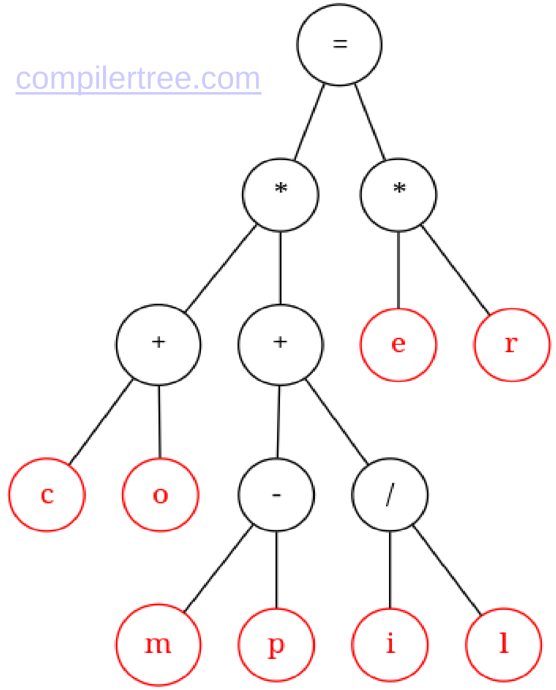
compilertree.com



Implementing Data Layout Optimizations in the LLVM Framework

Prashantha NR (Speaker)
CompilerTree Technologies
http://in.linkedin.com/in/mynrp

Vikram TV
CompilerTree Technologies
http://in.linkedin.com/in/tvvikram

Vaivaswatha N
CompilerTree Technologies
http://in.linkedin.com/in/vaivaswatha

Abstract

- Speed difference between processor and memory is increasing everyday
- Array/structure access patterns are modified for better cache behaviour
- We discuss the implementation of a few data layout modification optimizations in the LLVM framework
- All are Module Passes and implemented under lib/Transforms/DLO (currently not in llvm repo)

Outline

- Structure peeling, structure splitting and structure field reordering
- Struct-array copy
- Instance interleaving
- Array remapping

Outline

- **Structure peeling, structure splitting and structure field reordering**
- Struct-array copy
- Instance interleaving
- Array remapping

Structure Peeling: Motivation

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

A,C – Hot fields
B – Cold field

Structure Peeling: Motivation

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

A,C – Hot fields
B – Cold field

Peeled structures:

```
struct S.Hot {
  int A;
  int C;
};
```

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

Structure Splitting: Motivation

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

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

Presence of pointer to same type makes peeling invalid

Structure Splitting: Motivation

```
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;
};
```

Structure Peeling/Splitting
Implementation in LLVM

- Done in 5 phases:
  - Profile structure accesses
  - Legality
  - Reordering the fields
  - Create new structure types
  - Replace old structure accesses with new accesses

Structure Peeling/Splitting
Implementation in LLVM

- Profile structure accesses
  - Currently static profile is used
  - Each GetElementPtr of struct type is analyzed
  - Static profile count is maintained for each field of each struct
  - LoopInfo is used to get more accurate counts
  - This data is used in later phases to reorder the fields, decide whether to peel, split the structure

Structure Peeling/Splitting
Implementation in LLVM

- Legality
  - Not all structures can be peeled or split!
  - Cast to/from a given struct type
  - Escaped types / address of individual fields taken
  - Parameter types
  - Nested structures
  - Few others

Structure Peeling/Splitting
Implementation in LLVM

- Reordering the fields
  - Based on hotness of the fields
  - Based on affinity of the fields
  - Phase ordering problem

Structure Peeling/Splitting
Implementation in LLVM

- Creating new structure types
  - Decide to peel or split the structure
  - Split the structure if:
    - any of the fields of the StructType is a self referring pointer or
    - this StructType is a pointer in some other Struct Type
  - Otherwise peel
  - Don't split or peel if:
    - there is only one field in the structure or
    - fields already show good affinity or
    - just reordering the fields yield good profitability

Structure Peeling/Splitting
Implementation in LLVM

- Replace old structure accesses with new accesses:
  - Replace each getelementptr that computes address to a field of the old struct, with another one that computes the new address of that field.
  - Cold field access of a *split structure* need an additional getelementptr followed by a Load of the pointer in hot field that points to cold structure

Outline

- Structure peeling, structure splitting and structure field reordering
- **Struct-array copy**
- Instance interleaving
- Array remapping

Struct Array Copy: Motivation

Original access of structure field:

```
struct S {
  .
  int x;
  .
  .
} AoS[10000];

for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    sum = sum + AoS[j].x;
  }
}
```

After Structure to Array copy:

```
for (i = 0; i < n; i++) {
  temp[i] = AoS[i].x;
}

for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    sum = sum + temp[j];
  }
}
```

Struct Array Copy: Motivation

- We consider only Read-only loops. However, loops with writes can also be chosen if profitable
- Profitable when the access patterns of structure fields vary across the program – modifying the structure itself is not beneficial

Struct Array Copy
Implementation in LLVM

- Module Pass
- Analysis:
  - Identify Array of Structures
  - Identify loops with read-only struct field accesses
  - Legality
    - Trip count of the loop must be known before entering the loop
    - Type casts, escaped types, etc (as before)

Struct Array Copy
Implementation in LLVM

- Transformation
  - Allocate a temporary array of size equal to loop's trip count and structure field type
  - Create a loop before the read-only loop
  - Add instructions to initialize temporary array with specific field of AoS
  - Replace the AoS access in the read-only array with temporary array accesses. Index is translated if necessary
  - Free the temporary array after the loop

Outline

- Structure peeling, structure splitting and structure field reordering
- Struct-array copy
- **Instance interleaving**
- Array remapping

Instance Interleaving: Motivation

```
struct S {
  int a;
  int b;
  int c;
  int d;
} A[N];
```

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++)
    A[j].a /= 2;
  for (j = 10; j < (N/2); j++)
    A[j].b *= 5;
  for (j = 0; j < (N/4); j++)
    A[j].c *= 76;
  for (j = 0; j < N; j++)
    A[j].d /= 5;
}
```

Instance Interleaving: Motivation

```
struct S {
  int a;
  int b;
  int c;
  int d;
} A[N];
```

```
int a[N];
int b[N];
int c[N];
int d[N];
```

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++)
    A[j].a /= 2;
     a[j]
  for (j = 10; j < (N/2); j++)
    A[j].b *= 5;
     b[j]
  for (j = 0; j < (N/4); j++)
    A[j].c *= 76;
     c[j]
  for (j = 0; j < N; j++)
    A[j].d /= 5;
     d[j]
}
```

Array of structures to structure of arrays

Instance Interleaving
Implementation in LLVM

- Module Pass
- Identify arrays of structures whose different fields are accessed in different loops
- Identify the “length” of the array of structures
- Legality (as before)
- Create new arrays of size “length” and corresponding field types
- Modify getelementptr computations to reflect indexing a specific array, instead of an array of structures

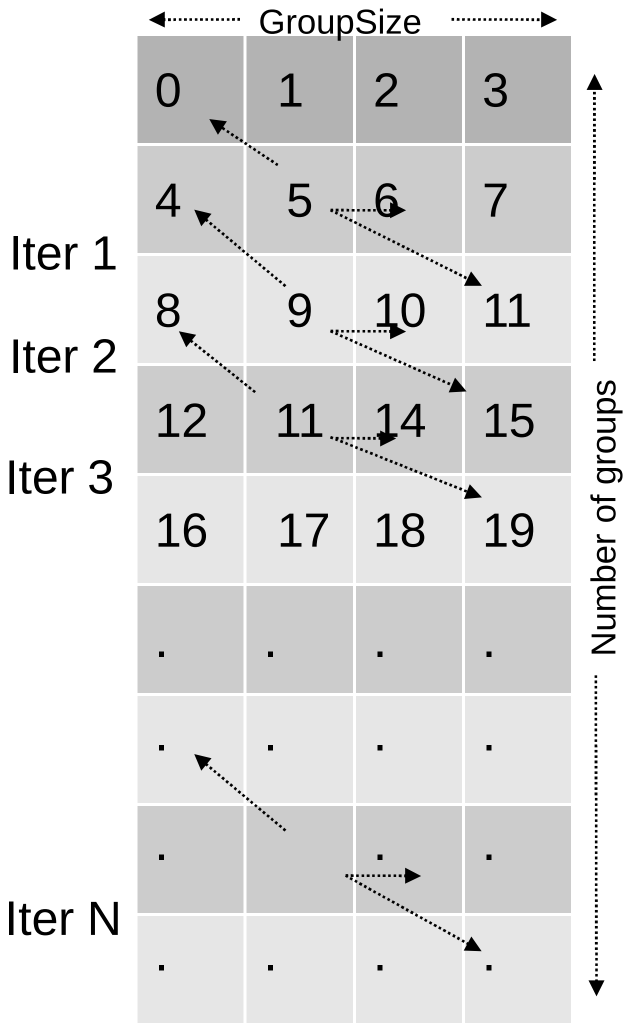
Outline

- Structure peeling, structure splitting and structure field reordering
- Struct-array copy
- Instance interleaving
- **Array remapping**

Array Remapping: Motivation

- Non-contiguous array accesses can be rearranged (remapped) to make them contiguous
- Array remapping is conceptually same as instance interleaving but happens with arrays

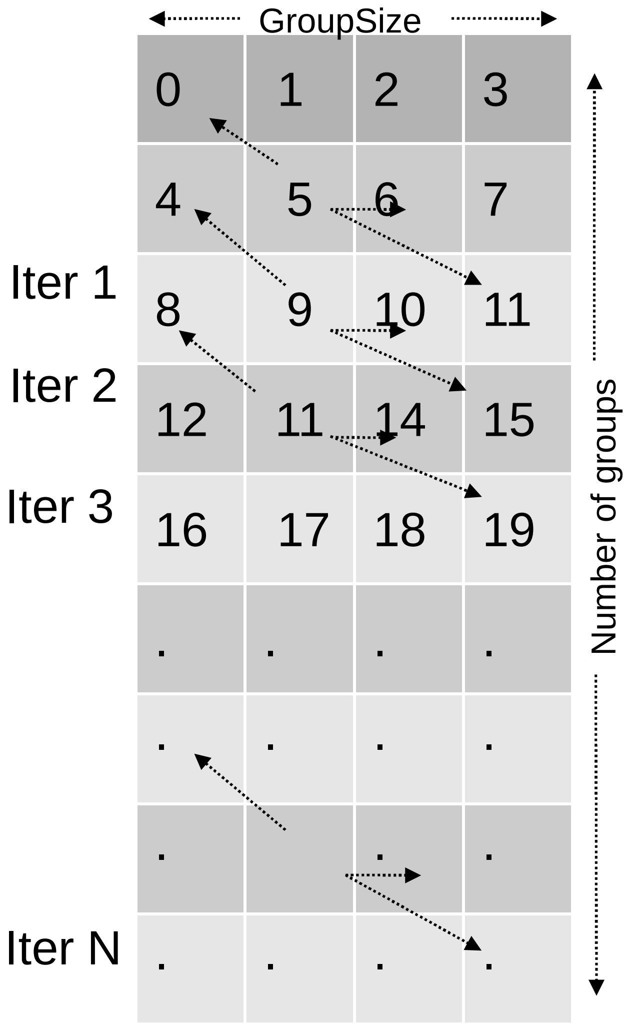
Array Remapping: Motivation




```
for (i = 5; i < 4004; i = i + 4)
{
      A[i + 6]
      A[i + 1]
      A[i + 0]
      A[i - 5]
}
```

- The locality here is very poor
  - No locality can be found in a single iteration
  - No locality can be found across iterations (think of large strides/less cache line size)
- What if we remap this array?

Array Remapping: Motivation



	Iter 1	Iter 2	Iter 3				Iter 1000	
0(0)	4(1)	8(2)	12	16	.	.	.	.
1(1000)	5(1001)	9(1002)	11	17	.	.		.
2(2000)	6(2001)	10(2002)	14	18	.	.	.	.
3(3000)	7(3001)	11(3002)	15	19	.	.	.	.

- Remap all accesses of A[i] as A[remap(i)]
- Fetching current iteration data also brings in the next iteration data. That is, we prefetch data of future “n” iterations in the current iteration

remap(i) = `i % GroupSize * NumberOfGroups + i / GroupSize`

Array Remapping
Implementation in LLVM

- Get Loop Information (IndVar, Stride, TripCount)
- Identify array remapping candidates
  - Get array access pattern by analyzing constants
    - GEP accesses are checked for A[i + const] type accesses
  - Identify groups
    - Remainder = constant % stride
    - Groups of constants which have same remainder are identified
    - All groups must have equal number of remainders

Array Remapping
Implementation in LLVM

- Compute new array-access locations
  - Insert new instructions in the entire module for every access of array A i.e. A[i] becomes A[remap(i)]
    - `remap(i) = i % GroupSize * NumberOfGroups + i / GroupSize`
    - (GroupSize = Stride, NumberOfGroups = TripCount)
  - ```
    %1 = add nsw i64 %indvars.iv, 19
    %arrayidx = getelementptr [100 x i32]* @a, i64 0, i64 %1
    ```

    becomes

    ```
    %1 = add nsw i64 %indvars.iv, 19
    %IterNum = urem i64 %1, %GroupSizeLD
    %Iter = mul i64 %IterNum, %NumGroupsLD
    %IterOffset = udiv i64 %1, %GroupSizeLD
    %NewIndex = add i64 %Iter, %IterOffset
    %arrayidx = getelementptr [100 x i32]* @a, i64 0, i64 %NewIndex
    ```
    ```

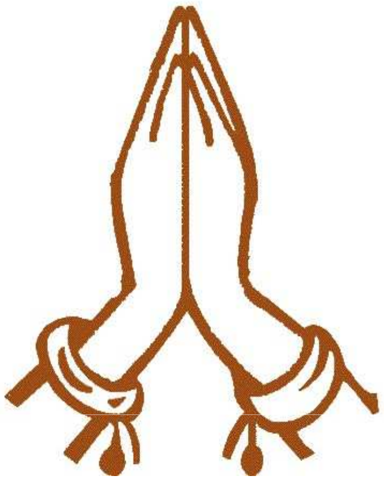
Experimental Observations

- Following benchmarks show significant gains with data layout optimizations
  - libquantum with struct splitting/peeling
  - mcf with array copy/instance interleaving
  - lbm with array remapping

Conclusion

- Different data layout optimizations are closely related
- Going forward ...
  - Framework for combined legality, profitability
  - Make Data layout optimizations work closely with Loop Optimizer (much harder)

Thank You



References

- D.C. Suresh et. Al. Multi-core scalability impacting compiler optimizations - Springer COMPUTER SCIENCE - RESEARCH AND DEVELOPMENT Volume 25, Numbers 1-2 (2010), 15-24,
- G Chakrabarti et. al. Structure Layout Optimizations in the Open64 Compiler
- Michael Lai – Extensions to Structure Layout Optimizations in the Open64 compiler
- Region Based Structure Layout Optimization by Selective Data Copying by Sandya S. Mannarswamy, R. Govindarajan and Rishi Surendran