Memory Optimization

D. Jiménez, E. Morancho and À. Ramírez

April 11, 2014

Index

Previous Work

Tools

1. Looking at `/proc` directory, using tools like `x86info` and processor manual, figure out the memory hierarchy parameters (capacity/associative of cache levels of memory, TLB characteristics, latencies, ...) of the processor you are using in order to do the exercises.

2. Which are the Oprofile events that you will need to look at to figure out the L1 misses, L1 hits, L2 misses, L2 hits, DTLB misses and hits, and ITLB misses and hits.

1

Getting knowledge of Our machine

1. `mountain.c` program (from *Computer Systems. A Programmer's Prespective* book, R. Bryant and D. O'Hallaron) let graphically show the memory hierarchy behaviour of your machine. This program performs memory accesses to different *workloads* (vectors of 2Kbytes up to 16 Mbytes) with different *strides* (element by element, every each element, etc., where each element is 4 bytes). With that, we want to evaluate the capacity of a cache and the cache line sizes.

   (a) Run the program forwarding the output to a file (for instance , `mountain.csv`).

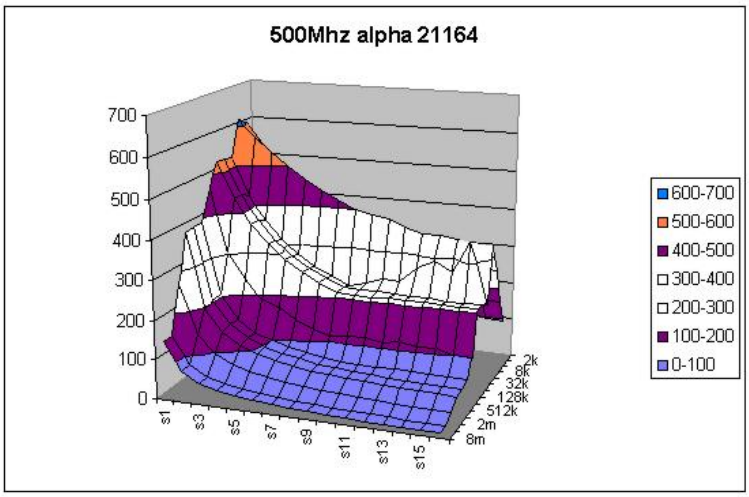
   (b) Open a new "excel" `mountain.ods` and file `mountain.csv` from `OpenOffice.org`. That will let us to graphically represent the data in `mountain.csv`. WARNING: maybe your openoffice configuration of decimals use "." or ",". In this case you have to see if the mountain.csv use the same format: "." or ",". Otherwise, nothing will be seen on the figure.

   (c) Load the `.csv` file. The figure shows the memory bandwdith you get for each combination of stride and data workload.

   (d) In order to figure out the number of caches and their size you can make a new figure using one of the columns of the table.

   (e) In order to figure out the cache line size you can make a new figure using one of the rows of the table.

   Here it is a figure obtained for an Alpha 21164 processor.



2. The *distribution sort* algorithm copies (distributes) the elements of a vector to another vector depending on a certain number of bits of the keys to be distributed.

   (a) Try to understand `distribution.c` program with an example in order to see:

- How the vector index is updated depending on the value of the most significant bits of the elements of S
- How the vector index is updated on the following loop in the code
- How the vector index is used to copy data from S to D

(b) Understand `distribution.sh` *shellscript*.

(c) Compile the program (make distribution.2) and run the *shellscript*. Once you have run the *shellscript* you should have two files:

i. `10000000-0-results.txt`: run of the program with parameters `n_elements`=10000000, `bits` from 1 to 22, `data_distribution`=0 (data is ramdomly distributed)

ii. `10000000-1-results.txt`: run of the program with parameters `n_elements`=10000000, `bits` from 1 to 22, `data_distribution`=1 (data is distributed with an order)

(d) Why there is performance difference between those two executions (data distributions)?

i. Perform the profiling with `oprofile` looking at the TLB events and L1/L2 misses, using the following parameters for the program for:

- `n_elements`=10000000, `bits`=10, `distribution`=0 (./distribution.2 10000000 10 0) and `n_elements`=10000000, `bits`=10, `distribution`=1 (./distribution.2 10000000 10 1).
- `n_elements`=10000000, `bits`=18, `distribution`=0 (./distribution.2 10000000 18 0) and `n_elements`=10000000, `bits`=18, `distribution`=1 (./distribution.2 10000000 18 1).

ii. Justify your answer based on your timings and the profiling you have just done.

2

Data aligment

3. The unaligned memory accesses may affect the program performance. In this exercise, you have to run the programs of `munge_vectors` directory, and make a figure in order to show the execution time depending on the data alignment.

   (a) Analyze the source codes that you will find on the directory, compile them and run them.

   (b) Create a figure for each program execution, and another for all of them together.

   (c) Try to justify the results obtained.

3

Memory Bandwidth

4. Using the same code of `munge_vectors` of previous exercise:

   (a) Create a figure for all the executions, but in this case, normalized by the number of accesses done. In order to create this figure you should modify comment out a printf of the `munge` function to print out *cyc/numberofaccesses*. The number of accesses is computed by (memoryEnd - memoryInit)/sizeof(data).

   (b) What are the memory accesses that take maximum profit of the memory bandwidth: 8-bit, 16-bit, 32-bit or 64-bit accesses? Which is the memory bandwidth of our machine to load/store to/from the general purpose registers?

5. Optimize your best `swap.c` program version (exercise of problem collection) so that the memory bandwidth exploitation is improved.

4

Spatial Locality

6. As a part of a graphic engine, there is a routine that has to fill a buffer with green colour before printing next frame. The screen is `ROWSxCOLUMNS` pixels.

   (a) Analyze the `rgb.c` code.

   (b) Obtain the first cache level misses (references to the second cache level) and the second cache level misses of the original code.

   (c) Propose an optimization of this code so that we can better exploit the spatial locality of the data. Compile and execute in order to obtain the new timing.

   (d) Propose a version that also reduces the number of memory accesess so that you can set more than one rgb structure to green, per memory access. Compile and execute in order to obtain the new timing.

   (e) Compute the speedup of the second optimitzation compared to the original code, and the spatial locality optimization. Why don't you get any speedup?

5

Temporal Locality

7. It is very common to find matrix per matrix on graphic engines. The basic code of that operation does not exploit very well the temporal and spatial locality. You have three different matrix per matrix code in `matriuAxB.c` file, and an example of *blocking* for a vector per matrix (exploiting the temporal locality of the vector) in `vectorxmatriu.c` file.

   (a) Compare the performance of the different implementations of the matrix product. Which is the fastest one? Why?

   (b) Codify a *blocking* version of the matrix per matrix function `mult1` in order to exploit the temporal locality of memory accesses to `B` matrix.

   (c) Run both versions (original and with *blocking*) for `N`=64, `N`=128, `N`=256, `N`=512, `N`=1024, `N`=2048 (if it is possible).

      i. Make different tests using different *blocking* sizes.

      ii. Make a table where you compare exectuion cycles, cache and TLB misses of the two versions: original and with *blocking*, varying the block size.

8. The PCA entreprise wants to sort their workers by NID very fast.

   Each worker is represented with a *struct*:

```
#define MAX_NOM 40
#define MAX_DIAS_MES 31
#define MAX_CATEGORIA 40

typedef struct
{
  long long int NID;              /* Identification number*/
  char Nom[MAX_NOM];
  char Cognoms[MAX_NOM];
  int  horesMes[MAX_DIAS_MES];
  char Categoria[MAX_CATEGORIA];
  unsigned int ptrClauForaneaDepartament;
  unsigned int ptrClauCategoria;
} Templeat
```

   (a) Program `empleats.c` sorts a parametrizable number of workers. In order to do that, the worker information is generated in a random way. In order to sort the workers quicksort (`qsort`) algorithm is used.

   (b) Make a figure showing the cycles spent per sorted worker (CPW, Cycles per worker), varying the number of workers to sort. In order to make that process automatic it would be usefull to create a *shellscript*. The figure can be created using OpenOffice.

(c) Now, the enterprise also wants to maintain the address information of the worker, the first and last name of the couple (if he/she has). If there is something that is not applicable, the field will be kept in blank. Activate the EXTES define of the *struct* definition and analyze the program performance with a figure where you show again CPW. Compare the performance of the previous code and the actual code.

(d) Optimize the program so that the size of the structure (activated or not EXTES) does not influence too much in the performance of the sorting process. This optimization should improve the spatial and temporal locality exploitation of the sorting.

(e) Compare that optimized code with the original code:

   i. Using CPW.

   ii. Using first and second cache level misses and TLB misses per worker sorted.