so strictly sequentially. The Web browser uses the text system in a different manner yet again, as it defines “rich” data types (such as pictures and hyperlinks) that extend the data types inherited from basic texts, and which must remain backward compatible with the basic services.

This raises the question how a library optimized for one particular client performs when it is used with another one. To this effect, we performed a series of

⁴These break-even results assume that re-optimization occurs in the background on the same processor that is executing user programs. As one of the authors has argued elsewhere [Franz 1997], this scenario may actually turn out to be unnecessarily pessimistic. It may very well be cost-effective to delegate the task of re-optimization to an additional processor, so that the CPU running the user’s application is not burdened by the re-optimization task. The reasoning is as follows: users are willing to pay a considerable premium for cutting-edge performance; a processor that is only half as fast as the current performance leader usually costs less than half as much (witness the cost of an Intel Celeron vs. a Pentium III). Hence, it may be worthwhile to add a cheap (slow) co-processor to a computer system whose sole task is to continuously fine-tune the instruction stream executed by the much more expensive main processor on which the user’s tasks are running.

An alternative argument in favor of disregarding re-optimization costs is that many future computer systems will be multiprocessors. More often than not, some of the processors will be sitting idle because the executing application is not fully parallelizable. These idle processors could be employed for re-optimization.

Benchmark	Phase I	Phase II	Phase III	Phase IV	Phase V	Total
TreeAdd	0.12	0.39	3.67	3.47	9.03	16.56
Bisort	0.10	1.33	1.27	2.80	1.72	7.12
Jigsaw	0.08	3.22	3.25	3.02	3.25	12.74
BTrees	0.86	19.05	19.49	12.47	18.91	69.92
Texts	0.60	5.42	5.85	16.02	8.37	35.66

Table III. Optimization Costs (in seconds): At load-time, Phase I generates the first “quick-and-dirty” native version of the application; At runtime, Phase 2 applies standard optimizations (conditional constant propagation, value numbering, loop invariant code motion, strength reduction, local instruction scheduling, hierarchical register allocation, and dead code elimination) to the application and inserts instrumentation utilized later by the memory optimization; Phase III reads the collected path profiling data and creates the TRG graph; Phase IV computes the new memory layout and changes the layout of all live objects; Phase V generates code for the new memory layout.

Break-even point (in seconds) for various opportunity costs			
	$O_i=60s$	$O_i=120s$	$O_i=180s$
TreeAdd	430	561	692
Bisort	∞	∞	∞
Jigsaw	205	336	468
BTrees	276	404	533
Texts	198	316	433

Table IV. Break-even Point: Illustrates the time required for the optimization to pay off. If the unoptimized program version ran longer than the break-even point, performing the data layout technique first and then running the optimized program version would perform better.

Data Layout Optimized For		Benchmark			
		Publishing Application	Web-Browser	Text Editor	Compiler
Programmer's Layout	PP	2,540	3,066	1,852	1,606
	NP	2,504	3,078	1,866	1,610
Publishing Application	PP	1,474	1,704	1,122	977
	NP	1,479	1,681	1,146	976
Web-Browser	PP	1,661	1,666	1,210	978
	NP	1,667	1,705	1,212	980
Text Editor	PP	1,466	1,669	1,131	984
	NP	1,473	1,684	1,143	990
Compiler	PP	1,682	1,684	1,223	982
	NP	1,668	1,679	1,220	985

Table V. Optimizing the *Texts* benchmark for different predominant access patterns (all numbers represent execution time in 10^6 cycles). The rows labeled *PP* include the overhead of path profiling instrumentation, whereas the rows labeled *NP* do not contain profiling overhead.

experiments that are summarized in Table V. We took an “original” data-structure layout in which the fields were arranged strictly in the order specified by the programmer in the source text with four layouts that were automatically obtained by our optimizer for four different uses of the *Texts* library, and correlated their performance across these four different usage scenarios. The rows labeled *PP* include the overhead of path profiling instrumentation, whereas the rows labeled *NP* do not include any profiling overhead. In order to simplify the comparison, the cost of re-generating the code itself has been disregarded. This is because this cost varies greatly depending on the order in which the layout of different types is modified. The cost of the first optimization cycle differs from that of subsequent ones, because the first cycle additionally needs to insert profiling code whereas subsequent cycles do not.

As can be seen in Figure 5, there is clearly a difference in optimizing for different text services. For example, the publishing application is 13% faster using a *Texts* library that is custom-tailored for it rather than a library tailored for the compiler or the Web browser. Similarly, the text editor is 7% faster with its custom-tailored version of the library versus the compiler’s custom tailored version. These results also confirm the expected result that dynamic compilation is superior to static compilation, because it can adapt to multiple behavior patterns instead of just a single one.

The reason why custom tailoring yields an additional benefit in this particular case is that the text service supports not only plain sequences of characters, but also enriched documents that have elements such as images, buttons, and hyperlinks embedded within them. The fields that support these additional “floating text elements” are accessed relatively frequently when dealing with Web pages and using the publishing application. But source programs rarely contain any embedded elements, and hence the program editor and the compiler access the corresponding fields more infrequently than other clients of the text service. This results in two

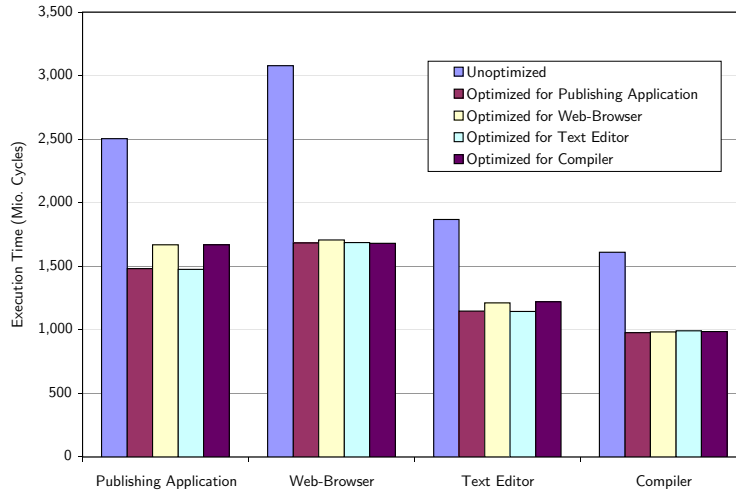


Fig. 5. Optimizing the *Texts* benchmark for different predominant access patterns.

very different usage scenarios, one in which the corresponding fields are placed with other frequently accessed ones on the same cache line, and another in which they are “demoted” in favor of other data members.

A similar case is shown in Table VI and Figure 6. This benchmark implements a binary tree that also links all the nodes in the tree in a linear list. Such data structures are often used in compilers for implementing symbol tables—the tree is used to rapidly look up an identifier, whereas the linked list is needed to preserve the declaration order of procedure parameters and the like. The first column in Table VI depicts an application that accesses both the linked list and the binary tree with the same likelihood. In contrast, the second and third column depict applications that primarily access either the linked list (Pattern I) or the binary tree (Pattern II). Again, for simplicity reasons, the cost of re-generating the code is not included.

These results show that optimizing for different behaviors is well worth the effort. If the data type is optimized for access to the binary tree, the benchmark that primarily accesses the list runs 11% slower than if it were optimized for accessing the list. Similarly, the benchmark that primarily accesses the tree runs 7% slower if it is optimized towards the list rather than the tree.

7. RELATED WORK

Cache optimizations aim to reduce the gap between memory and processor speeds. For example, data locality can be increased in scientific, array-based programs by applying techniques such as loop-reversal, loop-tiling, loop-skewing, and cache-blocking [Wolf and Lam 1991]. They change algorithmic behavior by reordering the execution sequence of iterations and by changing the shape of a loop’s iteration space and iteration depth. Rivera and Tseng’s algorithm [Rivera and Tseng 1998] inserts inter-variable and intra-variable padding to control the placement of

Data Layout Optimized For		Benchmark		
		Mixed Access Pattern	Primary Access Pattern I	Primary Access Pattern II
Programmer's Layout	PP	1,486	1,236	1,705
	NP	1,485	1,236	1,704
Mixed Access Pattern	PP	1,486	1,236	1,705
	NP	1,485	1,236	1,704
Access Pattern I	PP	1,486	1,236	1,705
	NP	1,485	1,236	1,704
Access Pattern II	PP	1,487	1,371	1,593
	NP	1,487	1,371	1,593

Table VI. Optimizing the *TreeInsert* benchmark for different predominant access patterns (all numbers represent execution time in 10^6 cycles). The rows labeled *PP* include the overhead of path profiling instrumentation, whereas the rows labeled *NP* do not contain profiling overhead.

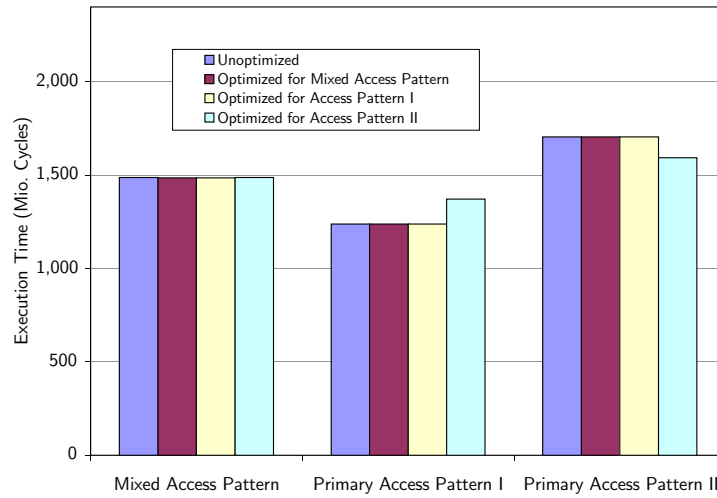


Fig. 6. Optimizing the *TreeInsert* benchmark for different predominant access patterns.

arrays in memory and to control the optimal row size of arrays. This technique can be applied orthogonally to control-transforming optimizations. Previous work has also studied the problem of data-prefetching in the context of array-based programs [Mowry et al. 1992]. Prefetching reduces memory latency by loading data values ahead of time into the cache. This is particularly beneficial for array-based programs that exhibit highly regular data access patterns.

Because of fundamental differences in program structure, most of these techniques developed for scientific programming cannot be applied directly to pointer-based applications that often exhibit much more complex access patterns. Also, the overall size of dynamically allocated data structures can usually not be determined at compile time. It is therefore not surprising that work on data prefetching [Luk and Mowry 1996], automatic placement of data in memory [Calder et al. 1998; Chilimbi and Larus 1998], and automatic data transformations in the context of pointer-based applications is happening only recently. With the ever-increasing memory hunger of pointer-based programs, further factors are becoming relevant for data cache performance, such as the effects of memory allocators [Grunwald et al. 1993; Seidl and Zorn 1997; Gay and Aiken 1998] and garbage collectors [Reinhold 1994] in modern operating systems.

Truong *et al.* [Truong et al. 1998] have lately proposed a field reorganization technique similar in spirit to ours. Quoting from their paper:

... the automatic detection of the most frequently used fields of a structure is beyond the possibility of current compiler technology. Therefore, data layout must be done by the programmer ...

Our work solves exactly this problem by fully automating the process. Unfortunately, Truong's paper simultaneously introduces another optimization and a new storage allocator and does not report separate speedup results for each of the three proposed techniques; hence our results are not immediately comparable with theirs.

More recently, Chilimbi *et al.* [Chilimbi et al. 1999] describe a cache-conscious memory reorganization technique that has many similarities with ours: fields are rearranged within dynamically allocated objects to assign temporally related fields into the same cache-line. Their approach differs in that they do not record the finer-grained ordering in which fields are accessed. Instead, they consider time intervals of 100 milliseconds and regard accesses to fields that occur in the same interval as being "simultaneous". As a consequence, they are unable to perform field ordering on a cache line. Their program transformation is also not fully automatic but requires programmer intervention, and as a consequence, cannot provide continuous re-optimization. It achieves speedups of 2–3% for an already-hand-optimized, I/O bound C program with external layout constraints (Microsoft SQL server 7.0).

In another thrust of their work, Chilimbi *et al.* [Chilimbi and Larus 1998] have also studied reordering the objects themselves in memory during a garbage collection cycle. This technique is entirely complimentary to the work described in this paper. Combinations of the two techniques are likely to bring benefits particularly in large garbage-collected environments.

Continuous optimization has also been studied outside the scope of memory reorganization. There have been several systems providing incremental ("staged") specialization of an already executing program at run-time [Engler et al. 1996;

Lee and Leone 1996; Marlet et al. 1999; Grant et al. 1999], based on manual annotation of the source program by a skilled programmer. In these approaches, a static compiler constructs a dedicated run-time code-generator that is able to dynamically create variants of the program to be executed, specialized depending on actual input data. In contrast to our background re-optimization engine, which is a full-fledged optimizing compiler, these dedicated code generators are much simpler and operate on the complexity level of macro expansion. They also have the problematic property that each possible optimization is enumerated explicitly for each potential optimization site. This requires a careful consideration of the trade-off between the size of the dedicated run-time code generator created for a particular application, and the speedup that can possibly be achieved by using it.

A variety of techniques have been proposed to speed up execution (and message dispatch in particular) of systems based on the “pure” object-oriented languages Smalltalk and SELF. The resulting dynamic compilation systems [Ungar and Smith 1987; Chambers et al. 1989; Hölzle et al. 1991; Chambers 1992; Hölzle 1994; Hölzle and Ungar 1996] provide incremental code generation, but once a piece of code has been optimized, it becomes static. Hence, no re-optimization is performed in response to changes in user behavior. One of the central findings of this paper, however, is that it is worthwhile to re-optimize an already executing program all over again when the users behavior changes.

8. CONCLUSION

In this paper, we have proposed a technique for improving the storage layout of dynamically allocated data structures. The technique can be implemented as an orthogonal addition to commonly used optimization techniques such as data-prefetching and optimal data placement. Our algorithm is based on a two-tiered strategy that first assigns fields to cache lines and then optimizes the order of fields within individual cache lines.

Our algorithm is an early representative of an emerging class of code optimizations that are applicable to programs that are already executing. As our results show quite conclusively, re-optimizing an already running program in response to changes in user behavior can give rise to real performance improvements.

ACKNOWLEDGEMENTS

We would like to thank Martin Burtscher, Peter Fröhlich, and Christian Stork who provided many helpful comments on an earlier version of this paper.

REFERENCES

- BRANDIS, M. 1995. *Optimizing Compilers for Structured Programming Languages*. Ph. D. thesis, Institut für Computersysteme, ETH Zürich.
- BURTSCHER, M. AND ZORN, B. 1999. *Exploring Last n Value Prediction*. Technical Report CU-CS-885-99 (April), University of Colorado at Boulder.
- CALDER, B., KRINTZ, C., JOHN, S., AND AUSTIN, T. 1998. Cache-Conscious Data Placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, pp. 129–149.
- CHAMBERS, C. 1992. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph. D. thesis, Stanford University.

- CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Volume 24, New York, NY, pp. 49–70. ACM Press.
- CHILIMBI, T. M., DAVIDSON, B., AND LARUS, J. R. 1999. Cache-Conscious Structure Definition. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI-99)*, Volume 34, 5 of *ACM SIGPLAN Notices*, New York, pp. 13–24. ACM Press.
- CHILIMBI, T. M. AND LARUS, J. R. 1998. Using Generational Garbage Collection To Implement Cache-Conscious Data Placement. In *Proceedings of the First International Symposium on Memory Management*, Vancouver, pp. 37–48. ACM Press.
- CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct.), 451–490.
- DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. 1997. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, Los Alamitos, pp. 292–302. IEEE Computer Society.
- DUTT, S. 1993. New Faster Kernighan-Lin-Type "Graph-Partitioning Algorithms". In *Proceedings of the IEEE/ACM International Conference on CAD*.
- ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 1996. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, pp. 131–144.
- FINKEL, D., KINICKI, R., LEHMANN, J., AND CARADONNA, J. 1992. *Comparisons Of Distributed Operating System Performance Using The Wpi Benchmark Suite*. Technical Report CS-TR-92-2, Worcester Polytechnic Institute.
- FRANZ, M. 1997. Run-Time Code Generation as a Central System Service. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pp. 112–117. IEEE Computer Society Press.
- FRANZ, M. AND KISTLER, T. 1997. Slim Binaries. *Communications of the ACM* 40, 12 (Dec.), 87–94. Also published as Technical Report TR 96–24, Department of Information and Computer Science, University of California, Irvine, June 1996.
- GABBAY, F. 1996. *Speculative Execution Based on Value Prediction*. Technical Report 1080 (Nov.), EE Department, Technion—Israel Institute of Technology.
- GAY, D. AND AIKEN, A. 1998. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, pp. 313–323.
- GLOY, N., BLACKWELL, T., SMITH, M. D., AND CALDER, B. 1997. Procedure Placement Using Temporal Ordering Information. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, Los Alamitos, pp. 303–313. IEEE Computer Society.
- GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison Wesley.
- GRANT, B., PHILIPSE, M., MOCK, M., CHAMBERS, C., AND EGGERS, S. J. 1999. An Evaluation of Staged Run-Time Optimization in DyC. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI-99)*, Volume 34, 5 of *ACM SIGPLAN Notices*, New York, pp. 293–304. ACM Press.
- GRUNWALD, D., ZORN, B. G., AND HENDERSON, R. 1993. Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico, pp. 177–186.
- GUTKNECHT, J. 1994. Oberon System 3: Vision of a Future Software Technology. *Software—Concepts and Tools* 15, 1, 26–33.
- GUTKNECHT, J. AND FRANZ, M. 1999. Oberon With Gadgets: A Simple Component Framework. *Object-Oriented Application Frameworks* 2.

- HÖLZLE, U. 1994. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. Ph. D. thesis, Department of Computer Science, Stanford University.
- HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, Springer Lecture Notes in Computer Science 512, Geneva, Switzerland, pp. 21–38.
- HÖLZLE, U. AND UNGAR, D. 1996. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Transactions on Programming Languages and Systems* 18, 4 (July), 355–400.
- KARYPIS, G. AND KUMAR, V. 1999. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (Jan.), 359–392.
- KERNIGHAN, B. W. AND LIN, S. 1970. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 291–307.
- KISTLER, T. 1997. Dynamic Runtime Optimization. In *Proceedings of the Joint Modular Languages Conference, JMLC'97*, Springer Lecture Notes in Computer Science 1204, Linz, Austria, pp. 53–66. Also published as Technical Report TR 96–54, Department of Information and Computer Science, University of California, Irvine, November 1996.
- KISTLER, T. AND FRANZ, M. 1998. *Computing the Similarity of Profiling Data—Heuristics for Guiding Adaptive Compilation*. Technical Report TR 98–30 (Sept.), Department of Information and Computer Science, University of California, Irvine.
- KISTLER, T. AND FRANZ, M. 1999. *Perpetual Adaptation of Software to Hardware: An Extensible Architecture for Providing Code Optimization as a Central System Service*. Technical Report TR 99–12 (March), Department of Information and Computer Science, University of California, Irvine.
- LEE, P. AND LEONE, M. 1996. Optimizing ML with Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, pp. 137–148.
- LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. 1996. Value Locality and Load Value Prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, pp. 138–147.
- LUK, C.-K. AND MOWRY, T. C. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, pp. 222–233. ACM Press.
- MARLET, R., CONSEL, C., AND BOINOT, P. 1999. Efficient Incremental Run-Time Specialization for Free. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI-99)*, Volume 34, 5 of *ACM SIGPLAN Notices*, New York, pp. 281–292. ACM Press.
- MOTOROLA INC. 1996. *PowerPC: Addendum to PowerPC 604 RISC Microprocessor User's Manual: PowerPC 604e Microprocessor Supplement and User's Manual Errata*.
- MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, pp. 62–73. ACM Press.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufman.
- REINHOLD, M. B. 1994. Cache Performance of Garbage-Collected Programs. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Florida, pp. 206–217.
- RIVERA, G. AND TSENG, C.-W. 1998. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, pp. 38–49.
- ROGERS, A. M., CARLISLE, M. C., REPPY, J. H., AND HENDREN, L. J. 1995. Supporting Dynamic Data Structures on Distributed-memory Machines. *ACM Transactions on Programming Languages and Systems* 17, 2 (March), 233–263.

- SEIDL, M. L. AND ZORN, B. 1997. *Predicting References to Dynamically Allocated Objects*. Technical Report CU-CS-826-97 (Jan.), Department of Computer Science, University of Colorado, Boulder.
- STICHNOTH, J. M., LUEH, G.-Y., AND CIERNIAK, M. 1999. Support for Garbage Collection at Every Instruction in a Java Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-99)*, Volume 34, 5 of *ACM SIGPLAN Notices*, New York, pp. 118–127. ACM Press.
- TRUONG, D. N., BODIN, F., AND SEZNEC, A. 1998. Improving Cache Behavior of Dynamically Allocated Data Structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Paris, France, pp. 322–329.
- UNGAR, D. AND SMITH, R. B. 1987. SELF: The Power of Simplicity. In *Proceedings of OOPSLA'87*, Special Issue of ACM SIGPLAN Notices, 22(12):227–242, Orlando, Florida.
- WIRTH, N. 1988. The Programming Language Oberon. *Software Practice and Experience* 18, 7 (July), 671–690.
- WIRTH, N. AND GUTKNECHT, J. 1992. *Project Oberon*. Addison-Wesley.
- WOLF, M. E. AND LAM, M. S. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Ontario Canada, pp. 30–44.