Praktikum im WS 2022/2023
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Sergej Breiter, MSc., Minh Thanh Chung, MSc.,Amir Raoofy, MSc., Bengisu Elis,MSc.,
Dr. Karl Fürlinger, Dr. Josef Weidendorfer

Assignment 07 – Due: 22.12.2022, 16:00

The aim of this assignment is to understand the performance properties of CPU cores: more details about instruction execution throughput and latency, as well as branch prediction.

Instruction-Level Parallelism

First, we will explore several aspects of Instruction-Level Parallelism (ILP). Modern superscalar out-of-order processors analyze an instruction stream for data dependencies, and they execute instructions in parallel if they do not depend on each other. This concept is also known as ILP, and it is facilitated by various mechanisms:

- **Multiple-issue:** instructions that are independent can be started at the same time.
- **Pipelining:** arithmetic units can deal with splitting up the computation for operations into multiple stages such that multiple operations can be worked on simultaneously, being in different stages at a given point in time.
- **Out-of-order execution:** the order of instructions can be rearranged if they are not dependent on each other, and if the resulting execution will be more efficient.
- **Branch prediction and speculative execution:** at a branch instruction, the processor guesses the execution path (which may be wrong), and it executes instructions accordingly speculatively. If the guess was wrong, calculated results get thrown away, and execution restarts at the correct branch target.
- **Prefetching:** data can be speculatively requested from memory before any instruction needing it is actually encountered.

Pipelining

- **Instruction latency:** The number of clock cycles that are required for the execution core to complete the execution of all of the micro operations that form an instruction.
- **Instruction throughput:** The number of clock cycles required to wait before the issue ports are free to accept the same instruction again. For many instructions, the throughput of an instruction can be significantly less than its latency.

1. **Instruction Latency**

   You learned that the floating-point units in a modern CPU are pipelined, and that pipelines require a number of independent operations to function efficiently. A typical pipelineable operation is the triad vector operation from the first assignment. An example of an operation that can not be pipelined is the computation of dot product as shown in Listing 1. The fact that `sum` gets both read and written in each iteration of the loop (*dependency chain*) halts the addition pipeline.

```
for (i=0; i<N; i++)
    sum += a[i] * b[i]
```

   Listing 1: Dot product

   To measure the latency of an instruction, we can repeatedly execute this particular instruction, and measure the average time per instruction execution. However, we have to make sure that pipelining and superscalar execution is prevented. Because the execution time depends on the core's frequency, instruction latency is typically specified in units of core cycles.

   (a) Theoretical question:
      i. Think of a way to modify the dot product code in Listing 1 that enables pipelining.

   (b) Measure the instruction latency of the add, multiply, and division instructions for the data types `double`, `float`, `int32_t`, and `int64_t`. You can use the provided code.

   (c) Summarize your results of the instruction latency in **units of cycles** in a table or bar plot. Explain your solution.

   (d) Find and list the vendor-provided (Intel, AMD, Fujitsu, Marvel) values for the corresponding instruction latencies of at least one architecture of your choice. Compare your measurements to the vendor-provided values.

2. **Instruction Throughput**

   (a) Theoretical questions:
      i. Assume an instruction latency of n cycles of a given instruction. How many independent dependency chains are necessary in a loop to fully utilize the pipeline?
      ii. Now assume that there are m available *execution ports* on the core for this instruction. How does that change your answer?

   (b) Measure instruction throughput for the same data types and instructions as in Task 1.

   (c) Summarize your results of instruction throughput next to the table or bar plot from Task 1.

   (d) Compare the number of depency chains required to minimize the execution time per instruction to the theoretical value from Part a.

   (e) Again, compare your results to vendor-provided values.

Branch Prediction

Branch prediction is needed because of pipeline execution of instructions (for high ILP): the input for a branch instruction, which decides about the next instruction to execute, may only be calculated/available a few clock cycles in the future. To not stall the pipeline, the CPU guesses the execution flow by predicting where jumps will go to. Branch predictors typically try to detect patterns, and are based on instruction address, local, and global jump behavior. In this task we

want to understand the capability of the branch prediction unit of architectures in BEAST using a given benchmark.
Different types of predictors exist, corresponding to branch instruction types.

- The first type are conditional jumps, which either jump to a given target address or proceed with the next instruction, depending on a boolean condition.
- The second type are indirect jumps: An input operand provides the address of the next instruction to execute. Here, the target address must be predicted.
- The final branch type is a special version of an indirect jump: ”return from a function”, where the jump target can be observed at the time the function gets called. This allows for a special predictor to be better than a generic indirect branch predictor.

The benchmark ”branch.cpp” is given. It provides different modes to analyze the capability of the three branch types. For all modes, command sequences of different lengths are generated; there is a loop going through these command sequences, executing different code associated with different commands. This will be done multiple times, and the run time is measured. This time depends on the successful prediction of jump instructions in the loop. There are two different types of command sequences generated:

1. all commands the same,
2. randomized command sequence.

The first sequence type will result in easy prediction, showing the best case. The other requires jump patterns of different lengths to be predicted. Run ”./branch -h” to see the command line options.

1. **Understand the benchmark**

   For this task, you can chose the architecture you want - even your own laptop.

   (a) Tests for indirect jumps: variant A, B, C

   What is the main difference between the command execution loops in variants A, B, and C? Look up for ”computed goto” to understand variant C. What influence on performance do you expect? Run ”branch -ABC -12 200” and look at the 4th column of each row. It shows the average per-command execution time. This invocation uses the two sequences ”fix1” and ”fix2” consisting of only commands 1 and 2, respectively (arg ”-12”), always with a length of 200 commands (arg ”200”), and runs variant A, B, and C (arg ”-ABC”).

   Compare the results and explain the differences you see.

   (b) Tests for conditional jumps: variant D

   Explain the code. Is it more similar to A, B, or C? What performance do you expect to see (for the case that branches get predicted correctly)? Test it with ”branch -ABCD -12 200”.

   (c) Tests for return instructions: variant E

   Predictors for return instructions typically use a stack of addresses. On a call, the return address is pushed; on a return, the top address is used as prediction. We want to measure the size of this stack structure used by the predictor. This test is very similar to variant A. However, to trigger different number of return instructions executed in a row, we do the command handling in batches with various lengths (”depth” in function ”runE”), and call the handler functions recursively, with the dispatcher code duplicated. Note that the time measurement here includes 2 branch types: the calls and the returns.

To test various batch lengths, variants E, F, G, and so on are actually just the same as variant E but with different lengths (F is similar to E6, with length 6). Run ”branch -AEFZ -12 200” to see the best case times. Compare the results.

2. **Branch Prediction Capacities**

   (a) Indirect jumps: Variant C

   For each of the architectures in BEAST, do measurements for different command sequences with different sizes. Draw a figure with command size on the X axes (max size 500000), per-command time on Y, and different curves for each command sequence type (fix1/fix2/rnd1): ”branch -128 -C”. Explain the results.

   Also, draw a figure just for ”rnd1”, but with curves for all 4 CPU architectures. Explain the differences.

   (b) Conditional jumps: Variant D

   Do the same measurements and figures as for Variant C. Explain the results.

   Create a figure comparing C and D for each architecture, for ”rnd1”. Try to explain the difference.

   (c) Return instructions: Variant E

   Find out the size of the internal return stack structure for return prediction on the various CPU architectures (testing variants E,F,...). To test lengths more fine-grained, you can modify the code. Hint: it is enough to use size 200, and even ”fix1” sequence should be fine for this. Why?

   (d) Relation to Performance Counter Results

   For selected cases (good/bad for the 3 branch types), measure the amounts of branches taken and branch mispredictions with performance counters (Likwid or Perf). Can you relate the results to the measurements?

3. **Bonus question: Optimization**

   Can you think about ways to reduce the penalty of branch mispredictions for a given workload? Hints: what are ”predicates”? Think about ways to reduce the number of (unpredictable) jumps.