LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

lrz

Technische Universität München

Praktikum im SS 2024
Evaluierung moderner HPC-Architekturen und -Beschleuniger (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

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

Assignment 06 – Due: 13.06.2024

In this assignment, we will look into details of the micro-architecture of CPUs using micro-benchmarks. There is a lot of text, but in the end it is about measurements to evaluate the capability of CPU hardware. The 3 parts are about (1) best-case core-to-core latencies, (2) instruction level parallelism (ILP) enabled by pipelined execution, and (3) branch prediction for speculative execution.

Core-to-core Latencies

Here, you will measure the best-case latency involved when cores want to talk to each other. This should reflect the hardware topology, so it is useful when you want to understand this from measurements. Furthermore, it gives insights into latencies to be expected when cores have to synchronize, e.g. within the runtime of OpenMP.
All communication between cores on modern multi-core systems works via shared memory. In the end, this is implemented by transactions of a cache coherence protocol, such as MOESI (AMD) or MESIF (Intel). Look up these protocols for more details. In this task you also will see how latencies change depending on the address used for communication.
Find the benchmark "c2c.c" in the sources for this assignment. It should compile on all systems with GCC or LLVM. It uses OpenMP to run a ping-pong test between given cores via writing values to a shared address and do busy-waiting to observe the address for changes on the other side. To get sensible data, you must bind threads to cores. The number of threads given triggers latency measurements between all pairs of threads.

1. **Tasks**
   (a) For each CPU multi-core system in BEAST (Milan, ThunderX2, IceLake, A64FX), find out the numbers given by Linux for the cores. Which OpenMP environment variable settings must be used to bind the range of physical cores to the corresponding number of threads? Give the answer using `OMP_PLACES` and `OMP_PROC_BIND`, and alternatively, `GOMP_CPU_AFFINITY`.

(b) For first tests, the benchmark may take long due to quadratic behavior in the number of cores. So, first run the test just for 2 threads, and set the environment variables to get measurements (1) between nearside cores within a socket, (2) between cores from different sockets, (3) if applicable: between cores on the same socket not sharing the last-level cache (this usually is L3, but L2 on A64FX). Document settings and results.

(c) Try to understand the benchmark. There is an option to get more verbose output, which shows results per memory address used, and also measurements for a ”warmup phase”. Why is the warmup phase before actual measurements useful? Find out the time interval used for the warmup phase. Is this enough for each of the systems in BEAST, or to high, such that measurements can be sped up?

(d) The benchmark uses multiple addresses in a range, 16 bytes apart, and measures the minimum, maximum, and average latency. Using verbosity mode, observe that these latencies may change depending on address used, also differently on each BEAST architecture. Try to explain why this is happening. For this, it is useful to understand how the cache is implemented, over which the communication has to happen, such as an “partitioned L3 cache” for Intel.

(e) Now run the benchmark for all physical cores of the available architectures with default settings, and create as result figure a heat map of minimum latencies as 2D matrix. That is, cell (x,y) should show a color from a fitting range of colors for the minimum latency between cores x and y. Use black for any (x,x) in the matrix, as a core does not communicate with itself, and no measurement can be done for that. Explain how the architecture topology can be derived from this result. Hints: to reduce time needed for the full run, try to use a minimal warmup phase. Also, you can assume the matrix to be symmetrical for cutting runtime by half (modify the code accordingly). See python scripts in sources as ways to produce the 2d matrix heat map.

(f) The benchmark takes a lot of time also due to being sequential. Try to parallelize it to make it faster. How much faster do you get? Make another 2d heat map out of it. Are the results still the same? If not, try to explain what the problem might be for the benchmark to go wrong with your parallelization approach.

Instruction-Level Parallelism

In this task, we 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:** functional 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 increases the processor instruction throughput–the number of instructions completed per unit of time–but it does not reduce the execution time of an individual instruction. In this part, we consider two micro-architectural properties of pipelined instruction execution:

- **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. We use the reciprocal values (cycles per instruction), because this makes comparisons between latency and throughput easier. The reciprocal throughput is also called *issue latency*.

2. **Understanding Instruction Latency and Throughput**

   You learned that execution units in modern CPUs are pipelined, and that pipelines require a number of independent operations to function efficiently. Computing a dot product, as shown in Listing 1, is an example that can not be pipelined. The fact that `sum` gets both read and written in each iteration of the loop (*dependency chain*) halts (*stalls*) the addition pipeline, because the add instruction depends on the result of the previous iteration (*data hazard*).

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

Listing 1: Dot product

   (a) Consider the dot product in Listing 1.
      i. Why is it possible to measure the add instruction latency, but not the multiply instruction latency with this code?
      ii. How can you derive the value of the add instruction latency from the execution time and an additional measurement of the number of clock cycles or frequency?
      iii. Think of a way to modify the dot product code that enables pipelining.

   (b) 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 a pipeline?
      ii. Now assume that there are m available *execution ports* on the core for this instruction. How does that change your answer?
      iii. How is a (reciprocal) instruction throughput lower than one cycle per instruction possible?

3. **Measuring Instruction Latency and Throughput**

   To measure the latency of an instruction, we can repeatedly execute the instruction, and measure the average time per instruction. However, we have to make sure that pipelining and superscalar execution are prevented. This can be achieved by executing the instruction in a single dependency chain. Instruction throughput can be measured by executing multiple independent dependency chains such that the pipeline is always full.

(a) Understanding the code (`instructions.cpp`).
   i. What is the purpose of the template parameter `CHAINS` in the function `instructions` on the innermost loop? Look at the generated assembly code, for example using *compiler-explorer* [1], or `objdump`.
   ii. Consider the operands of the instructions in the innermost loop. What happens when you set a high value for `CHAINS`?
   iii. How can you determine the instruction throughput from the measurements?

(b) Run the benchmark with the add, multiply, and division operations and the data types `float`, `double`, `int32_t`, and `int64_t` on **each** of the BEAST systems. You can use the provided Makefile to run the code with `make bench-instructions`. However, you need to adjust the maximum number of dependency chains, and also measure the average frequency, or number of clock cycles for the evaluation. [2]
   i. Compute instruction latency and (reciprocal) instruction throughput in units of **cycles per instruction** (CPI) and summarize your results in a table or bar plot. Explain your solution and interpret the results. Make a comparison between different instructions on the same architecture and between corresponding instructions on different architectures.
   ii. Compare the number of dependency chains required to minimize the execution time per instruction to the theoretical value derived in Task 2 b.
   iii. Find and list values taken from literature for the instruction latency and throughput of the instructions (look at the assembly code!) of one architecture of your choice (vendor-provided by Intel, AMD, Fujitsu, Marvel, or from Agner Fog, or from uops.info). Compare your measurements to values taken from literature.

Branch Prediction

Branch prediction is needed because of pipelined 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;

[1] `https://godbolt.org/z/qK8T9KTTz`
[2] Use Likwid or `perf`. Despite the potential inaccuracy, it is sufficient to perform whole-program profiling to measure the average frequency in this task.

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.

4. **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.

5. **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?

6. **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.