Assignment 3

Objectives

This assignment is divided into two parts. In the first part, we are going to observe the performance effects of Non-Uniform Memory Access (NUMA) behavior that is present in today's multi-socket platforms. Then, in the second part, we are going to analyze the memory reordering behavior observable in modern processors.

Assignment – Part 1

We provide a small program (`numa.c`) where we allocate a large 8GB array, and the base program traverses the array from the start to the end while accessing every byte. The given program performs 8G accesses, as it accesses each byte in the 8GB array.

First, we will analyze the effect of NUMA behavior on this program. Modern processors consist of multiple sockets, where each socket is a group of processors with their own cache and memory. Processors on each socket can access both their and the remote socket's memory. Multiple sockets are required because it is expensive to scale the number of cores within the same socket. On the SCITAS cluster, we have two sockets with 18 cores each.

To measure the performance impact of NUMA, you need to use the `numactl` command (`man numactl` or here) that is present on the cluster. This command will allow you to choose policies that can allocate program memory to different NUMA sockets in a local, remote, or interleaved fashion. You need to write a SCITAS job submission script that runs the `numa` program with these three memory allocation policies. This script should be named `execute_numa.sh` and is one of the deliverables. You need to document the results you obtain in your report.

In the results that you obtain, you will find that there is very little performance difference among the three NUMA memory allocation policies. This small difference is because with the current `numa` program, the hardware is able to hide most of the NUMA latency. To understand the effect of NUMA behavior on latency, you need to expose as much of the memory access latency as possible. Therefore, the next task requires you to *de-optimize* the given `numa` program.

A key part of this task is for you to bypass the three main optimizations used by modern servers to hide or reduce memory access latency – caching data, cache line prefetching, and out-of-order execution. As we have already seen, caches store recently accessed data which can be useful because of spatial or temporal locality. Modern caches also automatically prefetch the next cache line (next 64 bytes) when a cache line is accessed. They also employ a stride prefetcher that can detect linear access patterns in your memory accesses, and prefetch the required cache lines

(e.g. if you always access the 10th cache line relative to last access). Modern out-of-order cores can also perform memory accesses in parallel if they can calculate the address of the memory elements. Therefore, you need to change the given program so that:

- Your memory accesses do not benefit from caching.
- Your memory accesses do not benefit from cache line prefetching.
- The address of the next access can only be calculated after the first access completes.

You need to modify the `numa.c` program to traverse the 8GB array (you do not have to initialize or access every element). The goal here is to increase the (time taken/memory access) metric as much as possible by exposing the memory access latency (at least 1.0e-8 seconds per access while performing at least 10M accesses). You are only allowed to change the *next_addr* and *init_array* functions. The array indices you access should strictly monotonically increase i.e. you cannot go back or loop in the array. Only basic operations (+,-,/,*,%,if/else) and memory accesses are allowed in *next_addr* and *init_array*, and therefore you cannot include loops, functions calls, or external library calls. You can use the already provided random number generator functions in *init_array*, but not in *next_addr*. You cannot add irrelevant or complex calculations to increase execution time, and all the code modifications should be justified in the report.

Once you are done with the program, you can again use your `execute_numa.sh` script to run your modified `numa` program with the three NUMA memory allocation policies. Document the results in your report, and compare with the results you got previously.

Assignment – Part 2

The aim of this part is to analyze the hardware reordering of the instructions in the program assembly, and the effect of NUMA accesses on the reordering frequency. As x86 processors implement a model similar to processor consistency (actual model is not officially disclosed), we can only observe loads bypassing stores.

We provide a program (`order.c`) that uses three threads (two explicit + one main) to detect memory re-orderings on the processors in the SCITAS cluster. The main thread (always pinned to core 0) is responsible for controlling the execution of the two explicit threads, counting the re-ordering events, and reinitializing all the variables. The explicit threads follow the example given in the class slides, and perform reads and writes to the shared `X` and `Y` variables. The program already reports the number of re-orderings detected in 1M loop iterations. Using this program, you have to perform the following three mini-tasks:

- Dump the assembly of this program using `gcc -S` (you can use the *order_asm* target in the given Makefile) and identify the instructions that are re-ordered by the processor. Please use the `gcc` present on the SCITAS cluster to obtain the same assembly as us. In your report, explain the sequence in which these instructions actually access memory and lead to a reordering detection.
- Using the `numactl` command, create a `execute_order.sh` script that runs the `order` program twice while positioning the two explicit threads on the same socket and

on different sockets. In your report, analyze the impact of this NUMA behavior on the reordering frequency you observe in the program. (Hint: use numactl -C)

- Modify the explicit thread functions in the `order.c` program to force the required ordering between memory operations using the special ordering instructions that were discussed in the lectures. Run the program to verify that orderings are not detected anymore. Analyze the assembly for this new program, and explain the corresponding hardware behavior in your report.

Development Infrastructure

On the course's Moodle page, we have provided the tarball `A3.tgz`, which contains the following files:

- `Makefile` – compiles the given programs.
- `execute.sh` – A sample script to submit jobs to the SCITAS cluster. Please use the given parameters while using the script.
- `utility.h` – contains a set of helper functions:
  - `set_clock` & `elapsed_time`: to measure the start and end of execution.
  - `init_rand` & `next_rand`: generate random numbers.
- `numa.c` – the program for array traversal. You need to de-optimize the program to increase the (time taken/memory access) metric while obeying the constraints given in the program.
- `order.c` – the program for detecting memory reorderings. You have to modify this program to prevent memory reorderings by including special ordering instructions.

An example of running the programs and the output they produce:

```
# /bin/bash
$ ./numa
Traversing 8 GB array took total time = 2.564 seconds, number of
accesses = 44975068, 5.7e-08 seconds per access

$./order
999758 memory re-orderings detected in 1000000 iterations - 99
percent
```

Please make sure that the output is in the same format as given.

Deliverables

To submit your code and your report, create a tarball archive **(the only accepted format!)** called `a3_<yourGroupID>.tgz` and upload it on Moodle. Your file must contain:

- A de-optimized version of `numa.c`.

- A `execute_numa.sh` script containing commands to run `numa.c` with three NUMA policies.
- A modified version of `order.c` with no re-orderings possible.
- A `execute_order.sh` script containing commands to run `order.c` with the two explicit threads on the same and different NUMA nodes.
- A report, with the name `a3_<yourGroupID>.pdf`, that complies to the following description.

Report

In the report, we expect you to perform the following tasks:

1- Report the performance difference you got when running `numa` with different allocation policies (local, remote, interleaved). When you run the original optimized `numa` program with these policies, what is the difference you observe? Explain.

2- Explain how you de-optimized `numa`. List the different optimizations you removed, and how much did each of them effect the (time taken/memory access) metric.

3- Explain the assembly instructions that you analyzed as the cause behind memory re-orderings in `order`. What changes did you make to the program to disable these re-orderings? How does the hardware behave after your changes?

4- Report the change in the number of re-orderings observed when running the `order` program explicit threads in the same socket and different sockets. What is the cause of this behavior?

The report should be as precise as possible, addressing the questions above. Keep the answers of the above questions and your reasoning brief and precise. A regular report should not be longer than 2-3 pages.

Grading

The code will be graded automatically by a script and checked for plagiarism. The script will check if the results returned are consistent with what was expected. Plagiarized code will receive 0. We will escalate plagiarism cases to the student section.

The reports will be read and graded by humans and checked for plagiarism automatically.

The grade breakdown is as follows:

- Correctness: 50%
- Report: 50%