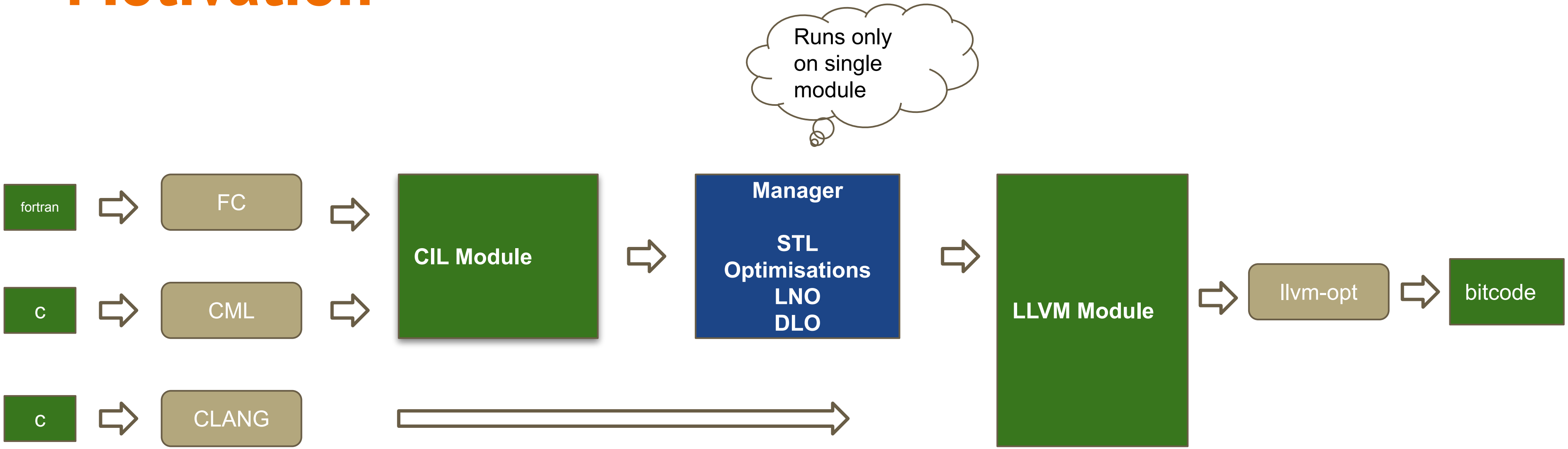
LTO and Data Layout Optimizations in MLIR

Ranjith/Prashantha
Compiler Tree Technologies

Agenda

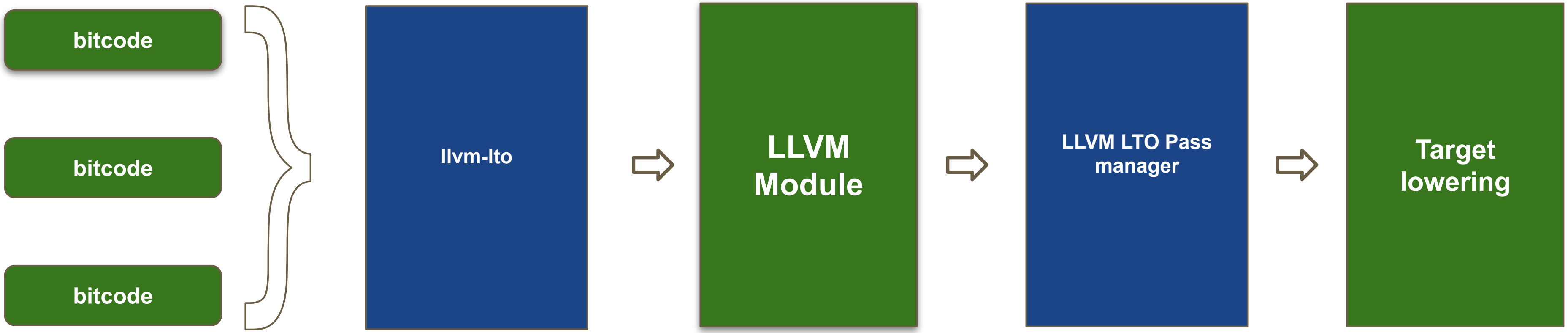
- Motivation for LTO in MLIR
- CIL - LTO
- Data layout optimisations
- Instance interleaving
- Dead field elimination

Motivation

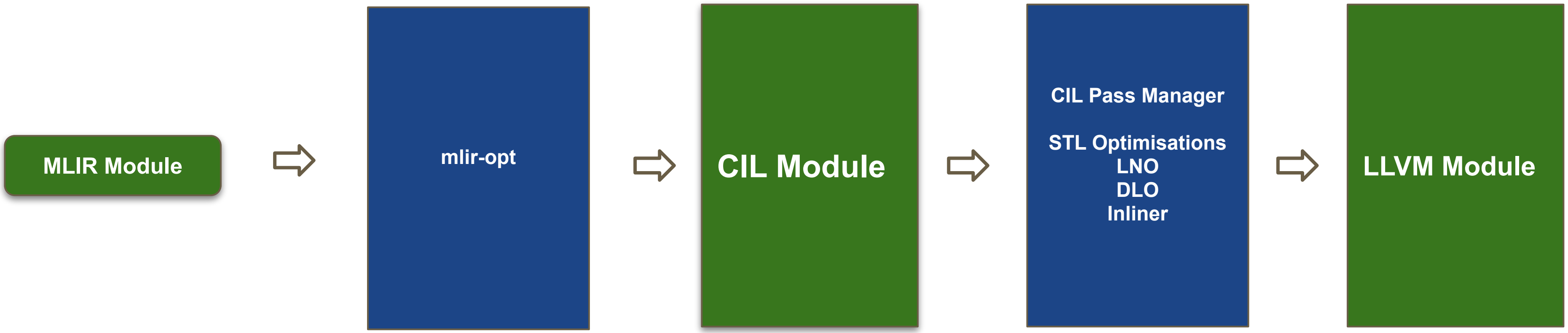


Runs only on single module
fortran
FC
c
CML
c
CLANG
CIL Module
Manager
STL
Optimisations
LNO
DLO
LLVM Module
llvm-opt
bitcode

Motivation

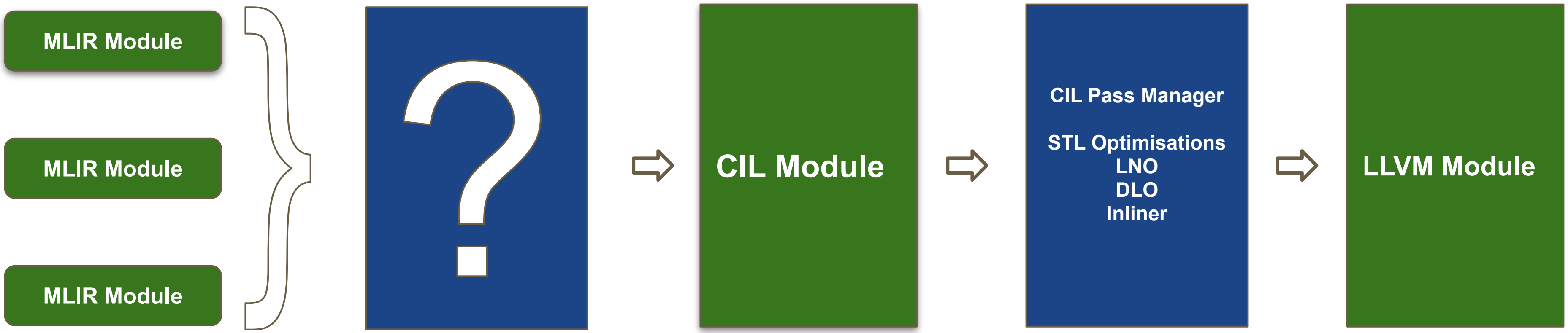

bitcode
bitcode
bitcode
llvm-lto
LLVM Module
LLVM LTO Pass manager
Target lowering

Motivation



MLIR Module
mlir-opt
CIL Module
CIL Pass Manager
STL Optimisations
LNO
DLO
Inliner
LLVM Module

Motivation


MLIR Module
MLIR Module
MLIR Module
?
CIL Module
CIL Pass Manager
STL Optimisations
LNO
DLO
Inliner
LLVM Module

Why LTO in MLIR?

- MLIR can represent high level source constructs like Multi-dimensional arrays.
- This reduces the analysis required during optimisations.
- Optimisations like LNO, DLO are best performed at MLIR.
- Coverage of these optimisations is limited if entire application is not represented using single MLIR module.
- Running LNO/DLO at MLIR on whole application requires a LTO support

Example - C

```c
extern void add(int *, int *, int *, int);
int main() {
  int a[N], b[N], c[N];
  for (int i = 0; i < N; i++) {
    c[i] = i + i;
    b[i] = i * i;
  }

  add(a, b, c, N);
  print(a);
  print(b);
  print(c);
  return 0;
}
```

f1.c

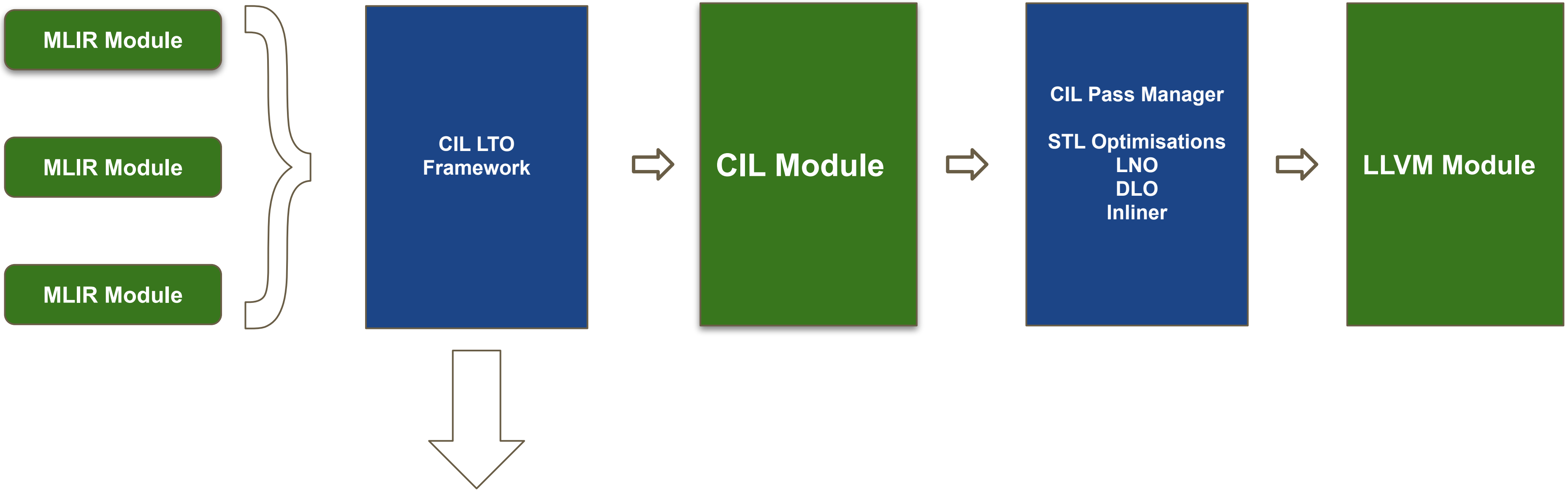
```c
void add(int *a, int *b, int *c, int n) {
  for (int i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
  }
}
```

f2.c

CIL - LTO

- CIL is a MLIR dialect designed to represent language constructs of C, C++ and Fortran.
- CIL LTO is a framework to link multiple MLIR modules into one.
- There is very less dependency on dialect and can be easily extended to work on any dialect.
- Multiple SPEC CPU 2017 benchmarks can be compiled with the framework
- https://llvm.org/devmtg/2020-09/slides/CIL_Common_MLIR_Abstraction.pdf

LTO Framework



LTO - Process

Read the module files → **Symbol Resolution** → **Create new module**

Read the module files

All the input modules are parsed and stored in a list

Symbol Resolution

Symbol resolution happens for top level operations like global variables operations and function operations

Create new module

Single MLIR module is created and fed to the pipeline

LTO Example - C

```c
extern void add(int *, int *, int *, int);
int main() {
  int a[N], b[N], c[N];
  for (int i = 0; i < N; i++) {
    c[i] = i + i;
    b[i] = i * i;
  }

  add(a, b, c, N);
  print(a);
  print(b);
  print(c);
  return 0;
}
```

```c
void add(int *a, int *b, int *c, int n) {
  for (int i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
  }
}
```

LTO Example - C

```c
extern int add(int, int);
int main() {
  printf (" Main function: %d \n", add(10, 5));
  return 0;
}
```

```c
int add(int a, int b) {
  printf ("Add function ");
  return a + b;
}
```

⇨

```
module {
  %0 = cil.global @__str_tmp0 {constant, sym_name = "__str_tmp0", ...
  func @main() -> !cil.int attributes {original_name = "main"} {
    %1 = cil.global_address_of @__str_tmp0 ..
    ...
    %5 = cil.call @add(%3, %4) ...
    %6 = cil.call @printf(%2, %5) ...
    %7 = cil.constant( 0 : i32 ): !cil.int
    return %7 : !cil.int
  }
  func @printf(!cil.pointer<!cil.char>) ...
  func @add(!cil.int, !cil.int) ...
}
```

```
module {
  %0 = cil.global @__str_tmp0 {constant, sym_name = "__str_tmp0", ...
  func @add(%arg0: !cil.int, %arg1: !cil.int) ... {
    ....
    %3 = cil.global_address_of @__str_tmp0 ....
    %5 = cil.call @printf(%4) : (!cil.pointer<!cil.char>) ...
    %8 = cil.addi %6, %7 : !cil.int
    return %8 : !cil.int
  }
  func @printf(!cil.pointer<!cil.char>) ...
}
```

LTO Example

```
module {
  %0 = cil.global @__str_tmp0 {constant, sym_name = "__str_tmp0", ...
  func @main() -> !cil.int attributes {original_name = "main"} {
    %1 = cil.global_address_of @__str_tmp0 ..
    ...
    %5 = cil.call @add(%3, %4) ...
    %6 = cil.call @printf(%2, %5) ...
    %7 = cil.constant( 0 : i32 ): !cil.int
    return %7 : !cil.int
  }
  func @printf(!cil.pointer<!cil.char>) ...
  func @add(!cil.int, !cil.int) ...
}

module {
  %0 = cil.global @__str_tmp0 {constant, sym_name = "__str_tmp0", ...
  func @add(%arg0: !cil.int, %arg1: !cil.int) ... {
    ....
    %3 = cil.global_address_of @__str_tmp0 ....
    %5 = cil.call @printf(%4) : (!cil.pointer<!cil.char>) ...
    %8 = cil.addi %6, %7 : !cil.int
    return %8 : !cil.int
  }
  func @printf(!cil.pointer<!cil.char>) ...
}
```

⇨

```
module {
  func @printf(!cil.pointer<!cil.char>) ...
  %0 = cil.global @__str_tmp0 {constant, sym_name = "__str_tmp0", ...
  func @main() -> !cil.int attributes {original_name = "main"} {
    %2 = cil.global_address_of @__str_tmp0 ...
    ...
    %6 = cil.call @add(%4, %5) : (!cil.int, !cil.int) -> !cil.int
    %7 = cil.call @printf(%3, %6) ...
    %8 = cil.constant( 0 : i32 ): !cil.int
    return %8 : !cil.int
  }
  %1 = cil.global @__str_tmp0_1 {constant, sym_name = "__str_tmp0_1" ..
  func @add(%arg0: !cil.int, %arg1: !cil.int) ... {
    ...
    %4 = cil.global_address_of @__str_tmp0_1 : ...
    ...
    %6 = cil.call @printf(%5) : (!cil.pointer<!cil.char>) ...
    ...
    %9 = cil.addi %7, %8 : !cil.int
    return %9 : !cil.int
  }
}
```

Data Layout Optimisations

Data Layout Optimisations

- Modifying structure/array patterns for better cache utilization.
- Implemented in both LLVM and MLIR

https://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf

Structure Splitting/Peeling

```
struct S {
   int A;
   int B;
   struct S *C;
};
```

A – Hot
B – Cold
C – Pointer to **struct** S

Split structures:

```
struct S {
   int A;
   struct S  *C;
   struct S.Cold *ColdPtr;
};
```

```
struct S.Cold {
   int B;
};
```

DLO - Instance Interleaving

```
struct S_Mod {
  int a;
  int b;
  int c;
  int d;
} Array[N];

int foo() {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j)
      Array[j].a += 10 + j;

    for (int j = 0; j < N/2; ++j)
      Array[j].b += 11 + j;

    for (int j = 0; j < N/4; ++j)
      Array[j].c -= 12 + j;

    for (int j = 0; j < N; ++j)
      Array[j].d *= 13 + j;
  }
  return 0;
}
```

Array[0].a	Array[0].b	Array[0].c	Array[0].d
Array[1].a	Array[1].b	Array[1].c	Array[1].d
Array[2].a	Array[2].b	Array[2].c	Array[2].d

https://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf

DLO - Instance Interleaving

```
struct S_Mod {
  int a;
  int b;
  int c;
  int d;
} Array[N];

int foo() {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j)
      Array[j].a += 10 + j;

    for (int j = 0; j < N/2; ++j)
      Array[j].b += 11 + j;

    for (int j = 0; j < N/4; ++j)
      Array[j].c -= 12 + j;

    for (int j = 0; j < N; ++j)
      Array[j].d *= 13 + j;
  }
  return 0;
}
```

⇨

```
struct S_Mod {
  int a[N];
  int b[N];
  int c[N];
  int d[N];
} Array;

int foo() {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j)
      Array.a[j] += 10 + j;

    for (int j = 0; j < N/2; ++j)
      Array.b[j] += 11 + j;

    for (int j = 0; j < N/4; ++j)
      Array.c[j] -= 12 + j;

    for (int j = 0; j < N; ++j)
      Array.d[j] *= 13 + j;
  }

  return 0;
}
```

https://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf

DLO - Instance Interleaving

```
struct S_Mod {
  int a[N];
  int b[N];
  int c[N];
  int d[N];
} Array;

int foo() {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j)
      Array.a[j] += 10 + j;

    for (int j = 0; j < N/2; ++j)
      Array.b[j] += 11 + j;

    for (int j = 0; j < N/4; ++j)
      Array.c[j] -= 12 + j;

    for (int j = 0; j < N; ++j)
      Array.d[j] *= 13 + j;
  }

  return 0;
}
```

Array.a[0]	Array.a[1]	Array.a[2]	Array.a[3]
Array.a[4]	Array.a[5]	Array.a[6]	Array.a[7]

https://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf

Data Layout Optimisations in MLIR

- Cross module optimization
- Instance interleaving and Dead field elimination optimisations are implemented in MLIR
- Runs as LTO passes
- Identification of struct access is simpler as compared to LLVM because there is separate operation for struct access.
- Approximately 35% improvement is seen in one of SPEC CPU 2017 benchmark.

DLO - Instance Interleaving

- Identify the profitable and legal structs to transform
- Identify arrays of structures whose different fields are accessed in different loops
- Create and allocate new structure type.
- Rewrite rewrite old accesses.

```
TheModule.walk([&](CIL::StructElementOp op) {
   CIL::StructType type = ...

   if (type != structType)
     return;

   for (auto &use : op.getResult().getUses()) {
     populateUse(..);
   }
   OpsToRewrite.push_back(op);
 });
```

DLO - Dead field elimination

```c
struct str {
  int a;
  int b;
  int c;
  int d;
};

int main() {
  struct str S;
  S.a = 10;
  S.b = 11;
  S.d = 13;
  printf (" %d %d \n", S.a, S.d);
  return 0;
}
```

b is write only field and c not at all accessed

```c
struct str {
  int a;
  int d;
};

int main() {
  struct str S;
  S.a = 10;
  S.d = 13;
  printf (" %d %d \n", S.a, S.d);
  return 0;
}
```

DLO - Dead field elimination

- Classify the struct fields
  - READ - Field is loaded in the use
  - WRITE - Some value is being written to the field
  - UNKNOWN - Any use other than read/write
  - NOACCESS - Field is not at all used.
- Remove the NOACCESS and WRITE only fields.
- Rewrite the uses

Dead field elimination - Transformation

```c
struct str {
  int a;
  int b;
  int c;
  int d;
};

int main() {
  struct str S;
  S.a = 10;
  S.b = 11;
  S.d = 13;
  printf (" %d %d \n", S.a, S.d);
  return 0;
}
```

→

```
====== Struct StructAnalysisInfo ======

struct.str {
    0 : READWRITE
    1 : WRITE
    2 : NOACCESS
    3 : READWRITE
  }

  Dead indices : 1 2
  Number of uses 5

 ========================================
Remap of Struct str
  Remap 0 0
  Removing index 1
  Removing index 2
  Remap 3 1
```

Dead field elimination - Unknown Access

```c
struct str {
  int a;
  int b;
  int c;
  int d;
};

int main() {
  struct str S;
  S.a = 10;
  S.b = 11;
  S.d = 13;
  ___ctt_lib_populate(&S.c);
  printf (" %d %d \n", S.a, S.d);
  return 0;
}
```

```
====== Struct StructAnalysisInfo ======

struct.str {
    0 : READWRITE
    1 : WRITE
    2 : UNKNOWN
    3 : READWRITE
  }

  Dead indices : 1
  Number of uses 6

 ========================================
Remap of Struct str
  Remap 0 0
  Removing index 1
  Remap 2 1
  Remap 3 2
```

Dead field elimination - 505.mcf_r

```
struct node
{
  cost_t potential;
  int orientation;
  node_p child;
  node_p pred;
  node_p sibling;
  node_p sibling_prev;
  arc_p basic_arc;
  arc_p firstout, firstin;
  arc_p arc_tmp;
  flow_t flow;
  LONG depth;
  int number;
  int time;
};
```

defines.h

⇨

```
struct.node {
    0 : READWRITE
    1 : READWRITE
    2 : READWRITE
    3 : READWRITE
    4 : READWRITE
    5 : READWRITE
    6 : READWRITE
    7 : READWRITE
    8 : READWRITE
    9 : NOACCESS
    10 : READWRITE
    11 : READWRITE
    12 : READWRITE
    13 : READWRITE
}

Dead indices : 9
Parent struct : struct.arc struct.network
Number of uses 169
```

Dead field elimination - 505.mcf_r

```c
typedef struct network
{
  char inputfile[200];
  char clustfile[200];
  LONG n, n_trips;
  LONG max_m, m, m_org, m_impl;
  LONG max_residual_new_m, max_new_m;

  LONG primal_unbounded;
  LONG dual_unbounded;
  LONG perturbed;
  LONG feasible;
  LONG eps;
  LONG opt_tol;
  LONG feas_tol;
  LONG pert_val;
  LONG bigM;
  double optcost;
  cost_t ignore_impl;
  node_p nodes, stop_nodes;
  arc_p arcs, stop_arcs, sorted_arcs;
  arc_p dummy_arcs, stop_dummy;
  LONG iterations;
  LONG bound_exchanges;
  LONG nr_group, full_groups, max_elems;
} network_t;
```

defines.h

⇨

```
struct.network {
  0 : UNKNOWN
  1 : UNKNOWN
  2 : READWRITE
  3 : READWRITE
  4 : READWRITE
  5 : READWRITE
  6 : READWRITE
  7 : READWRITE
  8 : READWRITE
  9 : READWRITE
  10 : NOACCESS
  11 : NOACCESS
  12 : NOACCESS
  13 : WRITE
  14 : NOACCESS
  15 : NOACCESS
  16 : READ
  17 : NOACCESS
  18 : READWRITE
  19 : READWRITE
  20 : NOACCESS
  21 : READWRITE
  22 : READWRITE
  23 : READWRITE
  24 : READWRITE
  25 : READWRITE
  26 : READWRITE
  27 : READWRITE
  28 : UNKNOWN
  29 : UNKNOWN
  30 : READWRITE
  31 : READWRITE
  32 : READWRITE
}

Dead indices : 10 11 12 13 14 15 17 20
Number of uses 285
```

Future works

- Improve coverage of DLO
- Implement other data layout optimisations like structure peeling, structure splitting etc.

Thank You