The LLVM compiler framework

The LLVM Machine Code Analyzer

Daniele Cattaneo

Politecnico di Milano

2021-03-16

Contents

LLVM Tools

LLVM provides several **tools** for various purposes:

- Working with LLVM-IR modules
  - `llvm-as`, `llvm-dis`, `opt`, `llc`, `lli`, `llvm-link`
- GNU binutils replacements
  - `llvm-addr2line`, `llvm-ar`, `llvm-cxxfilt`, `llvm-objcopy`
  - `llvm-objdump`, `llvm-ranlib`, `llvm-readelf`
  - `llvm-size`, `llvm-strings`, `llvm-strip`
- LLVM-Developer-specific tools
  - `FileCheck`, `tblgen`, `lit`, `llvm-build`
  - `llvm-exegesis`, `llvm-pdbutil`, `llvm-locstats`
- …

LLVM Tools

LLVM provides several **tools** for various purposes:

- Working with LLVM-IR modules
  - `llvm-as`, `llvm-dis`, `opt`, `llc`, `lli`, `llvm-link`
- GNU binutils replacements
  - `llvm-addr2line`, `llvm-ar`, `llvm-cxxfilt`, `llvm-objcopy`
  - `llvm-objdump`, `llvm-ranlib`, `llvm-readelf`
  - `llvm-size`, `llvm-strings`, `llvm-strip`
- LLVM-Developer-specific tools
  - `FileCheck`, `tblgen`, `lit`, `llvm-build`
  - `llvm-exegesis`, `llvm-pdbutil`, `llvm-locstats`
- …
- **Performance analysis**
  - **`llvm-mca`**

LLVM-MCA

`llvm-mca` is a **performance analysis tool**

- It uses information available in LLVM (e.g. scheduling models)
- Evaluates statically the performance of machine code in a specific CPU.

Performance is measured in terms of:

- throughput
- processor resource consumption

The tool currently works for processors with an out-of-order backend for which there is a scheduling model available in LLVM.

Throughput?

Throughput is defined as
***the total amount of work done in a given time**.[1]*

Translated in terms of *machine code instructions*:

Throughput is defined as
***the total number of instructions executed in a given time**.*

According to this definition,
the measurement unit of throughput is the **MIPS**
(Mega-Instructions Per Second)

Incidentally (or not...) this is also the name of a CPU architecture!

Throughput?

However...

In modern processors the machine code instruction is not the fundamental unit of computation anymore!

Throughput?

In modern processors:

1. Instructions are usually represented in a **CISC-like** compact encoding
   - This is now true even for architectures that have a RISC heritage like ARM
   - Memory latency $\gg$ decoding latency $\Rightarrow$ Code density $\gg$ Decoding overhead
2. The CISC-like encoding is *transformed on-the-fly* to **microcode instructions**
   - Select important instructions correspond directly to single microcode instructions
   - Implementation of more complex instructions stored as **microcode programs** in an internal ROM
   - Small amount of on-chip SRAM available for patches (microcode updates)

Throughput?

In modern processors:

3. The microcode instructions are put in a queue and issued to the functional units
   - Multiple instructions can be evicted from the queue at the same time
   - Superscalar architecture: multiple **functional units** can act in parallel to serve the microinstructions
   - **Reservation stations** (in Intel parlance: **Execution Ports**) are decoupled from the functional units (one station can serve multiple functional units)
   - Register accesses are handled using **explicit register renaming**

The entire execution pipeline revolves around **microinstructions**

Throughput?

In modern processors merely measuring the instructions executed per seconds is **insufficient**!

Instead we would like to measure the number of **microinstructions** per second.

Throughput?

It's not that easy!

- The microprocessor does not give any visibility into its microcode
- Microprocessor manufacturers consider their internal microcode ISA a **key competitive advantage** and won't release any detailed information about it

Performance analysis of a program is a **imperfect statistical science** based on **approximations** of the behavior of a processor derived from **black-box reverse-engineering**.

Processor Resource Consumption?

Another key element in performance analysis is that **we cannot derive realistic numbers without a reasonably accurate model of the processor**

- Latency of each Functional Unit
- Throughput of each Functional Unit
- Number and kind of Functional Units
- Number of Reservation Stations
- Size of the Internal Register File
- Number of Pipeline Stages (Issue Width)
- Microinstruction cache size
- Number of microinstructions per MC instruction

And there's already a lot to it, even though we are ignoring RAM fetch/store, branch prediction, forwarding...

Processor Resource Consumption?

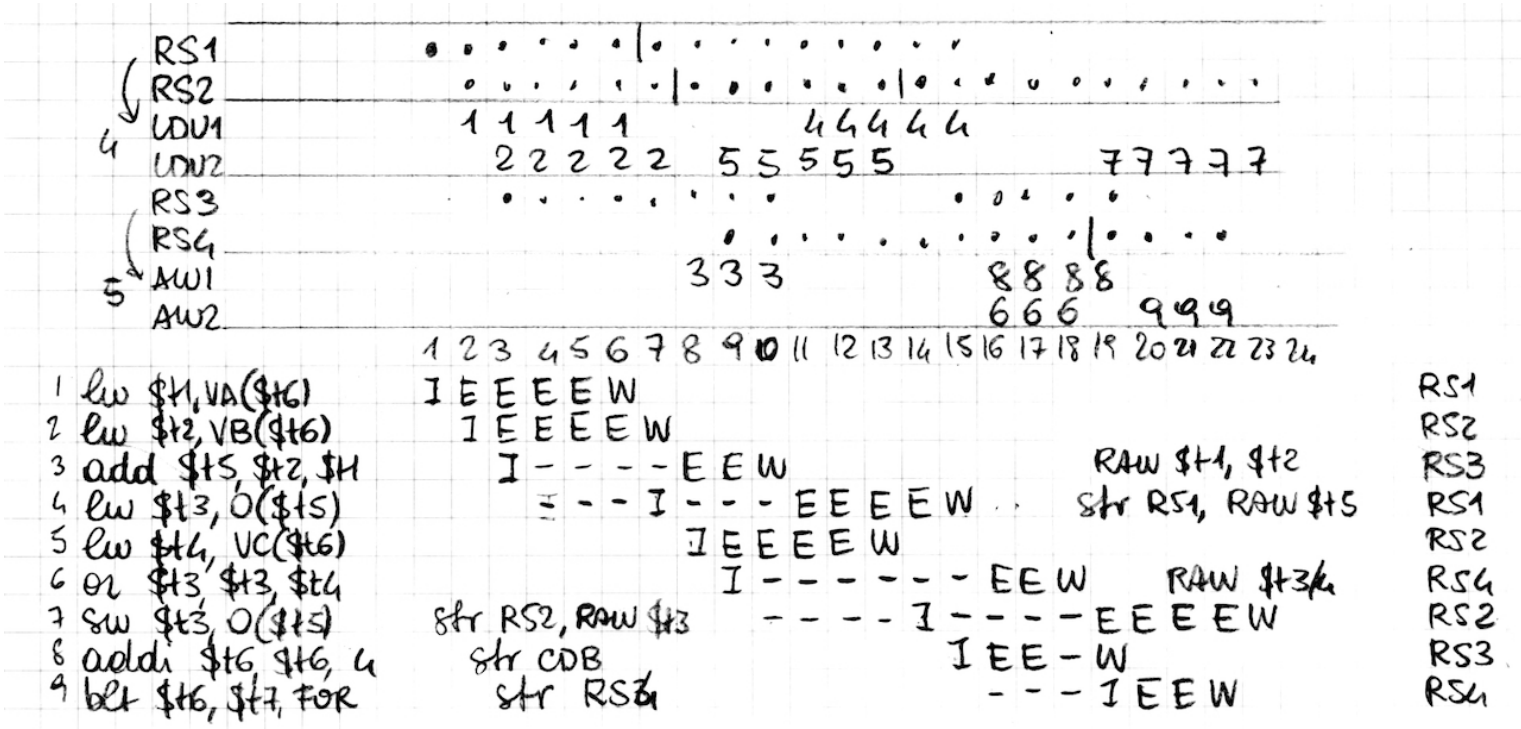
This is why **processor resource consumption** is important!

- It allows to compute the throughput of a piece of code accurately
- It gives information on the **margin of improvement** that can be achieved

Why LLVM-MCA

In the past, performance analysis was done:

- by **experimentation** with a wide variety of test machines and CPUs
- with **manual analysis** using microarchitectural information



Why LLVM-MCA

By using an automated tool, we can **improve** the quality of the analysis, and make it easier to perform.

```
Timeline view:
                    012345
Index     0123456789

[0,0]     DeeER.    .    .   vmulps   %xmm0, %xmm1, %xmm2
[0,1]     D==eeeER  .    .   vhaddps  %xmm2, %xmm2, %xmm3
[0,2]     .D====eeeER    .   vhaddps  %xmm3, %xmm3, %xmm4
[1,0]     .DeeE-----R    .   vmulps   %xmm0, %xmm1, %xmm2
[1,1]     . D=eeeE---R   .   vhaddps  %xmm2, %xmm2, %xmm3
[1,2]     . D====eeeER   .   vhaddps  %xmm3, %xmm3, %xmm4
[2,0]     .  DeeE-----R  .   vmulps   %xmm0, %xmm1, %xmm2
[2,1]     .  D====eeeER  .   vhaddps  %xmm2, %xmm2, %xmm3
[2,2]     .   D======eeeER   vhaddps  %xmm3, %xmm3, %xmm4


Average Wait times (based on the timeline view):

      [0]    [1]    [2]    [3]
0.     3     1.0    1.0    3.3       vmulps   %xmm0, %xmm1, %xmm2
1.     3     3.3    0.7    1.0       vhaddps  %xmm2, %xmm2, %xmm3
2.     3     5.7    0.0    0.0       vhaddps  %xmm3, %xmm3, %xmm4
       3     3.3    0.5    1.4       <total>
```

How LLVM-MCA works

LLVM incorporates a **dataset of instruction latency information** used for estimating the performance impact of various instructions.

LLVM-MCA expands this information to incorporate **machine-specific models** about several CPU cores.

LLVM-MCA uses the **backend** part of LLVM to do its job!

Who needs LLVM-MCA

1. Developers of performance-sensitive applications
   - Helps deciding where to vectorize
   - Tells you if the algorithm works well for the CPU
   - Analyzes if the implementation is memory-bound or not
   - Helps specifically if you are working in assembly language!
2. Compiler writers who want to check if their optimization is a good idea
   - Well it's a bit too low-level for that...

Fortunately, most of the time the compiler does a plenty good job optimizing stuff!

Why you don’t need LLVM-MCA!

In the days of the 8 and 16 bit processors, **code optimization** mostly meant *choosing the right instruction* because the memory was as fast as the CPU, there was no pipelining or speculative execution, and complex algorithms didn’t fit into the RAM.

Sometimes inexperienced programmers have the *impression* this is still true today.

This is not true anymore!

Instruction choice is **largely irrelevant**!

Why you don’t need LLVM-MCA!

What is **truly** important:

1. Complexity of your algorithm
2. Access locality
3. Explicit MIMD parallelization (threads)
4. Explicit SIMD parallelization (vector intrinsics)
5. Compiler flags: `-O3 -ffast-math`

Points 4 and 5 are where LLVM-MCA helps, but they are also the least important ones!

Contents

Preface

Let's look at how MCA allows us to predict the performance of some code **without running it**

We will look at:

- C language source code
- Assembly language produced by the compiler
- Analysis of the assembly language made by LLVM-MCA

We will learn about how to use LLVM-MCA and how to interpret its results as we go.

Contents

Sum Reduction

Algorithm

Given a vector $V = [v_1, v_2, \ldots v_N]$ we compute:

$$x = \sum_{i=1}^{N} v_i$$

Obvious Implementation

```
for (int i=0; i<N; i++) {
  accum += buffer[i];
}
```

- Let's measure the performance we get with LLVM-MCA!

Marking portions of code

- We need to tell LLVM-MCA which portion of code we want to measure!

Important things to know:

1. branch instructions are ignored
2. every piece of code is assumed to be executed in a loop
3. the behavior of memory accesses is not simulated

Consequences:

- We need to debug data locality issues **first**
- The only thing that can be analyzed feasibly is straight-line code

Marking portions of code

In this example there is not much possibility for data locality issues (only sequential reads...)

From what we said it's clear that the portion of code we are interested in is the **body of the loop**

To make LLVM-MCA analyze it we have to enclose it **in the assembly code** with:

```
# LLVM-MCA-BEGIN block_name
# LLVM-MCA-END block_name
```

block_name is any string identifier you want

Marking portions of code

If we are working in the C language, we can use `asm` blocks:

Obvious Implementation (annotated)

```
for (int i=0; i<N; i++) {
  __asm volatile("# LLVM-MCA-BEGIN plus_accum_body");
  accum += buffer[i];
  __asm volatile("# LLVM-MCA-END plus_accum_body");
}
```

Warning: Adding `asm` blocks can prevent optimizations!

In this particular case, it prevents **loop unrolling**. For the purpose of the example it's not a problem because we will talk about loop unrolling in a bit...

Marking portions of code

Let's compile with `-O3`...

Obvious Implementation (compiled)

```
        xorl    %eax, %eax
        vxorpd  %xmm0, %xmm0, %xmm0
LBB0_2:
        ## InlineAsm Start
        ## LLVM-MCA-BEGIN plus_accum_body
        ## InlineAsm End
        vaddsd  (%rbx,%rax,8), %xmm0, %xmm0
        ## InlineAsm Start
        ## LLVM-MCA-END plus_accum_body
        ## InlineAsm End
        addq    $1, %rax
        cmpq    $80000000, %rax
        jne     LBB0_2
```

Running LLVM-MCA

It's now time to actually run the analysis with LLVM-MCA.

There are several kinds of options available:

- Machine type options (defaults autodetected from host)
- Analysis configuration options

`-iterations`	Number of iterations to simulate
`-register-file-size`	Size of the shadow register file
`-bottleneck-analysis`	Enables bottleneck analysis

- Options for toggling **views** of the analysis data:

`-resource-pressure`	Enable the resource pressure view.
`-instruction-info`	Enable the instruction info view.
`-instruction-tables`	Prints resource pressure information based on the static information available from the processor model.

Running LLVM-MCA

We'll use the following options:

- `-timeline`
- `-all-stats`
- `-bottleneck-analysis`

The command line syntax is very simple.

LLVM-MCA invocation command

```
$ llvm-mca add_reduction.c \
    -timeline -all-stats -bottleneck-analysis
```

The Views of LLVM-MCA

The output is partitioned for each **code region** and for each **data view**.

In our example there is just one region.

The main view presents generic information about the simulation.

The Main View

```
Iterations:        100
Instructions:      100
Total Cycles:      408
Total uOps:        200

Dispatch Width:    6
uOps Per Cycle:    0.49
IPC:               0.25
Block RThroughput: 0.5
```

`Iterations:` Number of iterations of the loop that have been simulated.

`Instructions:` Total number of instructions simulated.

`Total Cycles:` Number of clock cycles taken for performing the code.

`Total uOps:` Number of CPU microcode instructions issued.

The Views of LLVM-MCA

The Main View

```
Iterations:        100
Instructions:      100
Total Cycles:      408
Total uOps:        200

Dispatch Width:    6
uOps Per Cycle:    0.49
IPC:               0.25
Block RThroughput: 0.5
```

`Dispatch Width:` Maximum number of microinstructions that can be issued simultaneously

`uOps Per Cycle:` Average number of microinstructions executed per clock cycle.

`IPC:` Average number of instructions executed per clock cycle.

`Block RThroughput:` Average **reciprocal throughput** of the code block

Reciprocal Throughput

The **throughput** is the *amount of work done in a given time*

The **reciprocal throughput** is *how much time is required to perform the unit of work*

unit of work	=	one iteration of the loop
time	=	clock cycles

The reciprocal throughput given by LLVM-MCA is **theoretical**.
It does not take into account data dependencies amongst multiple iterations.

It depends from the resource usage of the block of code.

Reciprocal Throughput

The IPC (Instructions per Cycle) estimation **does** take into account data dependencies.

Since IPC (like MIPS) is a measure of **throughput**, we can derive the **maximum throughput** by computing the **inverse of the RThroughput**

$$\text{IPC} \leq \frac{\text{N}_{\text{block}}}{\text{Block RThroughput}}$$

$$\text{N}_{\text{block}} = \frac{\text{Instructions}}{\text{Iterations}}$$

Data dependencies are important!

Let's try calculating the maximum IPC for our example...

$$\text{IPC}_{max} = \frac{\text{N}_{block}}{\text{Block RThroughput}} = \frac{1}{0.5} = 2$$

The maximum IPC is much higher than the **actual** IPC (0.25)!

This is because **every instruction executed depends on the result of the previous one**

The Timeline View

We can verify visually if an instruction is waiting for another one using the **timeline view**.

Timeline View

```
Timeline view:
                    0123456789
Index     0123456789          01234567

[0,0]     DeeeeeeeeeER   .    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[1,0]     D====eeeeeeeeeER    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[2,0]     D========eeeeeeeeeER.    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[3,0]     .D==========eeeeeeeeeER . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[4,0]     .D==============eeeeeeeeeER   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
```

The Timeline View

Timeline View

```
Timeline view:
                    0123456789
Index     0123456789          01234567

[0,0]     DeeeeeeeeeER   .    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[1,0]     D====eeeeeeeeeER    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[2,0]     D========eeeeeeeeeER.    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[3,0]     .D==========eeeeeeeeeER . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[4,0]     .D==============eeeeeeeeeER   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
```

On the *left* the **index** column contains the loop iteration counter and the index of the instruction in the block.

On the *right* the instructions are shown.

In the *middle* there is a **timing graph** showing in which stage the instruction is in at each CPU cycle (from left to right).

The Timeline View

Timeline View

```
Timeline view:
                    0123456789
Index     0123456789          01234567

[0,0]     DeeeeeeeeeER   .    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[1,0]     D====eeeeeeeeeER    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[2,0]     D========eeeeeeeeeER.    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[3,0]     .D==========eeeeeeeeeER . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[4,0]     .D==============eeeeeeeeeER   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
```

- **D** The instruction is **dispatched**.
  Reservation station and virtual register allocated.
- **e** The instruction is being **executed**
- **E** The instruction **finishes execution**.
  The result is ready and the reservation station is freed
- **R** The instruction is **retired**
  The virtual register is freed.

The Instruction Info View

Instruction Info View

```
Instruction Info:
[1]: #uOps
[2]: Latency
[3]: RThroughput
[4]: MayLoad
[5]: MayStore
[6]: HasSideEffects (U)

[1]    [2]    [3]    [4]    [5]    [6]    Instructions:
 2      9     0.50    *                    vaddsd        (%rbx,%rax,8), %xmm0, %xmm0
```

- The dispatch width we saw earlier is **6**
- In the timeline view, at most **3** instructions are dispatched together
- In fact, can see in the **instruction info view** that our instructions take 2 microinstructions to execute!

The Instruction Info View

Instruction Info View

```
Instruction Info:
[1]: #uOps
[2]: Latency
[3]: RThroughput
[4]: MayLoad
[5]: MayStore
[6]: HasSideEffects (U)

[1]    [2]    [3]    [4]    [5]    [6]    Instructions:
 2      9     0.50    *                    vaddsd        (%rbx,%rax,8), %xmm0, %xmm0
```

Other interesting information shown here:

`Latency` Number of cycles required to execute the instruction

`RThroughput` Reciprocal of the number of such instructions that can execute simultaneously

The Timeline and Microinstructions

Timeline View

```
Timeline view:
                    0123456789
Index     0123456789          01234567

[0,0]     DeeeeeeeeeER   .    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[1,0]     D====eeeeeeeeeER    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[2,0]     D========eeeeeeeeeER.    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[3,0]     .D==========eeeeeeeeeER . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[4,0]     .D==============eeeeeeeeeER   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
```

These two microinstructions (presumably!) perform, in order:

1. the **load from memory** of the next number to accumulate
2. the **sum** of the loaded number and the value in the register

This is why the executions partially overlap — only the **sum** microinstruction is blocking.

The Bottleneck Analysis

LLVM-MCA is able to automatically digest all this information for us, producing an analysis of the instruction sequence which has the biggest impact on the execution.

Bottleneck View

```
Cycles with backend pressure increase [ 38.24% ]
Throughput Bottlenecks:
  Resource Pressure       [ 0.00% ]
  Data Dependencies:      [ 38.24% ]
  - Register Dependencies [ 38.24% ]
  - Memory Dependencies   [ 0.00% ]

Critical sequence based on the simulation:

              Instruction                                 Dependency Information
 +----< 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0
 |
 |    < loop carried >
 |
 +----> 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0       ## REGISTER dependency:  %xmm0
 |
 |    < loop carried >
 |
 +----> 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0       ## REGISTER dependency:  %xmm0
```

The Bottleneck Analysis

We can see right away that there is a data dependency on the `%xmm0` register.

Bottleneck View

```
Cycles with backend pressure increase [ 38.24% ]
Throughput Bottlenecks:
  Resource Pressure       [ 0.00% ]
  Data Dependencies:      [ 38.24% ]
  - Register Dependencies [ 38.24% ]
  - Memory Dependencies   [ 0.00% ]

Critical sequence based on the simulation:

              Instruction                                 Dependency Information
 +----< 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0
 |
 |    < loop carried >
 |
 +----> 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0       ## REGISTER dependency:  %xmm0
 |
 |    < loop carried >
 |
 +----> 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0       ## REGISTER dependency:  %xmm0
```

Contents

Hold on a sec, you cheater bastard!

Wait... weren't there **some other instructions in the loop** apart from the sum!?

Exhibit A

```
        xorl    %eax, %eax
        vxorpd  %xmm0, %xmm0, %xmm0
LBB0_2:
        ## InlineAsm Start
        ## LLVM-MCA-BEGIN plus_accum_body
        ## InlineAsm End
        vaddsd  (%rbx,%rax,8), %xmm0, %xmm0
        ## InlineAsm Start
        ## LLVM-MCA-END plus_accum_body
        ## InlineAsm End
        addq    $1, %rax
        cmpq    $80000000, %rax
        jne     LBB0_2
```

Hold on a sec, you cheater bastard!

The results we looked at must have been completely wrong!

Exhibit A

```
        xorl    %eax, %eax
        vxorpd  %xmm0, %xmm0, %xmm0
LBB0_2:
        ## InlineAsm Start
        ## LLVM-MCA-BEGIN plus_accum_body
        ## InlineAsm End
        vaddsd  (%rbx,%rax,8), %xmm0, %xmm0
        ## InlineAsm Start
        ## LLVM-MCA-END plus_accum_body
        ## InlineAsm End
        addq    $1, %rax
        cmpq    $80000000, %rax
        jne     LBB0_2
```

I plead guilty!

I surrender! You are right! I apologize! Let's annotate all the instructions in the loop!

Remedial actions

```
LBB0_2:
        ## LLVM-MCA-BEGIN loop_trail
        vaddsd  (%rbx,%rax,8), %xmm0, %xmm0
        addq    $1, %rax
        cmpq    $80000000, %rax
        jne     LBB0_2
## %bb.3:
        ## LLVM-MCA-END loop_trail
```

I plead guilty!

Let's look at what's changed:

Before

```
Iterations:         100
Instructions:       100
Total Cycles:       408
Total uOps:         200

Dispatch Width:     6
uOps Per Cycle:     0.49
IPC:                0.25
Block RThroughput: 0.5
```

After

```
Iterations:         100
Instructions:       400
Total Cycles:       408
Total uOps:         500

Dispatch Width:     6
uOps Per Cycle:     1.23
IPC:                0.98
Block RThroughput: 0.8
```

IPC, *uOps Per Cycle* and *Block RThroughput* are very different!

I plead guilty!

Bottleneck View

```
Critical sequence based on the simulation:

              Instruction                                 Dependency Information
 +----< 1.    addq      $1, %rax
 |
 |    < loop carried >
 |
 |      0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0
 +----> 1.    addq      $1, %rax                          ## REGISTER dependency:  %rax
 |      2.    cmpq      $80000000, %rax
 |      3.    jne       LBB0_2
 |
 |    < loop carried >
 |
 +----> 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0       ## REGISTER dependency:  %rax
```

The bottleneck view unambiguously says that the biggest issue here is the update of the induction variable of the loop!

I plead guilty!

Bottleneck View

```
Critical sequence based on the simulation:

              Instruction                                 Dependency Information
 +----< 1.    addq      $1, %rax
 |
 |    < loop carried >
 |
 |      0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0
 +----> 1.    addq      $1, %rax                          ## REGISTER dependency:  %rax
 |      2.    cmpq      $80000000, %rax
 |      3.    jne       LBB0_2
 |
 |    < loop carried >
 |
 +----> 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0       ## REGISTER dependency:  %rax
```

The perfect cure to this horrible disease is a classic and ever-lasting optimization: **loop unrolling**!

First Performance Improvement

Now we're talking!

```
LBB0_2:
        vaddsd  (%rbx,%rax,8), %xmm0, %xmm0
        vaddsd  8(%rbx,%rax,8), %xmm0, %xmm0
        vaddsd  16(%rbx,%rax,8), %xmm0, %xmm0
        vaddsd  24(%rbx,%rax,8), %xmm0, %xmm0
        vaddsd  32(%rbx,%rax,8), %xmm0, %xmm0
        vaddsd  40(%rbx,%rax,8), %xmm0, %xmm0
        vaddsd  48(%rbx,%rax,8), %xmm0, %xmm0
        vaddsd  56(%rbx,%rax,8), %xmm0, %xmm0
        addq    $8, %rax
        cmpq    $80000000, %rax
        jb      LBB0_2
```

First Performance Improvement

Let's run the old and the new modified program side by side to show to the world that I am the best programmer ever!*

Speedup Speedup Speedup

```
$ ./add_reduction_v1
...
average time taken over 100 iterations: 0.100273 s
$
$ ./add_reduction_v2
...
average time taken over 100 iterations: 0.098941 s
$
```

*All experiments shown here were performed on a Intel Core i5-8259U CPU (Skylake microarchitecture) running at 2.30 GHz (peak 3.60 GHz).

First Performance Improvement

Let's run the old and the new modified program side by side to show to the world that I am the best programmer ever!*

Speedup Speedup Speedup

```
$ ./add_reduction_v1
...
average time taken over 100 iterations: 0.100273 s
$
$ ./add_reduction_v2
...
average time taken over 100 iterations: 0.098941 s
$
```

WHAAAAAT!? The improvement is MINIMAL!

---

*All experiments shown here were performed on a Intel Core i5-8259U CPU (Skylake microarchitecture) running at 2.30 GHz (peak 3.60 GHz).

A closer look at the data

Timeline Comparison

```
[0,0]     DeeeeeeeeeER   .    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[1,0]     D====eeeeeeeeeER    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[2,0]     D========eeeeeeeeeER.    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0

[0,0]     DeeeeeeeeeER   .    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[0,1]     DeE--------R   .    .    . .   addq   $1, %rax
[0,2]     D=eE-------R   .    .    . .   cmpq   $80000000, %rax
[0,3]     D==eE------R   .    .    . .   jne    LBB0_2
[1,0]     .D===eeeeeeeeeER    .    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[1,1]     .DeE----------R    .    . .   addq   $1, %rax
[1,2]     .D=eE----------R    .    . .   cmpq   $80000000, %rax
[1,3]     .D==eE---------R    .    . .   jne    LBB0_2
[2,0]     . D======eeeeeeeeeER.    . .   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
[2,1]     . DeE-------------R.    . .   addq   $1, %rax
[2,2]     . D=eE------------R.    . .   cmpq   $80000000, %rax
[2,3]     . D==eE-----------R.    . .   jne    LBB0_2
```

If we look at the timeline we can clearly see that LLVM-MCA **correctly predicted** that the loop condition did not affect the sum operations, even without loop unrolling.

A closer look at the data

Before

```
Iterations:         100
Instructions:       100
Total Cycles:       408
Total uOps:         200

Dispatch Width:     6
uOps Per Cycle:     0.49
IPC:                0.25
Block RThroughput: 0.5
```

After

```
Iterations:         100
Instructions:       400
Total Cycles:       408
Total uOps:         500

Dispatch Width:     6
uOps Per Cycle:     1.23
IPC:                0.98
Block RThroughput: 0.8
```

In fact, the ratio between the total number of CPU cycles and the number of iterations was the same as well, regardless of whether we were including the loop condition.

A closer look at the data

Before

```
Iterations:        100
Instructions:      400
Total Cycles:      408
Total uOps:        500

Dispatch Width:    6
uOps Per Cycle:    1.23
IPC:               0.98
Block RThroughput: 0.8
```

After

```
Iterations:        100
Instructions:      900
Total Cycles:      3208
Total uOps:        900

Dispatch Width:    6
uOps Per Cycle:    0.28
IPC:               0.28
Block RThroughput: 4.0
```

The ratio between the total number of CPU cycles and the number of additions performed is the same between the version with loop unrolling and the version without

With loop unrolling each iteration performs 8 additions

The Resource Pressure View

The reason why the loop condition computation can run in parallel with respect to the sum is that they require **different CPU resources**.

We can see in detail the pressure on each resource from the **Resource Pressure View**.

Resource Pressure View

```
Resource pressure per iteration:
[0]   [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]   [9]
 -     -    1.10  1.11  0.50  0.50   -    0.89  0.90   -

Resource pressure by instruction:
[0]   [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]   [9]  Instructions:
 -     -    0.24  0.76  0.50  0.50   -     -     -     -   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
 -     -    0.04  0.25   -     -     -    0.57  0.14   -   addq $1, %rax
 -     -     -    0.10   -     -     -    0.32  0.58   -   cmpq $80000000, %rax
 -     -    0.82   -     -     -     -     -    0.18   -   jne LBB0_2
```

The Resource Pressure View

Resource Pressure View

```
Resource pressure per iteration:
[0]   [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]   [9]
 -     -    1.10  1.11  0.50  0.50   -    0.89  0.90   -

Resource pressure by instruction:
[0]   [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]   [9]  Instructions:
 -     -    0.24  0.76  0.50  0.50   -     -     -     -   vaddsd (%rbx,%rax,8), %xmm0, %xmm0
 -     -    0.04  0.25   -     -     -    0.57  0.14   -   addq $1, %rax
 -     -     -    0.10   -     -     -    0.32  0.58   -   cmpq $80000000, %rax
 -     -    0.82   -     -     -     -     -    0.18   -   jne LBB0_2
```

For each resource, this view shows the average number of resource cycles consumed at every iteration by the instructions.

The concept of resources roughly corresponds to **reservation stations** and the number and type of resources depends on the microarchitecture which is target of analysis.

The Resource Pressure View

Resource Pressure View: List of Resources

```
Resources:
[0]   - SKLDivider
[1]   - SKLFPDivider
[2]   - SKLPort0
[3]   - SKLPort1
[4]   - SKLPort2
[5]   - SKLPort3
[6]   - SKLPort4
[7]   - SKLPort5
[8]   - SKLPort6
[9]   - SKLPort7
```

In our example, the CPU microarchitecture is **Intel Skylake**.
We have **7 `SKLPort` resources**.*

- *SKL* is a contraction of *SKyLake*
- *Port* means *Reservation Station* in Intel's terminology

Each one of these resources serves different Functional Units.

*Additional information: [2]

The Resource Pressure View

Resource Pressure View: List of Resources

```
Resources:
[0]   - SKLDivider
[1]   - SKLFPDivider
[2]   - SKLPort0
[3]   - SKLPort1
[4]   - SKLPort2
[5]   - SKLPort3
[6]   - SKLPort4
[7]   - SKLPort5
[8]   - SKLPort6
[9]   - SKLPort7
```

SKLPort0	Integer/float vector/scalar ALU, integer mul. and div., branch
SKLPort1	Integer/float vector/scalar ALU
SKLPort2	Load
SKLPort3	Load
SKLPort4	Store
SKLPort5	Integer vector/scalar ALU
SKLPort6	Integer scalar, branch
SKLPort7	Store address

*Additional information: [2]

The Resource Pressure View

Resource Pressure View: List of Resources

```
Resources:
[0]   - SKLDivider
[1]   - SKLFPDivider
[2]   - SKLPort0
[3]   - SKLPort1
[4]   - SKLPort2
[5]   - SKLPort3
[6]   - SKLPort4
[7]   - SKLPort5
[8]   - SKLPort6
[9]   - SKLPort7
```

The `SKLDivider` and `SKLFPDivider` resources are **dummy resources**

They are used by the LLVM model to represent some complex behavior that cannot be represented with a simulation based on simple pipelining.

Hardware-wise they don't exist.

---

*Additional information: [2]

A closer look at the data

We can attempt to **predict the execution time** using the total cycles / iteration count ratio.

Given that the actual program we are benchmarking executes 80×10^6 iterations:

$$T_{exc} = \frac{N_{cycles}}{f_{ck}}$$

$$N_{cycles} = \frac{\text{Total Cycles}}{\text{Iterations}} 80 \times 10^6 \approx \frac{400}{100} 80 \times 10^6 = 320 \times 10^6$$

$$T_{exc} = \frac{320 \times 10^6}{3.6 \times 10^9 s^{-1}} \approx 0.088\,889\,s$$

A closer look at the data

We measured an execution time of 0.10 seconds, while LLVM-MCA estimates an execution time of 0.089 seconds.

- The prediction by LLVM-MCA is close but not identical to the real value
- The discrepancy can be attributed by the fact that LLVM-MCA does not simulate exactly every detail of the CPU, *especially memory behavior*
  - In fact, if we modify the program to always read from the same array element, the execution time drops to $\approx 0.093\,175\ s$

What does this tell us?

1. We have to always pay attention to **which** resources get used, not just how many are used
   - Corollary: the loop condition usually does not interfere with the body of the loop
   - We can mark just the body of the loop without too much worry!
2. Traditional measurements of performance based on the instruction as the unit of work *are deceptive and useless*
   - Corollary: The performance numbers given by many CPU benchmarks are completely bogus and easy to manipulate
   - The problem is that not every instruction performs the same amount of work!

Contents

Next optimization!

Before we got sidetracked by the loop unrolling stuff, LLVM-MCA told us there was a **data dependency** bottleneck

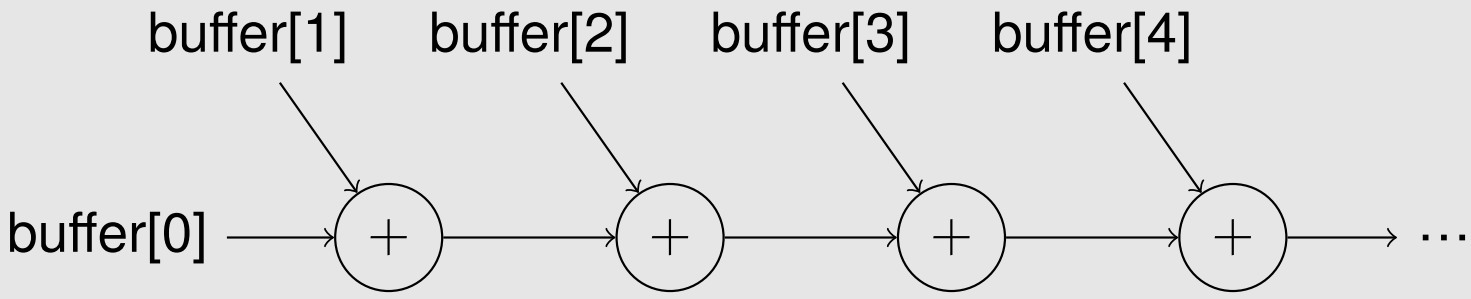
Bottleneck View

```
Cycles with backend pressure increase [ 38.24% ]
Throughput Bottlenecks:
  Resource Pressure       [ 0.00% ]
  Data Dependencies:      [ 38.24% ]
  - Register Dependencies [ 38.24% ]
  - Memory Dependencies   [ 0.00% ]

Critical sequence based on the simulation:

              Instruction                                 Dependency Information
 +----< 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0
 |
 |    < loop carried >
 |
 +----> 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0       ## REGISTER dependency:  %xmm0
 |
 |    < loop carried >
 |
 +----> 0.    vaddsd    (%rbx,%rax,8), %xmm0, %xmm0       ## REGISTER dependency:  %xmm0
```

Next optimization!

The problem is that every sum we perform depends on the result of the previous one.

Data Dependency Graph

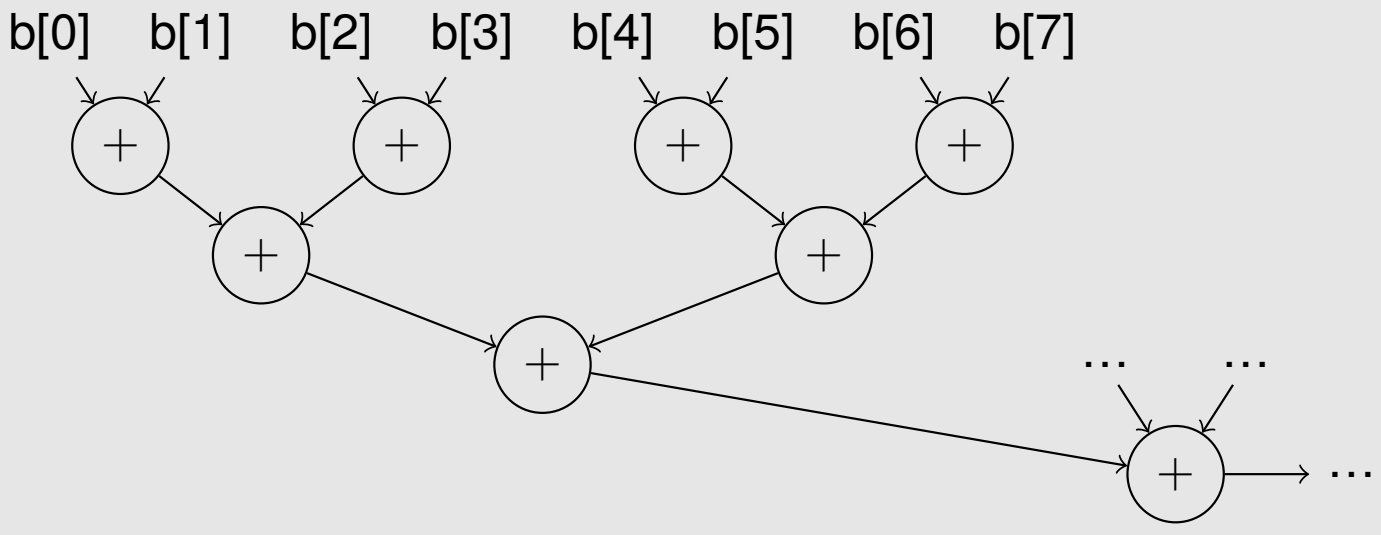


$$\text{result} = (((\text{buffer[0]} + \text{buffer[1]}) + \text{buffer[2]}) + \text{buffer[3]}) + \ldots$$

Next optimization!

If we exploit the associativity of addition, we can shorten the depth of our dependency chain!

Data Dependency Graph



$$\text{result} = ((\text{b[0]} + \text{b[1]}) + (\text{b[2]} + \text{b[3]})) + ((\text{b[4]} + \text{b[5]}) + (\text{b[6]} + \text{b[7]})) + \ldots$$

Next optimization!

Sum Reduction With Less Dependencies

```
for (int i=0; i<N; i+=8) {
  double t1 = buffer[i] + buffer[i+1];
  double t2 = buffer[i+2] + buffer[i+3];
  double t3 = buffer[i+4] + buffer[i+5];
  double t4 = buffer[i+6] + buffer[i+7];
  double t12 = t1 + t2;
  double t34 = t3 + t4;
  accum += t12 + t34;
}
```

Next optimization!

Sum Reduction With Less Dependencies

```
LBB0_2:
  vmovsd   (%rbx,%rax,8), %xmm0
  vmovsd   16(%rbx,%rax,8), %xmm1
  vaddsd   8(%rbx,%rax,8), %xmm0, %xmm0
  vaddsd   24(%rbx,%rax,8), %xmm1, %xmm1
  vaddsd   %xmm1, %xmm0, %xmm0
  vmovsd   32(%rbx,%rax,8), %xmm1
  vaddsd   40(%rbx,%rax,8), %xmm1, %xmm1
  vmovsd   48(%rbx,%rax,8), %xmm2
  vaddsd   56(%rbx,%rax,8), %xmm2, %xmm2
  vaddsd   %xmm2, %xmm1, %xmm1
  vaddsd   %xmm1, %xmm0, %xmm0
  vaddsd   %xmm0, %xmm3, %xmm3
  addq     $8, %rax
  cmpq     $80000000, %rax
  jb       LBB0_2
```

LLVM is already helping the CPU scheduler by moving the access to `buffer[i+2]` before the access to `buffer[i+1]`!

What LLVM-MCA says...

Before

```
Iterations:        100
Instructions:      900
Total Cycles:      3208
Total uOps:        900

Dispatch Width:    6
uOps Per Cycle:    0.28
IPC:               0.28
Block RThroughput: 4.0
```

After

```
Iterations:        100
Instructions:      1200
Total Cycles:      424
Total uOps:        1600

Dispatch Width:    6
uOps Per Cycle:    3.77
IPC:               2.83
Block RThroughput: 4.0
```

The amount of work per iteration has stayed the same, and the cycle count is massively reduced! From these figures, we expect the computation to complete in just $\approx 0.01\,s$ instead than $\approx 0.1\,s$!

What the reality of facts says

Let's run it!

```
$ ./add_reduction_v2
...
average time taken over 100 iterations: 0.098941 s
$
$ ./add_reduction_v3
...
average time taken over 100 iterations: 0.054087 s
$
```

Good: There is a huge improvement indeed!

Bad: The improvement is **much less** than what we expected.

The impact of memory accesses

add_reduction_v3_nocache.c

```
for (int i=0; i<N; i+=8) {
  double t1 = buffer[0] + buffer[0+1];
  double t2 = buffer[0+2] + buffer[0+3];
  double t3 = buffer[0+4] + buffer[0+5];
  double t4 = buffer[0+6] + buffer[0+7];
  double t12 = t1 + t2;
  double t34 = t3 + t4;
  accum += t12 + t34;
}
```

Execution Time

```
$ ./add_reduction_v3_nocache
...
average time taken over
100 iterations: 0.014668 s
$
```

- Experiment: let's reduce the impact of memory accesses as much as possible
- LLVM-MCA's analysis is the same even for the modified program
- The real execution time now matches LLVM-MCA's prediction!

The impact of memory accesses

- Memory accesses are **really slow**
- Once the code is reasonably optimized, it will spend **more time waiting for data to arrive** than performing actual calculations

- This doesn't mean that, when it reaches that stage, the code cannot be optimized further.
- Simply, any improvement from now on will be **marginal**, especially for an algorithm like this one where there is not much we can do to improve memory access patterns.

- To solve the problem of memory access cost, some computer scientists have theorized computing systems where many small CPUs are embedded inside memory chips ([3]).

What can we do

Bottleneck View

```
Cycles with backend pressure increase [ 90.57% ]
Throughput Bottlenecks:
  Resource Pressure       [ 90.57% ]
  - SKLPort0  [ 43.63% ]
  - SKLPort1  [ 43.63% ]
  - SKLPort2  [ 90.57% ]
  - SKLPort3  [ 90.57% ]
  Data Dependencies:      [ 2.36% ]
  - Register Dependencies [ 2.36% ]
  - Memory Dependencies   [ 0.00% ]
```

- We no longer have memory dependencies
- Now it seems that we have excessive resource pressure
  - `SKLPort2` and `SKLPort3` are used by **load instructions**
  - It seems we are issuing more loads than the CPU can perform

What can we do

- Let's scale back the ratio between memory accesses and add instructions a little bit...
- Note that we are actually **lengthening** the chain of data dependencies by doing this!

Sum Reduction V3b

```
for (int i=0; i<N; i+=4) {
  double t1 = buffer[i] + buffer[i+1];
  double t2 = buffer[i+2] + buffer[i+3];
  accum += t1 + t2;
}
```

Results

Sum Reduction V3b

```
$ ./add_reduction_v3
...
average time taken over 100 iterations: 0.054087 s
$
$ ./add_reduction_v3b
...
average time taken over 100 iterations: 0.051138 s
$
```

- The tradeoff was worthwhile! We squeezed out some residual speedup from the code.

In conclusion...

We just made a piece of code **worse** wrt. data dependencies... and there was a speedup!

- You must always strike the right balance between different contrasting behaviors
  - Data dependencies vs. resource usage
  - Loop overhead vs. code cache and branch prediction
  - Memory access patterns vs. algorithmic complexity
- You cannot reach the highest possible computation speed without machine-specific optimizations
  - On a CPU architecture with more than 2 load ports, the situation is reversed and the code with 4 additions per cycle is now slower

Contents

Recap

- LLVM-MCA analyzes the CPU performance of your machine code quite accurately
- **LLVM-MCA does not take into account the overhead of the memory hierarchy**
- Analyze the views provided by LLVM-MCA carefully! Do not get tricked by irrelevant metrics.
- Improve the code one bottleneck at a time
- Perform large improvements first
- When you find out that you are overfitting on your target machine, consider stopping

- And it should go without saying, but...
  Start with a good algorithm before you even consider using LLVM-MCA!

Final Remarks

- Most of the optimizations we made manually are applied automatically by LLVM when you use the `-ffast-math` flag
  - This does not mean the techniques we saw are useless!
  - It's just that our example was really simple
- A further optimization that can be made is **vectorization**
  - Try it at home!
  - Example code (alongside with all other examples shown here!): `https://github.com/danielecattaneo/LLVM-intro/tree/master/src/50-mca/example`
  - We can lower the execution time to $0.044\,824\,s$
- Not all CPUs are supported by LLVM-MCA!
  - Check if yours is supported or not using `llc --version`!

Contents

Quick look at how LLVM-MCA works

There are two main parts to LLVM-MCA:

- The simulation code
  - Frontend: `llvm/tools/llvm-mca`
  - Backend: `llvm/lib/MCA`
  - All simulated machines use *the same model*, just with different parameters
  - You can include `llvm/MCA/Context.h` and link with LLVM to use LLVM-MCA as a library!
- The machine models
  - `llvm/lib/Target/XXX/XXXScheduleYYY.td` (naming convention may vary)

    XXX Platform (X86, ARM, PowerPC...)
    YYY Microarchitecture (Skylake, [Cortex] A57, G5)
  - Determines instruction latency, reservation stations available, issue width...

* For those who got their BSc at PoliMi: **don't git-blame llvm-mca**

What's a .td file?

LLVM scheduling models are defined in **.td** files

- `.td` stands for **Tablegen Definition**
- It's a **standard syntax** of the LLVM project for **domain-specific languages**
- These DSLs are compiled by a tool named `llvm-tblgen`
- The usual compilation output for these DSLs are **data structures** (hence, **tables**) containing parameters used by LLVM at runtime
- More info:
  `https://llvm.org/docs/TableGen/index.html`

Machine Scheduler Definitions

- Root class of the scheduling model: `llvm::MCSchedModel`
  - See `llvm/MC/MCSchedule.h`
- The actual information is contained in `llvm::SchedMachineModel` subclasses auto-generated from the `.td` files
- More info in slides from previous LLVM developer meetings:

  `https://llvm.org/devmtg/2014-10/Slides/Estes-MISchedulerTutorial.pdf`

  `https://llvm.org/devmtg/2016-09/slides/Absar-SchedulingInOrder.pdf`

Contents

Conclusion

We have seen:

- On which principles LLVM-MCA is based
- How to use LLVM-MCA and how **not** to use it
- Which results can be achieved and what other factors influence the speed at which your code runs
- How LLVM-MCA is implemented and some hints on where to look if you want to expand or improve it

Have fun optimizing your programs!

Thank You!

Questions?

Contents

Bibliography I


John L Hennessy and David A Patterson.
Computer architecture: a quantitative approach.
Elsevier, 2011.


Agner Fog.
The microarchitecture of Intel, AMD and VIA CPUs.
Technical University of Denmark, 2020.


Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun.
Processing data where it makes sense: Enabling in-memory computation.
Microprocessors and Microsystems, 67:28 – 41, 2019.


Andrea di Biagio.
llvm-mca: a static performance analysis tool.
`https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html`.

Bibliography II


The LLVM Project.
llvm-mca - llvm machine code analyzer.
`https://llvm.org/docs/CommandGuide/llvm-mca.html`.


Agner Fog.
Instruction tables.
Technical University of Denmark, 2020.


Agner Fog.
Optimizing subroutines in assembly language.
Technical University of Denmark, 2020.


Agner Fog.
Optimizing software in C++.
Technical University of Denmark, 2020.