Practical Structure Layout Optimization and Advice

Robert Hundt, Sandya Mannarswamy, Dhruva Chakrabarti

Java, Compilers, and Tools Laboratory
Hewlett-Packard Company
{rhundt, dhruva}@cup.hp.com, sandyam@india.hp.com

Abstract

With the delta between processor clock frequency and memory latency ever increasing and with the standard locality improving transformations maturing, compilers increasingly seek to modify an application's data layout to improve spatial and temporal locality and to reduce cache miss and page fault penalties. In this paper we describe a practical implementation of the data layout optimizations Structure Splitting, Structure Peeling, Structure Field Reordering *and* Dead Field Removal, *both for profile and non-profile based compilations.*

We demonstrate significant performance gains, but find that automatic transformations fail for a relatively high number of record types because of legality violations or profitability constraints. Additionally, we find a class of desirable transformations for which the framework cannot provide satisfying results. To address this issue we complement the automatic transformations with an advisory tool. We reuse the compiler analysis done for automatic transformation and correlate its results with performance data collected during runtime for structure fields, such as data cache misses and latencies. We then use the compiler as a performance analysis and reporting tool and provide insight into how to layout structure types more efficiently.

1. Introduction

The delta between processor clock speed and memory latency continues to grow. Compilers are challenged to improve an application's cache locality and reuse. The standard locality improving transformations, such as loop transformations, are maturing and their applicability domain is limited to array and loop intensive scientific codes. For codes with pointer based data structures and irregular, pointer-chasing access patterns, these transformations don't apply. Therefore, in order to improve cache locality and cache reuse, compilers increasingly seek to modify an application's data layout.

This paper focuses on structure layout and placement on the heap. Many of the proposed methods dealing with record types have characteristics which make them unsuitable for commercial compilers: Some aren't fully automated [21][4], some are profile based [8][18][2][12], and some are program trace based [23][16]. The analysis and compile time can be prohibitive, such as for the trace based methods, or the usage patterns aren't adopted well by the community, such as for the methods using profiles. Some of the proposed methods assume type-safety [3][12], which in the case of C/C++ is rare in practice.

We present a practical framework for structure layout optimizations, such as structure splitting, structure peeling, dead field removal and field reordering, which we have implemented in the SYZYGY high level optimizer for the HP-UX C/C++ and FORTRAN compilers [15] for Intel Itanium ©. The framework is practical as it is simple, effective in finding optimization opportunities, and causes little compile time overhead (which is important for commercial compilers) at the expense of analysis accuracy.

We find that the number of transformable types increases drastically with an improved and more expensive analysis, but because of profitability filters most of these types are still not transformed. Additionally, there is a class of desirable transformations for hot and non-affine clusters of structure fields for which the framework cannot provide satisfying results.

To address these issues we develop a fast, compiler based advisory tool, which combines static compiler analysis with data collected at runtime. It generates as output annotated structure definitions providing insights into potentially more effective structure designs.

Our main contributions in this work are:

- Development and evaluation of a non-profile based heuristic that uses field affinity and hotness based on inter-procedurally propagated estimated edge weights for its layout decisions.

- Description and evaluation of a low-overhead, practical implementation in a commercial compiler which trades analysis accuracy for compile-time performance without losing any transformation opportunities (for our set of benchmarks).
- Development of a compiler-based performance advisory tool which combines static analysis with runtime measurements to guide in structure layout decisions.

This paper is organized as follows. In section 2 we briefly present our compiler framework and describe the actual transformations in detail before discussing the legality analysis. We explain the effects of a potentially more precise analysis. Then we outline the profitability analysis and compare several techniques we experimented with.

In section 3 we describe the advisory tool and show how we use the existing mechanism for feedback directed compilation effectively to attribute data cache events to structure fields. Then, by the means of example, we suggest possible uses of the presented data. We discuss related work in section 4 and conclude with a short summary in section 5.

In the following, the term *type* always refers to a record type, unless stated otherwise. The abbreviation *FE* denotes the front-end portion of the high-level optimizer and not the compiler front-end performing language parsing.

2. Framework

The framework has been implemented in the SYZYGY high level optimizer [15] for the HP-UX Itanium compilers, which offer a command-line option `-ipo` to enable inter-procedural optimizations. With this option, object files are emitted containing an intermediate representation of the input (IELF files). At link time, a dynamically loaded linker plug-in identifies the presence of IELF files and launches the inter-procedural optimizer, which performs inter-procedural analysis (IPA) and transformations, before writing the results back into IELF files in a temporary directory. It then creates a Makefile and invokes the utility "make" to parallelize the back-end and code generation. All this happens transparently to the user.

Structure layout optimizations are inter-procedural by nature. Cases where file- or procedure-local types can be modified are covered by the inter-procedural infrastructure. Types are identified which can be modified safely and attributes are collected (such as whether a type has been dynamically allocated or whether there are local or global variables of that type). These attributes are consulted to determine applicable transformations. Affinity and hotness analyses are performed to determine the final transformations.

Following the SYZYGY design philosophy we seek to push as much functionality as possible into the parallelizable front-end (FE) and back-end (BE) and to minimize work done in the monolithic inter-procedural analysis phase (IPA).

- The FE performs the bulk of the legality analysis and collects summaries for the profitability analysis in IPA.
- IPA aggregates the legality and affinity summaries, performs legality and type escape analysis, profitability analysis, and employs the heuristics. If types are to be split it emits control information for the BE.
- The actual transformations are performed in the BE

As we show in the next sections, these design decisions result in low compile time overhead, but lead to conservative analysis results.

2.1. Transformations

Structures can be modified in a variety of ways. In the following paragraphs we describe the four methods we have implemented and discuss some alternative implementations. The starting point for the descriptions is a dynamically allocated array of structures as shown in Figure 1 (a). Each structure has two interleaved hot and cold fields.

Structure Splitting – This transformation breaks a given structure into two (or more) pieces and inserts link pointers to allow addressing of all parts of a type via a pointer to it's root part. The transformation can be graphically illustrated as in Figure 1 (b).

Structure Peeling – For certain cases, types can be split without having to insert link pointers. The term structure peeling has been introduced by [8] for this special case of splitting. Instead of link pointers, new variables or pointers are created in either global or local scope to point to pieces of a structure.

For example, the SPEC2000 floating point benchmark 179.art [20] has a dynamically allocated array of structures containing only floating point fields (and a non-recursive pointer). The result of the dynamic allocation is assigned to a global pointer variable P; no other local or global pointers or variables of that type exist. The transformation breaks the type into multiple record types, each containing only one field corresponding to a field in the original type. The single dynamic allocation site is transformed into multiple allocations and multiple global pointer variables P_i are created to store the results of the allocations. All original accesses to a structure field via the global pointer P are transformed into accesses via one of the newly created pointers P_i. This transformation is graphically illustrated in Figure 1 (c).

Dead Field Removal – The transformations dead field removal is wrapped into the two previous transformations. If a type can be transformed and has at least one dead field, the

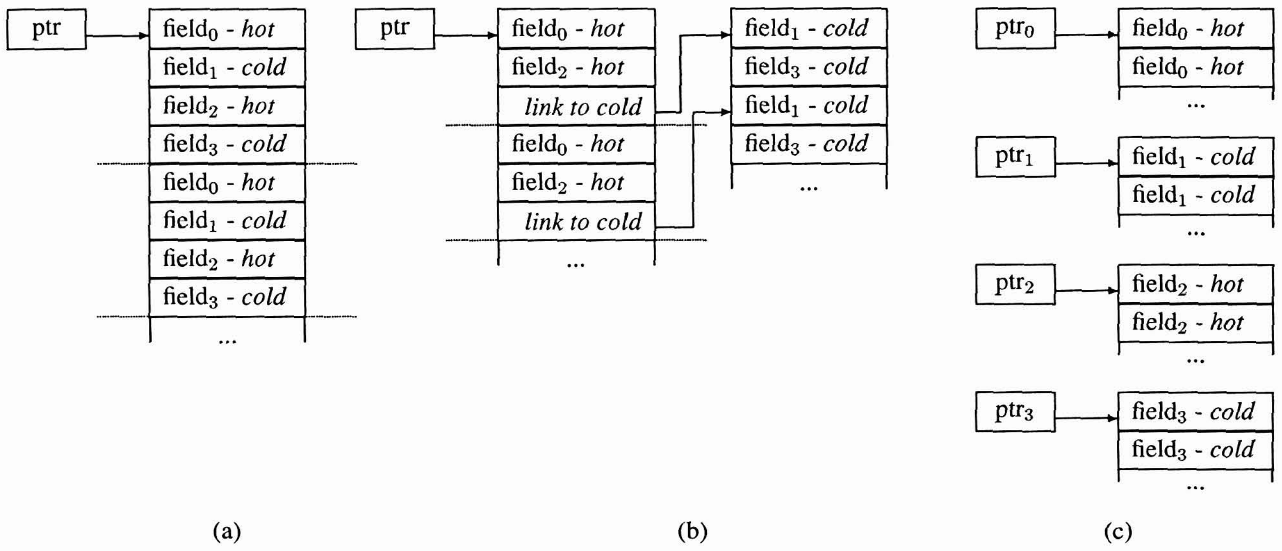

Figure 1. An array of record types (a), after splitting (b), and after peeling (c)

structure is split following the heuristics. Only live fields are moved into the newly created hot section. The cold section can be empty.

We distinguish between *unused* fields, for which it is sufficient to modify their parent type, and *dead* fields, which have stores to them, but who's value is never used. For dead fields, both the parent type need to be modified and (dead) store instructions need to be removed.

To identify dead fields for transformable types it is sufficient to find fields with no reads from them, but with writes to them. As we will explain later, the analysis can be kept simple as we guarantee that for a transformable type no aliases to individual fields exist.

Field Reordering - A similar mechanism as for dead field removal is used for field reordering. Once a record type is newly created, fields can be inserted in any desired order.

Alternative implementations are possible. For example, if one can prove that there is only one dynamic allocation of an array of structures, the type and the allocation site can be split into multiple pieces and the cold parts can be addressed with an offset to a pointer to the root portion of that type. If one can prove (or guarantee, for example, with help of an assertion) that there will be a limit on the size of a dynamically allocated array, in order to eliminate the need for link pointers one can use a technique similar to instance interleaving [21], but without the need for a special allocation library. Zhong et al [23], use a maximum fixed size for arrays of record types as well and convert pointers to integer indeces. Other implementation strategies are possible.

2.2. Legality Analysis

During FE's legality and property analysis, several small and efficient tests are performed in a single pass over our compiler's intermediate representation (IR) to determine whether it is safe to transform a type. A type is called *invalid* if it cannot be transformed. These are the main legality tests:

CSTT A cast to a type has been found. This indicates type-unsafe use of a type and such types are marked invalid. Note that in C/C++, for dynamically allocated types, at least one cast from a `(void *)` to the result type will be found, as both `malloc()` and `calloc()` return `(void *)`. The simple solution is to maintain a list of the return values from such calls and to tolerate casts made from these values. Since this analysis is done in the FE, types that are allocated in wrapper functions returning `(void *)` will be invalidated.

CSTF A cast from a record type has been found. Again, this indicates type-unsafe uses and types are invalidated.

ATKN The address of a field is taken. This may indicate address arithmetic on structure fields, which is incompatible with the modification of structure layouts. If the address of a field is taken in the context of a function call, we do not invalidate the type under the assumption that the called function will not try to access other structure fields from the pointer passed as argument.

LIBC Standard library functions are marked specially in the header files of the HP-UX compiler tool chain. If a type escapes to such a function, e.g. to `fwrite()`, the type is marked invalid, as it escapes to a function outside the current compilation scope. This analysis is done in the FE for efficiency reasons, as IPA will find escaped types as well.

IND A type escapes to an indirect function call. Since in the FE the targets of such calls are potentially unknown the type is invalidated.

SMAL A type has been dynamically allocated and the number of elements to be allocated is specified with a constant smaller than a threshold A. For example, in many or our benchmark programs we find allocation sites allocating arrays of size 1 (single objects).

MSET Our IR has special provisions for memory streaming operations corresponding to the C/C++ functions `memcpy()`, `memset()` etc. Types used in such expressions are marked invalid because of implementation limitations.

NEST A type is nested in another type. We mark these types as invalid also because of implementation limitations.

In order to determine applicable transformation types, we collect other attributes, such as whether a global or local variable, pointer, or array of a given type were found, whether a type has been dynamically allocated, free'd, or re-allocated. We also collect tuples <type, function> for record types escaping to (non-lib) functions for the escape analysis later in the IPA.

During IPA we read and aggregate the legality results from the FE and mark invalid types in the type-unified IPA symbol table. The escape summaries are read and aggregated as well. If a type escapes to a function outside the current (IPA) scope, it is invalidated.

Note that for some of above legality filters workarounds can be found which allow transforming a type. For example, for LIBC the original type can be reconstructed and be passed as a parameter. For MSET the IR can be converted into procedure calls with a loop assigning link pointers. So far, however, we haven't found performance opportunities that would warrant implementing such support.

There are other problematic constructs, for example, the operators `sizeof()` and `offsetof()`. The FE usually converts them into numeric constants based on it's knowledge of a type's layout. Code relying on these numbers can become unsafe after changing a structure layout. As a solution, the compiler front-end can avoid emitting indistinguishable integer constants for `sizeof()` constructs and instead emit attributed constants or introduce a new IR construct to enable proper analysis.

Benchmark	Types	Legal	%	Relax	%
181.mcf	5	1	20.0	3	60.0
179.art	3	2	66.7	2	66.7
milc	20	5	25.0	12	60.0
cactusADM	116	13	11.0	68	58.6
gobmk	59	9	15.3	45	76.3
povray	275	14	5.1	207	75.3
calculix	41	3	11.6	3	11.6
h264avc	42	3	7.1	25	59.5
moldyn	4	1	25.0	4	100.0
lucille	97	17	17.5	86	88.7
sphinx	64	4	6.2	52	81.2
ssearch	10	4	40.0	5	50.0
Average:			20.9		65.7

Table 1. Types and transformable types, with and without CSTF, CSTT, ATKN

Applying these simple and practical tests leads to very conservative results compared to what our field-sensitive points-to analysis (*Points-To*, for brevity) can derive. For example, if the address of a field is taken, Points-To may be able to derive that no other field can be accessed via this exposed address and that this operation is therefore not blocking transformations. If other fields can be accessed, Points-To will collapse the Points-To set for all fields.

During IPA, testing for collapsed Points-To sets can be used as a sharper legality test for ATKN, CSTT and CSTF. However, this would make it necessary to push the whole legality analysis into IPA.

We estimate an upper bound of the benefits of Points-To by introducing an internal flag to tolerate the type invalidating criteria CSTF, CSTT, and ATKN, for which Points-To can derive more accurate result.

Table 1 lists the benchmarks we are using in this paper. Among the SPEC2000 benchmarks, we found profitable opportunities for structure layout optimizations in only two benchmarks, 181.mcf and 179.art. We therefore add a number of open-source benchmarks representing a mix of floating point and integer programs. The total number of record types found is shown (column "Types") and the number of types passing through the practical analysis (column "Legal"). After relaxing the legality constraints, many more types are transformable (column "Relax"). The percentage of transformable types grows from 20.9% to 65.7%. However, the transformations are still being blocked by other legality tests than CSTF, CSTT, and ATKN, and by the profitability analysis. As a result, the number of transformed types (which can be seen in Table 3 below) remains constant.

2.3. Profitability Analysis

The profitability analysis computes read and write counts, as well as *hotness* for and *affinity* between structure fields. We provide simple informal definitions for the latter two terms:

- Two fields are affine to each other when they are accessed close to each other in the IR, for example, in the same statement or in the same loop.

- Hotness is computed from the aggregated total estimated accesses to a field. Fields that are accessed more often than others are hotter.

Our granularity for "closeness" is the loop level. The FE uses the loop optimizer's loop recognition, which is based on [9], to build a loop structure graph. It iterates over each loop's basic blocks and collects the field references for a given type into a weighted affinity group (weights are discussed below). Affinity groups can contain 1 or more fields. Field references found in remaining straight line code form another, single affinity group with the weight of the routine entry point.

Affinity groups containing identical fields are merged together via adding up the weights and stored as annotations in the IELF files. In our compiler infrastructure, annotations are nothing more than anonymous and indexable blocks of binary data.

Read and write counts are collected statement by statement using the basic blocks' incoming edge weights as counts. The aggregated information is also stored in the IELF file.

During IPA the annotations are read and aggregated, total read and write counts are computed and an affinity graph is constructed for every type. Nodes in the graph represent fields and an edge between two nodes indicates that both fields were found in at least one affinity group. The final edge weight is the result of summation of the weights of all affinities found in the IELF files. Hotness for fields is simply computed by adding up the incoming edge weights for a node in the affinity graph.

How weights are assigned to the affinity groups is what differentiates the various weighting mechanisms we experimented with. As an example we show in Table 2 below the relative hotness values for the fields of type `node_t` from the SPEC2000 integer benchmark 181.mcf for various experiments. Relative hotness is expressed in percent relative to the hottest field of a type. In the table, each column contains the relative field hotness in percent for a given experiment. As a baseline we use the hotness values computed from dynamic PBO (explained below). For all other methods we express the correlation to the baseline as a linear correlation coefficient r:

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2}\sqrt{\sum_i (y_i - \bar{y})^2}}$$

It takes on values between $+1$ (perfectly correlated) and -1 (complete negative correlation). A value close to 0 indicates no correlation at all. The x_i and y_i are the field specific percentages, and $\bar{x}, \bar{y}$ the respective arithmetic mean values. The last two rows in the table show the correlation r to the baseline (PBO) and the correlation r', which disregards field `potential`. The following paragraphs describe the various weighting mechanism in greater detail.

Profile Based Optimization (PBO) – If profile information is available, the incoming edge counts for loop headers are used as weights for the affinity group defined by the loop. This method resulted in the best affinity estimates. In general, the accuracy is determined by how good the training data sets predict the final executions of a program. It is more robust against a small number of problems in the training data sets than other parts of the compiler, such as the loop optimizer. In this phase a single important hot loop can be classified as cold because the training data set did not execute the loop. Since data structures are usually accessed in more than one loop, such problems are alleviated. PBO is our default method for feedback directed compilations.

Perfect PBO (PPBO) – As a reference we provide the values for "perfect PBO" where the feedback file is created using the reference data set. In the example, the correlation to the training data set is almost perfect ($r = 0.983$).

Static PBO (SPBO) – If no profile information is available, edge frequencies in a routine are estimated with help of probabilities for source constructs [22]. For example, a loop back edge is assumed to execute about 8 times on average and both branches of an if-then-else construct are assigned a 50% probability. As in the PBO case, incoming edge weights to loop headers are used as weights for the affinity groups.

Since the static heuristics for edge weights are local to procedures, they are not suitable for comparing inter-procedural affinities. For example, if a function foo() calls function bar() from an inner, deeply nested loop, the fields in bar() should be considered to be hotter than the ones in foo(), but with SPBO they are not. Correspondingly, the correlation to the baseline is poor ($r = 0.69$).

Inter-Procedurally Scaled Static PBO (ISPBO) – Similar to the method described in [22] we propagate the execution counts along call-graph edges to compute inter-procedural counts. The propagation happens top-down over the call-graph with the assumption that the main procedure is called once. The normalized execution count of a procedure is obtained by a summation of the normalized counts of its

Field	PBO	PPBO	SPBO	ISPBO	ISPBO.NO	ISPBO.W	DMISS	DLAT	DMISS.NO
number	0.2	0.0	5.3	4.1	5.1	1.9	0.2	0.2	0.1
ident	—	—	—	—	—	—	—	—	—
pred	73.7	74.7	100.0	82.3	100.0	100.0	13.7	11.7	12.8
child	20.8	21.7	37.3	28.1	35.0	35.1	1.2	1.1	0.6
sibling	20.7	21.7	28.7	20.4	25.5	25.4	0.4	0.2	0.2
sibling_prev	0.1	0.0	12.7	4.1	6.2	2.7	0.0	0.0	0.0
depth	3.1	1.3	19.2	10.6	13.8	6.5	2.6	1.7	2.6
orientation	23.2	22.6	52.7	38.5	47.4	42.2	32.4	38.6	30.9
basic_arc	39.9	42.5	61.5	53.2	64.4	68.4	1.6	1.8	2.1
firstout	0.8	0.2	11.4	4.3	6.0	1.2	0.9	1.9	0.9
firstin	0.7	0.2	6.4	1.0	1.7	0.3	0.2	0.1	0.1
potential	100.0	100.0	58.0	100.0	74.0	70.9	100.0	100.0	100.0
flow	2.8	0.9	38.5	20.1	26.3	12.0	1.1	0.7	1.0
mark	53.3	69.6	18.2	15.7	19.6	9.4	0.5	0.4	0.4
time	33.7	48.4	17.2	14.8	18.7	8.4	43.3	38.5	41.3
Correlation r		0.986	0.693	0.891	0.811	0.782	0.687	0.686	0.686
Correlation r'		0.983	0.727	0.799	0.795	0.764	0.211	0.207	0.207

Table 2. Relative field hotness for a variety of experiments and their correlation to PBO

incoming call-graph edges. The ratio of the normalized execution count of a procedure and its local execution count is used to scale every other static estimation within that procedure, such as branch and call frequencies. Our propagation algorithm properly handles recursion in the call-graph. The scaling is performed with the following algorithm.

Let $N_{loc}(f)$ be the local execution count of function f, and $N_g(f)$ it's global execution count. Let $E_{loc}(c)$ be the local execution count of call-site c and $E_g(c)$ it's global execution count. Let finally $C_{loc}(b)$ be the local execution count of a basic block b and $C_g(b)$ it's global execution count. For the main function m we set

$$N_g(m) = 1$$

For a function f we compute over all call sites c

$$N_g(f) = \sum E_g(c)$$

Once $N_g(f)$ is computed, the global counts of basic blocks within the function f are scaled using the following equation:

$$C_g(b) = C_{loc}(b) N_g(f) / N_{loc}(f)$$

Our probabilities for loop back-edges aren't high enough for certain benchmarks, resulting in histograms for field hotness which a are too flat, making it difficult to properly distinguish between hot and cold fields. We therefore scale up the inter-procedurally derived scaling factors S by an exponent E, which is currently set to 1.5. Since S is either bigger or smaller than 1.0 the scaling improves the separability between hot and cold fields. For reference, the non-scaled values are shown in column ISPBO.NO.

ISPBO correlates well with the baseline ($r = 0.89$) and is our default heuristic for compilations without profiles.

ISPBO with Modified Weights (ISPBO.W) – The experiment ISPBO.W shows how the effect of scaling correlates to the results obtained from increasing the back edge probabilities. We changed the probability for floating point loop back edges from 0.93 to 0.98 and that for other loops from 0.88 to 0.95 (this separation is an extension of [22]). We cannot change these probabilities permanently because of performance degradations in other benchmarks. The correlation between ISPBO.W and ISPBO is 0.94, indicating that using an exponent E in ISPBO is a valid approximation of higher back edge probabilities.

Cache Misses (DMISS, DLAT) – As we will show later, we also collect field specific data cache (d-cache) miss counts and latencies from the profile feedback files and attribute them to individual fields. Table 2 shows that misses and latencies are highly correlated to each other (0.96). However, the correlation to the baseline is poor (0.69).

If field `potential` is ignored the correlation sinks to only 0.21, indicating almost no correlation. The d-cache values are poor predictors for field hotness; however, in the future, they should be consulted to avoid splitting out of d-cache intensive fields. The column DMISS.NO presents the d-cache information gathered without instrumentation, which has a very high correlation to DMISS (0.996, not shown in the table). This means that the instrumentation has nearly no effect on the sampled d-cache values.

These numbers and their relative correlations have been confirmed by experiments with other benchmarks. However, in general, static heuristics, as well as heuristics based on profiles derived from non-representative training data sets, can potentially mis-classify fields.

2.4. Heuristics

Based on affinity, hotness, and type characteristics, the heuristics decide if and how to transform a type.

Dead structure fields are always removed as long as a limited set of alignment guarantees is preserved. In particular, removing bit-fields can result in more expensive access code sequences.

Structure peeling is always performed as well, subject to alignment constraints. It is possible to construct test cases for which this transformation results in a performance degradation. For example, a tight modulo-scheduled loop might need an additional cycle because of an additional load that needs to be performed. However, for all practical benchmarks and applications, no negative effects are expected.

Structure field reordering is currently only performed in the context of structure splitting; fields are only reordered if at least one field is eliminated or split out.

Structure splitting uses a threshold T_s. Fields with relative hotness lower than T_s are being split out. T_s is currently set to 3% for PBO and to 7.5% for ISPBO. T_s and the scaling factor E are subject to continuous tweaking. Since a link pointer needs to be inserted, at least two structure fields need to be split out for the transformation to be profitable.

Our observations for structure splitting are simple. We find that while the performance of hot loops improves significantly, the cost for loops accessing cold fields via link pointers grows disproportionately. Additional instructions need to be executed, which can negatively influence other optimizations, and the number of cache misses increases as well.

As an example, for 181.mcf's type `node_t`, the field `time` has a hotness of 14.8% in ISPBO, the field `mark` has a hotness of 15.6% (we chose these two fields because the resulting effects are significant). Splitting out field `time` results in a performance degradation of 9%. Splitting out the fields `time` and `mark` results in a performance degradation of 35%. We conclude that the single most important criteria for splitting is hotness - hot fields need to remain in the hot section, regardless of affinity or other metrics, such as access distance to other fields.

Only dynamically allocated objects are being transformed. The threshold for the allocation number in legality test SMAL is set to > 1, as we assume that splitting of single objects will be not be profitable. For the same reason we do not modify a type if it has only global or local variable instances and no static or dynamic array.

Our PBO infrastructure collects stride information for pointer-chasing loads and stores (and other information - the mechanism is explained below). The stride distance is usually a multiple of the size of the underlying type of an array or dynamic data structure a particular loop iterates over. Since type sizes change during structure splitting we were updating the stride distances as well. Interestingly, this had no or only slightly negative effects on runtime performance. This can be explained by the intricacies of our prefetch algorithm, which, unfortunately, are beyond the scope of this paper.

For multi-threaded applications a different set of heuristics can be applied. For example, there is a performance penalty if two threads access (write) disjoint hot structure fields on the same cache line due to costs associated with cache coherency. These fields should be separated to different cache lines instead of being moved together. Additionally, the multi-threaded heuristics can be augmented by analysis of whether fields are mostly read or written. While we perform read/write analysis, we do not currently consult these values in our heuristics. This is subject of current research.

2.5. Performance

The baseline for our performance analysis is a "SPEC base" configuration without using profiles, highlighting the quality of the non-profile based heuristics. The configuration specifies a high level of optimization and corresponds to the compiler options `+O3 -ipo +Onolimit +Olibcalls +Ofltacc=relaxed +DSnative +FPD`. All results were obtained on an HP server rx2600 with a 1500 MHz Intel Itanium © processor, 6 GB of memory, and 6 MB of L2 cache. For comparison, and to indicate the presence of potential second order effects of the transformation, we show the results obtained with and without profile for two benchmarks (181.mcf and moldyn).

In Table 3 we show the benchmarks (column "Benchmark") and whether a profile was used during compilation (column "PBO"). We show the number of types (column "T"), transformable types (column "T_t"), the number of split out and dead fields (column "S/D"), and finally the performance effects (column "Performance"). These range from -1.5% up to 78.2%. The three minor degradations for the benchmarks cactusADM, calculix, and h264avc are in the noise range. The benchmarks moldyn (21.8% – 30.9%), 181.mcf (16.7% – 17.3%) and 179.art (78.2%) gain significantly from the transformations.

For moldyn, the presence of profile information leads to improved behavior in other optimization phases, resulting in an additional 9% relative performance gain. For 181.mcf, however, there appear to be no second order effects of the transformations.

For all these benchmarks, the compile time overhead is low. For the FE it is 2.5% on average, with an observed

Benchmark	PBO	T	T_t	S/D	Performance
181.mcf	no	5	1	6 / 0	17.3 %
181.mcf	yes	5	1	6 / 0	16.7 %
179.art	no	3	2	0 /10	78.2 %
milc	no	20	5	3 / 0	0.0 %
cactusADM	no	116	13	14 / 1	-1.4 %
gobmk	no	59	9	2 / 0	0.5 %
povray	no	275	14	2 / 0	0.6 %
calculix	no	41	3	57 / 0	-0.6 %
h264avc	no	42	3	69 / 7	-0.3 %
moldyn	no	4	1	0 / 3	21.8 %
moldyn	yes	4	1	0 / 3	30.9 %
lucille	no	97	17	4 / 0	0.1 %
sphinx	no	64	4	17 / 0	0.5 %
ssearch	no	10	4	4 / 3	0.0 %

Table 3. Transformable/transformed types and performance impact

maximum of 5%. The overhead for IPA is always below 4%. For the BE the overhead is 1% on average, with an observed maximum of 2.5%. Our implementation is currently not optimized for speed and we believe that the overhead can be further reduced.

3. Advisory Tool

The performance results indicate that successful transformation can have significant positive impact on runtime performance. The number of automatic transformations is low because of legality violations and because of profitability constraints (the insertion of link pointers for split types has significant negative impacts on performance). The analysis, however, does provide valuable information and we offer an internal option to present the data to the user. In this section we first elaborate on how we correlate compiler analysis and runtime measurements. Then we provide an example and explain how to make use of the information presented. Finally we illustrate why a certain class of transformations cannot be performed automatically with the existing framework.

3.1. Combining Static Analysis and Dynamic Measurements

For a standalone performance tool it is difficult to correlate structure field accesses to individual loads and stores in a binary executable. Typically, the compiler has to emit additional annotations or tables into the binary to facilitate the correlation between instructions and debug information. It is difficult for the compiler itself to generate and maintain such tables because a mapping must be maintained between symbolic information in the high level optimizer and simple load and load-offset instructions in the low level optimizer.

We therefore reuse the existing profile-based optimization infrastructure (PBO) and use the compiler as a reporting tool. PBO is performed in two phases, a collection phase and a use phase.

In the PBO collection phase the application is instrumented and run with training input sets to produce feedback files. For the HP-UX Itanium tool chain, the instrumentation is compiler-based and performed at optimization level +O1. During the profile collection run of an application, the instrumented binaries additionally invoke the performance analysis tool, HP Caliper [11], to gather sampling data from the hardware performance monitoring unit (PMU), resulting in a feedback file that contains both edge counts and sampling results for data cache events. Note that other events could potentially be sampled and stored in the profile as well.

In the PBO use phase, the application's control flow graph (CFG) is constructed and matched against the CFG constructed from the data found in the feedback file. This matching is supported by source line information and an additional counting mechanism to distinguish between multiple expressions in a statement. After the matching, most or all arcs in the CFG have corresponding counts associated with them, and all loads and stores in the IR with attributed sampling events have this information available in the form of annotations.

As noted earlier, the instrumentation code itself has almost no effect on the sampling accuracy of the d-cache events. However, the collection (and conceptually the attribution) of profile information happens at +O1. This differs from an alternative approach which seeks to profile and sample a fully optimized binary. Both approaches have advantages and disadvantages. For example, the latter approach might deliver more accurate sampling results, but it is harder to attribute samples back to instructions because of the general problem of debugging optimized code. A full discussion of the differences is beyond the scope of this paper.

3.2. Reporting

With help of an option, IPA prints the annotated type layouts for all structure types, sorted by the hotness of the type, in a format similar to the one shown in Figure 2. For each type, it's name, total number of fields, and total size is shown. The type hotness is computed by adding up the hotness of the individual fields and comparing this sum against other types. Relative and absolute hotness are printed. Various attributes and legality violations are listed as abbreviations. For example, `LPTR` indicates that a local pointer variable of this type has been found.

It follows the list of fields and their attributes in field declaration order. For each field, it's relative hotness is shown in percent and as an absolute weight. To make the display more intuitive, we added a graphical bar. We distinguish between read and write references to a field and indicate their relation with a bar. If there are more reads than writes an uppercase "R" is used and a lowercase "w", else a lowercase "r" and an uppercase "W". In the example, field `number` has many more reads than writes, the bar therefore only contains uppercase "R" characters.

The d-cache miss count and average latency in cycles attributed to the field are shown next. The counts refer to the first level of cache for a given operation - L2 for floating point values and L1 for everything else on Itanium.

Finally, the affinities to other fields are shown, if present. Both fields and affinities are presented in declaration order. Only uni-directional edges are printed to make the output more compact.

Sometimes a graphical representation is helpful. For this purpose we also output control files for the VCG graph visualization tool [19] and use colors and line-thickness to indicate higher relative weights and affinities.

3.3. Combining D–Cache Misses, Hotness, and Affinity

More information can be derived from this output. Assuming a type T which has spatially close group of fields G_x (which may contain only 1 field), and a second, spatially distant group G_y, we differentiate these cases.

- G_x and G_y have high hotness, but low affinity to each other. This indicates that G_x and G_y are rarely used together in the same loops or probably used in separate program phases. The type should be split and G_x and G_y should be separated. Since the cost of link-pointers is prohibitive, the groups should be split conceptually at the source level. This may make algorithmic changes necessary in the program and is subject to more research. This important scenario cannot be handled well by the current, hotness based framework.

- G_x and G_y have high hotness and high affinities to each other. This indicates that G_x and G_y are used often together in loops or program phases. They should be grouped together, in particular if G_x and G_y have a high d-cache component. The cache effects of accesses to G_y might get hidden by the latencies for the accesses to G_x.

- G_x has low hotness. Splitting out G_x is an option (and probably can be performed by the automatic transformations). But, once again, since link pointers may be prohibitive, it is better to split out G_x conceptually at the source level.

- G_x is hot and has a high d-cache component. This may indicate that there are badly scheduled loops in a program or that a given composite data structure is too complicated (and breaks the compiler's analysis). When set in relation to other groups with lower d-cache components, this can give the compiler/scheduler hints about loads that need to be scheduled earlier.

- For multi-threaded applications, different conclusions can be found. For example, fields should additionally be grouped by read and write counts to minimize interprocessor cache coherency costs. Also, instead of relying on hotness, affinity may be used for field grouping to mirror thread specific references to fields.

Using the regular compiler as a reporting tool usually incurs unpractical overhead, as redundant compilations are performed. We believe this is not a general usability problem, as users will likely obtain the advisory output during their regular full builds. However, if a dedicated analysis tool is required, one can reconfigure the compiler to avoid unnecessary compilations and to only perform tasks needed for the analysis.

3.4. Experiences

This tool is currently being used by ourselves for hunting for opportunities in our set of benchmarks, including the upcoming SPEC2006 benchmark suite. Unfortunately, at time of this writing, this suite hasn't been finalized and we cannot disclose details. The tool is also used on the HP-UX kernel, about which we are only permitted to talk in general terms. The following paragraphs are therefore quite generic.

One of the C++ benchmarks in SPEC2006 has a hot structure S with a size larger than an L2 cache line (128 byte on Itanium). Looking at the affinity graphs derived from PBO clearly identified 4 hot fields in S which were not grouped together in the class definition. The affinity graph derived via ISPBO pointed out the exact same 4 fields. Grouping those fields together resulted in a performance improvement of 2.5%.

Another C benchmark in this suite is strongly dominated by three loops over an array of record types containing only two fields, a floating point field and an 8-byte integer field. Consequentially, this data structure is by far the hottest record type in the affinity graphs. Peeling of this type resulted in a performance improvement of almost 40%. After splitting, the three loops are iterating over an array of integers, performing only a few fast integer operations.

```
=== Affinity Graph ==================================================
Type      : node
Fields    : 15, 60 bytes
Hotness   : 100.0% rel, 52.6% abs
Transform: Splitting
Status    : *OK* / LPTR NSTP
---------------------------------------------------------------------
Field[0]    off: 0:0      |----------| "number"
  hot:    0.2%  weight: 5.367e+05
                read  : 9.375e+05, write: 2.072e+03     |RRRRRRRR|
                miss  : 2,    0.1%, lat:    9.5 [cyc]
  aff: 100.0% --> number
  aff:    1.3% --> firstout
  aff:   15.9% --> flow
Field[1]    off: 4:0      |----------| "ident" *unused*
Field[2]    off: 8:0      |#######---| "pred"
  hot:   73.6%  weight: 2.352e+08
                read  : 1.805e+08, write: 1.679e+05     |RRRRRRRR|
                miss  : 317,   12.8%, lat:    7.8 [cyc]
  aff:   41.0% --> pred
  aff:    0.3% --> child
  aff: 100.0% --> sibling
  aff:    7.2% --> depth
[. . .]
```

Figure 2. The advisory tool's output

The benchmark therefore hit a memory bandwidth barrier. When combined with a higher unroll factor for the three hot loops, or by using Itanium specific cache locality hints (completer .nt1) on loads, the splitting led to an overall performance gain of over 80%. This is another good example of potential second order effects of the transformations.

Currently the affinity information is used by the HP-UX kernel group to improve their structure definitions; some promising candidates have been identified. Since the kernel is a highly multi-threaded application, the analysis benefits heavily from the presence of the read/write counts. Based on the experiences, we plan to analyze large database systems and other key applications.

4. Related Work

We group related work into three areas, each consisting of a huge body of work. We only pick a few important representatives for each group.

First there is the work in locality improving transformations for scientific, array-based programs. These transformations are often referred to as loop nest transformations (for example [13], or the survey in [1]). These transformations do not apply to pointer intensive codes with complex data structures and control flow.

Secondly, for dynamically compiled languages such as Java, extensive work is being performed in many areas. There are studies on the effects of garbage collectors and memory allocators on data cache performance [17], [7], [10]. Chilimbi and Larus [5] use generational garbage collection to reorganize data structures so that objects with high temporal affinity are placed near each other, increasing the likelihood for them to reside in the same cache block. This work is different from ours as it seeks to re-arrange whole objects in memory to improve locality. We believe that there is some value of this approach even for static languages, as has been shown by Lattner [14], and we plan on augmenting our infrastructure with a similar mechanism.

Kistler and Franz [12] describe a technique that uses online path profiling data to reorder structure fields for type-safe languages, such as Java. The resulting layouts are aware of cache line fill buffer forwarding, a hardware feature supported by the PowerPC. They improve performance for their set of benchmarks by up to 24%.

Chilimbi et al [3] describes structure splitting and field reordering for Java. They report a reduction in cache miss rates of 10-27% and improved performance of 6-18% in five Java programs. They also describe the tool `bbcache` which has similarities with our advisor. The tool provides field reordering advice for C programs. Applying the advice they report improvements in the performance of Microsoft SQL server of about 2-3%; 5 types could be modified.

Finally there is a third group of work which deals with other aspects of data layout optimization.

Calder et al [2] apply a compiler directed approach using profile data to place global data, constants, stack variables and heap objects. Profile information guides compile time variable placement algorithms in finding a variable placement solution that decreases predicted inter-variable conflicts and increases predicted increased cache utilization. Their technique works on the placement of entire objects and not on the placement of fields within an object. Our compiler has a similar phase, which we call *global variable layout* (GVL). We plan to merge GVL with the presented framework in the future.

Chilimbi et al [4] describe a semi automatic tool called `ccmorph` that reorganizes the layout of homogeneous trees at runtime to improve locality. It relies on programmer annotations to identify the root of a tree and to indicate that the reorganization is safe. They also describe `ccmalloc` which is a malloc replacement that accepts hints to allocate one object near another. These hints only provide local information for an object pair and not any global information about entire data structures. This work targets a different problem domain and relies on user input. However, we believe there is value in user provided input and plan to investigate this area further.

Truong et al [21] propose a field reorganization technique called *instance interleaving* which is partially automated. They show that instance interleaving can have a large positive performance impact, but has limited applicability. Instance interleaving requires a special allocation library, `ialloc`, and type safe C/C++ programs. This transformation can be integrated into our current framework and thus be made fully automatic. So far, however, we haven't found opportunities for this extension in our set of benchmarks.

Rubin, Bodik and Chilimbi [18] developed a parameterized framework for data-layout optimization of general-purpose applications. They find an optimal layout is not only NP-hard, but also poorly approximable. Their framework finds a good layout by searching the space of possible layouts, with the help of profile feedback. The search process iteratively prototypes candidate data layouts, evaluating them by "simulating" the program on a representative trace of memory accesses.

Zhong et al [23] use the elegant concept of *reference affinity* to arrive at precise data transformation decisions, split structures in multiple parts, and achieve impressive performance results. However, their approach is based on program traces and therefore not applicable for commercial compilers. They use a maximum size for allocated arrays, which allows conversion of pointers to integer indices. This transformation does not incur the demonstrated penalties of link pointers. Consequentially, these effects are not modelled. We believe that there is value in the presented affinity concept even for non-profile based compilations and are trying to estimate similar metrics based on static, interprocedural analysis.

Rabbah and Palem [16] split structures by allocating objects in chunks and by remapping field locations into these block of memory. This corresponds, conceptually, to our structure peeling transformation, except that no external pointers to the remapped sub-arrays are needed. Instead, field locations are computed with address arithmetic based on the address of a first field. To ensure correctness, they use points-to analysis and, when needed, compiler-generated check code. Their approach is based on program traces. We performed manual experiments with similar transformations on a small set of benchmarks, but weren't able to achieve meaningful performance improvements over our existing transformations.

Finally, Hagog [8] describes an implementation of structure splitting based on profiles in GCC [6], which is, as such, close to our work. They report only partial implementation results and do not provide an advisory tool. In contrast to our experiences, they do split single global variables of a record type and suggest multi-level splitting using link pointers.

5. Conclusions and Future Work

We presented our framework for structure layout optimizations, the design constraints, and trade-offs we had to make because we are implementing a commercial compiler. After explaining our transformations, we developed and evaluated a set of simple legality tests, demonstrating that even with a more sophisticated analysis no more optimization opportunities were exposed in our set of benchmarks. Then we developed and evaluated several choices for out profitability analysis, introducing a correlation coefficient to measure their quality.

We find that many record types cannot be transformed automatically for a variety of reasons and introduce an advisory tool, which combines static compiler analysis with dynamic runtime measurements. The generated reports provide valuable insights into more effective structure layouts, as indicated in a few (very brief) case studies.

We plan to gradually extend our framework and handle more and more cases. In particular, many of the simple legality tests described in section 2.2 should be handled by an improved implementation. Field reordering appears to be underutilized at the moment and we plan to further enhance the heuristics for it. We also plan to augment our support for C++, for which we presently only have rudimentary support and will implement some form of automatic pool allocation to further improve locality. We are working with the HP-UX kernel team and based on their feedback we will further improve the reporting capabilities. Finally we are considering re-packaging the analysis phase into a standalone tool.

6. Acknowledgements

This work is the result of a strong team effort at HP. We would like to thank our team members in the SYZYGY high-level optimizer for their never ending willingness to help and Shin-Ming Liu for encouraging this work. We particularly like to thank Joe Mario for his relentless pointing out of the many flaws in the early versions of the advisor and for making the results available to a broader audience. 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] D. F. Bacon, J.-H. Chow, D. ching R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 3. IBM Press, 1994.

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

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

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

[5] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 37–48, New York, NY, USA, 1998. ACM Press.

[6] Gcc. The GNU compiler collection. http://gcc.gnu.org.

[7] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 177–186, New York, NY, USA, 1993. ACM Press.

[8] M. Hagog and C. Tice. Cache aware data layout reorganization optimization in gcc. In *GCC Summit Proceedings*, 2005.

[9] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[10] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[11] R. Hundt. HP Caliper: A framework for performance analysis tools. In *Concurrency, IEEE*, pages 64–71, Washington, DC, USA, 2000. IEEE Computer Society.

[12] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. In *ACM Transactions on Programming Languages and Systems, v.22 n.3*, pages 490–505, 2000.

[13] M. S. Lam and M. E. Wolf. A data locality optimizing algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[14] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, New York, NY, USA, 2005. ACM Press.

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

[16] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *Trans. on Embedded Computing Sys.*, 2(2):186–218, 2003.

[17] M. B. Reinhold. Cache performance of garbage-collected programs. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 206–217, New York, NY, USA, 1994. ACM Press.

[18] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 140–153, New York, NY, USA, 2002. ACM Press.

[19] G. Sander. Graph layout through the VCG tool. In *Lecture Notes in Computer Science 894*, pages 194–205. Springer Verlag, 1995.

[20] SPEC. Standard performance evaluation corporation. http://www.spec.org.

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

[22] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, New York, NY, USA, 1994. ACM Press.

[23] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI '04: Proceedings of the 2004 ACM SIGPLAN conference on Programming language design and implementation*, 2004.