Automatic indirect memory access instructions generation for pointer chasing patterns

Przemysław Ossowski

Co-authors:

Sebastian Szkoda, Adam Perdeusz, Łukasz Odzioba, Przemysław Karpiński, Marcin Koss, Marek M. Landowski

intel®

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Pointer chasing

Memory access characteristics

- A chain of dependent loads
- Serialized address generation and memory access

Pointer chasing

$x \leftarrow A[B[i] + j]$

Pointer chasing – an example of memory access pattern

Memory access characteristics

- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns

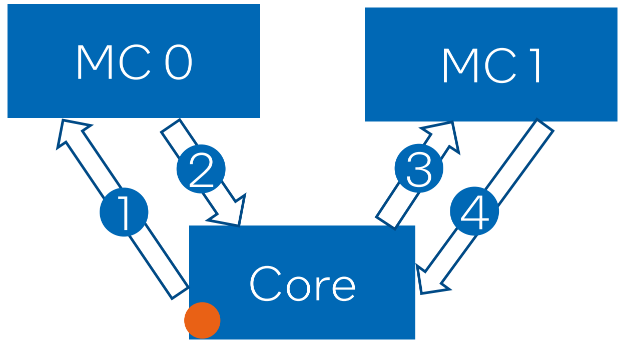
Pointer chasing

Memory access characteristics

- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns

x ⬅ A[B[i] + j]

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)

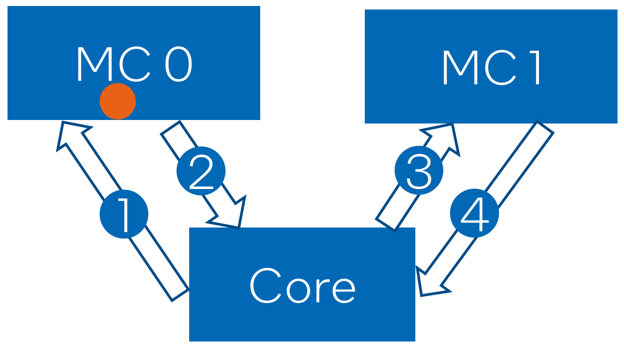
Pointer chasing

Memory access characteristics

- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns

x ⬅ A[B[i] + j]

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)

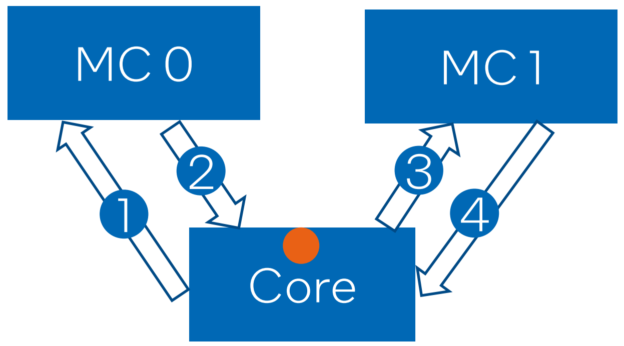
Pointer chasing

Memory access characteristics

- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns

x ⬅ A[B[i] + j]

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)

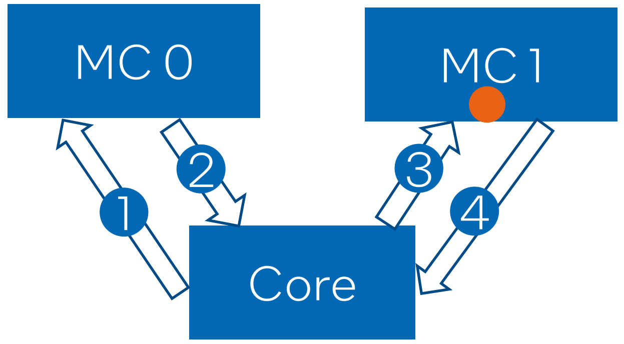
Pointer chasing

Memory access characteristics

- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns

x ⬅ A[B[i] + j]

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)

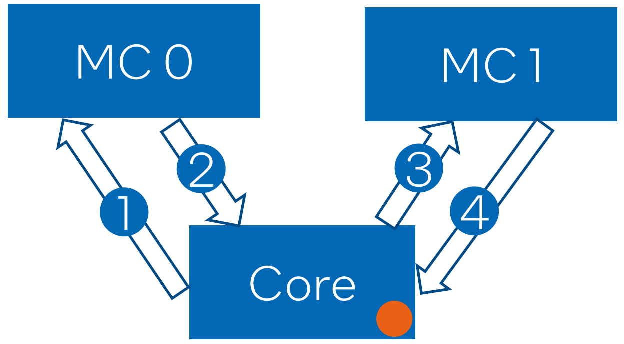
Pointer chasing

Memory access characteristics

- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns

$x \leftarrow A[B[i] + j]$

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)

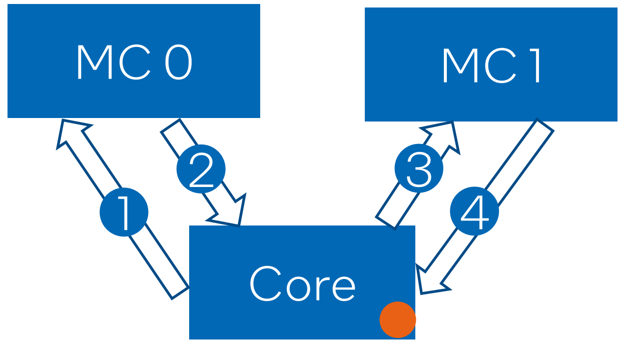
Pointer chasing

Memory access characteristics

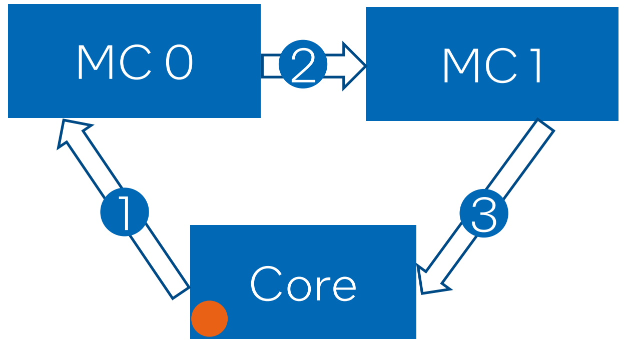
- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns
- Indirect Memory Access Instructions (IMAI)

$x \leftarrow A[B[i]+j]$

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)



Indirect load with pointer chasing scenario

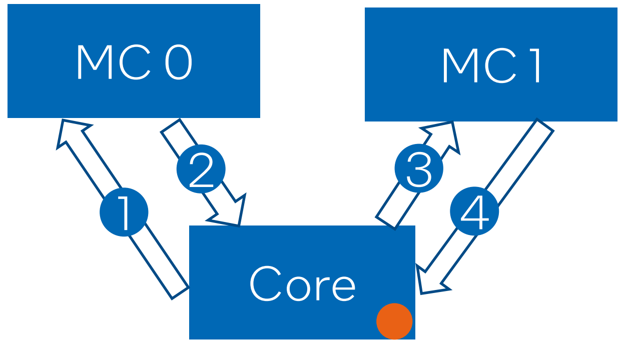
Pointer chasing

Memory access characteristics

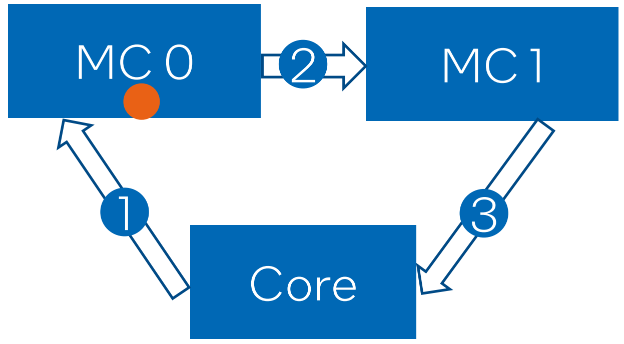
- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns
- Indirect Memory Access Instructions (IMAI)

$x \leftarrow A[B[i]+j]$

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)



Indirect load with pointer chasing scenario

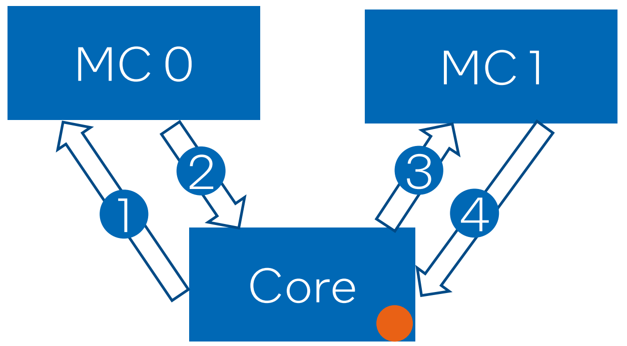
Pointer chasing

Memory access characteristics

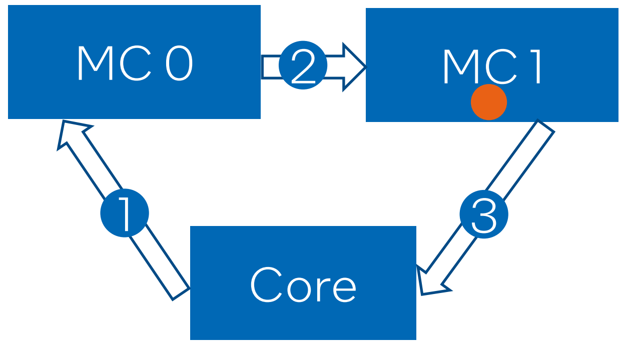
- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns
- Indirect Memory Access Instructions (IMAI)

x ⬅ A[B[i] + j]

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)



Indirect load with pointer chasing scenario

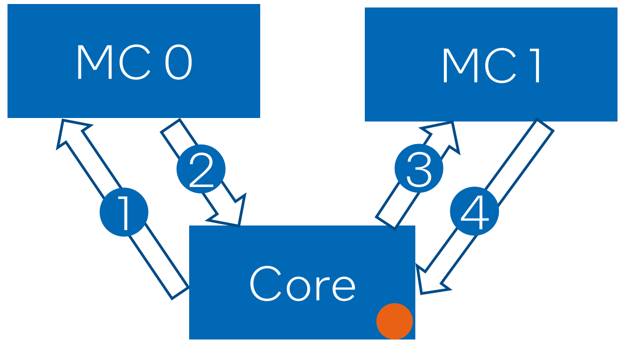
Pointer chasing

Memory access characteristics

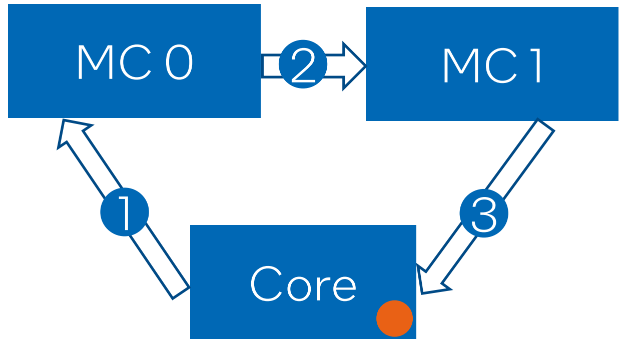
- A chain of dependent loads
- Serialized address generation and memory access
- Memory access patterns
- Indirect Memory Access Instructions (IMAI)

$x \leftarrow A[B[i]+j]$

Pointer chasing – an example of memory access pattern



Direct load with pointer chasing scenario – distributed memory example, each address is in a separate Memory Controller (MC)



Indirect load with pointer chasing scenario

Applying IMAI

New built-ins

```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

C function with indirect load operation...

```
  load       r0, r0, 0, 64
  load.idx   r1, r1, r0, 64
```

...compiled to two direct load instructions

Applying IMAI

New built-ins

- More complex instruction

```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

C function with indirect load operation...

```
  load       r0, r0, 0, 64
  load.idx   r1, r1, r0, 64
```

...compiled to two direct load instructions

```
  load.ind  r1, r0, r1, 64, O, 64, S
```

...compiled to indirect load instruction

Applying IMAI

New built-ins

- More complex instruction
- New built-ins
  - Complicated usage
  - Manual modification of code

```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

C function with indirect load operation…

```
  load       r0, r0, 0, 64
  load.idx   r1, r1, r0, 64
```

…compiled to two direct load instructions

```
  load.ind  r1, r0, r1, 64, O, 64, S
```

…compiled to indirect load instruction

```
double foo(double* x, u64* indices) {
  double loaded_value;
  __builtin_indirect_load_offset(
        &loaded_value, x, indices);
  return loaded_value;
}
```

C function with indirect load represented with a built-in

Applying IMAI

New built-ins

- More complex instruction
- New built-ins
  - Complicated usage
  - Manual modification of code
- LLVM IR with new intrinsic
  - Lacks common optimizations on load and store instructions

```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

C function with indirect load operation...

```
  load       r0, r0, 0, 64
  load.idx   r1, r1, r0, 64
```

...compiled to two direct load instructions

```
  load.ind  r1, r0, r1, 64, O, 64, S
```

...compiled to indirect load instruction

```
double foo(double* x, u64* indices) {
  double loaded_value;
  __builtin_indirect_load_offset(
        &loaded_value, x, indices);
  return loaded_value;
}
```

C function with indirect load represented with a built-in

* LLVM IR stands for LLVM *Intermediate Representation*

Automatic pattern detection

```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

C function with indirect load

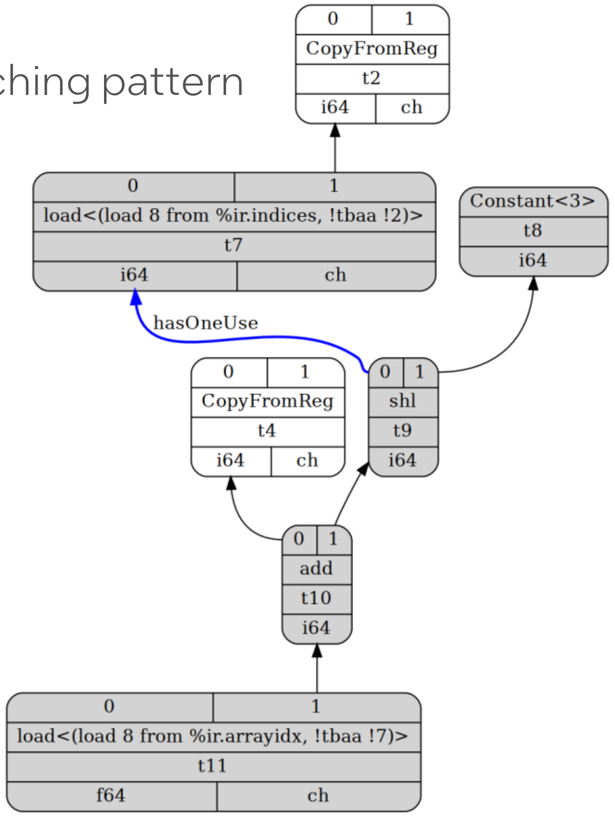
Automatic pattern detection

```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

C function with indirect load

- DAG Instruction Selection
  - Common optimizations on 'load' and 'store' instructions applied
  - Pattern with a constraint – first load '*hasOneUse*'

DAG with searching pattern



* DAG stands for *Directed Acyclic Graph*

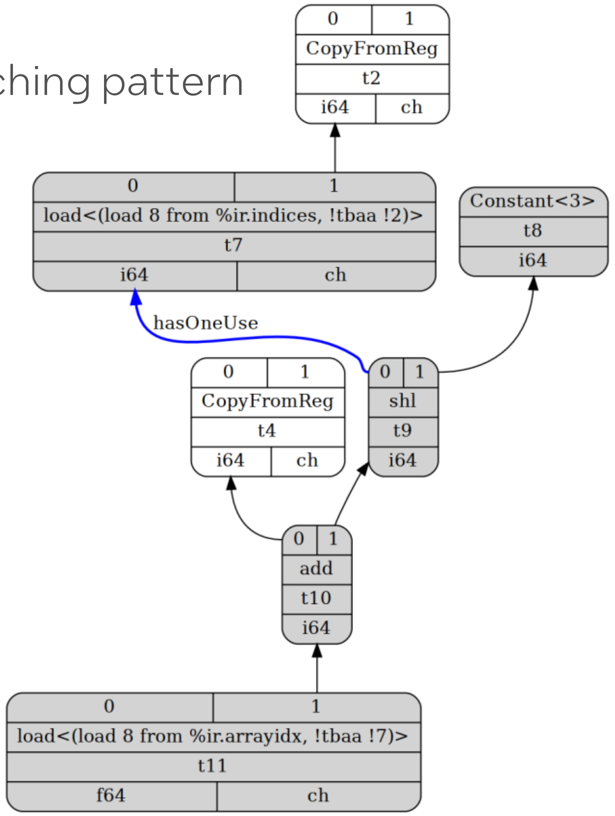
Automatic pattern detection

```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

C function with indirect load

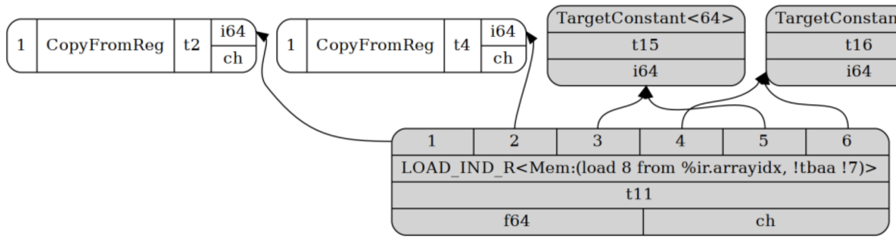
- DAG Instruction Selection
  - Common optimizations on 'load' and 'store' instructions applied
  - Pattern with a constraint – first load '*hasOneUse*'

DAG with searching pattern



CopyFromReg t2 i64 ch
load<(load 8 from %ir.indices, !tbaa !2)> t7 i64 ch
Constant<3> t8 i64
hasOneUse
CopyFromReg t4 i64 ch
shl t9 i64
add t10 i64
load<(load 8 from %ir.arrayidx, !tbaa !7)> t11 f64 ch

DAG with selected indirect load instruction – *LOAD_IND_R*



CopyFromReg t2 i64 ch
CopyFromReg t4 i64 ch
TargetConstant<64> t15 i64
TargetConstant<0> t16 i64
LOAD_IND_R<Mem:(load 8 from %ir.arrayidx, !tbaa !7)> t11 f64 ch

* DAG stands for *Directed Acyclic Graph*

Automatic pattern detection

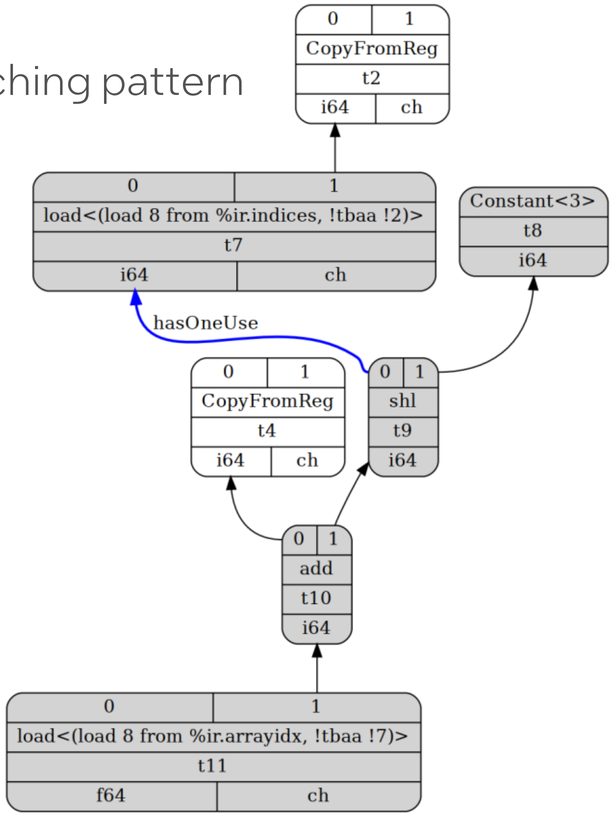
```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

C function with indirect load

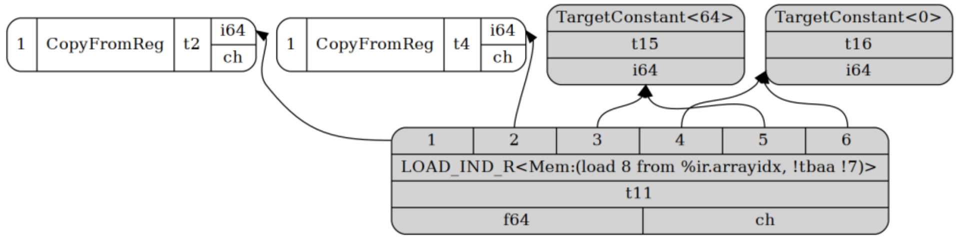
- DAG Instruction Selection
  - Common optimizations on 'load' and 'store' instructions applied
  - Pattern with a constraint – first load '*hasOneUse*'
  - It might be not enough – other constraints

* DAG stands for *Directed Acyclic Graph*

DAG with searching pattern



DAG with selected indirect load instruction – *LOAD_IND_R*



Intel® PIUMA

Programmable Integrated Unified Memory Architecture

- IMAI's on uncached data

Intel® PIUMA

Programmable Integrated Unified Memory Architecture

- IMAI's on uncached data
- Caching is configurable
  - User knows what is cached
  - Compiler doesn't know

Intel® PIUMA

Programmable Integrated Unified Memory Architecture

- IMAI's on uncached data
- Caching is configurable
  - User knows what is cached
  - Compiler doesn't know
- Compilation flag per module
  - Low flexibility

```c
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

Original C function with indirect load

Intel® PIUMA

Programmable Integrated Unified Memory Architecture

- IMAI's on uncached data
- Caching is configurable
  - User knows what is cached
  - Compiler doesn't know
- Compilation flag per module
  - Low flexibility
- **#pragma piuma indirect-allow**
  - Fine-granularity
  - Small code modification
  - Abstracts from instruction set details

```
double foo(double* x, u64* indices) {
    return x[*indices];
}
```

Original C function with indirect load

```
double foo(double* x, u64* indices) {
  #pragma piuma indirect-allow
  {
    return x[*indices];
  }
}
```

C function implementing indirect load with #pragma

Handling #pragma

- LLVM IR CodeGen – new basic blocks

```
define double @foo(i64* %indices, double* %x) {
entry:
 ...
 br label %allowind.start
allowind.start:
 %0 = load double*, double** %x.addr, align 8
 ...
 %3 = load double, double* %arrayidx, align 8
 br label %allowind.end
allowind.end:
 ret double %3
}
```

Handling #pragma

- LLVM IR CodeGen – new basic blocks

```
define double @foo(i64* %indices, double* %x) {
entry:
 ...
 br label %allowind.start
allowind.start:
 %0 = load double*, double** %x.addr, align 8
 ...
 %3 = load double, double* %arrayidx, align 8
 br label %allowind.end
allowind.end:
 ret double %3
}
```

- Pass - marking with Metadata

```
define double @foo(i64* %indices, double* %x) {
entry:
 %0 = load i64, i64* %indices, align 8, !allow.ind
 %idx = getelementptr inbounds double, double* %x, i64 %0
 %1 = load double, double* %idx, align 8, !allow.ind
 ret double %1
}
```

intel.

Handling #pragma

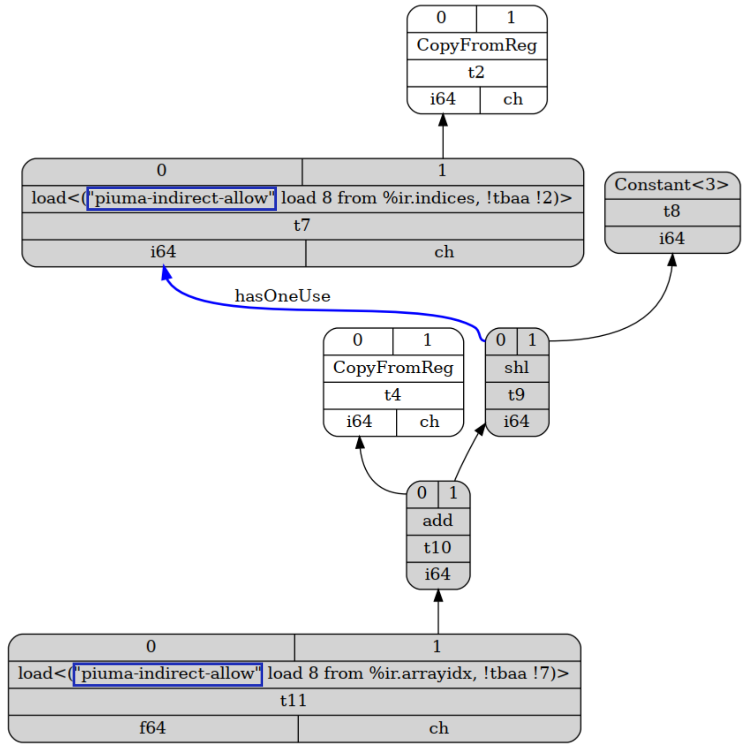
- LLVM IR CodeGen – new basic blocks

```
define double @foo(i64* %indices, double* %x) {
entry:
 ...
 br label %allowind.start
allowind.start:
 %0 = load double*, double** %x.addr, align 8
 ...
 %3 = load double, double* %arrayidx, align 8
 br label %allowind.end
allowind.end:
 ret double %3
}
```

- Pass - marking with Metadata

```
define double @foo(i64* %indices, double* %x) {
entry:
 %0 = load i64, i64* %indices, align 8, !allow.ind
 %idx = getelementptr inbounds double, double* %x, i64 %0
 %1 = load double, double* %idx, align 8, !allow.ind
 ret double %1
}
```

- DAG Builder – marking with new *MachineMemOperand::Flags*



0 1 CopyFromReg t2 i64 ch
0 1 load<("piuma-indirect-allow" load 8 from %ir.indices, !tbaa !2)> t7 i64 ch
Constant<3> t8 i64
hasOneUse
0 1 CopyFromReg t4 i64 ch
0 1 shl t9 i64
0 1 add t10 i64
0 1 load<("piuma-indirect-allow" load 8 from %ir.arrayidx, !tbaa !7)> t11 f64 ch

Pattern applied only on *MemSDNodes* marked with *MOIndirectAllow* flag

Summary

IMAI in Clang and LLVM:

- Built-in functions
- Automated pattern detection
- Compilation flag
- #pragma

Acknowledgments:

- Josh Fryman, Mariusz Sikora, Radosław Tyl, Maciej Grzywacz, Intel® PIUMA Team

More about Intel® PIUMA:

- https://arxiv.org/pdf/2010.06277.pdf