Cache-Conscious Coallocation of Hot Data Streams

Trishul M. Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
trishulc@microsoft.com

Ran Shaham[1]
Amuse Toy & Game Development
12/7 Yoni Netanyahu, Haifa 31905
Givat Shmuel 54423, Israel
ran.shaham@gmail.com

ABSTRACT

The memory system performance of many programs can be improved by coallocating contemporaneously accessed heap objects in the same cache block. We present a novel profile-based analysis for producing such a layout. The analysis achieves cache-conscious coallocation of a hot data stream *H* (i.e., a regular data access pattern that frequently repeats) by isolating and combining allocation sites of object instances that appear in *H* such that intervening allocations coming from other sites are separated. The coallocation solution produced by the analysis is enforced by an automatic tool, *cminstr*, that redirects a program's heap allocations to a run-time coallocation library *comalloc*. We also extend the analysis to coallocation at object field granularity. The resulting field coallocation solution generalizes common data restructuring techniques, such as field reordering, object splitting, and object merging, and allows their combination. Furthermore, it provides insight into object restructuring by breaking down the coallocation benefit on a per-technique basis, which provides the opportunity to pick the "sweet spot" for each program. Experimental results using a set of memory-performance-limited benchmarks, including a few SPECInt2000 programs, and Microsoft VisualFoxPro, indicate that programs possess significant coallocation opportunities. Automatic object coallocation improves execution time by 13% on average in the presence of hardware prefetching. Hand-implemented field coallocation solutions for two of the benchmarks produced additional improvements (12% and 22%) but the effort involved suggests implementing an automated version for type-safe languages, such as Java and C#.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors– *code generation, optimization, run-time environments*.

General Terms Performance, Measurement.

Keywords hot data streams, data profiling, dynamic allocation, memory layout, cache optimization, data locality.

1. INTRODUCTION

The rapidly growing processor-memory performance gap has made effective cache memory utilization an important determinant of overall program performance. Traditionally, hardware solutions to this problem include larger, more associative caches, non-blocking caches, speculation, and out-of-order execution. However, power considerations impose constraints on many of these techniques [9]. For example, larger caches reduce the number of cache misses but increase the power cost of each cache access. Consequently, software-based approaches to memory system performance optimization offer an attractive alternative as programs can be transformed to make more efficient use of the memory system, reducing memory power consumption as well as improving performance.

Memory and cache behavior studies of general-purpose programs indicate that a small fraction of data objects (10%) are responsible for most of the data references (90%) and cache misses (almost 90%) [3, 16]. This 90/10 rule makes these hot data objects attractive targets for software-based cache locality optimizations. In addition, recent research suggests that rearranging these hot data objects in memory could produce potential cache miss rate reductions of up to 80% [3].

While many cache-conscious data placement techniques exist [2,5,6,14,19,20], they suffer from two drawbacks. First, their placement decisions are guided by object/field frequency or pairwise affinity profiles, which are crude approximations of a program's temporal data reference behavior. Next, their layout decisions are determined by fairly ad-hoc heuristics. These can both be serious limitations in the light of research that has shown that layouts guided by inexact profiles can be far from optimal, and layout heuristics cannot be both robust and effective (i.e., work consistently well for a wide variety of programs) [15].

In contrast to many current cache-conscious data placement techniques, our technique relies on more precise and detailed profile information that is nevertheless cheap to collect [3,13], and relatively stable across different program runs [4]. In addition, it uses an efficient polynomial optimal-approximation algorithm to process the profile information and produce a good data layout, rather than rely on ad-hoc heuristics.

This paper builds on research that shows how to efficiently capture accurate temporal data reference profiles [3]. Chilimbi's *whole program streams (WPS)* is a compact yet complete representation of a program's data reference behavior. This WPS representation explicitly encodes regular access patterns and permits efficient extraction of *hot data streams*, which are sequences of consecutive data references that frequently repeat in the same order. This paper's key insight is that cache-conscious coallocation of a hot data stream, *H*, can be achieved by isolating and combining allocation sites of *H*'s data members, such that intervening allocations coming from other sites are separated. In more detail, coallocation is achieved through several separate heap regions where objects that are consecutively allocated in a region are placed contiguously. The analysis determines which allocation sites to direct to the same region to achieve

PLDI'06, June 11-14, 2006, Ottawa, Ontario, Canada.

[1] Work done while author was an intern at Microsoft Research.

(a) Original Layout (x_i is allocated at allocation site *X* and is the program's i-th allocation)

b_1	d_2	$\mathbf{a_3}$	d_4	$\mathbf{b_5}$	e_6	d_7	d_8	d_9	$\mathbf{c_{10}}$	c_{11}	a_{12}	d_{13}	a_{14}	b_{15}	...

(b) An access trace that contains a hot data stream H = $\mathbf{a_3b_5c_{10}}$

...$b_1e_6\mathbf{a_3b_5c_{10}}d_4\mathbf{a_3b_5c_{10}}d_8e_6a_3d_7a_{14}\mathbf{a_3b_5c_{10}}d_9$...

(c) Coallocating hot data stream H = $\mathbf{a_3b_5c_{10}}$

area for allocation sites {A, B, C}	b_1	$\mathbf{a_3}$	$\mathbf{b_5}$	$\mathbf{c_{10}}$	c_{11}	a_{12}	a_{14}	b_{15}	...

area for other allocation sites	d_2	d_4	e_6	d_7	d_8	d_9	d_{13}	...	...	...	...

Figure 1. Coallocating a hot data stream.

coallocation, and which allocation sites prevent coallocation and must be isolated in a separate region. We deliberately focus only on objects in hot data streams, since these typically incur most of the cache misses [3,16].

Figure 1(a) shows an example data layout, where objects are small and placed contiguously according to the program allocation order. x_i is allocated at allocation site *X*, and it is the *i*-th global allocation request in the program. For example b_1 is allocated in allocation site *B*, and this is the first object allocation in the program. d_2 is the next allocation request in the program. This object is allocated in allocation site *D*. Figure 1(b) shows a portion of a data reference trace containing a hot data stream $H=a_3\ b_5c_{10}$, a regular reference pattern that frequently repeats. Allocation sites *A, B, C* allocate objects a_3, b_5, c_{10} respectively. Colocating these objects, which are frequently accessed contemporaneously, can significantly reduce cache misses, especially if *H* fits in a cache block. Even when it does not, the resulting improvement in spatial locality can increase the effectiveness of hardware prefetching.

We achieve this as shown in Figure 1(c) by isolating allocation requests from sites *A, B, C* and separating allocation requests from other sites (i.e., *D, E*). Though other objects allocated at sites *A, B, C* are also placed in the same heap region as the hot data stream objects, this is acceptable as our primary objective to colocate hot data stream objects. Note that it is not always possible to colocate hot data stream members in this fashion. For example, if $a_3d_7a_{14}$ is a hot data stream, intervening allocations of objects allocated at allocation sites *A, D* (e.g., d_4, a_{12}) that cannot be separated, prevent coallocation.

This paper describes a profile-based analysis tool, **`cmanal`**, that produces a cache-conscious coallocation of heap objects that participate in hot data streams. We show that optimal coallocation can be reduced to the *weighted set packing(WSP)* problem, which is known to be NP hard [7]. Hence, **`cmanal`** uses the best known polynomial time approximation to the WSP problem [12] to arrive at the object coallocation solution.

In addition, we present an instrumentation tool, **`cminstr`**, that replaces a program's heap allocation requests with calls to a run-time coallocation library, **`comalloc`**, to enforce the coallocation solution layout. For example, **`cminstr`** would enforce the coallocation solution in Figure 1 by replacing allocations at sites *A, B, C* with calls to **`comalloc`**$_\mathbf{1}$, and allocations at other sites including *D, E* with calls to **`comalloc`**$_0$ (each **`comalloc`**$_\mathbf{i}$ manages a separate heap region where consecutively allocated objects are colocated). We demonstrate average execution time improvements of 13% using this technique.

Unfortunately, object coallocation cannot always be fully exploited on older machines where cache blocks are often only 32 bytes, though current-generation machines offer greater opportunities (for e.g., the Pentium 4 and the Itanium have 128 byte L2 cache blocks). However, hardware prefetching increases the effective cache block size by fetching the next few sequential cache blocks on a cache miss.

Previous work on cache-conscious data layout has addressed this by rearranging objects at a field granularity using techniques such as field reordering, hot/cold structure splitting and merging structures [6,14,19,20]. Extending **`cmanal`** to analyze potential coallocation at a field granularity produces significantly larger cache miss rate reductions. Since coallocating fields is a generalization of the combination of object field reordering, splitting, and merging, this analysis permits estimating the contribution of each individual technique to the field coallocation solution. We show that restrictions on field rearrangements yield a spectrum of coallocation solutions, which tradeoff ease of layout enforcement with larger cache miss rate reductions. Promisingly, in several cases most of the benefits of the general solution (arbitrary field rearrangement) can be achieved with a subset of the field restructuring techniques. Hand-implemented field coallocation solutions for two of the benchmarks produced additional improvements (12% and 22%) but the effort involved suggests implementing an automated version for type-safe languages, such as Java and C#.

The paper's main contributions include:

- A novel and efficient profile-based analysis for cache-conscious coallocation of hot heap objects that are contemporaneously accessed (Section 2).
- An extended analysis for coallocation at field granularity, which generalizes field reordering, structure splitting, and structure merging(Section 3).
- An automatic implementation of object coallocation and a semi-automatic implementation of field coallocation that shows that programs have significant coallocation opportunities and that can be exploited to produce execution time improvements (Section 4 and Section 5).

1.1 Related Work

Seidl and Zorn allocated heap objects in four pre-defined memory regions based on their summary reference characteristics [17]. Rubin et al., used a search-based learning technique to classify heap objects according to runtime characteristics such as allocation calling context, object size, etc., and allocate objects in separate heap regions based on this classification [16]. Both these techniques improve virtual memory performance by increasing page

utilization but have little, if any, impact on cache performance. **ccmalloc** is a cache-conscious heap allocator that uses programmer annotations to allocate contemporaneously accessed data objects in the same cache block [5]. We would like to achieve such cache-conscious coallocation automatically using the hot data stream profile. Guyer and McKinley used static analysis to determine object connectivity and used the garbage collector to colocate objects [10]. We have similar goals but use an efficient approximation algorithm in place of heuristics to guide data layout. Huang et al. use profile information to modify the garbage collector traversal order to improve locality [11]. Calder et al. applied placement techniques developed for instruction caches to data [2]. They use a compiler-directed approach that creates an address placement for stack variables, globals, and heap objects in order to reduce data cache misses. Their technique uses the temporal relationship graph (TRG) [8], shows significant gains for stack objects and globals but little improvement for heap objects. On the other hand, we focus solely on heap objects, use the hot data stream profile, which is more precise than the TRG since it does not depend on an arbitrary temporal reference window size, and demonstrate significant data cache miss rate reductions for some programs. Zhong et al. use whole-program reference affinity to perform field-level transformations, such as structure splitting [20]. In addition to field-level transformations, we perform automatic object coallocation using hot data streams as our locality model. Lattner et al. use compiler pool allocation to partition heap objects [21]. Their partitioning is statically enforced by the compiler whereas we use hot data stream profiles and an analysis to specifically improve cache locality.

2. COALLOCATING OBJECTS

This section describes our algorithm for coallocating heap objects that participate in hot data streams.We first briefly review hot data streams construction and then discuss a simple model to estimate the benefits of coallocation. Next, we describe when hot data stream objects may be coallocated. Finally, we present a cache-conscious object coallocation algorithm.

2.1 Hot Data Stream Sets

Chilimbi used a compression algorithm called Sequitur to construct a compact context-free grammar representation of a data reference trace, which can then be efficiently analyzed to detect hot data streams (see [3] for details). A hot data stream is a frequently repeated sequence of consecutive data references (in other words, a frequent data access pattern). The ``heat'' of a data stream is its length (number of stream references) multiplied by its frequency. A minimal hot data stream is the minimal prefix (exceeding length *1*) of a hot data stream with a heat *c* or more. The *hot data stream analysis* presented in [3] detects a set *B* of non-overlapping minimal hot data streams such that references to stream elements in *B* yields *P%* coverage of all trace references (usually the analysis sets *P=90%*).

The size of hot data streams detected by the analysis is adjustable. Based on our experimental results, we set the analysis to detect minimal hot data stream that contain at least *2* and not more than *20* elements. While longer hot data streams offer more coallocation opportunities, they also increase the computational cost of the coallocation analysis. The analysis also computes the normalized heat value for each hot data stream in *B*, such that a hot data stream *H* with normalized heat value H_c covers H_c% of the *reduced trace*, (i.e., *H* covers ($P*H_c$)% of the original trace).

For our purpose of coallocating hot data stream members contiguously, the exact order of objects in a hot data stream makes little difference as long the objects are in the same cache block, and ignoring the order also provides more coallocation opportunities. Hence, we reduce hot data streams to *hot data stream sets*, and our

(a) hot data streams with potential miss reduction

ht	stream	coalloc-atable objects	mr	coalloc-ation set	w'
0.6	$a_3b_5c_{10}$	a_3, b_5	1	{A,B}	0.6
		a_3, c_{10}	1	{A,C}	0.6
		b_5, c_{10}	1	{B,C}	0.6
		a_3, b_5, c_{10}	2	{A,B,C}	1.2
0.3	$d_4b_{15}e_6$	b_{15}, e_6	1	{B,E}	0.3
		d_4, e_6	1	{D,E}	0.3
0.1	$d_4e_6a_{12}a_{14}$	d_4, e_6	1	{D,E}	0.1
		e_6, a_{12}	1		
		e_6, a_{14}	1		
		a_{12}, a_{14}	1	{A}	0.1
		e_6, a_{12}, a_{14}	2	{A,E}	0.2

(b) accumulated and normalized coallocation set miss redn.

coallocation set	accumulated miss reduction	normalized miss reduction
{A,B}	0.6	0.29
{A,C}	0.6	0.29
{B,C}	0.6	0.29
{A,B,C}	1.2	0.57
{B,E}	0.3	0.14
{D,E}	0.4	0.19
{A,E}	0.2	0.1
{A}	0.1	0.05

(c) possible weighted set packing

	set packing	miss reduction
1	{A,B,C},{D,E}	0.76
2	{A,B},{D,E},{C}	0.48
3	{A,C},{D,E},{B}	0.48
4	{A,C},{B,E},{D}	0.43
5	{A,E},{B,C},{D}	0.38
8	{A},{B,C},{D,E}	0.52
9	{A},{B,E},{C,D}	0.19

(d) coallocation layout for set packing (1)

{A, B, C}	b_1	a_3	b_5	c_{10}	c_{11}	a_{12}	a_{14}	b_{15}	...
{D, E}	d_2	d_4	e_6	d_7	d_8	d_9	d_{13}	...	...

Figure 2. Coallocation algorithm example.

```
// first phase (see Figure 4):
// compute cache miss reduction weights for coallocation sets.
// object coallocatable communicates the criteria for coallocating
// objects (in Section 3 we use computeWeight with different
// coallocatability definitions).
computeWeight (object coallocatable, w)

// second phase: compute approximate weighted set packing
// according to Halldorsson's algorithm.
// ℋ is the set of hot data streams. ℋ_WSP is a partition of S_alloc(ℋ)
// with approximate maximal cache miss reduction.
ℋ_WSP = WSP(S_alloc(ℋ), {<C,w(C)>|w(C) > 0})
R_o = Σ_{C∈ℋ_WSP} w(C)
output ℋ_WSP

// third phase: compute cache miss reduction for a given layout,
// and then normalize weights.
for every H ∈ ℋ
   w_total = w_total + (|H| - 1) * heat(H)
   // C is a set of coallocatable objects.
   C = ϕ
   loop a ∈ H according to layout order
      if objects in C U a are contiguous
         C = C U a
      else
         w_l = w_l + (|C| - 1) * heat(H)
         C = {a}
R_l = w_l/w_total

// fourth phase: relate (sub)optimal coallocation and the given
// layout cache miss reduction numbers.
R_l^o = (R_o - R_l)/(1 - R_l)
output R_l^o
```

Figure 3. Coallocation Algorithm.

analysis also ignores global and stack references.

2.2 Avoiding Cache Misses by Coallocation

Figure 1 shows a sequence of objects allocated by a program during execution[1]. Now consider Figure 2, which is an elaboration of Figure 1. Figure 2(a) reports the result of the hot data stream analysis. The stream $a_3b_5c_{10}$ covers *60%* of the reduced trace, stream $d_4b_{15}e_6$ covers *30%* of the reduced trace, and stream $e_6d_4a_{12}a_{14}$ covers *10%* of the reduced trace.

In the worst case (denoted by *worst-case scenario*), each stream data access can result in a cache miss. Then, from Figure 2(a) coallocating b_5 and c_{10} could save one cache miss for every occurrence of the stream $a_3b_5\ c_{10}$ if b_5 and c_{10} fit in one cache block, and coallocating a_3, b_5 and c_{10} saves two cache misses if these fit in one cache block (shown in *mr* column).

For a more realistic estimate of cache miss reduction due to coallocation, we compute the potential cache miss reduction for two other layouts over the *worst-case scenario*, and then compare these reductions with those obtained with our coallocation solution: (i) the existing layout produced by the current heap allocator (denoted by *current layout*), and (ii) the layout produced if all objects are allocated contiguously according to allocation order (denoted by *allocation order*). In this manner we compare our solution against both *current layout* and *allocation order.*

[1] Our current implementation "linearizes" the layout to the access pattern and ignores both object and cache block size. This increases spatial locality and the effectiveness of hardware prefetching.

2.3 Coallocatable Objects

Informally, members of a hot data stream *H* can be coallocated if all intervening allocations between members of *H* come from allocations sites other than the ones used to allocate members of *H*. Even if all members of *H* cannot be coallocated it may be possible to coallocate some of them; thus our algorithm (see Section 2.4) considers coallocatable objects in a substream of a hot data stream *H*.

Definition: *Objects in H are said to be* coallocatable *if for every program object* $x_t \notin H$ *(where* x_t *denotes an object that is allocated at site X by the t-th global allocation request in the program):*

$$min(T_{alloc}(H)) <= t <= max(T_{alloc}(H)) \text{ implies } X \notin S_{alloc}(H)$$

where, (i) $H = \{x^{j1}_{i1}, \ldots, x^{jk}_{ik}\}$ is a set of *k* (unique) objects,

(ii)x^{jm}_{im} denotes an objects allocated at allocation site X_{jm} by the i_m-th allocation request of the program,

(iii)$S_{alloc}(H) = \{X_{j1}, \ldots, X_{jk}\}$ is the set of allocation sites for objects in *H*, and

(iv)$T_{alloc}(H) = \{i_1, \ldots, i_k\}$ is the set of allocation request times for the objects in *H*.

If objects in *H* are coallocatable, then they can be placed contiguously in memory, if (i) allocation sites in $S_{alloc}(H)$ allocate in a designated memory region *M*, (ii) Memory region *M* allocates consecutive allocation requests contiguously, and (iii) objects allocated at sites not in $S_{alloc}(H)$ are allocated in a separate memory region.

In Figure 2(a) b_5 and c_{10} (that come from allocation sites *B* and *C*, respectively) are coallocatable since the intervening allocations $e_6d_7d_8d_9$ come from other allocation sites ({*D, E*}). d_4 and b_{15}, which are part of a hot data stream that covers *30%* of the reference trace (heat = 0.3) are not coallocatable since d_7 (and also d_8, d_9) is an intervening allocation from *H* that comes from *D*, which allocates d_4. We use *coallocation set* to denote the set of allocation sites of coallocatable objects; thus {*A, B, C*} is the coallocation set for coallocatable objects a_3, b_5, c_{10}. Enforcing this coallocation set saves two cache misses for every occurrence of the hot data stream $a_3b_5c_{10}$ according to the simple model presented in Section 2.2.

2.4 Basic Algorithm

Figure 3 shows the algorithm for coallocating hot data streams. Our goal is to find a coallocation strategy that maximizes cache miss reduction. In a first phase, shown in Figure 4, for every hot data stream, coallocation sets are computed along with the number of cache misses avoided if the coallocation set is enforced. This step involves an exponential exploration of all possible coallocations. However, this is tractable since hot data streams are usually small (and recall that we set the analysis to detect minimal hot data stream with length at least 2 and not exceeding 20). For example, in Figure 2(a) a_3, b_5 are coallocatable; this coallocation will eliminate one cache miss, and the weighted miss reduction for $S_{alloc}(\{a_3, b_5\})$ = {A, B} is 0.6. If several objects from the same site are included in a stream care must be taken not to double count

```
// get cache miss reduction for coallocation sets.
// ℋ is the set of hot data streams.
// heat(H) gives the normalized heat of H.
// coallocatable expresses the coallocatability criteria.
computeWeight (coallocatable, w)
for H ∈ ℋ
    w_total = w_total + (|H| - 1) * heat(H)
    // Compute weights for coallocation sets
    // corresponding to subsets of H.
    for H'⊆H
        if objects in H' are coallocatable
            w(H') = (|H'| - 1)* heat(H)
        else
            w(H') = 0
    // attribute weights for coallocation sets avoiding double
    // contributions, by computing the maximal
    // partition contribution for a coallocation set
    for each coallocation set C
        𝒫 = {H_i'|H_i', H_j'⊆C ∧ w'(H_i') > 0 ∧
             S_alloc(H_i') = S_alloc(H_j') ∧
             H_i' ⊆ H_j' => w'(H_j') = 0 }
    w'(C) = Σ_Hi'∈𝒫 w'(H_i')

// normalize weights.
for each coallocation set C
    w(C) = w'(C)/w_total
```

Figure 4. Computing weights for coallocation sets.

cache miss reductions. For example, e_6, a_{12} and e_6, a_{14} and e_6, a_{12}, a_{14} are all coallocatable and come from the same coallocation set {A, E}. To avoid double counting we use a polynomial algorithm to find the maximal partition of coallocatable objects that come from a coallocation set. Thus, in Figure 2(a) a weighted miss reduction for coallocation set {A, E} only accounts for coallocating e_6, a_{12}, a_{14}. Finally, we accumulate the potential miss reduction per coallocation set and then normalize it shown in Figure 2(b).

In order to maximize the benefits of coallocation sets, in a second phase, we compute a partition of the set of allocation sites, such that cache miss reduction is maximized. The partition problem is an instance of a known NP-hard problem, {*weighted set packing* (WSP)} defined as follows: Given a set S of m base elements and a collection $C = \{C_1, C_2, \ldots, C_n\}$ of weighted subsets of S with a weight function w, find a subcollection $C' \subseteq C$ of disjoint sets of maximum total weight

$$\Sigma_{(C'_i \in C')} w(C'_i)$$

In our case, S is the set of hot allocation sites, i.e., the set of sites allocating at least one object that participates in a hot data stream, C is the set of coallocation sets, and $w(C_i)$ is the normalized potential cache miss reduction. According to [7] the best approximation algorithm for the WSP problem is that of Halldorsson [12]. For $m = |C|$, this algorithm approximates WSP to within $\sqrt{m}$ of the optimal solution in time proportional to the time it takes to sort the weights. Although the approximation algorithm may yield poor results for large values of m, we show in Section 5 that this is not a significant problem for most of our benchmarks. This is because: (i) the number of hot allocation sites m is small, and (ii) Halldorsson algorithm is a greedy-style algorithm. Our empirical results show that in most cases much of the cache miss reduction benefit comes from a few disjoint coallocation sets; thus even for a large m these coallocation sets will be selected by the approximation algorithm, yielding most of the potential benefits.

In Figure 2(c) possible partitions of allocation sites are presented. For {A,B,C},{D,E} we get 76% cache miss reduction over the worst-case scenario. Applying Halldorsson's WSP algorithm indeed yields this partition. Figure 2(d) presents the layout of objects using the {A,B,C},{D,E} partition. Hot data stream $a_3b_5c_{10}$ is allocated contiguously, and d_4e_6 which participates in two hot data streams is also colocated[1]. The result of the second phase is an approximate coallocation solution (``WSP solution''), and a number R_o reflecting cache miss reduction over a worst-case scenario.

In a third phase, the potential cache miss reduction for the "current layout" and "allocation order" layouts discussed previously are computed and related to R_o. The resulting number R_o^l reflects the expected cache miss reduction over a given layout. Since in our example, R_l for both layouts is 0 as R_l is assumed to be the worst-case layout, the benefits for coallocating {A,B,C},{D,E} are 76% over both layouts.

3. COALLOCATING FIELDS

Object coallocation is most beneficial when the colocated objects are smaller than a cache block, though next-cache-line hardware prefetching provides benefits for larger objects as well. However, even if colocated objects are smaller than a cache block, coallocation at an object field granularity should produce larger benefits.

Changing the trace abstraction level, so references are abstracted to object field accesses rather than object accesses, permits applying the object coallocation algorithm described previously to achieve field coallocation. The resulting field coallocation solution requires that fields of an object are independently allocated. While such layouts (though not completely arbitrary field placement without high cost) can be enforced with compiler support in strongly typed languages such as Java, the implementation effort and cost for enforcement in languages such as C is likely to be prohibitive.

To address this, we express a field coallocation solution in terms of common data restructuring techniques such as field reordering, object splitting, and object merging. Such restrictions on field rearrangements yield a spectrum of coallocation solutions, which tradeoff ease of layout enforcement with larger cache miss rate reduction. Promisingly, empirical results (see Section 5) show that for our benchmarks, most of the benefits arise from a combination of these restructuring techniques.

3.1 Split and Merge Field Coallocation

For field coallocation we assume (for the moment) that an allocation request for an object with n fields, is split into n field allocation requests coming from n different allocation sites, one for each field. In addition, we assume independent allocation of fields and fields of an object may be scattered in the heap. Figure 5(a) shows an allocation site A allocating a 3-field object, and sites B, C allocating 2-field objects. A_1,A_2,A_3 are the corresponding field allocation sites, which independently allocate the fields of A.

[1] This layout also coallocated $a1_2$, $a1_4$ for ``free''. We could in principle attribute this coallocation to {A,B,C} as well.

Figure 5(b) shows the layout of 4 objects allocated at *A,B,C*. In Figure 5(c) the same layout is expressed at a field granularity, where ai_j denotes the *j*-th field global allocation request in the program allocated at field allocation site A_i (*i*-th field of an object allocated at *A*). Thus, object a_2 is allocated by three consecutive field allocation requests coming from A_1, A_2, A_3 allocating fields $a1_6$, $a2_7$, $a3_8$ respectively.

With this abstraction, we can adapt the coallocation algorithm presented in Section 2.4, with objects replaced by fields. Consider the example in Figure 7, which assumes the allocation sites of Figure 5(a). The two leftmost columns of Figure 7(a) show the result of hot data stream analysis on a field access trace. The stream $a1_1\ a3_3\ a1_6\ a3_8$ covers 80% of the reduced trace, and the stream $a2_2\ b1_4\ c2_{10}$ covers 20% of the reduced trace. The next four columns show the computed weights for field coallocation sets. This and the next step shown in Figure 7(b) for accumulating and normalizing weights of field coallocation sets is exactly as described in Section 2.2. The meaning of (R),(S),(M) is explained later in Section 3.2. In Figure 7(d) $s \wedge m$ method indicates the results of the general field coallocation solution. Since this solution can be expressed in terms of splitting objects to independently allocate their fields, together with merging fields of different objects at different sites, we call it *split and merge*. For this example, the *split and merge* solution is optimal. We use $R_l^{s \wedge m}$ to denote the potential cache miss reduction over a given layout ("current layout" or "current order").

This independent allocation of fields gives finer control over object layout and our coallocation algorithm may find a larger number of cache-conscious placement opportunities. However, this comes at the cost of maintaining the program semantics with the transformed layout, since for example, the compiler assumes that fields of an object are placed contiguously (or at least not at arbitrary locations).

3.2 Split or Merge Field Coallocation

This section presents an algorithm for coallocating fields by limiting object restructuring to simple techniques, such as field reordering, object splitting and object merging. The key restriction is that an object can either be split, or it can be merged with other objects, but it cannot participate in both. Thus, while the coallocation algorithm considers coallocation opportunities at the field granularity, the solution itself is enforceable at the object allocation site level, eliminating the need to transform an object allocation into multiple field allocations. Implementing the resulting coallocation solution is much simpler, at the cost of some lost opportunity for field coallocation.

Before presenting the algorithm we define some field coallocation terms:

(i) fields in a hot data stream are said to be *field reordering coallocatable* if field reordering suffices to guarantee coallocation of these fields. For example, in Figure 7(a), $a1_1$,$a3_3$ are field reordering coallocatable, since reordering the fields of *A*, placing the first and the third field together, ensures coallocation. In Figure 7 benefits attainable by reordering are marked by **(R)**.

(ii) fields in a hot data stream are said to be *object split coallocatable* if object splitting (either with or without field reordering) suffices to ensure coallocation of these fields. For example, in Figure 7(a), $a3_3$,$a3_8$ are field split coallocatable, since splitting the fields of *A*, while placing the third field of instances of *A* contiguously ensures coallocation (see Section 4 for a practical implementation of splitting an object as described). In Figure 7 benefits attainable by splitting an object are marked by **(S)**.

(iii) fields in a hot data stream are said to be *object merge coallocatable* if merging the types of the respective object allocation sites (either with or without field reordering) ensures coallocation of these fields. For example, in Figure 7(a), $a2_2$,$b1_4$ are object merge coallocatable, since merging the types allocated at *A* and *B*, and reordering the fields to place the second field of *A* together with the first field of *B* ensures coallocation (see Section 4 for a practical implementation of object merging). In Figure 7 benefits attainable by merging objects are marked by **(M)**.

We now describe our split or merge field coallocation algorithm. First, we change the coallocation criteria in the first phase of the basic algorithm (see Figure 4). Instead of requiring that objects in a hot data stream *H* are coallocatable, we require that objects in *H* are field reorder coallocatable, object split coallocatable, object merge coallocatable, as shown in Figure 6 (first phase). Next, after exploring simple placement opportunities at the level of field allocation sites, we use the WSP approximation to compute the benefit of using a simple placement technique at the object allocation site level, as described in the second phase part of Figure 6. Split and also reorder techniques are not applicable to more than one allocation site at a time (i.e., we do now allow combining several allocation sites and then splitting them). Merge is applicable only for two or more allocation sites that have not been split. However, the algorithm includes the benefits of simple reordering in the merge coallocation solution.

For example, the results of applying the different coallocation definitions to the hot data streams is shown in Figure 7(a) and (b) using **(R), (S), (M)** denoting field reorder benefits, object split benefits and object merge benefits respectively. Then in Figure 7(c), we show the results of applying WSP at the level of object allocation sites. For field reordering, only *A* has reordering benefits, by placing its first and third fields together. For object split, again only *A* has benefits by splitting objects coming from *A* so the first and third fields of these objects are placed contiguously

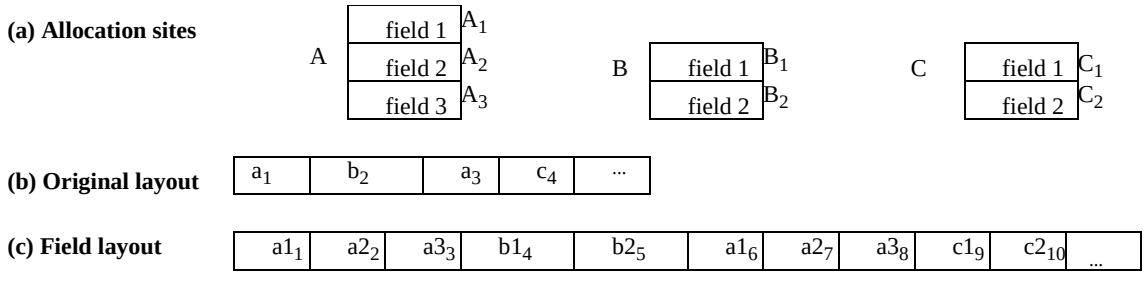

Figure 5. Expressing layout at field granularity.

```
// first phase: compute normalized weights for field coallocation
// sets according to reorder, split, merge coallocation conditions.
// computeWeight is given in Figure 4.
computeWeight(field reorder coallocatable,w_r)
computeWeight(object split coallocatable,w_s)
computeWeight(object merge coallocatable,w_m)

// second phase: approximate benefits for every placement
// technique at the granularity of object allocation sites.
// OBJ(X_i) gives the object allocation site for a given field
// allocation site, i.e., OBJ(X_i) = X.
// We also extend OBJ to sets of field allocation sites.
for each object coallocation set C^
   if |C| = 1 // compute split + reorder benefits
      C_rWSP = WSP(C,{<C,w_r(C)>|OBJ(C)=C})
      w_r(C) = Σ_{C∈C_rWSP} w_r(C)
      w_m(C) = w_r(C)
      C_sWSP = WSP(C,{<C,w_s(C)>|OBJ(C)=C})
      w_s(C) = Σ_{C∈C_sWSP} w_s(C)
      w_sVm(C) = max(w_r(C), w_s(C))
   else // |C| > 1 merge candidates
      C_mWSP = WSP(C,{<C,w_m(C)>|OBJ(C)=C})
      w_m(C) = Σ_{C∈C_mWSP} w_m(C)
      w_sVm(C) = w_m(C)

// third phase: find field coallocation solutions and show potential
// benefit over a given layout.
// R_l is potential cache miss reduction for a given layout.
ℜ_rWSP = WSP(OBJ(S_alloc(ℜ)),{<C,w_r>|w_r(C)>0})
R_l^r = (Σ_{C∈ℜ_rWSP} w_r - R_l)/(1 - R_l)
ℜ_sWSP = WSP(OBJ(S_alloc(ℜ)),{<C,w_s>|w_s(C)>0})
R_l^s = (Σ_{C∈ℜ_sWSP} w_s - R_l)/(1 - R_l)
ℜ_mWSP = WSP(OBJ(S_alloc(ℜ)),{<C,w_m>|w_m(C)>0})
R_l^m = (Σ_{C∈ℜ_mWSP} w_m - R_l)/(1 - R_l)
ℜ_sVmWSP=
WSP(OBJ(S_alloc(ℜ)),{<C,w_sVm>|w_sVm(C)>0})
R_l^sVm = (Σ_{C∈ℜ_sVmWSP} w_sVm - R_l)/(1 - R_l)
output R_l^r,R_l^s ,R_l^m ,R_l^sVm
```

Figure 6. Field Coallocation Algorithm.

(see Figure 7(e) for *A's* split layout). Finally, for merging objects, several opportunities exist. Merging either *A, B* or *A, C* or *B, C* gives 0.07 potential benefit, while merging A, B, C gives 0.14.

In Figure 6 (third phase part), the WSP approximation is once again applied for every placement technique to approximate the optimal disjoint subsets of object allocation sites, that are candidates for layout changes[1]. We denote by R^r_l, R^s_l, R^m_l reduction of cache misses over a given layout for field reorder coallocation, object split coallocation and object merge coallocation respectively.

[1] For object splitting and field reordering coallocation solutions WSP is actually the identity function since all the coallocation sets are of size 1, thus they are pairwise disjoint.

The results over a worst-case layout and over the layout *l* shown in Figure 5(c) are the same since R_l is 0. In Figure 7(d) *r* method shows the WSP solution for field reordering coallocation. By

(a) hot data streams: $H_1 = a1_1a3_3a1_6a3_8$ $H_2 = a2_2b1_4c2_{10}$

ht	strm	coallocatable objects	mr	coalloc-ation set	w'
0.8	H_1	$a1_1$, $a3_3$ (R)	1	$\{A_1,A_3\}$	0.8
		$a1_6$, $a3_8$ (R)	1	$\{A_1,A_3\}$	0.8
		$a3_3$, $a1_6$ (S)	1		
		$a1_1$, $a1_6$ (S)	1	$\{A_1\}$	0.8
		$a3_3$, $a3_8$ (S)	1	$\{A_3\}$	0.8
		$a1_1$, $a3_3$, $a1_6$ (S)	2		
		$a3_3$, $a1_6$, $a3_8$ (S)	2		
		$a1_1$,$a3_3$,$a1_6$,$a3_8$(S)	3	$\{A_1,A_3\}$	2.4
0.2	H_2	$a2_2$, $b1_4$ (M)	1	$\{A_2,B_1\}$	0.2
		$a2_2$, $c2_{10}$ (M)	1	$\{A_2,C_2\}$	0.2
		$b1_4$, $c2_{10}$ (M)	1	$\{B_1,C_2\}$	0.2
		$a2_2$, $b1_4$, $c2_{10}$ (M)	2	$\{A_2,B_1,C_2\}$	0.4

(b) Normalized miss reduction

coallocation set	accumulated miss reduction	normalized miss reduction
$\{A_1,A_3\}$(S)	2.4	0.86
$\{A_1,A_3\}$(R)	1.6	0.57
$\{A_1\}$ (S)	0.8	0.29
$\{A_3\}$ (S)	0.8	0.29
$\{A_2,B_1\}$ (M)	0.2	0.07
$\{A_2,C_2\}$ (M)	0.2	0.07
$\{B_1,C_2\}$ (M)	0.2	0.07
$\{A_2,B_1,C_2\}$ (M)	0.4	0.14

(c) WSP for fields at object allocation

tech	site(s)	field set packing	mr
r	A	$\{A_1,A_3\},\{A_2\}$	0.57
s	A	$\{A_1,A_3\},\{A_2\}$	0.86
m	A, B	$\{A_2,B_1\},\{A_1,A_3,B_2\}$	0.07
m	A, C	$\{A_2,C_2\},\{A_1,A_3,C_1\}$	0.07
m	B, C	$\{B_1,C_2\},\{B_2,C_1\}$	0.07
m	A, B, C	$\{A_2,B_1,C_2\},\{A_1,A_3,B_2,C_1\}$	0.14

(d) WSP field solutions

technique	set packing	miss red.
s/\m	$\{A_1,A_3\},\{A_2,B_1,C_2\},\{B_2,C_1\}$	1
r	A	0.57
s	A	0.86
m	A, B, C	0.14
s\/m	split A, merge B, C	0.93

(e) s\/m layout

{A split	$a1_1$ $a3_3$	$a1_6$ $a3_8$	...	a_{22}	a_{27}	...
{B, C} merge	$b1_4$ $c2_{10}$ $b2_5$ $c1_9$		...			

Figure 7. Coallocating Fields example.

simply reordering the fields of *A* (placing the first and third field as shown in Figure 7(b)) we get 0.57 potential cache miss reduction. WSP solution for *s* method shows that for splitting the objects at allocation site *A* (placing the first and third field together as shown in Figure 7(b)) we get 0.86 potential cache miss reduction. WSP solution for *m* method shows that for merging *A,B,C* we get 0.14 potential cache miss reduction.

Finally, the algorithm computes a hybrid solution, *split or merge*, where objects at an allocation site are either split, or participate in a merge with objects from other object allocation sites, or fields of these objects are just reordered. Thus, for every object coallocation set, the algorithm computes the maximal benefit obtained either by split, reorder or merge (denoted by $w_{s\backslash/m}$). Then applying WSP to object coallocation sets with $w_{s\backslash/m}$ (as shown in Figure 6, third phase) gives the desired coallocation solution.

In Figure 7(d) *s\/m* method shows the WSP solution for split or merge technique. Splitting *A* yields 0.86 potential cache miss reduction, and merging *B,C* gives 0.07 more potential, for a total of 0.93. Figure 7(e) shows the layout obtained by applying *split or merge* coallocation solution. The area between $a1_6$ $a3_8$ and $a2_2$ is reserved for placing the first and third fields of further allocations requests from site *A*.

We denote by $R_l^{s\backslash/m}$ the expected cache miss reduction of *split or merge* over a given layout (either "allocation order" or "current layout")We expect $R_l^r <= R_l^s <= R_l^{s\backslash/m} <= R_l^{s/\backslash m}$, and also $R_l^r <= R_l^m <= R_l^{s\backslash/m} <= R_l^{s/\backslash m}$.

4. IMPLEMENTING COALLOCATION

cmanal produces the coallocation solution that is enforced by **cminstr**, and a coallocation library, **comalloc**. **cmanal** uses Sequitur to compress a reference trace, which can be obtained with low-overhead using the technique described in [13]. It produces a context-free grammar representation that is efficiently analyzed to find hot data streams with their associated normalized heat value (see Section 2.1). **cmanal** applies the coallocation algorithm described in the previous sections to produce a collection of coallocation sets. Currently the reference abstraction granularity (object or field) is determined when the trace is produced, while the allocation context abstraction level (i.e., just the allocation site, or a calling context of some length *l*) is a tunable parameter.

cminstr is an instrumentation tool, based on a x86 binary-editing tool called Vulcan [18]. **cminstr** enforces the coallocation solution produced by **cmanal** by replacing the program's original heap allocation calls with calls to our coallocation library, **comalloc**. These comalloc calls are implemented by an independent dynamically linked library (**comalloc.dll)** that handles coallocation of objects or fields as described in Section 4.1. Then, the instrumented program running with **comalloc.dll** generates the coallocation solution layout. Note that the instrumentation tool updates debugging information in accordance with the modifications it makes to the target binary, which allows standard debuggers to be used with the optimized binary, if needed.

4.1 Comalloc Library

We enforce the coallocation solution by reserving a separate memory region, M_i for each coallocation set, C_i. Consecutive allocation requests from sites in the same coallocation set C_i, are assigned consecutive addresses in M_i. Allocation sites not in any coallocation set are assigned to heap region M_0.

We use the *heap layers* infrastructure [1] to implement this memory management policy. Heap layers permits managing several separate heaps, where each heap is independently managed, possibly with a different policy.

Each heap M_i is managed with **comalloc** functions, i.e, functions of the form **comalloc$_i$**, allocating memory from the respective heap (there is also support for *realloc*, *calloc* through *corealloc*$_i$, and *cocalloc*$_i$, respectively). Heap layers provides an easy way to set the characteristics and memory management of each heap, simply by changing its type defined by a mixin of templates providing different layers of heap functionality [1]; thus after enforcing a coallocation solution each heap M_i can be tuned independently.

In the current implementation, free operations are implemented by recording the owner heap of every object in its header. This is done by ensuring that every heap includes a **ownerHeap** layer as part of its type. When calling a **free**, the actual **free**$_i$ is dispatched to the owner heap of the object being freed. This ownership information is stored separately from the object to avoid reducing coallocation benefits. In addition, objects accesses are typically much more frequent than free operations.

4.2 Split, Merge using Comalloc

We use instance interleaving to split an object [19]. Instance interleaving splits object instances, such that frequently accessed instance field are laid out contiguously in memory. This is done by adding special padding fields to the object type definition and allocating the objects using an instance interleaving library, ialloc (described in [19]), which maintains the invariant of placing frequently accessed fields of object instances contiguously. We adapted the ialloc library to work with the heap layers infrastructure; thus a **iallocHeap** layer is used to enforce splitting an object.

Merging objects is done through combining the object type definitions (i.e., the combined type contains all the fields of the types being merged) and using a new heap layer **mergeHeap** for allocations. If *A, B, ... , K* are merged, upon allocation requests from these sites, **mergeHeap** determines whether a new object, large enough to store the combined object type, should be allocated, or if the last allocated combined object can be re-used to satisfy the allocation request. Consider an example where *A, B* are merged allocation sites, and allocation requests come from *A*, then *B* and again from *B*. **mergeHeap** will first return a new object **o** large enough to hold the combined type of *A, B*. The next allocation request from *B* will be returned the address of **o**, so that the *B* portion of **o** can be used. The third allocation request from *B* will allocate a new combined object, since the last allocated combined object was already used to satisfy the prior request from *B*.

4.3 Limitations

Our current implementation lacks access to type information and support for type manipulations. Thus, for example, field reordering is done by manual source code modification. Moreover, our instrumentation tool, **cminstr** relies on receiving the PC's of **malloc** calls to be replaced by **comalloc** calls. However, type manipulations affect the binary code, thus currently we have to adjust the allocation site PC's listed in a field coallocation solution. On the other hand, the object coallocation solution, which does not require type manipulations, is enforced automatically. In addition, splitting array elements is hard to enforce, since as noted in [19] it requires changes in the pointer arithmetic around array

Benchmark	hot sites	object	reorder	merge	split	split or merge	split and merge
boxsim	24	16	14	17	21	23	24
twolf	27	24	7	15	13	19	19
vpr	19	14	6	10	6	10	10
mcf	4	2	1	1	3	3	2
perl	17	12	2	8	5	11	11
foxpro	272	12	184	42	227	238	256

Table 1: Coallocation solution in terms of allocation sites.

expressions. In principle, a compiler can employ such layout assuming precise type information. We enforced array splitting manually (according to the coallocation solution) in one benchmark *mcf* to experiment with its benefits. For other benchmarks, we ignore array splitting suggested by the coallocation solution. Finally, the use of C benchmarks pose a problem with respect to type manipulation. As reported in Section 5, sometimes field reordering can crash an application due to hidden assumptions regarding structure layout. This problem is eliminated in strongly-typed languages, such as C# or Java, and techniques such as those described in [6][14] can be used to automatically implement the field coallocation solutions.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Methodology

We studied some memory-performance-limited programs that include vpr, twolf, mcf, and perl, from the SPECint2000 suite, boxsim, a graphics application that simulates spheres bouncing in a box, and, VisualFoxPro, a Microsoft database application. The program were instrumented with Microsoft's Vulcan tool to produce a data reference trace along with heap allocation information. The heap allocation information was used for trace abstraction at a object or field granularity depending on the experiment. We ran the SPECint2000 benchmarks on their test input and small inputs for boxsim and VisualFoxPro to generate traces for coallocation analysis. The traces were processed by **`cmanal`** to produce object/field coallocation solutions, which were enforced using **`cminstr`** and our **`comalloc`** allocation library. All performance improvements are reported using the train and ref inputs for the SPEC benchmarks and different larger inputs for boxsim and VisualFoxPro. Measurements of cpu time and hardware performance counters were carried out on a 2.8 Ghz Pentium4 processor with 1 GB of memory and a 512KB L2 cache running WindowsXP Professional. To minimize the effect of different allocator policies on runtime, we use the same allocator for allocating objects in the original benchmark and for allocating objects not in a coallocation set in the optimized benchmark. The results represent an average of five runs (variation was less than 3% across the runs).

5.2 Object Coallocation Results

Table 1 presents allocation sites characteristics of the benchmarks and the coallocations solutions. *hot sites* column presents the number of program allocation sites that allocate objects that participate in hot data streams. The next 6 columns present the number of sites participating in the corresponding coallocation solution. For object coallocation, the results indicate significant coallocation opportunity. Note that the number of hot allocation sites in VisualFoxPro is an order of magnitude larger than the other benchmarks, though the number of coallocation sets determined by the analysis is comparable. Figure 8 shows potential cache miss reduction numbers when coallocating at the granularity of object addresses, where the base (i.e., no improvement) is 0%. We experimented with using different allocation calling context abstractions, such as abstracting an object address to a calling context of length c leading to the allocation of that object, where c=1,2,3, but noticed no difference. Hence we used an allocation calling context of 1 for all subsequent experiments. For object-oriented languages, such as C# or Java, that make heavy use of container classes through standard libraries, the allocation calling context will likely be quite useful for distinguishing between different instances of these classes. In addition, our coallocation solution gives potential reduction in cache misses with respect to two alternative layouts, current layout and allocation order (see Section 2.2), but we do not notice any significant differences between these. Hence we compare only against current layout in subsequent experiments. The analysis time to determine the coallocation solutions from the access traces was reasonably low with the worst case (computing the field coallocation solution for VisualFoxPro) taking less than ten minutes.

Figure 8 indicates significant potential benefit for object level cache-conscious coallocation across the benchmarks. However, in *vpr* and *mcf* the potential cache miss reduction is attributed mainly (more than 90% of it in *mcf*) to coallocating larger objects (120 bytes and larger). Nevertheless, as we will see, the optimized placement still produces improvements in the presence of hardware prefetching. In *twolf* we get negative numbers, since in this case the WSP coallocation solution yields potential benefits over worst-case scenario, which are less than the benefits of the current layout over worst-case scenario. Our approximation algorithm which is bounded to $\sqrt{|\textit{hot-allocation-sites}|}$ of the optimal coallocation solution, is responsible for this. With the exception of twolf, the potential cache miss reduction indicate a probably un-achievable (approximate) best case bound for locality improvement.

Figure 9 shows executing time benefits from running the programs with optimized cache-conscious object coallocation layouts on two larger inputs (both different from the input used to generate the layout). In addition, we isolate the impact of hardware prefetching, which prefetches the next few sequential cache lines on a cache

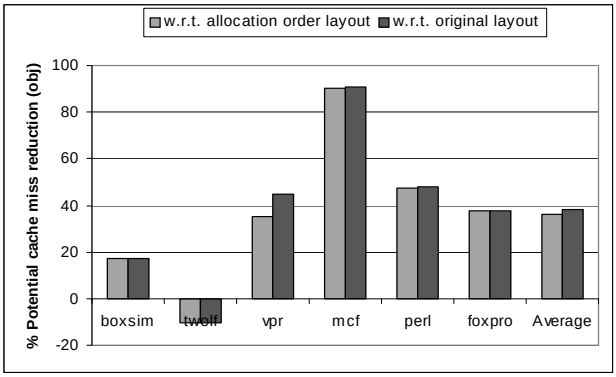

Figure 8. Potential cache miss reduction for obj. coallocation

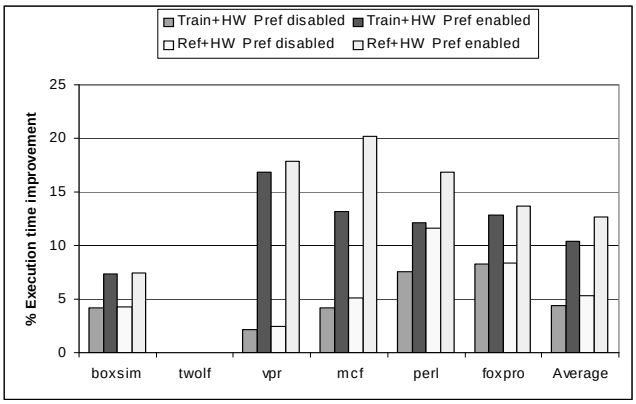

Figure 9. Benefits of automatic object coallocation.

miss, by disabling hardware prefetching for some experiments. Note that the base unoptimized configuration being compared against had hardware prefetching enabled.

The results in Figure 9 indicate that automatic object coallocation (in the presence of hardware prefetching) improves execution time by 13% on average for our set of benchmarks on their largest inputs. Without hardware prefetching, this improvement drops to 5%. Thus, object coallocation successfully "linearizes" the layout, improves its spatial locality, and consequently, makes hardware prefetching more effective. For vpr and mcf in particular, hardware prefetching is extremely effective as their coallocation solutions include larger objects. Our technique's effectiveness at improving VisualFoxPro by 14% suggests it can scale to large programs. On average the improvements are slightly larger on the program's ref inputs than on its train inputs. This is attributable to the difference in size of the data sets, which increases the importance and benefits of cache locality optimization.

5.3 Field Coallocation Results

Table 1 presents allocation sites characteristics of the field coallocations solutions. More sophisticated coallocation techniques create more coallocation opportunity. For example, field reordering finds far less coallocation opportunities than *split or merge*. The numbers for *split and merge* in terms of program allocation sites are quite similar to *split or merge* numbers indicating that much of the benefits of the general field coallocation solution can be attained with a more restricted solution, with VisualFoxPro being the sole exception.

For each benchmark we compare our 5 strategies for field coallocation. Figure 10 shows the results. For *boxsim*, *twolf*, and *mcf*, *split or merge* and *split* are comparable to *split and merge*. However, since *split and merge* technique requires sophisticated source transformations, we employ *split* to these benchmarks. In *perl*, merge suffices to obtain the coallocation benefits. In *vpr* we get negative numbers due to our approximation algorithm similar to the case for object allocation in *twolf*. VisualFoxPro gets maximum benefit from *split and merge* but *split or merge* in not too far behind.

Our current framework lacks type information. Thus, some type manipulation suggested by the coallocation solution may not be feasible, or may be hard to employ. Other problem arise from the fact that type changes are not always feasible in a non type-safe environment. We encounter the following problems: (i) We assume fields are up to 4-bytes long. Thus, reordering part of a 8-byte field is not feasible. This could happen for fields of type *double*. This

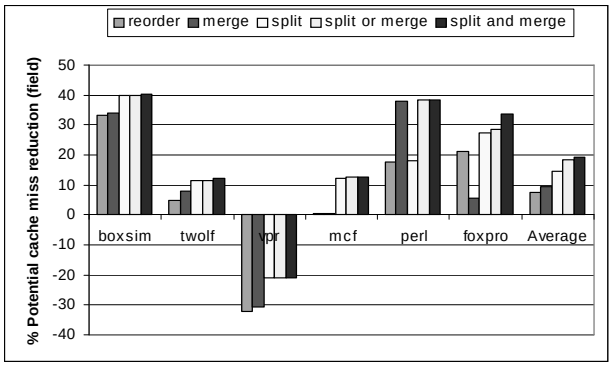

Figure 10. Potential cache miss reduction for different field coallocation strategies.

happens in *boxsim* and *twolf*. (ii) Several allocation sites may be associated with the same type. Thus, a coallocation algorithm assuming distinct allocation sites allocate distinct types may suggest conflicting type changes. In *boxsim* and *twolf* two allocation sites allocating the same type are candidates for a merge in a coallocation solution. (iii) splitting array elements is hard to employ, since as noted in [19] it requires changes in the pointer arithmetic around array expressions. Array splitting is suggested in the coallocation solution for *twolf*, *boxsim* and *mcf*. We employed array splitting only in *mcf* due to the large potential benefit. (iv) finally, in a non type-safe environment there may be hidden assumptions on the layout of a structure. In *boxsim* and *twolf* arbitrary field reordering causes these benchmarks to crash.

Given these constraints, we only attempted implementing the field coallocation by hand for *boxsim*, *twolf*, and *mcf*. In *boxsim*, a major part of the field coallocation solution was not enforceable (e.g., type manipulation problems), thus we were only able to employ one object split and field reordering to two types, obtaining only around 7% of potential benefit. The benefits for *mcf* come from splitting a hot array, and in *twolf* layout changes include mostly splitting small objects (12-88bytes). Thus, for *twolf* and *mcf* the potential and the actual cache miss improvement numbers are quite similar. With these changes**,** *boxsim* shows no speedup due to our problems enforcing the layout. In *twolf* we get a 22% speedup, and the time spent in cache misses is reduced by around 27%. For *mcf* we get a further 12% speedup (over object coallocation), reducing time spent in cache misses by a further 20%. These numbers indicate the potential benefit of field level coallocation. However, the effort required to obtain them and the hit-and-miss process involved suggests automatic implementation in a type-safe language, such as Java or C#.

6. CONCLUSIONS

We present a novel profile-based analysis aimed at coallocating contemporaneously accessed hot heap objects. The analysis attempts to obtain cache-conscious coallocation of a hot data stream *H* by isolating and combining allocation sites of *H* such that intervening allocations coming from other sites are separated. Our extended analysis aims for coallocation at object field granularity, generalizing common restructuring techniques, such as field reordering, object splitting, and merging. Our initial results indicate that programs possess significant coallocation opportunities. Automatic object coallocation produces average execution time improvements of 13% in the presence of hardware prefetching for the program's largest inputs. Hand-implemented field coallocation solutions for two of the benchmarks produced additional improvements (12% and 22%) but the effort involved

suggests implementing an automated version for type-safe languages, such as Java and C#.

ACKNOWLEDGEMENTS

We are grateful to Jim Larus, Ben Zorn and the anonymous referees for their comments on an earlier draft of this paper.

REFERENCES

[1] E. Berger, B. Zorn, and K. McKinley. "Composing high-performance memory allocators." In *ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[2] B. Calder et. al. "Cache-conscious data placement." In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1998.

[3] T.M. Chilimbi. "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality." In *ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[4] T.M. Chilimbi. "On the stability of temporal data reference profiles." In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Aug. 2001.

[5] T.M. Chilimbi, J.R. Larus, and M.D. Hill. "Cache-conscious structure layout." In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[6] T.M. Chilimbi, J.R. Larus, and B. Davidson. "Cache-conscious structure definition." In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[7] P. Crescenzi et al. "A compendium of NP optimization problems." ww.nada.kth.se/~viggo/problemlist/compendium.html

[8] N. Gloy et. al "Procedure placement using temporal ordering information." In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Dec. 1997.

[9] R. Gonzalez et al. "Energy dissipation in general-purpose microprocessors." In *IEEE Journal of Solid State Circuits, 31(9)*, Sept. 1996.

[10] S. Guyer and K. McKinley. "Finding your cronies: static analysis for dynamic object colocation." In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2004.

[11] X. Huang et al. "The garbage collection advantage: Improving program locality." In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2004.

[12] M. M. Halldorsson. "Approximations of weighted independent set and hereditary subset problems." In *Journal of Graph Algorithms and Applications, Vol. 4*, 2000.

[13] M. Hirzel and T.M. Chilimbi. "Bursty Tracing: A Framework for Low-Overhead Temporal Profiling." In *Workshop on Feedback Directed and Dynamic Optimizations (FDDO)*, Dec. 2001.

[14] T. Kistler and M. Franz. "Automated data-member layout of heap objects to improve memory-hierarchy performance." In *Transactions on Programming Languages and Systems (TOPLAS)*,volume 22, 2000.

[15] E. Petrank and D. Rawitz. "The hardness of cache-conscious data placement." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan 2002.

[16]S. Rubin, R. Bodik, and T.M. Chilimbi. "An Efficient Profile-Analysis Framework for Data-Layout Optimizations." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan 2002.

[17] M. Seidl and B. Zorn. "Segregating heap objects by reference behavior and lifetime." In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 1998.

[18] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. In *MSR-TR-2001-50*, 2001.

[19] D. Truong, F. Bodin, and A. Seznec. "Improving cache behavior of dynamically allocated data structures." In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1998.

[20] Y. Zhong et al. "Array regrouping and structure splitting using whole-program reference affinity." In *ACM SIGPLAN'04 Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[21] C. Lattner and V. Adve. "Automatic pool allocation: Improving performance by controlling data structure layout in the heap." In *ACM SIGPLAN'05 Conference on Programming Language Design and Implementation (PLDI)*, 2005.