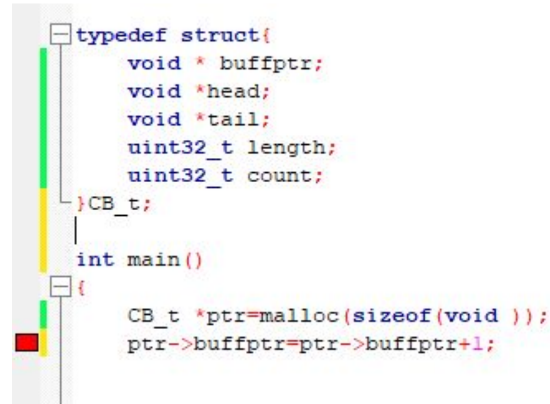
MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Practical Structure Layout Optimization and Advice

Group 21
Jaehyun Shim, William Wang, Christian Ronda, Yong Seung Lee

Problem statement

- Processor clock speed vs memory latency
- Need to improve an application's cache locality and reuse
  - Especially for pointer access patterns



Limitations of Previous Effort

- Previous Effort
  - Loop Transformation (Tiling)
  - Array Padding

- Challenges:
  - Limited to array and loop intensive scientific codes
  - Not applicable to pointer-chasing access patterns

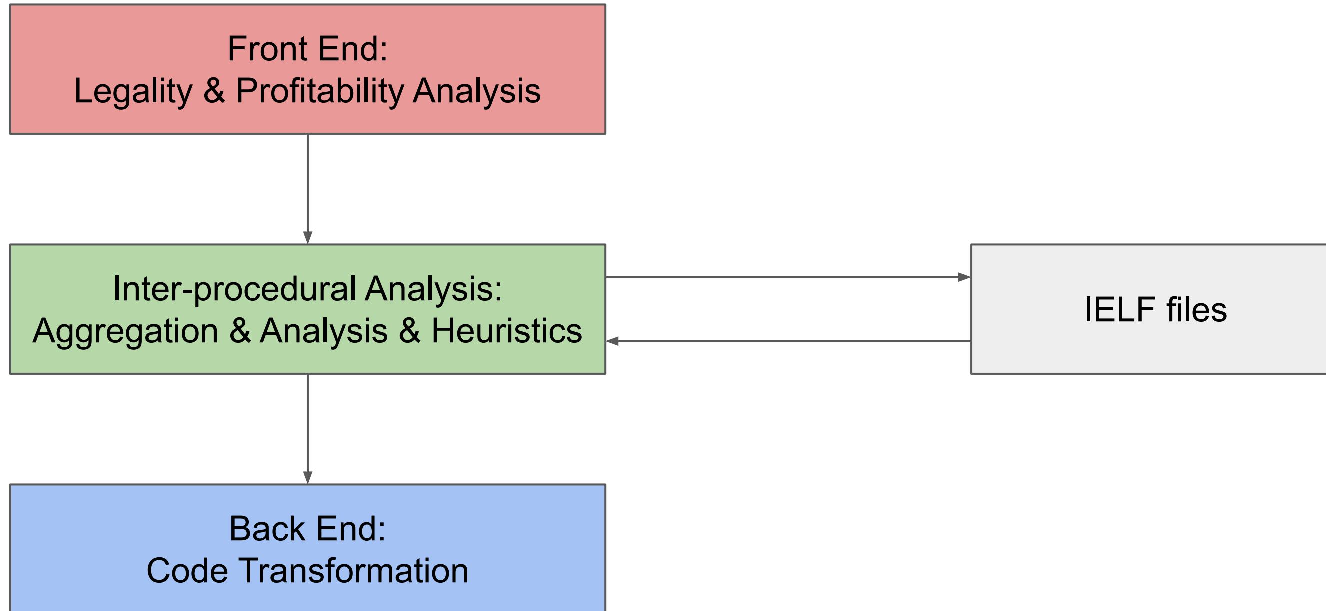
```
void calculate(std::vector<std::vector<long long>>& A, int i, int j) {
    A[i][j] = f(A[i][j], A[i + 1][j], A[i][j + 1], A[i+1][j+1]);
}


long long stencil(std::vector<std::vector<long long>>& A, long long sumAndCount[], int n, ...) {
    long long ghost[n];
    // tag identifier: the sender's rank + iteration
    for (int iter = 0; iter < 10; ++iter) {
        for (int i = 0; i < local_n ; ++i) {
            for (int j = 1; j < n - 1; ++j) {
                if (rank == 0) {
                    if(i > 0 && i < local_n - 1) {
                        calculate(A, i, j);
                    } else if (i == local_n - 1 && size != 1) {
                        A[i][j] = f(A[i][j], ghost[j], A[i][j + 1], ghost[j+1]);
                    }
                } else if (rank == size - 1) {
                    if (i != local_n - 1) {
                        calculate(A, i, j);
                    }
                } else {
                    if (i == local_n - 1) {
                        A[i][j] = f(A[i][j], ghost[j], A[i][j + 1], ghost[j+1]);
                    } else {
                        calculate(A, i, j);
                    }
                }
            }
        }
    }
```

Proposed Approach

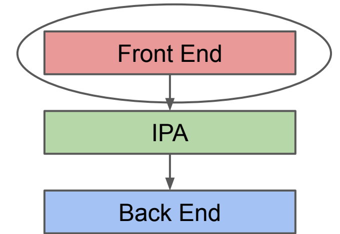
- Optimizing structure layout
  - 4 techniques:
    - Structure splitting
    - Peeling
    - Dead field removal
    - Reordering

- Advisory tool
  - Overcoming profitability constraints and struggles to provide satisfactory results
    - Static analysis & runtime data collection -> field affinity and hotness
      - Guiding structure layout decisions

Proposed Approach: Optimizing Structure Layout



Step 1: Front End - Legality Analysis

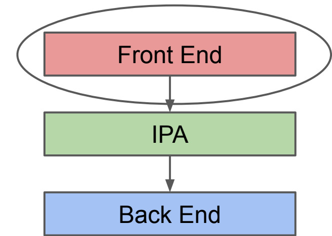


Before we can do a struct transformation, we have to do some checks first.

This is called Legality Analysis and is done by the Front End,

where legality is determined by a series of tests with a single pass over the IR.

Legality Analysis Flags



Here are some flags that are checked.

If flag conditions are met, the struct is deemed invalid for transformation.

To determine applicable transformation types, we also collect attributes such as global vs local scope, is it allocated or free, and escaping scopes.

Attributes and legality flags are then sent to the IPA stage

Flag examples:

CSTT: (void *) function returns are not valid.

CSTF: A casted record type is invalid.

ATKN: Address field usage in code is not invalidated if it is used as an argument, as we assume that the function will not use it to jump somewhere else unintentionally.

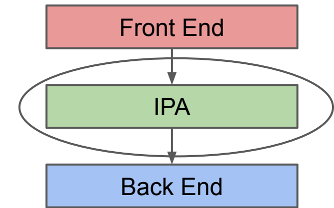
NEST: Nest types are invalid.

SMAL: Small dynamically allocated types with a size under a threshold are invalid, as it's not cost effective to optimize them.

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

not the drink unfortunately

Step 2: Inter-procedural Analysis (IPA)



The IPA is responsible for collecting legality information and giving the backend instructions on the most optimal restructuring methods to use.

A few things happen here:

1. Point-To Analysis: Additional address access tracking

Because of restructuring, struct addressing can get misdirected if function rely on moving from address to address.

Point-To Analysis aims to address and optimize this issue.

2. Profitability Analysis

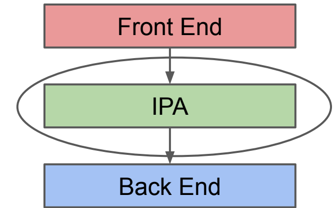
Track metrics such as read/write counts, access count, and affinity.

3. Profile Based Optimization

Because member variable accesses frequencies are relative to other variable, an weighted affinity graph is created to track relative accesses.

Confusing? Let's break them down next slides

Profitability Analysis



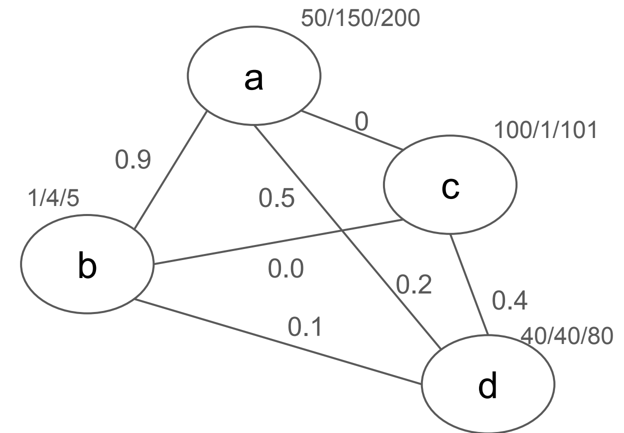
A weighted affinity graph is created to check which parts of our struct is worth restructuring.

We use three metrics

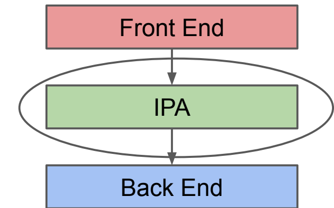
1. Hotness: The absolute summation of access of the field - can be estimated.
2. Affinity: Fields are affine when they are accessed close to each other.
3. Read/Write count

- Metrics are aggregated into an IELF file

```
struct MyStruct {
   int a;
   long b;
   char c;
   float d;
}
```



Profile Based Optimization



Based on the metrics collected from Profit Analysis, the IPA declares an optimization mode based on information available.

This are just example optimization modes >>

Regular Profile Based Optimization (PBO):

- If profile information is available, the incoming edge counts for loop headers are used as weights. (Yields overall best optimization)

Static PBO:

- If profile information is not available, edge frequencies are estimated with probabilities based on back edge tracking.
- However, effectiveness is limited to local scopes.

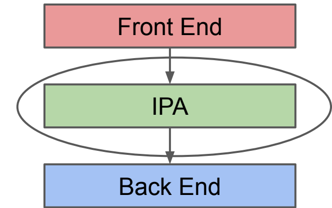
Inter-Procedurally Scaled Static PBO:

- Execution counts are further propagated along call-graph edges. The counts are scaled based on local vs global execution count.

Cache Misses and Latency (DLAT):

- D-caches miss counts and latenties also provide profile information due to high correlation to affinity.
- But weak for predicting hotness.

Transformation Heuristics



After deciding on an Optimization Mode:

‘Correlation r’ or relative hotness determines how much weight should be placed on transforming a specific affinity field group.

Additional heuristics are added as well:

- Dead structure fields are always removed as long as most of the struct alignment is preserved.

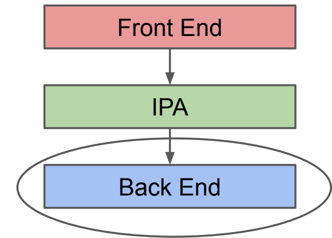
- Peeling is always performed as long as it mostly preserves struct alignment.

- Field reordering is only performed in context of splitting, with at least one field eliminated or split out.

- Splitting uses a different threshold for each type of IPA optimization.

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

Step 3: Backend - Struct Transformations



The IPA hands the heuristic weights and decide which fields are *hot* or *cold*.

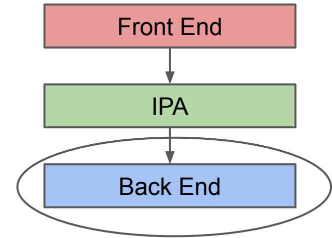
The Backend then uses transformation methods to change the struct organization using this information.

With the heuristics previously mentioned,

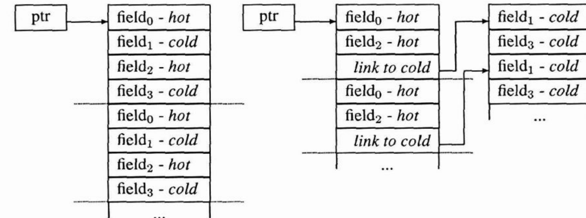
4 Transformation Methods are considered

- Structure splitting
- Peeling
- Dead field removal
- Reordering

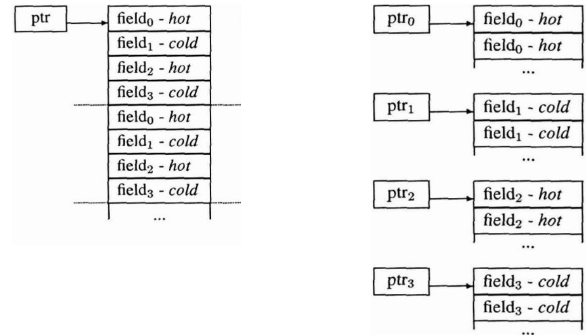
Struct Grouping Transformations



- Struct **Splitting**
  - Break a given Struct into two (or more) pieces
  - Insert link pointers from the root to the splitted
    - Tree like structure

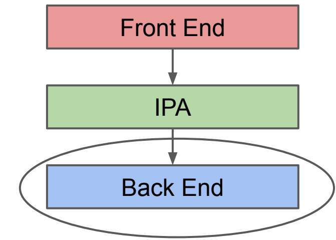


- Struct **Peeling**
  - Splitting without having to insert link pointers
  - Much more distinct groupings in memory.
  - New Variables instead of link pointers are created



MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Optimization Transformations



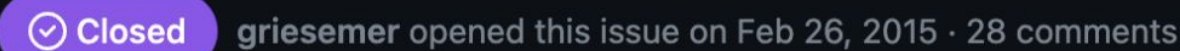
- Dead Field Removal
  - Similar to Dead Code removal
  - Dead Fields: Stores, but never read
    - Split/Peel structs
    - Remove Inst
  - Only Live Fields are moved into a new hot section
- Field Reordering
  - Insert fields in any desired order into newly transformed type

Problem of struct optimization

- Problem: Due to **legality violations** & **profitability constraints** -> Low transformations
- gcc does not reorder the elements of a struct, because that would violate the C standard. Section 6.7.2.1 of the C99 standard states:

> Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared.

proposal: spec: define/imply memory layout of tagged struct fields #10014



That said, no Go compiler should probably ever reorder struct fields. That seems like it is trying to solve a 1970s problem, namely packing structs to use as little space as possible. The 2010s problem is to put related fields near each other to reduce cache misses, and (unlike the 1970s problem) there is no obvious way for the compiler to pick an optimal solution. A compiler that takes that control away from the programmer is going to be that much less useful, and people will find better compilers.

Advisory Tool

- Providing statistics & useful information regarding struct fields.
- Correlating structure field accesses to individual loads and stores in a binary executable is HARD.
    - Mapping symbolic information in the high level optimizer <> simple load and load-offset instructions in the low level optimizer.
- 2 step
    - PBO collection phase: collect data such as edge count and data cache events
    - PBO use phase: create CFG and maps profiled data.

Result report

```
=== 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
[. . .]
```

Size, relative/absolute hotness, attributes & legality violation (Local Pointer)

Each field: offset / name/ Hotness / Read, Write counts / Cache miss rate and avg latency / Affinities to other fields

Result analysis

Spatially CLOSE group of fields A & Spatially DISTANT group of fields B

Case	Analysis
A & B hot, low aff	Separate A & B
A & B hot, high aff	Group A & B
A cold	Split out A

Multi-threaded application: Grouping based on Read & Write counts

→ minimize **inter-processor cache coherency** costs

Usage experience

1. PBO clearly **identified hot fields** in struct which were not grouped together in the class definition.

   => **Grouping** results in **2.5%** performance increase

2. Array of instance (float, int) & lots of loop

   => **Peeling** improved **40%** + additional loop optimizations & splitting improved **80%**

3. Affinity information used by the HP-UX kernel group to improve their structure definitions

   => Multi-threaded kernel benefits from read/write counts

Related Work

- Two groups of related work
- Extensive work in many different areas for dynamically compiled languages (e.g. Java)
- Other aspects of data layout optimization

Dynamically Compiled Languages

- Chilimbi and Larus: Generational garbage collection to reorganize data structures
  - Objects with high temporal affinity are placed near each other → increases likelihood to reside in the same cache block
- Kistler and Franz: Use online path profiling data to reorder structure fields for typesafe languages such as Java
  - Improve performance for their set of benchmarks by up to 24%

Other Aspects of Data Layout Optimization

- Calder et al:
  - Compiler-directed approach for variable placement using profile data.
  - Targets global data, constants, stack variables, and heap objects.
- Zhong et al:
  - Use the concept of reference affinity for precise data transformation decisions.
  - Split structures into multiple parts for optimization, achieving impressive performance results.

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Conclusion + Group Commentary

- Future Framework Enhancements:
  - Gradual Extension
  - Improved Legality Tests
  - Field Reordering
- Group comments
  - Lack of benchmarks, but great adaptation of optimizations techniques on insightful advisory report.
  - Not just focusing on a single optimization technique but utilized all of them together.
  - Rich analyses and modularization of different analyses enriched authors' endeavors.
  - A lot emphasis on optimization schemes for a variety of access scopes.