Integer Programming (WS2018/19)
Practial Project

Motivation

Dynamic Random Access Memories (DRAMs) are fundamental components of todays computers allowing data storage at a very high density. However, due to architectural limitations only a certain amount of data can be cached and made available for fast access. Accessing non-cached data comes at high costs in both time and energy. The achievable bandwidth and energy efficiency of DRAM memories strongly depends on the access patterns to the memory devices. Even tough, we usually do not know in advance how these memory devices are accessed, many applications have a regular or fixed memory access pattern. Especially embedded systems, which are designed for specific applications, such as streaming, real-time image and signal processing, often have deterministic memory access patterns. For these applications we wish to store the data in such a way that the energy and time consumption is as small as possible.

The Mathematical Model

In the following we give a mathematical model of the problem stated above [1]:
Let f be a sequence of elements over some set V_f. Furthermore, let b, c, r be positive integral numbers with $brc \geqslant |V_f|$. Let $\mathcal{B} = (B_1, \ldots, B_b)$ be a b-tuple such that $V_f = B_1 \dot\cup \ldots \dot\cup B_b$ is a partition of V into b (possibly empty) disjoint sets with cardinality of at most rc. The b-tuple $\mathcal{B}$ is called *bank allocation* and the sets B_i, $i = 1 \ldots, b$, *banks*. Each bank B_i is further partitioned into r *rows* $R_{i,1}, \ldots, R_{i,r}$ which are (possibly empty) disjoint sets of cardinality of at most c, i.e. $B_i = R_{i,1} \dot\cup \ldots \dot\cup R_{i,r}$. We call the r-tuple $\mathcal{R}_i = (R_{i,1}, \ldots, R_{i,r})$ a *row-allocation* of B_i and c is said to be the number of *columns*. The sets $R_{i,j}$, $i = 1, \ldots, b$, $j = 1, \ldots, r$, are called rows. The B-tuple $\mathcal{A} = (\mathcal{R}_1, \ldots, \mathcal{R}_b)$ is called a *bank-row-allocation* of V_f or simply an *allocation*.
Given an allocation of V_f its elements are requested in the order of their occurrence in f and have to be accessed in the row they are allocated to. While doing so any row can be in exactly one of two states: either it is *cached* or *non-cached*. At any point in time the number of cached rows per bank is at most one. At the beginning all rows are non-cached. A requested element can only be accessed if the row where it is allocated to is cached. Thus, by requesting an element in a non-cached row, the row has to be cached (and possibly causing another row to be non-cached). Caching an non-cached row comes at a cost of one unit called a *row miss*. If the row of the requested element is already cached then no further costs are charged. In this case, we speak of a *row hit*.
We are now ready to formulate our main optimization problems:

MINIMUM ROW MISSES
INSTANCE: A sequence f over a set V_f, nonnegative integral numbers $b, r, c \in \mathbb{Z}_{>0}$ with $brc > |V_f|$
TASK: Find an allocation of V_f such that the number of row misses is minimum

MAXIMUM ROW HITS
INSTANCE: A sequence f over a set V_f, nonnegative integral numbers $b, r, c \in \mathbb{Z}_{>0}$ with $brc > |V_f|$
TASK: Find an allocation of V_f such that the number of row hits is maximum

[1] For our purpose it is not necessary to understand how a DRAM works in detail; it suffices to have a mathematical model that describes the problem appropriately.

Observe that finding an optimal allocation with a minimum number of row misses is equivalent to finding an allocation with a maximum number of row hits.

Example Consider the sequence $f = (1,2,3,1,5,6,7,1,7,6)$ and the allocation given in Figure 1. At the beginning all rows are non-cached. Consequently, the request of 1 causes the row R_{11} to be cached, incurring a row miss. The next element 2 is also contained in R_{11} and its requests admits a row hit. As in bank $\mathcal{R}_2$ no row is cached yet, the request of 3 causes a row miss. Requesting the next element 1 incurs a row hit. The request of 5 causes the row R_{12} to be cached and in turn the row R_{11} to be non-cached admitting a row miss. Table 1 summarizes the cached rows in each bank and the hits and misses at every point in time.

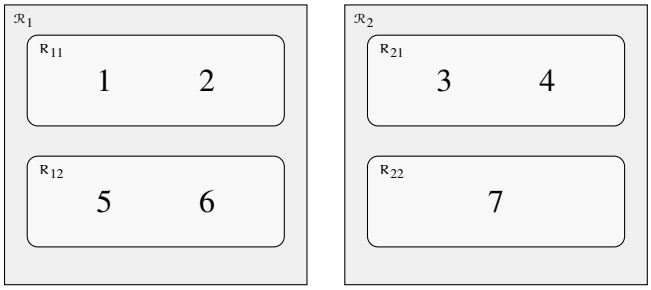

Figure 1: An allocation of the sequence $f = (1,2,3,1,5,6,7,1,7,6)$ with 6 row misses and 4 row hits.

t	0	1	2	3	4	5	6	7	8	9	10
f(t)	-	1	2	3	1	5	6	7	1	7	6
$\mathcal{R}_1$	-	R_{11}	R_{11}	R_{11}	R_{11}	R_{12}	R_{12}	R_{12}	R_{11}	R_{11}	R_{12}
$\mathcal{R}_2$	-	-	-	R_{21}	R_{21}	R_{21}	R_{21}	R_{22}	R_{22}	R_{22}	R_{22}
		miss	hit	miss	hit	miss	hit	miss	miss	hit	miss

Table 1: The cached rows, hits and misses in each bank for the allocation of V_f given in Figure 1.

Assignment

Your task is to write a program that calculates a feasible solution to MINIMUM ROW MISSES and MAXIMUM ROW HITS, respectively, that is as "good" as you can manage.Your code should be written in `python 3`. You are free to use any library you wish (e.g. `time`, `random`, `math`, `networkx`, etc). The main file of your code must be called `dram_optimization.py` and contains the following function:

```
def dram_optimization(sequence, number_of_banks, number_of_rows, number_of_columns):
''' Calculates a feasible allocation for sequence
param sequence :  list
param number_of_banks    : an integer specifying the number of banks
param number_of_rows     : an integer specifying the number of rows
param number_of_columns : an integer specifying the number of columns
return:
hits     : an integer specifying the number of row hits with respect to the returned allocation
misses : an integer specifying the number of row misses with respect to the returned allocation
```

```
banks  : a dictionary specifying to which bank an element is allocated to;
         the keys are the elements of the sequence and the values are the integers
         from 1 up to number_of_banks
rows   :  a dictionary specifying to which row an element is allocated to;
         the keys are the elements of the sequence and the values are the integers
         from 1 up to number_of_rows
'''
```

The due date is the **15th February 2019**. You are allowed to hand in your code in **groups up to 3 persons**. Your code has to run without raising an error and return a feasible allocation, i.e. the number of elements per row is at most c, the number of rows per bank is at most r and the number of banks is at most b. Moreover, the number of row misses and hits must be calculated right. Beside your working code you have to assign a summary on how your algorithm works and the obtained results. For each sequence the minimum and maximum number of row misses and hits obtained and the time needed to calculate these allocations must be given. The parameters for each sequence are given in the appendix.

Further Informations

Note that we do not expect that you are able to tackle this problem perfectly at this point in the lecture. Throughout this course you get to know a couple of techniques that can be used in order to solve it. It is up to you which one you choose. Of course you can also combine several techniques. Moreover, every now and then the weekly assignments will contain exercises regarding the problem and its structure. These results might be helpful.
We point out that solving the problem optimally – especially for large-sized instances – is not an easy task. Hence, we do not expect from you to find a provably optimal solution. However, we award the participants with the best solutions on the benchmark instances provided to you (cf. Appendix).

Appendix

Provided Data

On the lecture site we provide a couple of sequences which you can use to test and benchmark your code. The names of the sequences are of the form "<number of elements>_<length>.seq" (for example 6121_12155.seq is of length 12155 and contains 6121 elements). Of course, you can also create your own sequences to test you code. The benchmarks should be done with the sequences provided to you with the following parameters:

Sequence	Number of columns	Number of rows
10_50.seq	4	4
99_500.seq	8	16
190_950.seq	16	16
278_140.seq	16	32
546_2750.seq	32	32
792_19024.seq	32	32
4224_8402.seq	64	128
6121_12155.seq	64	128
10336_453358.seq	128	128
10400_294960.seq	128	128
11296_316796.seq	128	128
11520_644456.seq	128	128
131072_393216.seq	128	128
356400_1198800.seq	512	1024

The number of banks should be $1, 2$ and 4. Thus, for each sequence there are 3 settings of the parameters. The sequence can be read with the function read_sequence.py which is provided on the lecture site as well:

```
def read_sequence(file_path):
''' Reads a *.seq-file and returns a list corresponding to the sequence in the file
param file_path :  a string containing the relative path to the *.seq-file
return:
sequence : a list containing the sequence of the *.seq file
'''
```

Materials and furhter readings

On the lecture site there will be an Introduction to python, networkx and guorbi. The introduction to gurobi contains also an explanation how to install python. For more detailed informations we refer to

- http://www.python-course.eu
- https://networkx.github.io/documentation/stable/
- http://www.gurobi.com/documentation/