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

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.

Distribution Statement A – Approved for Public Release, Distribution Unlimited
Pointer chasing

Memory access characteristics

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

$\text{x } \leftarrow \text{A[B[i] + j]}$

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

LLVM Developers’ Meeting 2022
Pointer chasing

Memory access characteristics

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

define 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

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

C function with indirect load operation...

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

...compiled to two direct load instructions

```c
load.ind r1, r0, r1, 64, 0, 64, S
```

...compiled to indirect load instruction
Applying IMAI

New built-ins

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

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

C function with indirect load operation...

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

...compiled to two direct load instructions

```c
load.ind r1, r0, r1, 64, 0, 64, S
```

...compiled to indirect load instruction

```c
double foo(double* x, u64* indices) {
    double loaded_value;
    __builtin间接加载偏移(
        &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

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

C function with indirect load operation...

```assembly
load r0, r0, 0, 64
load.idx r1, r1, r0, 64
```
...compiled to two direct load instructions

```assembly
load.ind r1, r0, r1, 64, 0, 64, S
```
...compiled to indirect load instruction

```c
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

1 double foo(double* x, u64* indices) {
2     return x[*indices];
3 }
Automatic pattern detection

```c
1 double foo(double* x, u64* indices) {
2     return x[*indices];
3 }
```

C function with indirect load

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

* DAG stands for Directed Acyclic Graph
Automatic pattern detection

```c
1  double foo(double* x, u64* indices) {
2      return x[indices];
3  }
```

C function with indirect load

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

* DAG stands for Directed Acyclic Graph
Automatic pattern detection

```
1 double foo(double* x, u64* indices) {
2     return x[*indices];
3 }
```

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
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
1 double foo(double* x, u64* indices) {
2     return x[*indices];
3 }
```

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

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

Original C function with indirect load

```c
int 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

```c
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

```llvm
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

```llvm
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
}
```
Handling #pragma

- **LLVM IR CodeGen – new basic blocks**

```assembly
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**

```assembly
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**

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:
