NAME

llvm-mca - LLVM Machine Code Analyzer

SYNOPSIS

llvm-mca [*options*] [input]

DESCRIPTION

llvm-mca is a performance analysis tool that uses information available in LLVM (e.g. scheduling models) to statically measure the performance of machine code in a specific CPU.

Performance is measured in terms of throughput as well as processor resource consumption. The tool currently works for processors with a backend for which there is a scheduling model available in LLVM.

The main goal of this tool is not just to predict the performance of the code when run on the target, but also help with diagnosing potential performance issues.

Given an assembly code sequence, **llvm-mca** estimates the Instructions Per Cycle (IPC), as well as hardware resource pressure. The analysis and reporting style were inspired by the IACA tool from Intel.

For example, you can compile code with clang, output assembly, and pipe it directly into **llvm-mca** for analysis:

```
$ clang foo.c -O2 -target x86_64-unknown-unknown -S -o - | llvm-mca -mcpu=btver2
```

Or for Intel syntax:

```
$ clang foo.c -O2 -target x86_64-unknown-unknown -mllvm -x86-asm-syntax=intel -S -o - | llvm-mca -mcpu=btve
```

(**llvm-mca** detects Intel syntax by the presence of an *.intel_syntax* directive at the beginning of the input. By default its output syntax matches that of its input.)

Scheduling models are not just used to compute instruction latencies and throughput, but also to understand what processor resources are available and how to simulate them.

By design, the quality of the analysis conducted by **llvm-mca** is inevitably affected by the quality of the scheduling models in LLVM.

If you see that the performance report is not accurate for a processor, please *file a bug* against the appropriate backend.

OPTIONS

If **input** is "-" or omitted, **llvm-mca** reads from standard input. Otherwise, it will read from the specified filename.

If the *-o* option is omitted, then **llvm-mca** will send its output to standard output if the input is from standard input. If the *-o* option specifies "-", then the output will also be sent to standard output.

-help
Print a summary of command line options.

-o <filename>
Use **<filename>** as the output filename. See the summary above for more details.

-mtriple=<target triple>
Specify a target triple string.

-march=<arch>
Specify the architecture for which to analyze the code. It defaults to the host default target.

-mcpu=<cpuname>
Specify the processor for which to analyze the code. By default, the cpu name is autodetected from the host.

-output-asm-variant=<variant id>
Specify the output assembly variant for the report generated by the tool. On x86, possible values are [0, 1]. A value of 0 (vic. 1) for this flag enables the AT&T (vic. Intel) assembly format for the code printed out by the tool in the analysis report.

-print-imm-hex
Prefer hex format for numeric literals in the output assembly printed as part of the report.

-dispatch=<width>
Specify a different dispatch width for the processor. The dispatch width defaults to field 'IssueWidth' in the processor scheduling model. If width is zero, then the default dispatch width is used.

-register-file-size=<size>
Specify the size of the register file. When specified, this flag limits how many physical registers

are available for register renaming purposes. A value of zero for this flag means "unlimited number of physical registers".

-iterations=<number of iterations>
Specify the number of iterations to run. If this flag is set to 0, then the tool sets the number of iterations to a default value (i.e. 100).

-noalias=<bool>
If set, the tool assumes that loads and stores don’t alias. This is the default behavior.

-lqueue=<load queue size>
Specify the size of the load queue in the load/store unit emulated by the tool. By default, the tool assumes an unbound number of entries in the load queue. A value of zero for this flag is ignored, and the default load queue size is used instead.

-squeue=<store queue size>
Specify the size of the store queue in the load/store unit emulated by the tool. By default, the tool assumes an unbound number of entries in the store queue. A value of zero for this flag is ignored, and the default store queue size is used instead.

-timeline
Enable the timeline view.

-timeline-max-iterations=<iterations>
Limit the number of iterations to print in the timeline view. By default, the timeline view prints information for up to 10 iterations.

-timeline-max-cycles=<cycles>
Limit the number of cycles in the timeline view, or use 0 for no limit. By default, the number of cycles is set to 80.

-resource-pressure
Enable the resource pressure view. This is enabled by default.

-register-file-stats
Enable register file usage statistics.

-dispatch-stats
Enable extra dispatch statistics. This view collects and analyzes instruction dispatch events, as well as static/dynamic dispatch stall events. This view is disabled by default.

-scheduler-stats
Enable extra scheduler statistics. This view collects and analyzes instruction issue events. This view is disabled by default.

-retire-stats
Enable extra retire control unit statistics. This view is disabled by default.

-instruction-info
Enable the instruction info view. This is enabled by default.

-show-encoding
Enable the printing of instruction encodings within the instruction info view.

-show-barriers
Enable the printing of LoadBarrier and StoreBarrier flags within the instruction info view.

-all-stats
Print all hardware statistics. This enables extra statistics related to the dispatch logic, the hardware schedulers, the register file(s), and the retire control unit. This option is disabled by default.

-all-views
Enable all the view.

-instruction-tables
Prints resource pressure information based on the static information available from the processor model. This differs from the resource pressure view because it doesn't require that the code is simulated. It instead prints the theoretical uniform distribution of resource pressure for every instruction in sequence.

-bottleneck-analysis
Print information about bottlenecks that affect the throughput. This analysis can be expensive, and it is disabled by default. Bottlenecks are highlighted in the summary view. Bottleneck analysis is currently not supported for processors with an in-order backend.

-json
Print the requested views in valid JSON format. The instructions and the processor resources are printed as members of special top level JSON objects. The individual views refer to them by index. However, not all views are currently supported. For example, the report from the bottleneck analysis is not printed out in JSON. All the default views are currently supported.

-disable-cb
Force usage of the generic CustomBehaviour and InstrPostProcess classes rather than using the target specific implementation. The generic classes never detect any custom hazards or make any post processing modifications to instructions.

EXIT STATUS

llvm-mca returns 0 on success. Otherwise, an error message is printed to standard error, and the tool returns 1.

USING MARKERS TO ANALYZE SPECIFIC CODE BLOCKS

llvm-mca allows for the optional usage of special code comments to mark regions of the assembly code to be analyzed. A comment starting with substring **LLVM-MCA-BEGIN** marks the beginning of a code region. A comment starting with substring **LLVM-MCA-END** marks the end of a code region. For example:

```
# LLVM-MCA-BEGIN
  ...
# LLVM-MCA-END
```

If no user-defined region is specified, then **llvm-mca** assumes a default region which contains every instruction in the input file. Every region is analyzed in isolation, and the final performance report is the union of all the reports generated for every code region.

Code regions can have names. For example:

```
# LLVM-MCA-BEGIN A simple example
  add %eax, %eax
# LLVM-MCA-END
```

The code from the example above defines a region named "A simple example" with a single instruction in it. Note how the region name doesn't have to be repeated in the **LLVM-MCA-END** directive. In the absence of overlapping regions, an anonymous **LLVM-MCA-END** directive always ends the currently active user defined region.

Example of nesting regions:

```
# LLVM-MCA-BEGIN foo
  add %eax, %edx
# LLVM-MCA-BEGIN bar
  sub %eax, %edx
```

```
# LLVM-MCA-END bar
# LLVM-MCA-END foo
```

Example of overlapping regions:

```
# LLVM-MCA-BEGIN foo
  add %eax, %edx
# LLVM-MCA-BEGIN bar
  sub %eax, %edx
# LLVM-MCA-END foo
  add %eax, %edx
# LLVM-MCA-END bar
```

Note that multiple anonymous regions cannot overlap. Also, overlapping regions cannot have the same name.

There is no support for marking regions from high-level source code, like C or C++. As a workaround, inline assembly directives may be used:

```
int foo(int a, int b) {
  __asm volatile("# LLVM-MCA-BEGIN foo":::"memory");
  a += 42;
  __asm volatile("# LLVM-MCA-END":::"memory");
  a *= b;
  return a;
}
```

However, this interferes with optimizations like loop vectorization and may have an impact on the code generated. This is because the **__asm** statements are seen as real code having important side effects, which limits how the code around them can be transformed. If users want to make use of inline assembly to emit markers, then the recommendation is to always verify that the output assembly is equivalent to the assembly generated in the absence of markers. The *Clang options to emit optimization reports* can also help in detecting missed optimizations.

HOW LLVM-MCA WORKS

llvm-mca takes assembly code as input. The assembly code is parsed into a sequence of MCInst with the help of the existing LLVM target assembly parsers. The parsed sequence of MCInst is then analyzed by a **Pipeline** module to generate a performance report.

The Pipeline module simulates the execution of the machine code sequence in a loop of iterations (default is 100). During this process, the pipeline collects a number of execution related statistics. At the end of this process, the pipeline generates and prints a report from the collected statistics.

Here is an example of a performance report generated by the tool for a dot-product of two packed float vectors of four elements. The analysis is conducted for target x86, cpu btver2. The following result can be produced via the following command using the example located at **test/tools/llvm-mca/X86/BtVer2/dot-product.s**:

```
$ llvm-mca -mtriple=x86_64-unknown-unknown -mcpu=btver2 -iterations=300 dot-product.s

Iterations:        300
Instructions:      900
Total Cycles:      610
Total uOps:        900

Dispatch Width:    2
uOps Per Cycle:    1.48
IPC:               1.48
Block RThroughput: 2.0


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

[1]    [2]    [3]    [4]    [5]    [6]    Instructions:
 1      2     1.00                        vmulps      %xmm0, %xmm1, %xmm2
 1      3     1.00                        vhaddps     %xmm2, %xmm2, %xmm3
 1      3     1.00                        vhaddps     %xmm3, %xmm3, %xmm4


Resources:
[0]   - JALU0
[1]   - JALU1
[2]   - JDiv
[3]   - JFPA
```

```
[4]   - JFPM
[5]   - JFPU0
[6]   - JFPU1
[7]   - JLAGU
[8]   - JMul
[9]   - JSAGU
[10]  - JSTC
[11]  - JVALU0
[12]  - JVALU1
[13]  - JVIMUL


Resource pressure per iteration:
[0]    [1]    [2]    [3]    [4]    [5]    [6]    [7]    [8]    [9]    [10]   [11]   [12]   [13]
 -      -      -     2.00   1.00   2.00   1.00    -      -      -      -      -      -      -

Resource pressure by instruction:
[0]    [1]    [2]    [3]    [4]    [5]    [6]    [7]    [8]    [9]    [10]   [11]   [12]   [13]   Instructions:
 -      -      -      -     1.00    -     1.00    -      -      -      -      -      -      -     vmulps      %xmm0, %xmm1, %xmm2
 -      -      -     1.00    -     1.00    -      -      -      -      -      -      -      -     vhaddps     %xmm2, %xmm2, %xmm3
 -      -      -     1.00    -     1.00    -      -      -      -      -      -      -      -     vhaddps     %xmm3, %xmm3, %xmm4
```

According to this report, the dot-product kernel has been executed 300 times, for a total of 900 simulated instructions. The total number of simulated micro opcodes (uOps) is also 900.

The report is structured in three main sections. The first section collects a few performance numbers; the goal of this section is to give a very quick overview of the performance throughput. Important performance indicators are **IPC**, **uOps Per Cycle**, and **Block RThroughput** (Block Reciprocal Throughput).

Field *DispatchWidth* is the maximum number of micro opcodes that are dispatched to the out-of-order backend every simulated cycle. For processors with an in-order backend, *DispatchWidth* is the maximum number of micro opcodes issued to the backend every simulated cycle.

IPC is computed dividing the total number of simulated instructions by the total number of cycles.

Field *Block RThroughput* is the reciprocal of the block throughput. Block throughput is a theoretical quantity computed as the maximum number of blocks (i.e. iterations) that can be executed per simulated clock cycle in the absence of loop carried dependencies. Block throughput is superiorly limited by the dispatch rate, and the availability of hardware resources.

In the absence of loop-carried data dependencies, the observed IPC tends to a theoretical maximum which can be computed by dividing the number of instructions of a single iteration by the *Block RThroughput*.

Field ’uOps Per Cycle’ is computed dividing the total number of simulated micro opcodes by the total number of cycles. A delta between Dispatch Width and this field is an indicator of a performance issue. In the absence of loop-carried data dependencies, the observed ’uOps Per Cycle’ should tend to a theoretical maximum throughput which can be computed by dividing the number of uOps of a single iteration by the *Block RThroughput*.

Field *uOps Per Cycle* is bounded from above by the dispatch width. That is because the dispatch width limits the maximum size of a dispatch group. Both IPC and ’uOps Per Cycle’ are limited by the amount of hardware parallelism. The availability of hardware resources affects the resource pressure distribution, and it limits the number of instructions that can be executed in parallel every cycle. A delta between Dispatch Width and the theoretical maximum uOps per Cycle (computed by dividing the number of uOps of a single iteration by the *Block RThroughput*) is an indicator of a performance bottleneck caused by the lack of hardware resources. In general, the lower the Block RThroughput, the better.

In this example, **uOps per iteration/Block RThroughput** is 1.50. Since there are no loop-carried dependencies, the observed *uOps Per Cycle* is expected to approach 1.50 when the number of iterations tends to infinity. The delta between the Dispatch Width (2.00), and the theoretical maximum uOp throughput (1.50) is an indicator of a performance bottleneck caused by the lack of hardware resources, and the *Resource pressure view* can help to identify the problematic resource usage.

The second section of the report is the *instruction info view*. It shows the latency and reciprocal throughput of every instruction in the sequence. It also reports extra information related to the number of micro opcodes, and opcode properties (i.e., ’MayLoad’, ’MayStore’, and ’HasSideEffects’).

Field *RThroughput* is the reciprocal of the instruction throughput. Throughput is computed as the maximum number of instructions of a same type that can be executed per clock cycle in the absence of operand dependencies. In this example, the reciprocal throughput of a vector float multiply is 1 cycles/instruction. That is because the FP multiplier JFPM is only available from pipeline JFPU1.

Instruction encodings are displayed within the instruction info view when flag *-show-encoding* is specified.

Below is an example of *-show-encoding* output for the dot-product kernel:

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

[1]    [2]    [3]    [4]    [5]    [6]    [7]    Encodings:                    Instructions:
 1      2     1.00                          4    c5 f0 59 d0                   vmulps %xmm0, %xmm1, %xmm2
 1      4     1.00                          4    c5 eb 7c da                   vhaddps     %xmm2, %xmm2, %xmm3
 1      4     1.00                          4    c5 e3 7c e3                   vhaddps     %xmm3, %xmm3, %xmm4
```

The *Encoding Size* column shows the size in bytes of instructions. The *Encodings* column shows the actual instruction encodings (byte sequences in hex).

The third section is the *Resource pressure view*. This view reports the average number of resource cycles consumed every iteration by instructions for every processor resource unit available on the target. Information is structured in two tables. The first table reports the number of resource cycles spent on average every iteration. The second table correlates the resource cycles to the machine instruction in the sequence. For example, every iteration of the instruction vmulps always executes on resource unit [6] (JFPU1 - floating point pipeline #1), consuming an average of 1 resource cycle per iteration. Note that on AMD Jaguar, vector floating-point multiply can only be issued to pipeline JFPU1, while horizontal floating-point additions can only be issued to pipeline JFPU0.

The resource pressure view helps with identifying bottlenecks caused by high usage of specific hardware resources. Situations with resource pressure mainly concentrated on a few resources should, in general, be avoided. Ideally, pressure should be uniformly distributed between multiple resources.

Timeline View

The timeline view produces a detailed report of each instruction’s state transitions through an instruction pipeline. This view is enabled by the command line option **-timeline**. As instructions transition through the various stages of the pipeline, their states are depicted in the view report. These states are represented by the following characters:

- D : Instruction dispatched.
- e : Instruction executing.
- E : Instruction executed.
- R : Instruction retired.
- = : Instruction already dispatched, waiting to be executed.
- - : Instruction executed, waiting to be retired.

Below is the timeline view for a subset of the dot-product example located in **test/tools/llvm-mca/X86/BtVer2/dot-product.s** and processed by **llvm-mca** using the following command:

```
$ llvm-mca -mtriple=x86_64-unknown-unknown -mcpu=btver2 -iterations=3 -timeline dot-product.s

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]: Executions
[1]: Average time spent waiting in a scheduler’s queue
[2]: Average time spent waiting in a scheduler’s queue while ready
```

```
[3]: Average time elapsed from WB until retire stage

      [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>
```

The timeline view is interesting because it shows instruction state changes during execution. It also gives an idea of how the tool processes instructions executed on the target, and how their timing information might be calculated.

The timeline view is structured in two tables. The first table shows instructions changing state over time (measured in cycles); the second table (named *Average Wait times*) reports useful timing statistics, which should help diagnose performance bottlenecks caused by long data dependencies and sub-optimal usage of hardware resources.

An instruction in the timeline view is identified by a pair of indices, where the first index identifies an iteration, and the second index is the instruction index (i.e., where it appears in the code sequence). Since this example was generated using 3 iterations: **-iterations=3**, the iteration indices range from 0-2 inclusively.

Excluding the first and last column, the remaining columns are in cycles. Cycles are numbered sequentially starting from 0.

From the example output above, we know the following:

- Instruction [1,0] was dispatched at cycle 1.
- Instruction [1,0] started executing at cycle 2.
- Instruction [1,0] reached the write back stage at cycle 4.
- Instruction [1,0] was retired at cycle 10.

Instruction [1,0] (i.e., vmulps from iteration #1) does not have to wait in the scheduler’s queue for the operands to become available. By the time vmulps is dispatched, operands are already available, and pipeline JFPU1 is ready to serve another instruction. So the instruction can be immediately issued on the JFPU1 pipeline. That is demonstrated by the fact that the instruction only spent 1cy in the scheduler’s queue.

There is a gap of 5 cycles between the write-back stage and the retire event. That is because instructions must retire in program order, so [1,0] has to wait for [0,2] to be retired first (i.e., it has to wait until cycle 10).

In the example, all instructions are in a RAW (Read After Write) dependency chain. Register %xmm2 written by vmulps is immediately used by the first vhaddps, and register %xmm3 written by the first vhaddps is used by the second vhaddps. Long data dependencies negatively impact the ILP (Instruction Level Parallelism).

In the dot-product example, there are anti-dependencies introduced by instructions from different iterations. However, those dependencies can be removed at register renaming stage (at the cost of allocating register aliases, and therefore consuming physical registers).

Table *Average Wait times* helps diagnose performance issues that are caused by the presence of long latency instructions and potentially long data dependencies which may limit the ILP. Last row, **<total>**, shows a global average over all instructions measured. Note that **llvm-mca**, by default, assumes at least 1cy between the dispatch event and the issue event.

When the performance is limited by data dependencies and/or long latency instructions, the number of cycles spent while in the *ready* state is expected to be very small when compared with the total number of cycles spent in the scheduler's queue. The difference between the two counters is a good indicator of how large of an impact data dependencies had on the execution of the instructions. When performance is mostly limited by the lack of hardware resources, the delta between the two counters is small. However, the number of cycles spent in the queue tends to be larger (i.e., more than 1-3cy), especially when compared to other low latency instructions.

Bottleneck Analysis

The **-bottleneck-analysis** command line option enables the analysis of performance bottlenecks.

This analysis is potentially expensive. It attempts to correlate increases in backend pressure (caused by pipeline resource pressure and data dependencies) to dynamic dispatch stalls.

Below is an example of **-bottleneck-analysis** output generated by **llvm-mca** for 500 iterations of the dot-product example on btver2.

```
Cycles with backend pressure increase [ 48.07% ]
Throughput Bottlenecks:
  Resource Pressure       [ 47.77% ]
  - JFPA  [ 47.77% ]
  - JFPU0  [ 47.77% ]
  Data Dependencies:      [ 0.30% ]
  - Register Dependencies [ 0.30% ]
  - Memory Dependencies   [ 0.00% ]

Critical sequence based on the simulation:

              Instruction                         Dependency Information
 +----< 2.    vhaddps %xmm3, %xmm3, %xmm4
 |
 |    < loop carried >
 |
 |      0.    vmulps  %xmm0, %xmm1, %xmm2
 +----> 1.    vhaddps %xmm2, %xmm2, %xmm3         ## RESOURCE interference:  JFPA [ probability: 74% ]
 +----> 2.    vhaddps %xmm3, %xmm3, %xmm4         ## REGISTER dependency:  %xmm3
 |
 |    < loop carried >
 |
 +----> 1.    vhaddps %xmm2, %xmm2, %xmm3         ## RESOURCE interference:  JFPA [ probability: 74% ]
```

According to the analysis, throughput is limited by resource pressure and not by data dependencies. The analysis observed increases in backend pressure during 48.07% of the simulated run. Almost all those pressure increase events were caused by contention on processor resources JFPA/JFPU0.

The *critical sequence* is the most expensive sequence of instructions according to the simulation. It is annotated to provide extra information about critical register dependencies and resource interferences between instructions.

Instructions from the critical sequence are expected to significantly impact performance. By construction, the accuracy of this analysis is strongly dependent on the simulation and (as always) by the quality of the processor model in llvm.

Bottleneck analysis is currently not supported for processors with an in-order backend.

Extra Statistics to Further Diagnose Performance Issues

The **-all-stats** command line option enables extra statistics and performance counters for the dispatch logic, the reorder buffer, the retire control unit, and the register file.

Below is an example of **-all-stats** output generated by **llvm-mca** for 300 iterations of the dot-product example discussed in the previous sections.

```
Dynamic Dispatch Stall Cycles:
RAT     - Register unavailable:                      0
RCU     - Retire tokens unavailable:                 0
SCHEDQ  - Scheduler full:                            272  (44.6%)
LQ      - Load queue full:                           0
SQ      - Store queue full:                          0
GROUP   - Static restrictions on the dispatch group: 0


Dispatch Logic - number of cycles where we saw N micro opcodes dispatched:
[# dispatched], [# cycles]
 0,              24  (3.9%)
 1,              272  (44.6%)
 2,              314  (51.5%)


Schedulers - number of cycles where we saw N micro opcodes issued:
[# issued], [# cycles]
 0,          7  (1.1%)
 1,          306  (50.2%)
 2,          297  (48.7%)

Scheduler's queue usage:
[1] Resource name.
[2] Average number of used buffer entries.
[3] Maximum number of used buffer entries.
[4] Total number of buffer entries.

 [1]            [2]        [3]        [4]
JALU01           0          0          20
JFPU01           17         18         18
JLSAGU           0          0          12


Retire Control Unit - number of cycles where we saw N instructions retired:
[# retired], [# cycles]
 0,           109  (17.9%)
 1,           102  (16.7%)
 2,          399  (65.4%)

Total ROB Entries:               64
Max Used ROB Entries:            35  ( 54.7% )
Average Used ROB Entries per cy:  32  ( 50.0% )


Register File statistics:
Total number of mappings created:    900
Max number of mappings used:         35

*  Register File #1 -- JFpuPRF:
   Number of physical registers:     72
   Total number of mappings created: 900
   Max number of mappings used:      35

*  Register File #2 -- JIntegerPRF:
   Number of physical registers:     64
   Total number of mappings created: 0
   Max number of mappings used:      0
```

If we look at the *Dynamic Dispatch Stall Cycles* table, we see the counter for SCHEDQ reports 272 cycles. This counter is incremented every time the dispatch logic is unable to dispatch a full group because the scheduler's queue is full.

Looking at the *Dispatch Logic* table, we see that the pipeline was only able to dispatch two micro opcodes 51.5% of the time. The dispatch group was limited to one micro opcode 44.6% of the cycles, which corresponds to 272 cycles. The dispatch statistics are displayed by either using the command option **-all-stats** or **-dispatch-stats**.

The next table, *Schedulers,* presents a histogram displaying a count, representing the number of micro opcodes issued on some number of cycles. In this case, of the 610 simulated cycles, single opcodes were issued 306 times (50.2%) and there were 7 cycles where no opcodes were issued.

The *Scheduler's queue usage* table shows that the average and maximum number of buffer entries (i.e., scheduler queue entries) used at runtime. Resource JFPU01 reached its maximum (18 of 18 queue entries). Note that AMD Jaguar implements three schedulers:

- JALU01 - A scheduler for ALU instructions.

- JFPU01 - A scheduler floating point operations.
- JLSAGU - A scheduler for address generation.

The dot-product is a kernel of three floating point instructions (a vector multiply followed by two horizontal adds). That explains why only the floating point scheduler appears to be used.

A full scheduler queue is either caused by data dependency chains or by a sub-optimal usage of hardware resources. Sometimes, resource pressure can be mitigated by rewriting the kernel using different instructions that consume different scheduler resources. Schedulers with a small queue are less resilient to bottlenecks caused by the presence of long data dependencies. The scheduler statistics are displayed by using the command option **-all-stats** or **-scheduler-stats**.

The next table, *Retire Control Unit*, presents a histogram displaying a count, representing the number of instructions retired on some number of cycles. In this case, of the 610 simulated cycles, two instructions were retired during the same cycle 399 times (65.4%) and there were 109 cycles where no instructions were retired. The retire statistics are displayed by using the command option **-all-stats** or **-retire-stats**.

The last table presented is *Register File statistics*. Each physical register file (PRF) used by the pipeline is presented in this table. In the case of AMD Jaguar, there are two register files, one for floating-point registers (JFpuPRF) and one for integer registers (JIntegerPRF). The table shows that of the 900 instructions processed, there were 900 mappings created. Since this dot-product example utilized only floating point registers, the JFPuPRF was responsible for creating the 900 mappings. However, we see that the pipeline only used a maximum of 35 of 72 available register slots at any given time. We can conclude that the floating point PRF was the only register file used for the example, and that it was never resource constrained. The register file statistics are displayed by using the command option **-all-stats** or **-register-file-stats**.

In this example, we can conclude that the IPC is mostly limited by data dependencies, and not by resource pressure.

Instruction Flow

This section describes the instruction flow through the default pipeline of **llvm-mca**, as well as the functional units involved in the process.

The default pipeline implements the following sequence of stages used to process instructions.

- Dispatch (Instruction is dispatched to the schedulers).

- Issue (Instruction is issued to the processor pipelines).
- Write Back (Instruction is executed, and results are written back).
- Retire (Instruction is retired; writes are architecturally committed).

The in-order pipeline implements the following sequence of stages: * InOrderIssue (Instruction is issued to the processor pipelines). * Retire (Instruction is retired; writes are architecturally committed).

llvm-mca assumes that instructions have all been decoded and placed into a queue before the simulation start. Therefore, the instruction fetch and decode stages are not modeled. Performance bottlenecks in the frontend are not diagnosed. Also, **llvm-mca** does not model branch prediction.

Instruction Dispatch

During the dispatch stage, instructions are picked in program order from a queue of already decoded instructions, and dispatched in groups to the simulated hardware schedulers.

The size of a dispatch group depends on the availability of the simulated hardware resources. The processor dispatch width defaults to the value of the **IssueWidth** in LLVM's scheduling model.

An instruction can be dispatched if:

- The size of the dispatch group is smaller than processor's dispatch width.
- There are enough entries in the reorder buffer.
- There are enough physical registers to do register renaming.
- The schedulers are not full.

Scheduling models can optionally specify which register files are available on the processor. **llvm-mca** uses that information to initialize register file descriptors. Users can limit the number of physical registers that are globally available for register renaming by using the command option **-register-file-size**. A value of zero for this option means *unbounded*. By knowing how many registers are available for renaming, the tool can predict dispatch stalls caused by the lack of physical registers.

The number of reorder buffer entries consumed by an instruction depends on the number of micro-opcodes specified for that instruction by the target scheduling model. The reorder buffer is responsible for tracking the progress of instructions that are "in-flight", and retiring them in program order. The number of entries in the reorder buffer defaults to the value specified by field *MicroOpBufferSize* in the target scheduling model.

Instructions that are dispatched to the schedulers consume scheduler buffer entries. **llvm-mca** queries the scheduling model to determine the set of buffered resources consumed by an instruction. Buffered resources are treated like scheduler resources.

Instruction Issue

Each processor scheduler implements a buffer of instructions. An instruction has to wait in the scheduler's buffer until input register operands become available. Only at that point, does the instruction becomes eligible for execution and may be issued (potentially out-of-order) for execution. Instruction latencies are computed by **llvm-mca** with the help of the scheduling model.

llvm-mca's scheduler is designed to simulate multiple processor schedulers. The scheduler is responsible for tracking data dependencies, and dynamically selecting which processor resources are consumed by instructions. It delegates the management of processor resource units and resource groups to a resource manager. The resource manager is responsible for selecting resource units that are consumed by instructions. For example, if an instruction consumes 1cy of a resource group, the resource manager selects one of the available units from the group; by default, the resource manager uses a round-robin selector to guarantee that resource usage is uniformly distributed between all units of a group.

llvm-mca's scheduler internally groups instructions into three sets:

- WaitSet: a set of instructions whose operands are not ready.
- ReadySet: a set of instructions ready to execute.
- IssuedSet: a set of instructions executing.

Depending on the operands availability, instructions that are dispatched to the scheduler are either placed into the WaitSet or into the ReadySet.

Every cycle, the scheduler checks if instructions can be moved from the WaitSet to the ReadySet, and if instructions from the ReadySet can be issued to the underlying pipelines. The algorithm prioritizes older instructions over younger instructions.

Write-Back and Retire Stage

Issued instructions are moved from the ReadySet to the IssuedSet. There, instructions wait until they reach the write-back stage. At that point, they get removed from the queue and the retire control unit is notified.

When instructions are executed, the retire control unit flags the instruction as "ready to retire."

Instructions are retired in program order. The register file is notified of the retirement so that it can free the physical registers that were allocated for the instruction during the register renaming stage.

Load/Store Unit and Memory Consistency Model

To simulate an out-of-order execution of memory operations, **llvm-mca** utilizes a simulated load/store unit (LSUnit) to simulate the speculative execution of loads and stores.

Each load (or store) consumes an entry in the load (or store) queue. Users can specify flags **-lqueue** and **-squeue** to limit the number of entries in the load and store queues respectively. The queues are unbounded by default.

The LSUnit implements a relaxed consistency model for memory loads and stores. The rules are:

1. A younger load is allowed to pass an older load only if there are no intervening stores or barriers between the two loads.

2. A younger load is allowed to pass an older store provided that the load does not alias with the store.

3. A younger store is not allowed to pass an older store.

4. A younger store is not allowed to pass an older load.

   By default, the LSUnit optimistically assumes that loads do not alias (*-noalias=true*) store operations. Under this assumption, younger loads are always allowed to pass older stores. Essentially, the LSUnit does not attempt to run any alias analysis to predict when loads and stores do not alias with each other.

   Note that, in the case of write-combining memory, rule 3 could be relaxed to allow reordering of non-aliasing store operations. That being said, at the moment, there is no way to further relax the memory model (**-noalias** is the only option). Essentially, there is no option to specify a different memory type (e.g., write-back, write-combining, write-through; etc.) and consequently to weaken, or strengthen, the memory model.

   Other limitations are:

- The LSUnit does not know when store-to-load forwarding may occur.
- The LSUnit does not know anything about cache hierarchy and memory types.
- The LSUnit does not know how to identify serializing operations and memory fences.

The LSUnit does not attempt to predict if a load or store hits or misses the L1 cache. It only knows if an instruction "MayLoad" and/or "MayStore." For loads, the scheduling model provides an "optimistic" load-to-use latency (which usually matches the load-to-use latency for when there is a hit in the L1D).

llvm-mca does not (on its own) know about serializing operations or memory-barrier like instructions. The LSUnit used to conservatively use an instruction's "MayLoad", "MayStore", and unmodeled side effects flags to determine whether an instruction should be treated as a memory-barrier. This was inaccurate in general and was changed so that now each instruction has an IsAStoreBarrier and IsALoadBarrier flag. These flags are mca specific and default to false for every instruction. If any instruction should have either of these flags set, it should be done within the target's InstrPostProcess class. For an example, look at the *X86InstrPostProcess::postProcessInstruction* method within *llvm/lib/Target/X86/MCA/X86CustomBehaviour.cpp*.

A load/store barrier consumes one entry of the load/store queue. A load/store barrier enforces ordering of loads/stores. A younger load cannot pass a load barrier. Also, a younger store cannot pass a store barrier. A younger load has to wait for the memory/load barrier to execute. A load/store barrier is "executed" when it becomes the oldest entry in the load/store queue(s). That also means, by construction, all of the older loads/stores have been executed.

In conclusion, the full set of load/store consistency rules are:

1. A store may not pass a previous store.
2. A store may not pass a previous load (regardless of **-noalias**).
3. A store has to wait until an older store barrier is fully executed.
4. A load may pass a previous load.
5. A load may not pass a previous store unless **-noalias** is set.
6. A load has to wait until an older load barrier is fully executed.

In-order Issue and Execute

In-order processors are modelled as a single **InOrderIssueStage** stage. It bypasses Dispatch, Scheduler and Load/Store unit. Instructions are issued as soon as their operand registers are available and resource requirements are met. Multiple instructions can be issued in one cycle according to the value of the **IssueWidth** parameter in LLVM's scheduling model.

Once issued, an instruction is moved to **IssuedInst** set until it is ready to retire. **llvm-mca** ensures that writes are committed in-order. However, an instruction is allowed to commit writes and retire out-of-order if **RetireOOO** property is true for at least one of its writes.

Custom Behaviour

Due to certain instructions not being expressed perfectly within their scheduling model, **llvm-mca** isn't always able to simulate them perfectly. Modifying the scheduling model isn't always a viable option though (maybe because the instruction is modeled incorrectly on purpose or the instruction's behaviour is quite complex). The CustomBehaviour class can be used in these cases to enforce proper instruction modeling (often by customizing data dependencies and detecting hazards that **llvm-mca** has no way of knowing about).

llvm-mca comes with one generic and multiple target specific CustomBehaviour classes. The generic class will be used if the **-disable-cb** flag is used or if a target specific CustomBehaviour class doesn't exist for that target. (The generic class does nothing.) Currently, the CustomBehaviour class is only a part of the in-order pipeline, but there are plans to add it to the out-of-order pipeline in the future.

CustomBehaviour's main method is *checkCustomHazard()* which uses the current instruction and a list of all instructions still executing within the pipeline to determine if the current instruction should be dispatched. As output, the method returns an integer representing the number of cycles that the current instruction must stall for (this can be an underestimate if you don't know the exact number and a value of 0 represents no stall).

If you'd like to add a CustomBehaviour class for a target that doesn't already have one, refer to an existing implementation to see how to set it up. The classes are implemented within the target specific backend (for example */llvm/lib/Target/AMDGPU/MCA/*) so that they can access backend symbols.

Custom Views

llvm-mca comes with several Views such as the Timeline View and Summary View. These Views are generic and can work with most (if not all) targets. If you wish to add a new View to **llvm-mca** and it does not require any backend functionality that is not already exposed through MC layer classes (MCSubtargetInfo, MCInstrInfo, etc.), please add it to the */tools/llvm-mca/View/* directory. However, if your new View is target specific AND requires unexposed backend symbols or functionality, you can define it in the */lib/Target/<TargetName>/MCA/* directory.

To enable this target specific View, you will have to use this target's CustomBehaviour class to override the *CustomBehaviour::getViews()* methods. There are 3 variations of these methods based on where you want your View to appear in the output: *getStartViews()*, *getPostInstrInfoViews()*, and *getEndViews()*. These methods returns a vector of Views so you will want to return a vector containing all of the target specific Views for the target in question.

Because these target specific (and backend dependent) Views require the *CustomBehaviour::getViews()* variants, these Views will not be enabled if the *-disable-cb* flag is used.

Enabling these custom Views does not affect the non-custom (generic) Views. Continue to use the usual command line arguments to enable / disable those Views.

AUTHOR

Maintained by the LLVM Team (https://llvm.org/).

COPYRIGHT