2009 18th International Conference on Parallel Architectures and Compilation Techniques

Region Based Structure Layout Optimization by Selective Data Copying

Sandya S. Mannarswamy
Hewlett-Packard,
Bangalore, India
sandya.s.mannarswamy@hp.com

R. Govindarajan
SERC, Indian Institute of Science
Bangalore, India
govind@serc.iisc.ernet.in

Rishi Surendran
Hewlett-Packard
Bangalore, India
rishi.surendran@hp.com

Abstract **—** As the gap between processor and memory continues to grow, memory performance becomes a key performance bottleneck for many applications. Compilers therefore increasingly seek to modify an application's data layout to improve cache locality and cache reuse. Whole program Structure Layout [WPSL] transformations can significantly increase the spatial locality of data and reduce the runtime of programs that use linked list-based data structures, by increasing the cache line utilization. However, in production compilers WPSL transformations do not realize the entire performance potential possible due to a number of factors. Structure layout decisions made on the basis of whole program aggregated affinity/hotness of structure fields can be sub-optimal for local code regions. WPSL is also restricted in applicability in production compilers for type unsafe languages like C/C++ due to the extensive legality checks and field sensitive pointer analysis required over the entire application. In order to overcome the issues associated with WPSL, we propose Region Based Structure Layout (RBSL) optimization framework using selective data copying. We describe our RBSL framework, implemented in the production compiler for C/C++ on HP-UX IA-64. We show that acting in complement to the existing and mature WPSL transformation framework in our compiler, RBSL improves application performance in pointer intensive SPEC benchmarks ranging from 3% to 28% over WPSL.

1 INTRODUCTION

As the gap between processor and memory continues to grow, memory performance becomes a key performance bottleneck for many applications. Compilers are challenged to improve an application's cache locality and reuse. Standard locality improving transformations [22, 23, 24], such as loop transformations, for improving cache locality are maturing. However their applicability is limited to array and loop intensive scientific codes. For codes with pointer based data structures and irregular pointer chasing access patterns, these transformations are not applicable [3]. Therefore, in order to improve cache locality and cache reuse, compilers increasingly seek to modify an application's data layout. Data layout transformations [1, 2, 3, 4, 5, 6, 10, 11, 12, 14, 15] are a class of optimizations that seek to improve the memory performance of applications by controlling the way data is arranged in memory. Data layout transformations include global variable layout [10], stack layout, heap layout [1] and structure layout optimizations [2, 3, 4, 5, 6, 11, 12, 14, 15]. Our focus in this paper is on structure layout.

There are many techniques that optimize the placement of fields within a structure. These include structure splitting [3], structure peeling [4, 11, 12], field reordering [3, 4, 11, 12] and dead field removal [4]. These techniques use various heuristics to improve locality. For instance, a common heuristic is to simply separate hot and cold fields so that cold fields are not unnecessarily brought into the cache as they decrease the cache-line utilization. However the runtime data access pattern might not be consistent with the access frequency distribution of the fields. In other words, not all hot fields of a structure are accessed together in a region of program. Hence many of the structure layout techniques use the notion of affinity between fields accessed [3, 4, 5, 6, 12]. Fields f1 and f2 have a strong affinity to each other if they are often accessed close to each other. Placing fields that have stronger affinity together in the same cache line would improve spatial locality.

Structure layout optimizations are essentially whole program by nature since they operate on global types which are visible and passed across multiple functions. These analyses identify the structure types that can be modified safely. Affinity and hotness analyses are performed on these structures to determine the splitting decisions. Once a decision is made to transform a particular type, it is applied across the entire program by modifying all references to that type. Because of their potential to improve application performance dramatically, structure layout transformations have been the focus of considerable research of late.

However whole program structure layout transformations (WPSL) in production compilers do not realize the full performance potential possible due to a number of factors. Applications often exhibit different affinity behavior across the fields of the structure in different program regions for the same data structure type. Two different hotspots in an application can be accessing two different sets of fields of a hot data type. If the data layout framework bases its splitting decisions by aggregating the affinity/hotness information across all regions, then it will split the structure by combining the affinity information of both hotspot regions. Such a global decision may be sub-optimal for each of the local hotspot regions. On the other hand, if it decides to split the structure type for the entire application based on field access affinity information for one region only, it will result in poor locality and cache line utilization for other region.

Previous research work on field placement and data structure splitting appears in [3, 5, 17, 14]. Early work in this area uses error-prone human inspection of C applications to make sure that the transformation is safe [3, 5, 14]. In a program written in a pointer-rich language, such as C and C++, splitting a structure type might impact

1089-795X/09 $25.00 © 2009 IEEE
DOI 10.1109/PACT.2009.43

the whole program because of aliasing relationships. Therefore, a compiler needs to modify all the affected references when it splits a particular structure type. Applying a type-safe optimization to a type-unsafe language without a proper safety assurance mechanism is unacceptable in production compilers. Zhao et al show that whole program field sensitive inter-procedural pointer analysis is needed in order to safely perform structure splitting for the application [12].

Hundt et al show that extensive legality checks are required for the WPSL transformation [4]. For the transformation to be legal, the compiler needs to ensure that the type is not passed to opaque library calls, and that there are no dangerous type casting transformations on the type. Since legality checks need to be satisfied throughout the whole program, it reduces the potential applicability of the transformation. Even if there is a single opaque call site to which the type that is selected for structure splitting is exposed, the compiler is prevented from performing the transformation.

All these disadvantages owe their origin to the single fact that the existing structure layout frameworks make WPSL decisions. That is, the layout decision needs to be based on and applied to the entire program. Every reference to structure type selected for layout transformation needs to undergo the same layout transformation. It is not possible to have different splitting decisions for different local regions. For instance, we cannot decide to split a type locally for one region and not split it in another code region. Fields selected to be placed in the hot and cold part cannot be different for different code regions. The requirement of uniformity of the layout decisions across the whole program prevents WPSL transformations from extracting the maximum performance possible.

In order to overcome the above disadvantages associated with WPSL, we propose a new local or region based structure layout (RBSL) transformation framework. RBSL is complementary and can co-exist with WPSL. RBSL transformation phase uses data copying (partial structure cloning) to enable local data layout decisions that are best suited for each local region, which can be different from the WPSL decision. Thus our RBSL framework trades off the data copying overhead with the increased cache line utilization for that local region. Although there has been prior work on using copying to reduce conflict misses in the case of array based programs [22, 27], to the best of our knowledge, ours is the first work in selectively applying data copying to enable region based structure layout automatic optimization for linked data structures in type unsafe languages like C/C++.

We have implemented RBSL in the HP-UX IA-64 production compiler for C/C++ [7]. We show that working in complement to the existing and mature WPSL transformation framework in the compiler, our new optimization improves application performance in certain SPEC benchmarks by up to 28%. More importantly, this work establishes that RBSL as an effective region-based transformation which facilities the application of data layout transformation, perhaps locally, on structures that were not amenable under the WPSL framework.

In Section 2, we provide the necessary background and motivation for RBSL optimization. Section 3 introduces data utilization metrics and describes the local region based data layout transformation. In Section 4, we briefly describe the steps involved in RBSL transformation. We present our experimental evaluation results in Section 5. We discuss related work in Section 6 and conclude with a short summary in Section 7.

2 BACKGROUND

In this section we present the necessary background on WPSL optimization. Subsequently we motivate the need for RBSL optimizations with the help of a few examples.

2.1 Structure Layout Optimizations

There are regions of code in an application which have poor utilization of data, in terms of the ratio of the amount of data actually used by the application to the amount of data fetched. Structure layout optimizations attempt to improve the data utilization for such delinquent code by modifying the structure layout. It typically splits the structure type into 2 parts. The hot part contains only those fields which are actually used in that region and the unused fields are moved to the cold part. Consider the following nested loop from 179.art benchmark shown in Fig. 1

```
for (tj=0; tj < numf2s; tj++)     {
    Y[tj].y = 0;
        if ( !Y[tj].reset )     {
            for (ti=0;ti<numf1s;ti++)     {
                Y[tj].y += f1_layer[ti].P* bus[ti][tj];
            }
        }
}
```

Figure 1. A loop from 179.art

Each access to structure element f1_layer[ti] of type f1_neuron brings in one cache line of data, out of which only one field (P) is used. This results in poor data utilization for the above loop as only 8 bytes of data out of the 64 bytes (L1 D-cache line size) of the cache line data fetched is actually used. Such poor data utilization can lead to wasted memory bandwidth, poor cache utilization and low TLB hit rate. If structure splitting [3] creates a new array of structures f1_layer_P, containing only field P, then it results in 100% cache-line utilization for f1_layer_P for the above loop.

Structure layout optimizations are inter-procedural by nature as they operate on global types that are visible and passed across multiple functions. Structures are identified as whether they can be modified safely and attributes are collected (such as whether a type has been dynamically allocated or whether they are local or global variables of that type). These attributes are consulted to determine applicable transformations. Affinity and hotness analyses are performed across the entire program to determine the

final transformations [3, 4]. Once a decision is made to transform a particular structure, it is applied across the entire application by modifying all references to that type.

2.2 Limitations of WPSL Decisions

Many applications, however, contain frequently executed code regions in which the groups of fields accessed from a hot data type T are different. For instance, f1 and f2 can be the set of fields of type T that are accessed in one region whereas f1 and f4 are the set of fields accessed in another region. This happens, for instance, in the SPECint2006 benchmark 429.mcf. The two loops shown in Fig 2 and 3 access different hot fields of the structure 'arc', which is one of the hottest data structures of the application.

The first loop in Fig. 2 accesses the fields nextout, nextin, tail and head from the 'arc' structure, while the second loop shown in Fig. 3 accesses the fields head, tail, ident and cost. These are two distinct program regions where the affinity groups for the structure 'arc' are different.

```
arc = net->arcs;
for (stop = (void *) net-> stop_arcs; arc < (arc_t *) stop; arc++ )   {
      arc->nextout = arc->tail->firstout;
      arc->tail->firstout = arc;
      arc->nextin = arc->head->firstin;
      arc->head->firstin = arc;
}
```

Figure 2. Loops from mcf with different hot fields

```
arc = net->arcs;
for ( ; arc < stop_arcs; arc += nr_group) {
      if (arc->ident > BASIC)  {
            red_cost = arc->cost-arc->tail->potential +
                        arc->head->potential;
            ……..
      }
}
```

Figure 3. Loops from mcf with different hot fields

Existing data layout frameworks typically employ whole program splitting decisions. Let us consider the effect of this on the data structure 'arc' shown in Fig. 2 and Fig. 3. If the data layout framework bases its splitting decisions by aggregating the affinity information across all program regions, it will make the decision to split the structure 'arc' into two parts, a hot part consisting of the fields cost, tail, head, ident, nextin and nextout and a cold part consisting of the fields flow and org_cost. Though such a layout will bring together the high affinity fields of both loop regions, it results in sub-optimal decisions for the hottest loop region L2, where only the 4 fields head, tail, cost and ident are accessed. Thus WPSL decisions may not always achieve the maximum performance potential that are realizable by data layout transformations that are specific to each local region.

Moreover the access patterns exhibited by a particular data structure can be different for different regions. This can also happen when an array of structures is traversed with different stride patterns in different local regions, or a data structure such as a tree is traversed either depth first or breadth first in different local regions. A WPSL optimization cannot optimize for such varying access patterns for different regions. This brings up the possibility of having a region based structure layout framework which can decide on an optimal data layout for each local region, based on the regional affinity of the fields and access patterns for that data structure specific to that region.

Extensive legality checks are required for the WPSL transformation. For a transformation to be legal, the compiler needs to ensure that the type is not passed to opaque library calls and there are no dangerous typecasting transformations on the type.

TABLE I. DATA TYPES AMENABLE UNDER WPSL

Benchmark	***Number of types***	***Number of types eligible for WPSL***
400.perlbench	101	12
401.bzip2	6	0
403.gcc	384	28
429.mcf	4	3
445.gobmk	53	9
450.sjeng	8	4
464.h264ref	39	7
462.libquantum	3	2
456.hmmer	30	5

The potential applicability of WPSL transformation becomes restricted since the legality checks need to be satisfied throughout the whole program. Even if there is a single opaque call site to which the type selected for structure splitting is exposed, the compiler is prevented from performing the transformation. This results in leaving potential performance on the table since all possible data layout opportunities cannot be realized. In Table I, we present the number of types which pass the whole program legality checks in our WPSL framework, for a set of SPECint2006 benchmarks, compared to the number of data structure types present in the benchmark. The WPSL framework employed in our compiler uses a set of legality checks such as "cast not applied", "sizeof operator not applied" and "structure not passed to external shared library" to filter candidates for WPSL. We find that nearly 80% of the types become ineligible for transformation as they fail whole program legality checks. Another main reason for considering a region based structure layout framework comes from the fact that any WPSL decision requires that all pointers in the application code which can point to the type being split need to be updated to the newly created split type after the transformation. Since the compiler needs to identify all pointers that point to the

transformed type, in order for the transformation to be correct, an accurate alias analysis is required which is often expensive. Last, most of the existing automatic WPSL frameworks cannot handle array of data structures allocated non-contiguously [11], unless there is support for automatic pool allocation [16]. This also contributes to WPSL not being able to address all of the structure layout transformation opportunities available in the application.

2.3 Region Based Structure Layout (RBSL)

In order to enhance the efficacy of structure layout transformations by mitigating some of the disadvantages which are associated with the WPSL transformation, we propose a new local, or region based structure layout transformation framework. RBSL can co-exist with the existing WPSL framework and is complementary to it. RBSL uses data copying to enable local data layout decisions that are best suited for each local region, which can be different from the WPSL decision. The framework trades off the data copying overhead with the increased cache line utilization for that local region resulting in improving the overall application performance. In the case of RBSL, candidates for RBSL need to pass the legality checks only over the local region and not over the whole program.

3 Region Based Structure Layout Optimization

In this section we discuss the metrics for quantifying the data utilization of local regions and how these metrics can be used for RBSL candidate selection.

3.1 Data Utilization Metrics

In order to quantify our discussion on poor data utilization for a local region of code, we introduce the following definitions: Data Volume is in general, total volume of data accessed over a local code region such as a loop. The volume of data intended by the programmer to be accessed and the volume of data actually accessed by the program can differ considerably. Hence we define two quantities namely Programmer Intended Data Volume (PIDV) and Actually Accessed Data Volume (AADV). Programmer Intended Data Volume (PIDV) is the volume of data actually intended by the programmer to be accessed over the region, via explicit references to the data by the application code in that region. PIDV can be defined for different granularities of the code region such as functions, loops etc. In this work, we focus on those inner loops which traverse over a list/array of structures.

We can estimate programmer intended data volume accessed over the loop region L which traverses over a list of structures of type T denoted as *PIDV(L,T)* to be equal to

$$PIDV(L,T) = N_L * (SF_u + \sum SF_{cf} * p_f)$$

where NL is the number of loop iterations for L, SFu is the aggregated size of all unconditionally accessed fields of type T in each iteration by the application code, SFcf is the size of field f in type T that is conditionally accessed in a loop iteration, and pf is probability of accessing field f in an iteration.

We also define the term Actually Accessed Data Volume over a local region (AADV) as the volume of data the application actually ends up fetching (into the cache) to satisfy the programmer intended data accesses over that region. Given a loop region L which traverses over a list of structures of type T, we denote *AADV(L, T)* to be the volume of data actually fetched by the application in order to meet *PIDV(L,T)* required in that region. For example, consider a loop region L iterates over a list of structures with each structure element size 64 bytes, where each element of the list is aligned to the cache line and occupies only a full cache line size. If only one integer word field *f* is accessed in the loop, then the application actually ends up fetching 64 bytes for each reference of the field, whereas the programmer intends to use only 4 bytes out of the 64 bytes accessed. We estimate

$$AADV(L,T) = N_L * C_L * CS$$

where C_L is the number of cache lines that were brought in per iteration to meet the *PIDV(L,T)* for *L,* and *CS* is the cache line size.

Next we define *PIDV(L)* to be the total programmer intended data volume for all types *T* in loop *L*. Thus

$$PIDV(L) = \sum T \ (PIDV(L,T))$$

Similarly, *AADV(L)* is the total data volume actually brought in by the loop L. Assuming different data types accessed in a loop do not fit within a cache line,

$$AADV(L) = \sum T \ (AADV(L,T))$$

AADV(L) is always greater than or equal to the size of *PIDV(L)*. The closer AADV is to the PIDV, better the data utilization of the local region under consideration. We define the data utilization ratio (DU) for a local region as the ratio of PIDV to AADV for that region.

$$DU(L) = PIDV(L) / AADV(L)$$

We define Data Volume OverHead for a given region as AADV – PIDV for that region.

$$DVoh(L) = AADV(L) - PIDV(L)$$

If the *DU* ratio is closer to 1, then *DVoh* is smaller and we say the local region is better behaved with respect to data utilization. We define those loop regions with high *DVoh,* and hence low DU ratio, as *delinquent regions*.

We explain the above metrics with the help of our example innermost loop shown in Fig. 1, from the application 179.art. For every structure element accessed in the loop, the application ends up fetching 64 bytes of data, but utilizes only 8 bytes of data. Assuming a loop iteration count of 10000 from the profile data, we compute

the *PIDV* as 80000 and the *AADV* as 640000 for this loop region. The *DU* ratio is 0.125 and the data volume overhead is 560000. In our work, we choose a threshold of 0.7 and below for *DU* ratio to mark a loop as delinquent loop. Hence this loop is a delinquent loop.

Next, we demonstrate that there exist delinquent loops in programs even after applying a reasonably good WPSL transformation. Our production compiler [7] employs a WPSL framework which includes various optimizations such as structure splitting, structure peeling and field reordering. After all the WPSL optimizations have been employed, we added a phase to identify delinquent loops as defined above in order to understand the potential for further local structure layout transformations. We ran the analysis on a suite of SPEC benchmarks and the results are shown in Table II. We see that even after mature WPSL transformations have been employed on the code, eight out of the eleven benchmarks still exhibit delinquent loops.

TABLE II. DELINQUENT LOOPS IN BENCHMARKS AFTER WPSL

Benchmark	***Number of Delinquent Loops found post-WPSL***	***Number of Data Types involved in Delinquent Loops***
400.perlbench	3	1
401.bzip2	1	1
403.gcc	7	3
429.mcf	6	1
445.gobmk	12	5
456.hmmer	0	0
458.sjeng	4	2
462.libquantum	0	0
468.h264ref	0	0
471.omnetpp	3	1
473.astar	3	2

Therefore we target our region RBSL transformation on such delinquent loop regions to improve their data utilization. In order to determine the candidates for RBSL we need to quantify the potential gain due to RBSL. The Data Volume OverHead (*DVoh*) has an impact on application's execution time since there is a finite cost associated with each unit of data that is accessed by the application. This cost is due to multiple factors such as (a) the actual cost of data access for each unit of data (b) the impact of the amount of data accessed on the cache misses, memory bandwidth and DTLB etc. Assuming an average fixed unit cost for each byte of data accessed, we estimate the Data Volume Overhead Cost as,

$$DVohc(L) = DVoh(L) * \text{average cost for each unit of data accessed}$$

We approximate the average cost of each unit of data accessed by the value of *k* cycles. Hence

$$DVohc(L) = DVoh(L) * k \text{ cycles}$$

As a conservative approximation, we use the value of k equal to 1 (In practice, k can be many cycles).

Data volume overhead cost is borne by the application and hence impacts its execution time. Hence the positive impact of a particular structure layout on the application performance can be approximated by the reduction in data volume overhead cost achieved by a given layout for a region. Let L_{WPSL} be the WPSL layout used for the type T in the loop region L, if WPSL is possible for L and $L_{current}$ be the current layout. Note that if WPSL is not possible for T, then L_{WPSL} would be same as $L_{current}$. Similarly, let L_{RBSL} represent the RBSL-based optimized structure layout applied to L for the type T. The estimated reductions in the data overhead due to WPSL and RBSL optimizations over the original loop are

$$DVoh(L_{current}) - DVoh(L_{WPSL}) \text{ and}$$

$$DVoh(L_{current}) - DVoh(L_{RBSL})$$

respectively. Finally we can estimate benefit of the RBSL over WPSL in terms of cycles using the *k* value as;

$$(DVoh(L_{WPSL}) - DVoh(L_{RBSL})) * k$$

which equals,

$$(AADV(L_{WPSL}) - AADV(L_{RBSL}) * k).$$

Hence we denote the benefits due to applying RBSL as

$$Benefits(L_{RBSL}) = ((AADV(L_{WPSL})) - AADV(L_{RBSL})) * k \text{ cycles}$$

Now using these metrics, we try to answer the question of when the compiler needs to employ RBSL on top of a WPSL framework. For each delinquent region R, we compute the benefits of applying the local layout decision compared to the WPSL decision. If the benefit computed is positive for a given region R, then R is considered eligible for RBSL transformation. As already mentioned, if WPSL is not possible for a type due to either legality checks not passed for the whole application or pointer analysis declaring the type to be not transformable, then we compute the benefits compared with respect to the original loop $L_{current}$.

3.2 Overheads associated with RBSL

The above discussion on benefits of RBSL does not include any overheads introduced by the RBSL transformation itself. We perform RBSL by selective data copying. The data layout of the application is adjusted by making a copy of the original data such that the data utilization of the delinquent loop is improved and the

application can amortize or even overcome the cost of copying overhead. The copy has a better (reduced) footprint in the cache since it is packed and for cases of non-sequential access pattern which varies from the allocation pattern of the structure list, the copy can be set up to follow the access pattern for the delinquent loop which helps improve the data utilization for that local region. We denote the original data structure as the original structure list and the copy instantiated by RBSL as the copy structure list.

Although, at first glance, copying appears to be a very simple idea, it can be difficult to

a) ensure coherence between original structure list and the copy list and
b) estimate at compile time, whether the benefits of improved data utilization for the delinquent loop outweighs the cost of copying.

We define the cost involved in setting up the copy initially as the copy *SetupCost* and any cost involved in keeping the copy in synch with the original data structure as *SyncCost*. Both these costs have to be taken into account when computing the net benefits due to RBSL. The overall *COPYCOST* is the sum of *SetupCost* and *SyncCost*. The copy setup cost is nothing but the AADV cost incurred over the delinquent loop for the original (whole program) structure layout.

$$\text{CopySetupCost} = \text{AADV}(L_{WPSL}) * k \text{ cycles} .$$

We approximate the synch cost by upper bounding it by the copy set up cost. Hence the overall copy cost becomes

$$\text{CopyCost} = 2 * \text{CopySetupCost} .$$

All elements of the structure list over which the delinquent loop traverses must be brought into the cache once, in order to set up the copy list. Hence the extra runtime overhead incurred in setting up the copy must be offset completely by the improved data utilization of the transformed loop. Otherwise the RBSL enabled by data copying can degrade application performance. In order to ensure this criterion conservatively, for recovering the cost of copying incurred by the transformation, we use the simple filtering criterion that the delinquent loop region is nested inside an outer loop. For ease of reference in further discussion, we refer to the inner loop as the delinquent loop (DL) and we refer to the outer loop enclosing the delinquent inner loop as the ancestor loop (AL). Since the delinquent loop is nested inside the ancestor loop, we can estimate the RBSL benefits over the ancestor loop as

$$\text{Benefits}_{AL}(\text{RBSL}) \text{ excluding the copy cost} = \text{Benefits}_{DL}(\text{RBSL}) * N_{AL}$$

where N_{AL} is the number of iterations of the ancestor loop. Therefore the net benefits of RBSL over the ancestor loop region when we account for the CopyCost is

$$\text{NetBenefits}_{AL}(\text{RBSL}) = \text{Benefits}_{AL}(\text{RBSL}) - \text{CopyCost} = \text{Benefits}_{DL}(\text{RBSL}) * N_{AL} \text{ CopyCost}.$$

Recall that,

$$\text{Benefits}_{DL}(L_{RBSL}) = ((\text{AADV}(L_{WPSL})) - \text{AADV}(L_{RBSL})) * k$$

Using this in the above equation,

$$\text{NetBenefits}_{AL}(\text{RBSL}) = ((\text{AADV}(L_{WPSL}) * (N_{AL} - 2) - \text{AADV}(L_{RBSL}) * N_{AL}) * k.$$

And we apply RBSL only if the $\text{NetBenefits}_{AL}(\text{RBSL})$ as computed above is positive.

3.3 Granularity of Region for RBSL

The granularity of region considered for structure layout transformation can vary, being innermost loop level, function level or inter-procedural. WPSL considers the whole program as a region and makes layout decisions which represent one end of the spectrum. The other end of the spectrum is our RBSL framework which considers each individual delinquent loop region as the candidate for layout decisions. It is possible to consider a region granularity between these two points and consider combination of delinquent regions for transformation. In general, one should consider all possible *r*-combinations of regions and need to choose the most appropriate one. Note that when *r* equals *n* and if all regions are selected for the combination, the layout decision becomes WPSL. When *r* equals 1, then the RBSL decision is per delinquent loop region. In this paper we focus on single delinquent loop regions, enclosed by an Ancestor Loop. We defer the question of considering all possible *r*-combinations of regions where the copy cost can be shared, and hence redundant copies can be avoided, for future work.

4 IMPLEMENTATION OF RBSL TRANSFORMATION IN OUR COMPILER

Next, we describe our compiler infrastructure and the implementation of the RBSL transformation.

4.1 Our Compiler framework

We have implemented the RBSL transformation in the SYZYGY [7] high level optimizer for the HP-UX IA-64 production C/C++ compilers. Our compiler employs an Inter-Procedural WPSL phase which includes various structure layout transformations such as structure splitting, peeling and field reordering [4]. RBSL happens after all the transformations identified by the WPSL optimization phase have been carried out on the code. Hence it targets those delinquent loops which were not amenable to WPSL or which did not benefit fully from WPSL.

4.2 Region Based Structure Layout Algorithm

The basic algorithm for RBSL using selective data copying consists of identifying the delinquent loops,

performing legality checks on them and code transformation.

4.2.1 Identification of delinquent loops

We use our optimizer's loop recognition phase to build a loop structure graph. For each loop, our optimizer iterates over the basic blocks and collects the field references for the structure type which is traversed in that loop. For each loop region L, it computes the PIDV and AADV values using the profile information for the loop iteration counts and conditional branch probabilities as mentioned in Section 2.1. Nested loops with a DU ratio of 0.7 and below are marked as delinquent loops and for each such delinquent loop, its ancestor loop is also identified. These delinquent loops constitute the basic set of RBSL candidates for further checking and transformation.

4.2.2 Legality and Profitability Checking

Our WPSL framework has an exhaustive set of legality checks to ensure layout of structure layout transformation [4]. We use the same set of legality checks in RBSL. However in our case, they are applied only over the local region and not over the entire application. We also perform legality checks over the ancestor loop region to ensure that we can correctly identify all updates to the original structure list. In cases where the structure list elements can escape the ancestor region, the loop is discarded from the candidate list. For amortizing the cost of copying incurred by the RBSL transformation, we use the simple filtering criterion that the delinquent loop region is nested inside an outer loop. We then apply the NetBenefits computation as described in Section 3.2 to identify the candidates profitable for RBSL transformation.

4.2.3 Code Transformation

RBSL transformation includes the following steps:

(i) Creation of a new structure type containing the hot fields from the information obtained in 4.2.1 for the delinquent candidate loop

(ii) Instantiation of the copy list at the entry point to the ancestor loop

(iii) Replacing the references in the delinquent loop body from the original structure list to the copy list

(iv) Insertion of code to add updates to the copy list, at points of update to the original structure list in the ancestor loop body to maintain the original list and copy list in sync; and

(v) Insertion of code to free the copy list in the post-pad of the ancestor loop.

5 Experimental Evaluation

5.1 Experimental Methodology

We used SPEC2000 and SPEC2006 C/C++ applications as the set of benchmarks for our experiments and evaluated the performance of our RBSL framework in HP-UX IA-64 production compiler for C/C++. We compiled the benchmarks at the highest optimization level (level 4 with IPO) with RBSL enabled. Many of the structure layout opportunities present in the application are already addressed satisfactorily by the WPSL framework enabled at optimization level 4 in our compiler. We find that structure layout optimization opportunities in benchmarks like 177.mesa, 188.ammp, 300.twolf. 462.libquantum and 458.sjeng are transformed by the WPSL framework in our compiler. RBSL identifies additional structure layout transformation opportunities in the 6 benchmarks listed in Table III.

We report performance results only for those benchmarks in which RBSL identified candidates for transformation in Table III. RBSL had no impact on the other SPEC benchmarks since no candidates were identified for RBSL transformation in them. We also evaluated RBSL on two proprietary applications namely HP-UX operating system kernel and our standard system library 'libc'. We report these results in Table IV. Though 181.mcf (from SPEC2000) is also transformed by RBSL, we do not include it in our results since the same code region is transformed in 429.mcf (from SPEC2006) also. The baseline for our performance comparison is a SPEC base configuration with reference input sets and using the HP-UX compiler's non-profile based heuristics for the branch frequencies and loop iteration counts. We report results with RBSL enabled and without RBSL enabled on the base configuration, to show the extra performance that is extracted by RBSL over WPSL. All results were obtained on an HP rx2600 server with a 1500 MHz Intel Itanium 2 processor, 6 GB of memory, and 6 MB of last level cache.

5.2 Performance Comparison

In Table III we show, for each the benchmark, the structure modified by RBSL in the benchmark and the performance improvement in terms of % improvement in execution time.

TABLE III. Performance Improvement of RBSL over WPSL

SPEC BM	*Structure modified by RBSL*	*Improvement in Execution time (%)*	*Reduction in Dcache Miss latency (%)*	*Reduction in DTLB Miss (%)*
mcf	Arc_t	28.5	5.94	36.12
milc	Site	10.2	7.71	1.02
gobmk	String_data	4.2	3.32	1.6
moldyn	Mol_t	17.3	22.31	25.78
omnetpp	cMessage	3.8	4.59	3.24
art	Fl_neuron	11.6	5.12	14.3

We also report the data cache miss latency cycles reduction along with reduction in the number of data TLB

misses. The data cache and data TLB miss data were obtained using the performance profile tool HP-Caliper by sampling the IA-64 hardware performance counters [8].

The performance improvements range from 3% to 28% for these benchmarks. In the case of 429.mcf, our WPSL framework already performs structure splitting for the 2 structure types 'node' and 'arc'. The structure 'arc' has 8 fields, and WPSL splits the fields cost, tail, head, ident and flow into the most frequently accessed part (hot part) and the rest of the fields in to the cold part during structure layout. However WPSL layout still performs poorly in the delinquent region identified by RBSL, in the function 'primal_bea_mpp'. This region has poor cache line utilization due to non-sequential stride. For every access to the fields of arc in this loop, 128 bytes of data is brought in, out of which only 16 bytes corresponding to the 4 fields used in this loop are actually accessed by the application. There is also no spatial locality in this loop since the next 'arc' element that is accessed is not sequential, but is separated by a stride value equal to the variable 'nr_group'.

RBSL identifies the above delinquent region as a candidate for transformation. It creates a new structure type which contains only the 4 fields accessed in this region. When it sets up the copy, it uses the stride information obtained from the compiler analysis for this delinquent region to set up the copy array such that consecutive elements in the copy array are those which are apart by the stride value in the original structure. Hence the delinquent region is transformed from a region of no-spatial locality to a region of high spatial locality and there is no wastage of cache line since only the fields accessed in the region are brought into the cache. Hence RBSL improves performance for 429.mcf by 28% over WPSL.

CPU stalls of 433.milc have a very large data cache component, about 75%. The major data structure is 'site' whose size is about 2K bytes. In case of 433.milc, WPSL is unable to transform the type due to legality checks not passing over the entire program for this type. RBSL identifies 4 delinquent regions where only one field of the structure type 'site' is accessed over a loop. By copying the accessed field in each 'site' structure to an array of structures with each structure only containing this field as its member, we can improve the spatial locality for this delinquent region. For instance, RBSL identifies a delinquent region where only the field 'tempmat1' is accessed from the site structure over all the sites. The size of tempmat1 field is 144 bytes. Hence 2 cache lines (each of size 128bytes) of data are brought in, out of which only 144 bytes are actually intended to be used by the programmer. This results in a low cache line utlization of only around 57%. RBSL transforms this delinquent region to improve the cache line utilization to 100%.

In case of 445.gobmk, RBSL identifies 2 delinquent regions where there is traversal of the structure type 'string_data'. RBSL splits the fields which are accessed in those regions from string_data. In 471.omnetpp, RBSL identifies a delinquent region where an array of structures 'cMessage' is traversed. The delinquent region contains references only to the fields arrival_time, priority and insert_order. RBSL transformation improves overall performance by 3.8%. In case of moldyn, RBSL identifies 3 delinquent regions over the traversal of structure type 'Mol' and transforms them resulting in performance improvement of 17.26%. Performance results for HP-UX kernel and HP-UX libc are shown in Table IV.

TABLE IV. PERFORMANCE IMPROVEMENT FOR HP-UX

Other Applications	*Structure Transformed by RBSL*	*Improvement in Exec. Time over WPSL*
HP-UX kernel	Structure A	1.4%
HP-UX libc	Structure B	3.2%

Because of the proprietary nature of HP-UX kernel and standard library source code, we omit the names of the structures and refer to them as structures A and B respectively. Structure A is one of the hottest structures of the kernel, having 190 fields, containing data belonging to different subsystems. WPSL is not able to transform this structure due to legality considerations over the entire HP-UX kernel. RBSL identified 2 delinquent regions where only one field was accessed and transformed the loop regions to obtain 1.4% improvement for the SPEC 057.sdet benchmark which is used for HP-UX kernel performance testing. We also compiled HP-UX standard library libc for finding opportunities with RBSL. WPSL does not find any candidates in standard library 'libc' due to legality constraints. RBSL identified 2 delinquent regions in the memory allocation routines involving the structure B. Applying RBSL gave a performance improvement of 3.2% for 'libc' when tested with the memory allocator performance benchmark [26].

We find that though the number of delinquent loops identified is much larger as found in Table II, only a few of them are transformed by RBSL due to RBSL's filtering criterion that the delinquent loop needs to be a nested loop. We are investigating whether this criterion can be relaxed.

5.3 Copying Overhead

For the benchmarks transformed by RBSL, we measured the overheads due to setting up and maintenance of the copy of the original data structure. To measure this overhead, in this experiment, RBSL inserts the copy and sync operations, but does not transform the delinquent loop region field references to access the copy data structure. The copying overheads measured as the

performance degradations over our baseline are shown in Table V.

TABLE V. COPY OVERHEAD

Benchmark	*copy overhead*
429.mcf	1.29%
433.milc	3.2%
471.omnetpp	3.87%
445.gobmk	1.1%
472.moldyn	2.1%
179.art	0.9%

We find that copying overheads range from 0.9% to 3.8%. Currently our RBSL framework does not do any specific optimization for reducing the copying overhead. It is possible to reduce the copying overhead by delaying the sync-up of original data structure until the point of use of the original data structure after the exit from the delinquent region (lazy sync-up) and by chunking the copying instead of inserting point-updates. We plan to investigate this as part of our future work.

6 RELATED WORK

The area of research most closely related to this work is the area of automatic data transformations. Chilimbi et al. first used structure splitting to improve data locality [3]. Rabbah and Palem split structured data in C programs by allocating objects in large chunks where structure fields were stored in separate arrays [17]. Chilimbi later improved structure splitting using the frequency of data sub-streams called hot-streams [9]. Zhong et al. defined a model to measure the closeness of references in a memory trace, known as reference affinity and show how it can be used for structure splitting and array regrouping [14]. Although they perform structure splitting in a compiler they assume that the language is type-safe and use programmer intervention to ensure safety. The Forma framework for array grouping and structure layout automatically and safely reshapes single-instantiated arrays [12]. Curial et al [11] combine structure splitting with automatic pool allocation [16], and use a comprehensive field sensitive pointer analysis to ensure safety.

Many of the structure layout frameworks mentioned above employ WPSL decisions. WPSL transformations have also been discussed in detail in [4, 5, 11, 12, 23]. Hundt et al [4] show that the applicability of WPSL transformations is limited and propose using a semi-automatic tool to address some of the layout opportunities not amenable for WPSL. However the semi-automatic tools require code changes by the programmer unlike RBSL which is performed automatically by the compiler. The applicability of RBSL is wider compared to WPSL since the legality checks need to be satisfied only over the delinquent region targeted for transformation. RBSL is fully automatic and does not need any programmer intervention. RBSL is complementary to the existing WPSL transformations and can co-exist with them.

Alternatives to the automatic transformations described above are transformations that require programmer intervention or special allocator libraries [28, 18], whereas RBSL does not require any special libraries. There has been work on loop transformations where copying has been proposed to aid loop tiling in order to avoid conflict misses [22, 27]. Yi et al [30] use data copying based on standard array dependence analysis for improving the data layout of array based computations.

Chilimbi and Larus [30] used the copying garbage collection [GC] mechanism for object movements guided by locality information, aimed at type safe languages with GC support. Our work uses a purely compile time analysis for RBSL transformations aimed at type unsafe languages like C/C++ which do not support automatic GC. There has been work on runtime data layout and data relocation [20, 25. 28]. However these require special OS techniques [28] or special hardware [20].

7 CONCLUSIONS AND FUTURE WORK

In this paper we have proposed Region Based Structure Layout optimization for improving cache utilization and locality. Our paper proposes, for the first time, local or region based approach which is especially attractive for structures that are not amenable for transformation under WPSL. We have implemented RBSL in our production C/C++ compiler for HP-UX IA-64 and evaluated its performance for certain SPEC benchmarks and HP-UX proprietary applications. We showed that RBSL working complementary to a mature WPSL framework can help improve performance from 3% to 28% in a set of benchmarks. We plan to investigate RBSL for multithreaded applications as part of our future work. We also plan to investigate whether a WPSL scheme which is aware of a down-stream RBSL phase can make better layout decisions. Profitability analysis for RBSL candidate selection can be inaccurate when loop counts cannot be determined at compile-time. We plan to address this in future-work, using loop multi-versioning with RBSL applied selectively at runtime based on iteration count.

ACKNOWLEDGEMENTS

We like to thank our team members in the IA-64 optimizer group for their help and support. We particularly like to thank Teresa Johnson and Kaushik Rajan for their comments/suggestions. We would also like to extend our thanks to the anonymous reviewers; their feedback helped greatly to improve the quality of this paper.

REFERENCES

[1] B. Calder, C. Krintz, S. John, and T. Austin. 1998. Cache-conscious data placement. In ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems, NY, USA, pp. 139–149, ACM Press, 1998.

[2] T. M. Chilimbi, B. Davidson, and J. R. Larus. 1999. Cache-conscious structure definition. In PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, NY, USA, pp. 13–24, ACM Press, 1999.

[3] T. M. Chilimbi, M. D. Hill, and J. R. Larus. 1999. Cache-conscious structure layout. In PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, NY, USA, pp. 1–12, ACM Press, 1999.

[4] R. Hundt, S. Mannarswamy, and D. Chakrabarti. 2006. Practical structure layout optimization and advice. In CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, (Washington, DC, USA), pp. 233–244, IEEE Computer Society, 2006.

[5] T. Kistler and M. Franz. 2000. Automated data-member layout of heap objects to improve memory-hierarchy performance. ACM Trans. Program. Lang. Syst., vol. 22, no. 3, pp. 490–505, 2000.

[6] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. 2004. Array regrouping and structure splitting using whole-program reference affinity. In PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, (New York, NY, USA), pp. 255–266, ACM Press, 2004.

[7] S. Moon, X. D. Li, R. Hundt, D. R. Chakrabarti, L. A. Lozano, U. Srinivasan, and S.-M. Liu. 2004. Syzygy - a framework for scalable cross-module ipo. In CGO '04: Proceedings of the international symposium on Code generation and optimization, (Washington, DC, USA), p. 65, IEEE Computer Society, 2004.

[8] R. Hundt. 2000. HP Caliper: A framework for performance analysis tools. IEEE Concurrency, vol. 8, no. 4, pp. 64–71, 2000.

[9] T. M. Chilimbi and R. Shaham. 2006. Cache-conscious coallocation of hot data streams. SIGPLAN Not., vol. 41, no. 6, pp. 252–262, 2006.

[10] N. McIntosh, S. Mannarswamy, and R. Hundt. 2006. Whole program optimization of global variable layout. In PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, (Seattle, WA, USA), IEEE Computer Society, 2006.

[11] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S.Cui, R. Silvera, and R. Archambault. 2008. MPADS: memory-pooling-assisted data splitting. In *Proceedings of the 7th international Symposium on Memory Management* (Tucson, AZ, USA, June 07 - 08, 2008). ISMM '08. ACM, New York, NY, 101-110.

[12] P. Zhao, S. Cui, Y. Gao, R. Silvera, and N. J. Amaral. 2007. *Forma*: A framework for safe automatic array reshaping. *ACM Trans. Program. Lang. Syst.* 30, 1 (Nov. 2007).

[13] E. Petrank and D. Rawitz. 2005. The hardness of cache conscious data placement. *Nordic J. of Computing* 12, 3 (Jun. 2005), 275-307.

[14] Y. Zhong, M. Orlovich, X. Shen and C. Ding. 2004. Array regrouping and structure splitting using whole-program reference affinity. *SIGPLAN Not.* 39, 6 (Jun. 2004), 255-266.

[15] M. Franz and T. Kistler. 1998. Splitting data objects to increase cache utilization. Tech. Report ICS-TR-98-34, Dept. of Information and Computer Science, Univ. of California, Irvine, Irvine, CA, Oct.

[16] C. Lattner, and V. S. Adve. 2002. Automatic pool allocation for disjoint data structures. In ACM SIGPLAN Workshop on Memory System Performance (Berlin, Germany). ACM, New York, 13–24.

[17] R. Rabbah and S. Palem. 2003. Data remapping for design space optimization of embedded memory systems. ACM Trans. Embed. Comput. Syst. 2, 2, 186–218.

[18] D. N. Truong, F. Bodin, and A. Seznec. 1998. Improving cache behavior of dynamically allocated data structures. In PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, (Washington, DC, USA), p. 322, IEEE Computer Society, 1998.

[19] M. Hirzel. 2007. Data layouts for object-oriented programs. *SIGMETRICS Perform. Eval. Rev.* 35, 1 (Jun. 2007), 265-276.

[20] X. Huang, Z. Wang, and K.S. McKinley. 2001. Compiling for the Impulse Memory Controller. In Proceedings of the 2001 international Conference on Parallel Architectures and Compilation Techniques (September 08 - 12, 2001). PACT. IEEE Computer Society, Washington, DC, 141-150.

[21] E. Raman, R. Hundt and S. Mannarswamy. 2007. Structure layout optimizations for multithreaded programs. In CGO '07: Proceedings of the International Symposium on Code Generation and Optimization, pages 271–282, Washington, DC, USA, 2007. IEEE Computer Society.

[22] M. S. Lam, E Rothberg, and M. E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. In Proc. ASPLOS-IV, 1991.

[23] K.S. McKinley, S. Carr, and C. Tseng. 1996. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.* 18, 4 (Jul. 1996), 424-453

[24] D F Bacon, S L Graham, and O J Sharp. Compiler transformations for high-performance computing. ACM Computing Surveys, 26:346–420, 1994.

[25] C. Ding and K. Kennedy. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. *SIGPLAN Not.* 34, 5 (May. 1999), 229-241. [26] Chuck Lever and David Boreham. malloc() performance in a multithreaded linux environment. In Proceedings of the USENIX Annual 2000 Technical Conference, June 2000.

[26] O. Temam, E. D. Granston, and W. Jalby. 1993. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (Portland, Oregon, United States). Supercomputing '93. ACM, New York, NY, 410-419.

[27] C. Luk and T. C. Mowry. 1999. Memory forwarding: enabling aggressive layout optimizations by guaranteeing the safety of data relocation. *SIGARCH Comput. Archit. News* 27, 2 (May. 1999), 88-99.

[28] Y. Feng and E. D. Berger. 2005. A locality-improving dynamic memory allocator. In Proceedings of the 2005 Workshop on Memory System Performance (Chicago, Illinois, June 12 - 12, 2005). MSP '05. ACM, New York, NY, 68-77.

[29] T. M. Chilimbi and J. R. Larus. 1998. Using generational garbage collection to implement cache-conscious data placement. In Proceedings of the 1st international Symposium on Memory Management (Vancouver, British Columbia, Canada, October 17 - 19, 1998). ISMM '98. ACM, New York, NY, 37-48.

[30] Q. Yi. 2005. Applying Data Copy to Improve Memory Performance of General. Array Computations. Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, 91-105.