An Efficient Profile-Analysis Framework for Data-Layout Optimizations

Shai Rubin
Computer Sciences Dept.
University of Wisconsin-Madison
Madison, WI 53705
shai@cs.wisc.edu

Rastislav Bodík
Computer Sciences Dept.
University of Wisconsin-Madison
Madison, WI 53705
bodik@cs.wisc.edu

Trishul Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98502
trishulc@microsoft.com

ABSTRACT

Data-layout optimizations rearrange fields within objects, objects within objects, and objects within the heap, with the goal of increasing spatial locality. While the importance of data-layout optimizations has been growing, their deployment has been limited, partly because they lack a unifying framework. We propose a parameterizable framework for data-layout optimization of general-purpose applications. Acknowledging that finding an optimal layout is not only NP-hard, but also poorly approximable, our framework finds a good layout by searching the space of possible layouts, with the help of profile feedback. The search process iteratively prototypes candidate data layouts, evaluating them by "simulating" the program on a representative trace of memory accesses. To make the search process practical, we develop space-reduction heuristics and optimize the expensive simulation via memoization. Equipped with this iterative approach, we can synthesize layouts that outperform existing non-iterative heuristics, tune application-specific memory allocators, as well as compose multiple data-layout optimizations.

1. INTRODUCTION

The goal of memory optimizations is to improve the effectiveness of the memory hierarchy [18]. The memory hierarchy, typically composed of caches, virtual memory, and the translation-lookaside buffer (TLB), reduces the memory access time by exploiting the execution's locality of reference. The opportunity for the optimizer is to help the memory hierarchy by enhancing the program's inherent locality, either temporal or spatial, or both.

To alter the temporal locality, one must modify the actual algorithm of the program, which has proved possible for (stylized) scientific code, where transformations such as loop tiling and loop interchange can significantly increase both temporal and spatial locality [4, 14, 26]. However, when the code is too complex to be transformed—a situation common in programs that manipulate pointer-based data structures—one must resort to transforming the layout of data structures, improving spatial locality. Many data-layout optimizations have been proposed [3, 4, 7, 8, 9, 11, 15, 16, 21, 23, 24], with their specific goals ranging from reordering structure fields [7] to object inlining [11].

POPL '02, Jan. 16-18, 2002 Portland, OR USA
 ISBN 1-58113-450-9/02/01...$5.00

Data-layout optimizations synthesize a layout with good spatial locality generally by (i) attempting to place contemporaneously accessed memory locations in physical proximity (i.e., in the same cache block or main-memory page), while (ii) ensuring that frequently accessed memory cells do not evict each other from caches. It turns out that these goals make the problem of finding a "good" layout not only intractable but also poorly approximable [20]. The key practical implication of this hardness result is that it may be difficult to develop data-layout heuristics that are both robust and effective (i.e., able to optimize a broad spectrum of programs consistently well).

The hardness of the data-layout problem is also reflected in the lack of tools that static program analysis offers to a data-layout optimizer. First, there appear to be no static models for predicting the dynamic memory behavior of general-purpose programs that are both accurate and scalable (as was shown possible for scientific code [5]), although initial successes have been achieved for small C programs [1, 13]. Second, while significant progress has been made in deducing shapes of pointer-based data structures [22], in order to create a good layout, one is likely to have to understand also the temporal nature of accesses to these shapes. This problem appears beyond current static analyzers (although a combination of [1] and [22] may produce the desired analysis power).

The insufficiency of static analysis information has been recognized by existing data-layout optimizations, which are all either profile-guided or exploit programmer-supplied application knowledge [3, 7, 8, 9, 11, 15, 16, 21, 23, 24]. Although many of these techniques manage to sidestep the problem of analyzing the program's memory behavior by observing it at run-time, they are still fundamentally limited by the hard problem of synthesizing a good layout for the observed behavior [20]. Typically based on greedy profile-guided heuristics, these techniques provide no guarantees of effectiveness and robustness. Indeed, our experiment show that sometimes their optimization is far from optimal (not effective), and that some programs are actually impaired (not robust).

In this paper, we propose a framework that enables effective and robust data-layout optimizations. Our framework finds a good data layout by searching the space of possible layouts, using profile feedback to guide the search process. A naive approach to profile-guided search may transform the program to produce the candidate data layout, then recompile and rerun it. Clearly, such a search cycle is uninformative and too slow (or cumbersome) to be practical.

Indeed, to avoid this tedious cycle our framework evaluates a candidate layout by *simulating* its fault rate (e.g., cache-miss rate or

page-fault rate) on a representative trace of memory accesses. Simulation is more informative than rerunning since it allows us not only to measure the resulting miss rates of the candidate data layouts, but also to identify the objects that are responsible for the poor memory performance. We use this information to narrow the layout search space by (a) determining which levels of memory hierarchy need to be optimized, (b) selecting optimizations (and thus layouts) that may influence the affected levels of memory hierarchy, and finally by (c) excluding infrequently accessed objects from consideration.

Trace-based fault-rate simulation is not only more informative but it is also faster than editing, recompiling and rerunning the program. To simulate the program without re-compilation, we "pretend" the program was optimized by remapping the original addresses of data objects to reflect the candidate data layout. To make the iterative search even more practical, we speed up the simulation by compressing the trace as a context-free grammar, which in turn allows us to develop algorithms for memoized simulation.

This paper makes the following contributions:

- We present a framework for data-layout optimization of general-purpose programs that permits composing multiple optimizations. Unifying existing profile-based data-layout optimizations, the framework operates by first heuristically selecting a space of profitable layouts, and then iteratively searching for a good layout, using profile-guided feedback. (Section 3).
- We develop techniques for efficiently evaluating a candidate layout optimization using a trace of memory references. These techniques, based on memoization, miss-based trace compaction, and on-demand simulation, make the framework's iterative data layout search efficient enough to be practical (Section 4).
- Using the framework, we re-implement two existing data-layout optimizations: Field Reordering and Custom Memory Allocation. The experimental results show that our iterative-search versions outperform the existing single-pass heuristic optimizations (Section 5).

2. RELATED WORK

In this section we describe previous work on profile-driven data-layout optimization. First, we discuss general concepts that apply to all optimizations. Next, we illustrate the optimization steps common to all, and finally, we summarize existing optimizations and highlight their unique features.

2.1 Common Optimization Concepts

Data-layout optimization aims to improve a program's *memory performance*. Usually, memory performance is characterized in terms of the *fault rates* a program incurs on various *memory resources.* Memory resources are hardware or software elements that are part of the memory sub-system: all levels of the processor cache, the translation lookaside buffer (TLB), and the virtual-memory page table [18, 19]. The fault rate is the fraction of references not found in the cache/TLB/page table.

To reduce fault rate, data-layout optimizations attempt to find a good *data layout* for a program's *data objects*. Data objects are the optimization's "building blocks". For example, structure fields represent the data objects for Field Reordering and Class Splitting optimizations [7], while structure instances are data objects for Linearization and Custom Memory Allocation [8, 21, 23]. A data layout is a placement of data objects in memory, namely it is a map from data objects to their memory locations. A *good* memory layout is one that improves a program's memory performance.

Finding an optimal memory layout is intractable [20]. Indeed, existing data-layout optimizations use heuristic methods to produce a good memory layout. The heuristics are usually based on two simple objectives. First, optimizations such as Class Splitting [7], Coloring [8], and Custom Memory Allocation [23] attempt to prevent having highly referenced data being evicted from the cache by infrequently accessed data. Second, optimizations such as Field Reordering [7, 15, 24], Linearization [8, 9, 16, 21], Global Variable Positioning [3], and Custom Memory Allocation try to increase the implicit prefetch that cache blocks provide, by packing contemporaneously accessed data in the same cache block.

Most optimizations *target* only specific types of data objects. For example, Global Variable Positioning targets global variables, Field Reordering targets structures that are larger than the cache block size, and Custom Memory Allocation targets heap objects that are smaller than the virtual-memory page size.

Most optimizations use profile information to guide the heuristic layout techniques. A *profile* is a map from a set of data objects to a set of attributes. For example, the Field Pair-wise Affinity profile maps a pair of fields to the number of pre-defined intervals in which they are both referenced. A field reordering heuristic can then use this pair-wise profile to place fields according to their temporal affinity [7, 15].

2.1.1 Common Optimizations Steps

The first optimization step is to *identify* the memory resources that limit program performance, which we call *bottlenecks.* This step is usually done manually, using profiling tools such as Intel Vtune [25] that measures a program's memory performance (e.g., its cache miss rate). The next step is to select, out of many available optimizations, one that can potentially improve the bottleneck's performance. Selecting a "good" optimization requires a deep understanding of the program memory behavior, and is typically done by a programmer familiar with the program. Then, the selected optimizer is invoked and the automatic phase of the optimization begins.

The optimizer first identifies the data objects it intends to target. For example, the target objects can be the structure fields that should be reordered, the classes that should be split, or the global variables that should be repositioned in memory. Next, it chooses a heuristic method to construct a new data layout for the target objects. The choice of heuristic is often tied to the profile information used to drive the optimization. For example, the Fields Access Frequency profile can be used to identify the "splitting point" between highly referenced and rarely accessed member fields of a class.

Finally, the program must be modified to produce the new data layout. This step is often done manually. The program is re-written to generate the new layout, then re-compiled and re-executed to measure the performance benefit. In most cases, the initial benefit is small and the optimization must be tuned further. This *tuning* often requires using a different profile and/or a different heuristic to produce the optimized layout. In the worst, though not uncommon, case the new layout may not yield the expected results despite best tuning efforts, and the entire process must be repeated with a different optimization.

Table 1. data-layout optimizations

Memory Resources and Data Objects Targeted	Profile Information Used	Heuristic Objective and Methods
Field Reordering [7, 15, 24]. Increases cache block utilization. Targets structures that have more than two fields and are larger than the cache block size.	Two profiles are commonly used: 1. Fields Pair-wise Affinity: $f_1 \times f_2 \rightarrow count$. Maps pairs of fields to their temporal affinity. The fields temporal affinity is the number of intervals in which they were both referenced, for a given length of a time interval. 2. Fields Access Frequency: $f_1 \rightarrow count$. Maps a single structure field to the number of its references.	Objective: place fields that are concurrently accessed in the same cache block. The heuristic uses the profile to build the "affinity graph" of fields: nodes are fields, arcs represent the affinity between them. Based on graph clustering algorithms [12], the optimization clusters fields with high temporal affinity into the same cache block.
Class Splitting [7, 24]. Increases cache utilization. Targets data objects whose collection of hot fields are smaller than the cache block size.		Objective: Avoid bringing rarely accessed data into the cache. The heuristic uses the Frequency profile to split classes into "hot" and "cold" parts. Hence, when accessing the "hot" part, the "cold" part is not brought into the cache, leaving more space for other "hot" data.
Global Variable Positioning [3]. Reduces cache conflicts among global variables. Targets globals.		Similar to the Field Reordering optimization.
Coloring [8]. Reduces cache conflicts. Usually targets dynamically allocated objects.	The two profiles mentioned above, but at an object granularity. Alternatively, an implicit profile based on knowledge of data structure topology.	Objective: to prevent frequently referenced data being evicted from the cache by rarely accessed data. Using the Frequency profile, or programmer knowledge, the heuristic distributes the objects among the cache sets, so that frequently accessed objects do not conflict with rarely accessed objects.
Linearization, Clustering [7, 8, 11, 16, 21]. Increases cache-block utilization. Targets recursive data structures with elements smaller than the cache block size.		Objective: to increase the locality of reference among structure nodes (e.g., link list nodes). The heuristic usually exploits the topology of data structure nodes to re-position them in memory. For example, the optimization clusters two linked list nodes that are linked via the 'next' pointer into the same cache block.
Custom Memory Allocation [23]. Increases virtual-memory utilization.	Heap Object Lifetime profile: $x \rightarrow l$. Where x is a heap object, l is an attribute describing its lifetime behavior. Previous work used four attributes: (i) highly referenced, (ii) rarely referenced, (iii) short lived, and (iv) other.	Objective: to increase heap object reference locality by allocating objects with the same lifetime behavior within the same memory region. Two heuristic methods are used. First, the programmer heuristically defines the "correct" lifetime attributes to use (e.g., one for "hot" and one for "cold" objects). Second, the optimizer heuristically partitions the objects into memory regions, according to their lifetime behavior (see Section 5.1 for more details).

2.2 Existing Data-Layout Optimizations

To the best of our knowledge, all previous work on (implicit) profile-driven data-layout optimization [3, 7, 8, 11, 15, 16, 21, 23, 24] follows the process described above. Three features distinguish the optimizations: the memory resources targeted, the heuristic method used to construct a new layout, and the profile information used to guide the heuristics. Table 1 summarizes these features for existing data-layout optimizations.

However, prior work does not directly address several key issues such as systematic identification of bottlenecks, optimization selection, profile and heuristic selection, and, perhaps most importantly, tuning the optimization to the program being optimized. Our data-layout optimization framework, described in the next sections, uses profile feedback and an iterative search process to address these issues.

3. NEW DATA-LAYOUT OPTIMIZATION PROCESS

We now present the overall structure of our data-layout optimization process. First, we give a general description of the process, then describe each step in detail.

3.1 Process Overview

Figure 1 shows the overall structure of the process. The goal of the process is to find a data layout that improves the program's memory performance. Since efficiently computing even an approximation of an optimal layout is intractable [20], our process is based on a search in the space of data layouts. We define this space, informally, to be all possible layouts that can result from applying any data-layout optimization in any order. Because the layout space is too large and an exhaustive search is usually not practical, we propose two techniques to increase the search feasibility: analyses to narrow the search space, and the possibility to use a hill climbing to guide the search into more "promising" areas.

The goal of the first three steps is to narrow the search space. In the *Bottleneck Identification* step the process identifies the memory resources (e.g., cache, TLB) that limit the program memory performance. Then, for each resource bottleneck the process finds the *data objects* that mostly influence its memory behavior by performing the *Data-Objects Analysis* step. Last, it selects optimizations that affect the memory behavior of these "critical" objects (*Select Optimizations* step). By selecting these optimizations, the process effectively narrows the search space only to layouts that have the potential to improve the bottleneck's memory behavior.

The *Build Profiles* step computes profile information for the selected optimizations. Next, a search to find better layouts begins (*Apply Optimizations* and *Evaluation* steps): Starting with the original program layout, the process may evaluate a large number of candidate layouts, incrementally improving the best layout it has found so far. The *Apply Optimizations* step provides the next candidate data layout, by applying (possibly simultaneously) the optimizations chosen in the *Select Optimization* step. The *Evaluate* step evaluates the candidate layout and provides feedback that can guide a hill-climbing search.

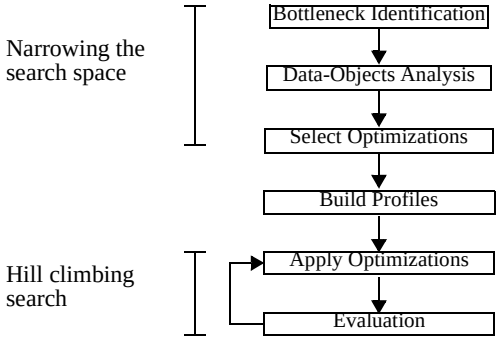

Figure 1. Overall Process Structure

3.2 Process Steps

We now describe each process step in detail. All steps use the *data-objects trace* which is the sequence of data objects (and their addresses) referenced by the program during execution. Each candidate optimization must provide the following inputs: the memory resources the optimization targets, a *Build Profile* function that builds the profile(s) the optimization requires, an *Optimization Predicate* which specifies the data objects the optimization targets, and a *Next Layout* function that iterates over different layouts the optimization may produce. The formal definitions of these inputs, together with other definitions, are found in Table 3; the formal process is described in Table 4.

1. ***Bottleneck Identification***. This step identifies the memory resources that limit program performance. The process defines a memory resource as a *bottleneck* if its miss rate is higher than a pre-defined threshold (e.g., L1 miss rate higher than 2%). To calculate the miss rate for a specific memory resource, the process uses the *Simulate* function (see Table 4) that simulates the resource memory behavior on the *data-objects trace*.
2. ***Data-Objects Analysis***. For each bottleneck, the *Data-Objects Analysis* identifies critical data objects. Critical objects are those objects that are responsible for "most" of the faults. Finding the critical objects helps narrow the search space to layouts that affect the placement of these objects.

The *Bottleneck Critical Objects* set contains two type of objects: (i) The *faulting objects* that frequently miss in the given resource. These objects are critical since changing their layout can reduce the resource's memory faults. (ii) The *hot objects* are the frequently referenced data objects. These objects are critical because introducing too many new memory faults in them will degrade performance.

To find the bottleneck critical objects, the process uses the *Bottleneck Critical Objects* function (*BCO* in Table 4). This function traverses the *data-objects trace* and uses two parameters, *k* and *j*, to identify the frequently faulting and frequently referenced objects. Ideally, the function should return the smallest set of objects that covers at least *k%* of the misses and *j%* of the references. Since finding the smallest set is difficult, the process finds a larger set that has the desired coverage. The *BCO* function builds this set by adding the frequently referenced and frequently missed objects incrementally, until the set satisfies the coverage requirements. Our empirical results (Section 5.2) show that setting *k=j=*80% usually yields a small critical set (typically 10% of the total number of data objects) that covers both 90% of the references and 90% of the misses.

Table 2. Optimization Predicates. Return true iff the object is a suitable target for the optimization

Field Reordering [7, 15, 24]. (i) Structure size is larger than the cache block size, and (ii) the structure has more than two fields.
Class Splitting [7, 24]. (i) Some structure fields are accessed more frequently than others, and (ii) the set of frequently accessed structure fields is smaller than the cache block size.
Global Variable Positioning [3]. A global variable (data object) that is smaller than the cache block.
Linearization, Clustering [8, 9, 11, 21]. a node of a recursive data-structure (e.g., a node of a link list) that is smaller than the cache block size.
Coloring [8]. All data objects that incur cache conflicts.
Custom memory allocation [23]. A dynamically allocated data object that is smaller than the page size.

3. ***Select Optimization***. The goal of the *Select Optimization* step is to narrow the search to those optimizations (and their resulting layouts) that can potentially improve the bottleneck's memory performance.

Since critical data objects affect bottleneck performance, the process looks for optimizations that target these objects. To this end, the process builds, for each optimization, the *Optimization Target Objects* set: the set of critical objects that are also targets for the given optimization. Optimizations with a non-empty Target Objects set have the potential to improve program performance. To quantify the optimization potential, the process calculates the Target Objects set *coverage*, which is the fraction of the trace accounted for by references to objects in the set. Larger coverage implies greater optimization potential.

To build the Target Objects set, the process uses the *Optimization Target Objects* function (*OTO* function in Table 4). For a given bottleneck and optimization, this function places *Bottleneck Critical Objects* that satisfy the *optimization predicate* into the O*ptimization Target Objects* set. The optimization predicate determines if a data object is likely to benefit from the optimization. For example, field reordering benefits structures that are larger than the cache block size and have more than two fields. Table 2 lists the optimization predicates the process uses, which were selected based on previous work [3, 7, 8, 11, 15, 16, 21, 23, 24], and our experience with these data-layout optimizations.

From this point on, the search focuses only on optimizations with the potential to improve program performance.

4. ***Build Profile***. For each optimization selected, the process uses the *Build Profile* function (see Table 3) to build the profile(s) needed for that optimization. The profiles are needed for the next step (*Apply Optimization*).

At this point the process begins its search process.

5. ***Apply Optimizations***. In this step the process produces a new candidate data layout for the program data objects. It uses the *Layout Selector* function (*LS* in Table 4) to iterate over the space of possible layouts. Each time the function is called, it returns a valid data layout, or a *null* value. When the function returns a null, the process terminates. When the function returns a data layout, the process continues to the next step where the layout is evaluated and feedback is provided to guide the search (if desirable).

Table 3. Process Definitions

1. A *data object x* is an elementary piece of data, such as an object, a global variable of a basic type, a field in an instance of a class type, or an array. A *trace T* is a sequence of references to data objects. *O* is the set of all data objects in the trace *T*. The *coverage* of $S \subseteq O$ is the sub-trace length induced by *S* objects, divided by the length of *T*.
2. Let *M* be the set of all memory addresses (locations) available for the program. A *data layout* is a map $DL: O \rightarrow M$.
3. *Memory Resource* is a pair: *r=(SimulationFunction, Threshold)* where
   - *Simulation_Function:* $Trace \rightarrow double$. A function that returns the fault rate (e.g., cache miss rate on *T*) of *r* on trace *T*.
   - *Threshold* is a fault rate above which a resource is defined as a bottleneck.

   For a resource *r*, the *Resource Critical Objects Set*, $O_r \subseteq O$, contains data objects that are responsible for most of the misses in resource *r*. *R* is the set of all memory resources.
4. A *Profile* is a map $P: 2^O \rightarrow 2^{Profile-attributes}$ .The profile attributes used are optimization specific.
5. Optimization is a tuple *opt = (resources, ep, bp, nl)*.
   - $resources \subseteq R$ is a set of memory resources the optimization targets to improve.
   - *ep* is the *Optimization Predicate:* $ep: T \times O_r \rightarrow bool$ which holds if the optimization targets the given data objects. (The trace, *T* is an input since some predicates need to calculate the miss rate).
   - $O_{OPT_r} \subseteq O_r$ is the *Optimization Target Objects Set* for resource *r*. $O_{OPT_r} = \{ x \in O_r \mid ep(T,x) \}$.
   - *bp* is the *Build Profile* function: $bp: T \times O_{OPT_r} \rightarrow 2^{Profile}$. It returns a set of possible profiles needed to drive the optimization.
   - *nl* is the *Next Layout* function: $nl: 2^{Profile} \times T \times 2^O \times misc \rightarrow DL$. The function is used to iterate over possible layouts of the optimization. It returns the 'next' layout from the set of possible layouts for this optimization. misc. is miscellaneous information needed to create a legal layout (e.g., order of allocations of heap objects).

Table 4. The Framework data-layout optimization Process

<table>
<tr><td></td><td colspan="3">Process input: (i) Optimizations: a set of available optimizations (ii) Resources: a set of memory resources to optimize. (iii) T: the program memory reference trace. (iv) restrictions: information needed to build a legal data layout (e.g., heap objects allocation order). (v) k, j: coverage percentages for the BTO function. The defaults are k=j=80% and were determined by the experimental results (see Section 5.2).</td></tr>
<tr><td></td><td>Step</td><td>Procedural Description</td><td>Functions Used</td></tr>
<tr><td></td><td>Initialization</td><td>double benefit=0, best_benefit=0;
pr: an empty vector; // optimization profiles
CS: a map from resources to Critical Sets.
TS: a map from <optimization,resource> to Target Sets
SO: an empty set; // selected optimizations</td><td rowspan="6">Simulate(trace T, Resource r) → MR_R
The Simulate function returns the miss rate of resource r on trace T.
BCO(trace T, resource r, double k, double j) → CS_r
The Bottleneck Critical Objects function returns the critical data-objects set, $CS_r \subseteq O$, for the resource r. CS_r covers more than k% of T, and more than j% of memory faults incurred at resource r.
OTO(trace T, Optimization o, critical set CS_r) → O_{OPT_r}
The Optimization Target Objects (OTO) function calculates the Optimization Target Objects set. Using the optimization predicate, ep the function filters out only the critical objects that are targeted by the optimization.
LS(trace T, Optimization[] v, Profile[] p, Target Objects T, double benefit, restrictions m) → DL
The Layout Selector function uses the Next Layout function of the selected optimizations to return the next data layout the process should evaluate. It can use the benefit variable, which indicates the performance benefit of the last layout returned, to guide its decision about which data layout to return.
Evaluate(trace T, data-layout DL) → benefit
T_{new} is a trace defined by assigning to each data object, x, its address: DL(x). The function returns the performance benefit of the layout T_{new}.
The benefit is defined as $\sum_{r \in Resources} W_r(Simulate(T,r) - Simulate(T_{new},r))$.
W_r is the 'weight' of the resource r: the number of machine cycles it takes to access r.</td></tr>
<tr><td rowspan="4">Analysis Phases</td><td>Bottleneck Identification.</td><td rowspan="4">for each $r \in Resources$ do {
if (Simulate(T,r) > r.threshold
CS[r] = BCO(T, r, k, j);
}
for each $o \in Optimizations$ do {
for each resource $r \in o.resources$ do {
TS[o,r]=OTO(T, o, CS[o.r]);
if (| TS[o,r] | > 0) do
pr = pr + o.bp(T, TS[o,r]);
SO = SO + o;
}}}
while ((DL= LS(T, SO, pr, TS,
benefit, restrictions) != NULL) do {
benefit = Evaluate(T,DL);
if (benefit > best_benefit) do {
best_benefit = benefit;
BestDL = DL;
}}
return BestDL; // the selected layout</td></tr>
<tr><td>Data-Objects Analysis.</td></tr>
<tr><td rowspan="3">Optimization Phases</td><td>Select Optimization.</td></tr>
<tr><td>Build Profile.</td></tr>
<tr><td>Apply Optimizations</td></tr>
<tr><td></td><td>Evaluation</td></tr>
</table>

The goal of the *LS* function is to guide the search towards more "promising" layouts. Unfortunately, selecting good layouts is a difficult task, especially when applying several optimization simultaneously. For example, it is not known what is the best order in which to apply Field Reordering and Class Splitting together with a Custom Memory Allocation. Ideally, the *LS* function should synthesize a layout by combining several optimizations. However, we considered only one (described next) simple approach to produce new layouts.

The *Layout Selector* presented in the paper applies each optimization separately, starting with the optimization whose target set has the largest coverage. The *LS* function uses the optimization's *Next Layout* function (see Table 3), to return a new layout that has not yet been evaluated. If feasible, the optimization's *Next Layout* function can simply iterate over all possible layouts, as we did successfully for the Field Reordering optimization in Section 5.5.1. If such an approach is too time consuming, the *Next Layout* function can use heuristics to limit the search, as we did for the Custom Memory Allocation optimization in Section 5.5.2.

6. ***Evaluate Optimization***. To evaluate the optimization, the process uses the *Evaluate* function (see Table 4) that returns the performance benefit (i.e., the difference in memory resource fault rates) of the data layout obtained in the *Apply Optimizations* step.

Previous work on data-layout optimization [3, 7, 23] evaluates a layout using a tedious, manual, cycle of editing the program, recompiling it, and re-executing it. By contrast, in the process implementation presented in Section 4, the *Evaluate* step uses the *trace* to perform the evaluation in an extremely efficient, automatic way: The *Evaluate* function simulates the data-objects trace, but, instead of using the original object addresses, it uses the new addresses from the data layout being evaluated. This efficiency is crucial to making the iterative search feasible.

4. EFFICIENT FRAMEWORK FOR DATA-LAYOUT OPTIMIZATION

The previous section described a process for profile-based data-layout optimization at a fairly abstract level. This section presents a concrete instantiation of such a process and describes how it can be efficiently implemented in a general framework for data-layout optimizations. The section begins with the description of the framework components, continues with algorithms needed for efficient process implementation, and ends with quantitative measurements of the framework efficiency. The general structure of the framework is presented in Figure 2.

4.1 Major Framework Components

As described in Section 3, all process phases require profiling information; in most cases the data-objects trace. In this section we describe this trace in more detail, along with how the framework *compacts* it to facilitate process efficiency.

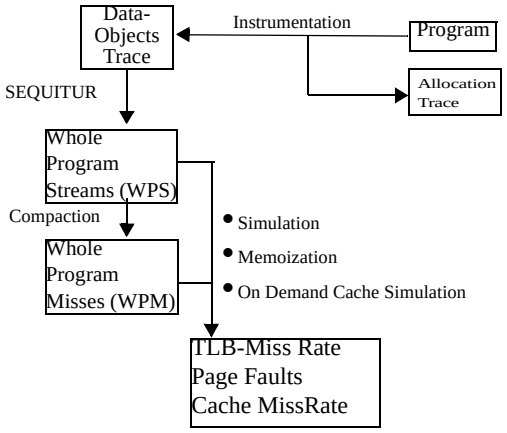

Figure 2. Framework for data-layout optimization

4.1.1 Data-Objects Trace

The *data-objects trace* is an extension of the program data-reference trace. Each entry in the trace contains not only the address that was referenced, but also a unique identifier (a symbolic name) of the referenced data object[1]. For example, assume a program references a global variable *c*, and two dynamically allocated instances, *a* and *b*, of the same structure type *foo*, where *foo* has two fields: *x* and *y*. Assume that the program references these objects in the following order <*a.x, b.y, b.x, c, a.x*>. Then, the data-objects trace is the following sequence of pairs: <$(A_1$,*a.x*), (A_2,*b.y*), (A_3,*b.x*), (A_4,*c*), (A_1,*a.x*)> where A_1, A_2, A_3, and A_4 are the addresses of the symbolic names *a.x*, *b.y*, *b.x*, and *c*, respectively.

The data-objects trace contains more information than the data-reference trace. For example, in the reference trace one cannot always distinguish between instances of dynamically allocated objects (or stack variables) since two objects can share the same address at two different points in the program execution. However, in the data-objects trace one can always distinguish between such cases.

This unique feature of the data-objects trace enables one of the framework's novel capabilities; it enables the framework to evaluate the future effects of a candidate data layout without program re-execution. As described in Table 3, a data layout is a map from data objects to memory locations. Hence, to evaluate the memory performance of a candidate data layout, the *Evaluate* function in Table 4 uses the data layout to assign to each data object its new memory location, then, it simulates the trace with the new addresses to measure the new memory behavior.

Evaluating a new layout without re-execution is not the only feature that enables an efficient search process. To increase efficiency further, the framework uses additional two methods. First, to enable fast trace traversal, it compresses the trace, so that the trace resides entirely in memory rather than on disk. Second, to enable fast memory simulation, the framework uses novel simulation techniques that exploit the internal structure of the compressed trace. The following sections describe the compressed trace representation and our efficient simulation techniques.

1. Unique object id's can be generated by techniques presented in [3, 23].

4.1.2 Compact Trace Representation

For most programs, the data-objects trace is very large, composed of about 100M memory accesses, or about a few GBytes of data [6]. Since the trace is used in all process phases, it is important to *compactly represent* it. Compact representation enables the framework to place the trace in physical memory so it can be easily analyzed (for example by the *Simulate* and the *Evaluation* functions).

Our compression scheme is based on Nevill-Manning's SEQUITUR algorithm, which represents the trace as a context-free grammar that derives a single string—the original trace [6]. In the SEQUITUR representation, the grammar terminals represent the program data objects, and non-terminals represent sub-traces of the original trace. Figure 3b illustrates how SEQUITUR compresses the trace in Figure 3a. The compression is possible because sub-traces 24, 25, and 2525 are repeated. These sub-traces are represented with non-terminals A, B, and C, respectively. The SEQUITUR context-free grammar is internally represented as a directed acyclic graph (DAG), as in Figure 3c. Internal nodes are non-terminals and leaf nodes are the terminals. Outgoing edges are ordered and connect a non-terminal with the symbols of its grammar production. The grammar's start symbol is the root of this DAG. Given the DAG representation, which is called Whole Program Streams (WPS), the original trace can be regenerated by traversing the DAG in depth-first order, visiting children of a node left to right, and repeatedly visiting each shared sub-DAG. Chilimbi describes this process in more detail [6].

4.2 Efficient Algorithms for Profile Analysis and Optimization Evaluation

Compact trace representation is the first method the framework uses to achieve the efficiency that facilitates the iterative data-layout search. The iterative process requires efficient methods to implement the *Build Profile*, *Evaluate*, and the *Bottleneck-Identification* functions (see Table 4). This section describes three techniques to obtain this vital efficiency.

4.2.1 Memoization-Based Profile Analysis

Our *memoization* technique exploits the SEQUITUR grammar structure to increase the computation efficiency of profile analysis. To illustrate the memoization technique, assume we want to calculate the length of the trace represented by the grammar in Figure 3. In a brute-force grammar traversal, each edge $X \rightarrow Y$ is traversed *paths(X)*[1] times (e.g., the edge $A \rightarrow 2$ is traversed 3 times). However, if we *memoized* the length of a rule after we traverse its sub-dag (e.g., the memoization value of the rule (sub-dag) $A \rightarrow 24$ is 2), then during the memoized-length computation each edge is traversed only once. Although the memoization speed-up depends on SEQUITUR representation's compression ratio (which is trace dependent), empirical results (Section 4.3) show that the benefits from memoization in terms of the computation time can be an order of magnitude.

For a more realistic example, we describe the memoization values needed for memoizing cache simulation of a fully-associative cache with a Least Recently Used (LRU) replacement policy.

1. The total number of paths from the start non-terminal *S* to *X*.

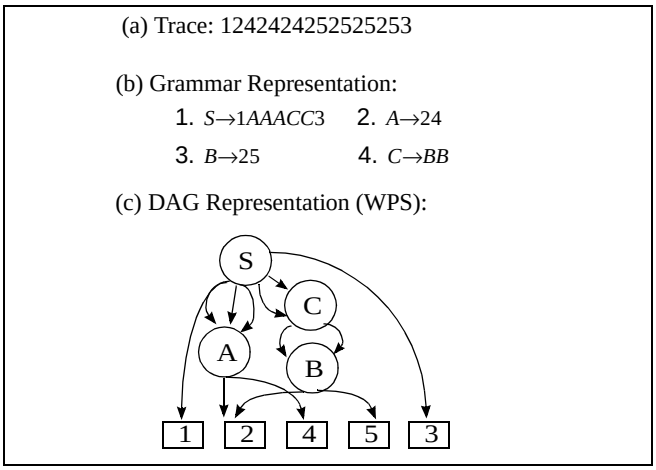

Figure 3. SEQUITUR Compression Scheme

Assume a cache with *K* blocks. For each grammar rule we perform cache simulation on the string defined by the rule, starting simulation *with an empty cache*. During simulation of the rule, we will compute the *LRU state* which is composed of 3 summary values:

1. The *CompulsoryList*. This list keeps the first *K* compulsory misses—defined as the first access to a cache block—that occur during simulation.
2. The *LRUcontext* which is the context of the *LRU* queue at the end of the simulation. The LRU queue is a list of cache blocks ordered by accessed time: the most recent access at the head, the oldest access at the tail of the queue.
3. The *misses* variable, which gives the total number of cache misses that occurred during the simulation.

To understand why these three values are sufficient to memoize the total number of cache misses in the trace, let us consider a simple example. Assume that *T* is a data-reference trace that is divided to two consecutive sub-traces T_1, and T_2. Assume that we have already computed (using cache simulation starting with an empty cache) the above summary values for these two sub-traces. Since we simulated T_2 starting with an empty cache, some of the cache misses we counted would not have been occurred if we started simulation with the *LRU* queue state at the end of the simulation of T_1. We call these cache misses *false* misses. To obtain the number of *false* misses, we compare T_2.*CompulsoryList* and T_1.*LRUcontext*. Each reference that is found in both of these lists is a *false* miss.

Now let us consider how to obtain the final *LRU* context of *T* by composing the summary values of T_1 and T_2. Since the cache policy is *LRU*, all references in T_2 are "newer" than the references in T_1. Hence, to obtain *T.LRUcontext*, we keep the first *K* entries in the concatenation of [T_2.*LRUcontext*] and [T_1.*LRUcontext* \ T_2.*LRUcontext*][2].

2. T_1.*LRUcontext* \ T_2.*LRUcontext* means T_1.*LRUcontext* after removing elements that appear in T_2.*LRUcontext*.

```
// K is the number of blocks in the cache
// input: LRU state after simulation of two consecutive sub-traces T1, T2, starting with an empty cache.
// output: LRU state after simulation of T1 followed by T2.
1.  Compose(LRU T1, LRU T2) {
2.    LRU ANSWER;
3.    ANSWER.misses=T1.misses+T2.misses-|T1.LRUcontext∩T2.CompulsoryList|
4.    ANSWER.LRUcontext=head(concat(T2.LRUcontext,[T1.LRUcontext\T2.LRUcontext]),K);
5.    ANSWER.CompulsoryList=
        head(concat( T1.CompulsoryList,{T2.CompulsoryList\{T1.LRUcontext∪T1.CompulsoryList}} ,K)
6.    return ANSWER
7.  }
```

Figure 4. Memoization Algorithm for a Fully-Associative LRU Cache with *K* Blocks

Figure 4 presents the algorithm for producing the summary value of a trace *T* when *T* is divided to two consecutive sub-traces. To obtain the cache simulation result for a trace that is split into more than two sub-traces (e.g., the *WPS* representation), we compute summary values for all rules traversing the grammar structure in a bottom-up manner and recursively composing the memoized-rule values.

In practice the memoization algorithm (Figure 4) has two disadvantages: (i) It is not suitable for simulation of large LRU caches since such caches have a large number of blocks (e.g., 512 blocks for a typical L1 cache), which causes the *LRUcontext* to become quite long. Long lists increase the overhead of memoization (especially the set operations in lines 3, 4 and 5 in Figure 4); (ii) It is difficult to use the same algorithm when we simulate a *D*-way set-associative cache[1] since we need to keep recently used lists for every cache-set.

To overcome these overhead problems, the framework simulates large caches using a more compact trace representation, the Whole Program Misses (WPM) described in the next section. In addition, we describe an on-demand cache simulation technique (Section 4.2.3) that enables yet more efficient simulation for measurement of the effect of small data layout changes on the cache miss rate.

4.2.2 Miss-Based Compaction

Although the grammar already significantly compresses the trace (about 10-fold), the resulting grammar can be compressed further. In [6], Chilimbi describes several ways to further compress the SEQUITUR grammar. Here we extend this work and present a new compression scheme, the *Whole Program Misses (WPM)* representation. From our framework's perspective, the *WPM* representation offers an advantage over the WPS representation and previous work: It is a more compact trace representation and enables more efficient cache simulation in cases where memoization is less effective. It preserves the number of cache misses in the program and consequently, the accuracy of cache simulation. The *WPM* representation retains the ability to remap all program data objects to new addresses, so it is also useful for evaluating of the impact of an optimization. In addition, the *WPM* representation is easy to build directly from the trace or from the *WPS*.

The *WPM* contains only memory references that may cause a memory fault. In other words, the representation omits references that can never suffer any memory fault. For example, consider the reference sequence '*abba*', where *a* and *b* are two different data objects in the original trace. Regardless of the cache (or TLB or page table) configuration, the second reference to *b* never causes any cache miss. Hence, this reference can be omitted from the trace (and also from its SEQUITUR representation) without affecting the total number of misses. Furthermore, for a specific cache configuration, it is easy to determine that the second reference to *a* will always hit in the cache (even if we re-map *a* and *b* to different addresses). For example, since we have only one unique access between the two consecutive references to *a*, the second reference to *a* will never suffer a cache miss if the cache is at least 2-way set-associative, each set has at least two entries, and each set is managed using the *LRU* policy. To retain the original reference behavior, we never omit the first reference to a data-object. This maintains the ability of the framework to perform accurate cache simulation even if the addresses of all symbols are changed.

To build the WPM, we traverse the trace and keep a list of the last *K* data-objects seen (*K* depends on the particular cache configuration, see below). When processing a new reference, we compare it to the references in the list to see if it never causes a cache miss, and if so, we ignore the reference. Although this technique depends on our cache configuration, a single *WPM* can be used for a several cache configurations. For example, a *WPM* built for a fully-associative LRU cache with 4 entries can be used with any fully-associative LRU-cache with more than 4 entries, or in any cache that is N-way set associative cache where $N \geq 4$.

4.2.3 On-Demand Cache Simulation

The goals of the on-demand cache simulation are: (i) Calculate the number of cache misses that result from 'small' changes to the original memory layout (e.g, changing the locations of a few global variables); (ii) Perform such simulation faster than whole-trace re-simulation. Such fast simulation capability is especially useful for optimizations such as global variables positioning [3], where the optimization needs to determine the effect of alternative layouts on the cache miss rate. We describe such an algorithm for a *D*-way set-associative cache with *K* sets.

On demand set-associative cache simulation is based on the observation that it can be accomplished by simulating each cache

1. A *D*-way set associative cache is a cache that is divided into sets, each set containing *D* lines. Each memory block can reside in only one cache set. The replacement policy inside each set is LRU.

set separately[1], and later summing up the number of misses and the number of references to obtain the overall cache misses. More formally, let $miss_i$ denote the number of cache misses to the ith cache set, then the total number of cache misses can be computed by $Misses=\sum_{i=1}^{k} miss_i$. Hence re-mapping a single data object from set m to set n only affects the number of misses and the number of references of those sets. Let us denote the new number of misses in sets n and m as $missNew_n$ and $missNew_m$, then the new number of misses is calculated by:

$$MissesNew=\left(\sum_{i\neq m,n}^{k} misses_i\right)+missNew_n+missNew_m$$

Since the total number of references in the new layout is the same as in the original layout the new miss rate is easily obtained from the last equation.

The above discussion implies that for small data layout changes we need to simulate only two or a small number of cache sets. Unfortunately, to simulate two sets we need to traverse the *whole* trace (or the equivalent representative grammar) to reveal where these two sets are referenced, so it would appear that the benefits from this approach are minimal. However, given the hierarchical structure of the SEQUITUR grammar, we can use memoization to avoid the linear trace traversal. After the first traversal of a sub-trace (a rule in the grammar, see Section 4.1.2) we record a single boolean value indicating whether set *m* or set *n* are referenced in this sub-trace. If *m* or *n* are not referenced in the sub-trace, subsequent traversals of this sub-dag are unnecessary.

4.3 Evaluation of Framework Efficiency

This section presents empirical results for evaluating the framework's efficiency based on the three methods discussed (i.e., memoization, miss-based compaction, and on-demand cache simulation). To evaluate how the WPM representation and the memoization technique improve the efficiency of TLB simulation, we perform the following experiments. First, we simulate a TLB with 32 entries and LRU replacement policy on the two grammar types, using a complete linear grammar traversal. Second, we apply the simulation again, this time using the memoization algorithm (Figure 4) to avoid the linear traversal.

In a different experiment we evaluate the effectiveness of the on-demand simulation method using a 4-way set-associative 64Kb cache with 512 sets (i.e., each cache-block is 32 bytes). First, we perform a full cache simulation of the two trace representation (i.e., WPS and WPM). Second, for the same cache we performed cache simulation of only two sets. Table 5 presents the results from these two experiments (all times are average of three runs), and shows that these techniques can improve simulation efficiency by over an order of magnitude.

The results in Table 5 serve as a summary for this section. We present algorithms that form the basis of the framework's efficiency. The *memoization*, *compaction*, and *on-demand cache simulation* algorithms are used to efficiently implement the functions required by the process (Table 4). They are especially useful for the implementation of the *Simulate* function and *Evaluation* functions. Table 5 shows that incorporating these methods into the framework enables it to evaluate the future optimization effect, in most cases, within seconds. This efficiency enables the iteration that is necessary for our data-layout optimization process. Indeed, as presented in Section 5.5, this efficient iterative process enables us to develop new types of iterative search-based optimizations.

5. FRAMEWORK APPLICATION TO OPTIMIZE PROGRAM DATA LAYOUT

This section describes the process of using our data-layout optimization framework to apply two optimizations: Field Reordering and Custom Memory Allocation. First, we define the goals of the two optimizations and then detail how the optimizations interact with the framework at each step.

1. In such a cache, each address can be mapped to a single, unique set.

Table 5. Framework Efficiency (all times measured on 500-Mhz 20164-Alpha machine)

Benchmark	Number of heap references in trace[a]	TLB simulation using complete grammar traversal (sec.)		TLB simulation using memoization (sec.)		Cache Simulation times (sec.)		On demand cache simulation (simulating two cache sets)
		Whole Program Streams	Whole Program Misses	Whole Program Streams	Whole Program Misses	Whole Program Streams	Whole Program Misses	
espresso	13,604,108	5.7	1.2	2.6	1	4.74	1.11	0.74
boxsim	36,157,141	25.8	12.3	1.4	1.4	17.69	8.9	0.51
twolf	41,354,395	22.7	12.5	1.4	1.3	17.3	10.82	0.45
perl	38,171,324	25.8	11.2	2.6	1.3	18.87	8.63	0.74
gs	82,872,046	60.7	22.3	11.0	5.0	40.64	16.1	3.44
lp_solve	28,050,510	19.2	7.1	1.2	0.5	14.39	5.6	0.42
Average speed up		1	2.62[b]	11.1[b]	18.2[b]	1	2.22[c]	18.0[c]

a. Since our main interest lies in data-layout optimizations for heap-intensive programs, our grammars contain only heap references. However, grammars for all program references can be built easily.
b. Speed up over TLB simulation using complete grammar traversal when using Whole Program Streams.
c. Speed-up over full cache simulation using Whole Program Streams.

5.1 Optimization Description

The goal of *Field Reordering* is to reduce the number of cache misses by reordering the fields of a structure type [7, 15, 24]. The cache miss rate is reduced by increasing cache block utilization, which is achieved by grouping fields with high temporal affinity in the same cache block.

Existing Field Reordering heuristics use the Pair-wise Affinity profile (see Table 1) to identify fields with high temporal affinity [7, 15]. Starting with the "hottest" pair, the heuristic incrementally builds a layout by appending a field that maximizes the temporal affinity with the fields already in the layout.

First proposed by Seidl and Zorn [23], the *Custom Memory Allocation* (CMA) optimization aims to improve virtual memory performance (i.e., page faults, TLB faults, memory consumption) by increasing memory page utilization. To achieve this goal, CMA attempts to place heap objects with high temporal affinity in the same memory page.

CMA decomposes the optimization task into two sub-problems: (i) finding a layout for heap objects that improves virtual memory performance, and (ii) defining an *allocation policy* that enforces this layout at runtime. An example of an allocation policy might be to cluster all objects that were allocated in a procedure *foo* in a separate memory region. Typically, the allocation policy is based on predictors [2], which are runtime attributes associated with each allocated object (e.g., the call-stack context at object allocation time).

Seidl and Zorn solved the two sub-problems separately [23]. First, using a heap-object lifetime behavior, they defined a "good" layout based on four pre-defined memory regions corresponding to four object lifetime behaviors: the *highly referenced* objects region, the *rarely referenced* objects region, the *short-lived* objects region, and the *other* objects region. Then, they proposed a set of run-time predictors (i.e., the object size and allocation call stack context) that could enforce an approximation of their desired layout.

5.1.1 Optimization Instantiation

Table 6 presents the parameters for the two optimizations that permit them to be instantiated in our framework. For both optimizations, the *Next Layout* functions encapsulate the main difference between previous work and our implementation of these optimizations. The Field Reordering *Next Layout* function drives an exhaustive search in the data layout space to find a good layout for each structure (as described in Section 5.5.1). The CMA *Next Layout* function drives a hill climbing search to find a good allocation policy (as described in Section 5.5.2). Our search based approaches inherently differ from the heuristic approaches used in the past. Since heuristics are based on certain assumptions to construct a new layout (e.g., reordering fields according to their temporal affinity improves cache behavior), if an assumption does not hold for a particular program this may result in a layout that actually degrades performance (as in the case of Field Reordering for perl in Table 10). In contrast, our search will never find a layout that lowers performance since our framework commits to a data layout only if it is proven better—through simulation on the whole program trace—than the original layout.

.

Table 6. Optimization Parameters

	Field Reordering	Custom Memory Allocation
Optimized resource	L1 cache: r=(*CS*, 0.02). *CS* is a Cache Simulator function that returns the miss rate of *r* on the trace *T*.	Virtual Memory r=(*WSS*, 10). *WSS* is a Working Set Simulator function that returns the average working set size of the trace *T*.
Optimization Predicate	see Table 2	
Build Profile function	Field Frequency Profile: objects→real , maps fields to their access frequencies.	Heap Objects Profile: objects→attribute set . maps objects to runtime attribute (see Section 5.5.2).
Next Layout function	Exhaustive search for a candidate layout. In case this search is too costly, perform an exhaustive search only using frequently accessed fields (see Section 5.5.1).	Recursively partition the objects into memory regions (see Section 5.5.2).

Another difference from previous work arises in the way we measure the performance benefit of CMA. Workstations today have large physical memories (e.g., at least 512MB), that accommodate the data set of most programs. Hence, most programs suffer only a few compulsory page faults (i.e., faults resulting when a page is first referenced) when running in isolation on a machine. Thus, we measure the benefit of CMA using a program's Average Working Set size—the average number of virtual memory pages the program uses [10]—rather than its page fault rate. Improving this performance metric allows more applications to concurrently run on a machine without paging to disk.

As discussed in Section 3, the framework requires a threshold for each memory resource to identify bottlenecks. Selecting the "correct" threshold is a difficult task that requires experience and tuning. For a 16KB directed mapped cache, miss rates lower than *2%* appear "normal" behavior for well tuned programs (see page 391 in [18]). For the virtual memory system, we use our own estimation that a program using less than 10 pages (each page is 8Kb) is not limited by its virtual memory performance.

5.2 Bottleneck Analysis

The process starts with *Bottleneck Identification* which finds the memory resources that limit the program's memory performance (see Section 3). Table 7 presents the cache miss rates and the working set sizes for our six benchmarks. Using the resource thresholds presented in Table 6, all programs except *espresso* suffer from L1 cache and virtual memory bottlenecks.

Table 7. Memory Bottlenecks[a]

Bench-mark	Number and percentage of heap references.	Cache miss rate[b]	Working set size (pages)[c]
espresso	13,604,108 (65%)	0.67%	8.27
boxsim	36,157,141 (36%)	**7.70%**	**60.04**
twolf	41,354,394 (44%)	**9.31%**	**17.35**
perl	38,171,324 (37%)	**3.21%**	**25.90**
gs	82,072,046 (45%)	**8.22%**	**128.30**
lp_solve	28,050,510 (58%)	**13.80%**	**26.30**

a. A bold number indicates that this resource is a bottleneck for that benchmark.
b. 16KB direct-mapped cache.
c. Average number of pages touched every 100,000 heap references. The size of each page is 8Kb.

We compute these metrics, and all further results in this section, only for heap references for the following reasons. First, in "general purpose" programs (and also in our benchmarks as the percentage of heap references demonstrates), heap objects have a large influence on overall memory performance [3]. Hence, improving the memory behavior of these objects is important for good overall memory performance. Second, both optimizations mainly target heap objects—structure fields or structure instances [7, 23]—by clustering them into the same memory unit—a cache block or a memory page—to increase utilization. Since other objects, such as global or stack variables cannot be clustered together with heap objects, their presence in the trace does not affect the decision of which objects to cluster together.

As mentioned in Table 4, the process continues with *Data-Objects Analysis*, which builds the critical set for each bottleneck. Since the bottleneck memory behavior is mostly influenced by the critical objects, the process uses them, in the next step, to narrow the search only to those optimization that target these objects. Our L1-cache critical set covers at least 80% of both the cache misses and data references (i.e., *k=j=80%* in function *BCO* in Table 4). It turns out that the virtual memory critical set is almost identical to the L1-cache critical set, since both sets contain objects that cover at least 80% of data references.

Table 8 presents critical set sizes and their coverage rates for all benchmarks. For completeness, we show the results even for cases where the framework does not actually perform the calculation for a given program (*espresso* in Table 8); these results are in strike-through text. Table 8 suggests an analogy between code optimization and data-layout optimization. Similar to the commonly used rule-of-thumb that 10% of the code is executed 90% of the time, Table 8 suggests that 10% of data-objects cover 90% of all memory references *and* (almost) 90% of cache misses. Indeed, some data-layout optimizations such as Field Reordering and Class Splitting do optimize only the highly referenced objects [7]. However, whether other optimizations such as CMA can derive almost all their benefit from placing only critical objects requires further investigation.

Table 8. Critical Objects[a]

Bench-mark	Total # of objects	Critical set		
		Size[b]	Reference coverage	Cache-miss coverage
~~espresso~~	~~131,326~~	~~9.6%~~	~~86%~~	~~85%~~
boxsim	99,881	5.2%	96%	90%
twolf	16,287	10.3%	90%	96%
perl	108,552	11.7%	91%	85%
gs	368,205	14%	88%	85%
lp_solve	19,928	13.7%	84%	84%
Average		10.70%	89.1	87.5%

a. We present data for all benchmarks even if no bottlenecks were identified. Such data is marked with strike-through text.
b. Percentage of all data objects.

5.3 Select Optimizations

To narrow the search, the process selects optimizations that produce layouts with the potential to influence bottleneck behavior. To achieve this, the process computes the *Optimization Target Objects* set for each candidate optimization. The set is computed by applying the *optimization predicate* (presented in Table 2) to the critical objects, retaining only those that qualify as optimization targets.

Table 9 shows the target sets *coverage rates* for our two optimizations. The coverage rate, which is the fraction of the trace covered by the set, measures the potential of the optimization to affect the program memory behavior; the higher the rate, the higher the potential (see Section 3 for a more formal definition of these concepts). The process eliminates optimizations whose target sets are empty[1]; these optimizations have very little potential to improve the bottleneck performance. In our case, Field Reordering is eliminated for *espresso*, *lp_solve*, and *ghostscript*.

Table 9. Target Objects Set Coverage Rate

	Coverage Rates		Selected Optimizations	
	Field Reorder-ing	CMA	Field Reorder-ing	CMA
~~espresso~~	~~0%~~	~~71%~~	~~no~~	~~yes~~
boxsim	27%	18%	yes	yes
twolf	42%	45%	yes	yes
perl	22%	33%	yes	yes
gs	0%	2.5%	no	yes
lp_solve	0%	11%	no	yes

5.4 Build Profile

The *Build Profile* step builds the profile information needed to drive the optimizations. As we will shortly discuss in Section 5.5, our Field Reordering optimization is based on an exhaustive search

1. The framework can be easily changed so that it eliminates optimizations with target sets below a preset threshold.

in the space of field layouts. However, when the exhaustive search is too time consuming, the optimization uses the Field Frequency profile (see Table 6) to narrow the search space. First, the optimization groups fields into pairs according to their frequency, and then iterates over all possible field-pair permutations.

To build an allocation policy, which maps sets of heap object attributes to memory regions, the CMA requires a profile of all the heap objects allocated during the sample execution with each object annotated with the values of its runtime attributes. For this purpose, we used an annotated *allocation trace*. The allocation trace is the history of all program allocations and de-allocations. Each entry in the trace corresponds to one physical dynamically allocated data object, and each entry is annotated with the values of seven runtime attributes: the last three procedures on the stack, the sizes of the last three allocated objects, and the current allocated object size. The values of these attributes are recorded at the entry point of the allocation routine (e.g., C malloc).

5.5 Iterative Profile-Feedback Search

To find a better memory layout, the process continues with an iterative search process. Starting with the optimization that has the highest potential (i.e., the highest coverage rate for the optimization target set), our framework applies a separate hill climbing search for each selected optimization; each search process starts with the resulting layout of the optimization applied previously.

In this section, we describe the search for each of our example optimizations. The search combines two process steps: The *Apply Optimizations* step, which uses the *Next Layout* function to produce a candidate data layout, and the *Evaluate* step, which provides the necessary profile-feedback by measuring the performance benefit of the candidate data layout.

5.5.1 Field Reordering Exhaustive Search

Since finding the optimal structure layout is intractable [20], our Field Reordering Next Layout function iterates over all possible layouts (i.e., over field permutations) for each targeted structure type. As specified in Table 2, Field Reordering targets data structures with more than two fields that are larger than the cache block size. Starting with the most frequently referenced structure, we iterate over all field permutations within each target structure, committing the best layout produced. This exhaustive search strategy finds "almost optimal" layouts for each structure. The layout is "almost optimal" since it depends on the order in which we iterate over the structures themselves.

If the number of the structure fields is too large and iteration over all possible structure layouts becomes too time consuming—in our current framework implementation if the number of layouts exceed one thousand—we narrow the exhaustive search using the "field pairs" technique. Since the cache miss rate is mostly influenced by the *distribution* of fields among cache blocks rather than the *order* of fields inside a cache block, our goal is to quickly explore the set of fields that should reside in the same cache block rather than their order within the block. Consequently, we first order the fields according to their access frequency. Then, we group the fields into pairs according to this order (such that the most frequently accessed field is paired with the second most frequently accessed, the third with the forth, and so on). Last, we iterate over all the layouts resulting from all pair permutations. For example, a structure with 12 fields that originally requires evaluation of 12! layouts, now requires the evaluation of only 6! layouts. Indeed, our results show that the difference between full iteration and the "field pairs" approach is negligible (less that 0.1% reduction in miss rate).

5.5.2 CMA Hill Climbing Search

As mentioned, Seidl and Zorn's CMA solves one sub-problem at a time: First they find a "good" memory layout using a predefined number of memory regions, and then they build, using runtime predictors (attributes), an allocation policy to enforce it at runtime. This decomposition has a drawback: the *desired* layout may not be enforceable by the set of predictors, so the *actual* layout enforced at runtime may not yield the expected performance benefit. We tackle this difficulty by combining the two problems into one *classification* problem: given a set of attributes, find an allocation policy that classifies the objects into different memory regions such that the resulting memory layout has "good" memory behavior. Solving the this classification problem yields an enforceable layout that should improve performance.

We adopt a simple approach to solving the classification problem: we synthesize an allocation policy by building a decision tree from a given set of attributes [17]. Our CMA *Next Layout* function synthesizes an allocation policy by incrementally partitioning the heap objects into different regions. Starting with all objects in a single memory region, the function works as follows. Given a memory layout with n regions, it produces all possible layouts with $n+1$ regions. To obtain a layout with $n+1$ regions the function divides a region into two sub-regions by using a single attribute to distinguish between the objects in the original region. After evaluating all possible layouts with $n+1$ regions, the function selects the layout with the highest performance benefit. If there is no additional benefit from this partitioning, the function returns null.

5.5.3 Evaluation Step

The evaluation step is similar for both optimizations. The framework "pretends" that the optimization was applied by substituting the address of each data object in the data-object trace with its new address from the candidate data layout. After the addresses are remapped, the framework uses the Simulate function (see Table 4) to evaluate the new trace.

5.6 Optimizations Results

This section presents the results of our two example optimizations. We also discuss the cost and effectiveness of the iterative approach.

5.6.1 Field Reordering Results

We have compared our iterative approach with other existing methods for Fields Reordering. First, we ordered fields according to their frequency profile, starting with the fields with the highest frequencies [24]. Second, we ordered the fields according to their pairwise affinity profile, as discussed in Section 5.1 [7]. Last, we iterated over the fields layout space, exhaustively exploring many possible layouts, as described in Section 5.5.1.

Table 10 presents the results for the three version of the Field Reordering optimization. The frequency-based approach and the affinity approach reduced the cache miss rate by 19.3% and 16% respectively, while the exhaustive search reduced the miss rate by 25.6%. The exhaustive search is consistently better than both the frequency and affinity profiles.

Table 10. Field Reordering Results[a]

	Original miss rate	Reduction in cache miss rate		
		Iteration	Affinity Profile	Frequency Profile
boxsim	7.70%	18%	10%	7%
twolf	9.31%	42%	39%	38%
perl	3.21%	17%	-1%	13%
Average **reduction** in miss rate		25.6%	16%	19.3%
~~lp_solve,gs, espresso~~	~~Less than 0.01% reduction in cache miss rate~~			

a. The results were obtained using the "train" input — the input that is used to build the grammar and drive the search. Results for the test input are not available yet because some of the layouts require considerable code modifications to unfold nested structures into the "containing" structure. However, previous work [3, 7] indicates that there is high correlation between the train and test inputs.

When using the "field pair" heuristic (see Section 5.5.1), the search was completed in less than an hour. Although this may seem costly, both the Frequency and Affinity approaches require a time consuming effort, (i.e., instrumenting the program, building the profile, etc.). Furthermore, Field Reordering is usually applied rarely, towards the end of the application development cycle.

5.6.2 CMA Results

We have measured the working set size of our benchmarks both with the "train" input, which was used to learn the allocation policy, and a "test" input, which was used to measure how well the allocation policy behaves on an unfamiliar input.

Table 11 presents the results for these two inputs. On average, our CMA reduces the working set size on "test" inputs by *6.1%*. We compared the iterative approach to a "static approach" in which the allocation policy uses a predefined number of memory regions. We built a Custom Memory Allocation that uses two regions: the "hot" region for frequently accessed heap objects, and the "cold" region for rarely accessed objects. Using this allocator, only *perl* and *boxsim* showed a reduction in the working set size; and we obtained these results only after we manually tuned the definition of hot objects for each program. We believe that because the iterative process adapts the number of regions to the program under consideration, it can build a better CMA than the static approach.

The allocation policy learning method requires more layouts to be evaluated than the Field Reordering case (i.e., thousands vs. hundreds), and the learning is completed in a few hours. As in the case of Field Reordering, we do not consider this too costly. Previous work on CMA requires almost the same effort [23], and CMA is usually applied once at the end of the development process.

Table 11 presents another interesting observation. Although the virtual memory was not identified as a bottleneck for *espresso*, it shows considerable reduction in the working set size. Note that, eliminating a memory resource as a bottleneck does not imply that its memory behavior can not be improved; it just questions the necessity of doing so. In fact, Table 9 shows that the CMA target set of *espresso* covers 71% of the trace, indicating the high potential of the optimization to reduce the working set size.

Table 11. Custom memory allocation - results

Benchmarks	**Input**	**Original Working Set Size[a]**	**New Working Set size**	**Improvement**	**Number of Regions Used[b]**
~~espresso~~	~~mlp4~~	~~8.27~~	~~7.97~~	~~3.65%~~	~~2~~
	~~largest~~	~~19.81~~	~~16.51~~	~~16.6%~~	
boxsim	N 2 T 250	60.04	57.22	4.7%	8
	N4 T 500	71.16	61.52	13.5%	
twolf	spec 2000 test	17.35	14.14	18.5%	5
	spec2000 ref	114.7	112.8	1.6%	
perl	recurse	25.9	24.4	6.0%	5
	scrabble	35.4	30.5	7.9%	
gs	intro	128.3	120.7	5.9%	10
	report	121.4	115.8	4.6%	
lp_solve	etamacro	26.3	23.8	9.4%	6
	fit1p	52.6	50.8	3.2%	
Average reduction in working set size (measured only on **test** input without *espresso*).				6.1%	

a. Average number of pages touched every 100,000 references.
b. The number of memory regions used is also the number of sets in the partition that the decision tree found.

6. CONCLUSIONS

We present a generalized process for profile-driven data-layout optimization. Unlike prior work on data-layout optimization, the process is based on a profile-guided iterative search in the data layout space. Profile-feedback is used both to narrow the data layout search space and to select among many candidate layouts. The process unifies all existing data-layout optimizations and permits composing multiple layout optimizations. In addition, it enables selection, application, and tuning of the more profitable optimizations for the program under consideration. We implement the process in an efficient framework. The framework efficiency stems from new techniques for fast memory simulation: memoization-based profile analysis, miss-based profile compaction, and on-demand cache simulation. These techniques, together with the ability to evaluate a data layout without program re-execution, drive the iterative search. Using the framework, we instantiate two example optimizations: Field Reordering and Custom Memory Allocation. Our Field Reordering technique, based on an exhaustive search, outperforms existing non-iterative techniques and reduces the cache miss rate by 25%. Our Custom Memory Allocation is based on a hill climbing search that classifies heap objects according to their runtime attributes rather than a pre-defined partition of program data objects. Without our new data-layout optimization framework and its efficient implementation, development of this allocator would not have been possible.

ACKNOWLEDGMENTS

We thank Brian Fields, Denis Gopan, Pacia Harper, and Martin Hirzel, as well as the anonymous referees for their comments on

early drafts of the paper. Andy Edwards patiently answered our x86 questions. This work was supported in part by the National Science Foundation with grants CCR-0093275 and CCR-0103670, the University of Wisconsin Graduate School, and donations from IBM and Microsoft Research. Shai Rubin was supported in part by a Fullbright Scholarship. The work was started when Shai Rubin was an intern at Microsoft Research.

References

[1] ALT, M., FERDINAND, C., MARTIN, F., AND WILHELM, R. Cache behavior prediction by abstract interpretation. *Lecture Notes in Computer Science 1145* (1996), 52–66.

[2] BARRETT, D. A., AND ZORN, B. G. Using lifetime predictors to improve memory allocation performance. In *Proceedings of PLDI'93, Conference on Programming Languages Design and Implementation* (Albuquerque, NM, June 1993), pp. 187–196.

[3] CALDER, B., CHANDRA, K., JOHN, S., AND AUSTIN, T. Cache-conscious data placement. In *Proceedings of ASPLOS'98, Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose CA, 1998).

[4] CARR, S., MCKINLEY, K. S., AND TSENG, C.-W. Compiler optimizations for improving data locality. In *Proceedings of ASPLOS'94, Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, Oct. 1994), pp. 252–262.

[5] CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. Exact analysis of the cache behavior of nested loops. In *Proceedings of PLDI'01, Conference on Programming Languages Design and Implementation* (Snowbird, UT, June 2001), pp. 286–297.

[6] CHILIMBI, T. M. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of PLDI'01, Conference on Programming Languages Design and Implementation* (Snowbird, UT, June 2001), pp. 191–202.

[7] CHILIMBI, T. M., DAVIDSON, B., AND LARUS, J. R. Cache-conscious structure definition. In *Proceedings of PLDI'99, Conference on Programming Languages Design and Implementation* (Atlanta, GA, May 1999), pp. 13–24.

[8] CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. Cache-conscious structure layout. In *Proceedings of PLDI'99, Conference on Programming Languages Design and Implementation* (Atlanta, GA, May 1999), pp. 1–12.

[9] CHILIMBI, T. M., AND LARUS, J. R. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the International Symposium on Memory Management (ISMM-98)*, vol. 34, 3 of *ACM SIGPLAN Notices*, pp. 37–48.

[10] DENNING, P. J., AND SCHWARTZ, S. C. Properties of the working set model. *Communications of the ACM 15*, 3 (Mar. 1972), 191–198.

[11] DOLBY, J., AND CHIEN, A. An automatic object inlining optimization and its evaluation. In *Proceedings of PLDI'00, Conference on Programming Languages Design and Implementation* (Vancuver, Canada, June 2000), pp. 345–357.

[12] DUTT, S. New faster kernighan-lin-type graph-partitioning algorithms. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (Santa Clara, CA, Nov. 1993), M. Lightner, Ed., IEEE Computer Society Press, pp. 370–377.

[13] FERDINAND, C., AND WILHELM, R. Fast and efficient cache behavior prediction.

[14] GANNON, D., JALBY, W., AND GALLIVAN, K. Strategies for cache and local memory management by global programming transformation. *Journal of Parallel and Distributed Computing 5*, 5 (Oct. 1988), 587–616.

[15] KISTLER, T., AND FRANZ, M. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems 22*, 3 (2000), 490–505.

[16] LAMARCA, A., AND LADNER, R. E. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithms 1* (1996), 4.

[17] MITCHELL, T. M. *Machine learning*. McGraw Hill, New York, US, 1996.

[18] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Inc., San Mateo, CA, 1990.

[19] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design: Hardware and Software Interface*. Morgan Kaufmann Publishers, San Mateo, California, June 1993.

[20] PETRANK, E., AND RAWITZ, D. The hardness of cache conscious data placement. In *Proceedings of POPL'02, Conference on Principles of Programming Languages* (Portland, OR, Jan. 2002).

[21] RUBIN, S., BERNSTEIN, D., AND RODEH, M. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. *Lecture Notes in Computer Science 1575* (1999), 259–273.

[22] SAGIV, M., REPS, T., AND WILHELM, R. Parametric shape analysis via 3-valued logic. In *Proceedings of POPL'99 Conference on Principles of Programming Languages* (San Antonio, TX, Jan. 1999), pp. 105–118.

[23] SEIDL, M. L., AND ZORN, B. G. Segregating heap objects by reference behavior and lifetime. In *Proceedings of ASPLOS'98, Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 1998), pp. 12–23.

[24] TRUONG, D. N., BODIN, F., AND SEZNEC, A. Improving cache behavior of dynamically allocated data structures. In *Proceedings of PACT'98, Conference on Parallel Architectures and Compilation Techniques* (Paris, France, Oct. 1998), pp. 322–329.

[25] VAN DER WAL, R. Programmer's toolchest: Source-code profilers for Win32. *Dr. Dobb's Journal of Software Tools 23*, 3 (Mar. 1998), 78, 80, 82–88.

[26] WOLF, M. E., AND LAM, M. S. A data locality optimizing algorithm. In *Proceedings of PLDI'91, Conference on Programming Languages Design and Implementation* (Toronto, Canada, June 1991), pp. 30–44.